Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Optimización de cómputo QM/MM empleando arquitecturas masivamente paralelas

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Manuel Ferrería y Juan Pablo Darago

Director: Dr. Esteban Mocskos
Codirector: Dr. Mariano Camilo González Lebrero
Buenos Aires, 2015

RESUMEN

Este trabajo se enfoca en acelerar cálculos de estructura electrónica mediante el uso de arquitecturas de procesadores especializados en cómputo masivos. Se partió de LIO, una implementación ya existente de los algoritmos dados por la Teoría de los Funcionales de la Densidad (DFT) que ya hacía uso de placas de video para cómputo general (GPGPU). Este programa se adaptó para hacer uso de las prestaciones ofrecidas por multiprocesadores, GPUs modernas, y para el coprocesador numérico Xeon Phi de Intel.

Las mejoras se enfocaron en la sección de código más computacionalmente intensiva: el cálculo de la energía de intercambio y correlación. En GPU se buscó cambiar la estrategia de paralelización en la estructura del cómputo y aprovechar la mayor cantidad de memoria de las placas para almacenar más resultados intermedios. La implementación en CPU y Xeon Phi es compartida, aprovechando las prestaciones del compilador para explotar vectorización y paralelismo de manera portable. Se estudió el uso de algoritmos de partición de trabajo y balance de cargas para mantener una división equilibrada de trabajo entre unidades de cómputo, notando el impacto de estas decisiones en la escalabilidad de la implementación.

Las mejoras logradas en el rendimiento de la implementación son de 8 veces por sobre la original en GPU, y de 22 veces en el caso de CPU, para los casos de prueba y configuraciones de *hardware* usados. Los resultados en Xeon Phi resultan comparativamente inferiores a los obtenidos con las otras arquitecturas pero se lograron identificar puntos claves del coprocesador que permitirán continuar con la tarea de optimización. En GPU se implementó y estudio, además, el uso de múltiples placas de video en una misma plataforma, escalando linealmente en función de la cantidad de los dispositivos usados para algunos casos.

Adicionalmente, se observó que las implementaciones en CPU y GPU tienen un rendimiento complementario con respecto a las tareas necesarias para el cálculo de la energía de intercambio y correlación. Esto lleva a pensar que puede lograrse mejoras muy significativas con una implementación híbrida CPU-GPU, usando el *hardware* ya disponible.

Por último, se realizó una comparativa de las arquitecturas estudiadas desde un punto de vista pragmático en el diseño de *clusters* de cómputo. Se espera que las técnicas expuestas en este trabajo sirvan como guía para optimizar aplicaciones de índole similar mediante el uso de estas arquitecturas, y para decidir de manera informada entre las mismas según la clase de problema a resolver.

Palabras claves: QM/MM, DFT, Xeon Phi, CUDA, HPC, *scheduling*.

ABSTRACT

The present work is focused on accelerating electronic structure calculations using massively parallel processor architectures. As a starting point, an existent implementation of the algorithms derived from the Density Functional Theory (DFT), LIO, was studied. LIO employs General Purpose Graphics Processing Unit (GPGPU) for several computations. This software was adapted to use specific features offered by multiprocessors, modern GPUs and the numerical coprocessor Intel Xeon Phi.

The optimization efforts were put on the most time-consuming sections, the exchange-correlation calculations. In GPU the goal was to improve parallelism in the structure of the code and store more temporary results, previously discarded due to insufficient device memory. The CPU and Xeon Phi implementation is shared, exploiting vectorization and paralelization techniques portably, present in modern compilers. The use of work splitting and load balancing algorithms is also studied, in order to keep a balanced work partition between compute units. The impact of different approaches regarding the scalability of the implementation is observed.

The improvements obtained in the performance of the application are up to 8 times over the original in GPU, and over 22 times in the case of the CPU in the hardware employed. The results in Xeon Phi are comparatively inferior to other architectures although key points in the coprocessor architecture were identified to encourage further optimizations. An implementation to employ multiple GPUs in a single system was developed, obtaining linear speedups in some of the studied configurations.

Furthermore, it was noted that CPU and GPU have complementary performance regarding the necessary steps needed to perform the exchange-correlation calculation. This points towards potentially significant improvements regarding a hybrid CPU-GPU implementation, using already available resources.

Finally, a comparison was made between the architectures from the pragmatic point of view of building a compute cluster. Hopefully, the techniques presented in this work can lead to the optimization of similar applications in the usage of these architectures, and help make an educated decision between them according to the problem at hand.

Keywords: QM/MM, DFT, Xeon Phi, CUDA, *scheduling.*

AGRADECIMIENTOS

A Nano y Esteban, por su dedicación a este trabajo, su guía, su paciencia y la buena onda en momentos de crisis o de pánico.

A Manuel, por todos los proyectos que hicimos, los éxitos que tuvimos, los fracasos que enseñaron y por los desastres que haremos en el futuro.

A mi mamá Cecilia y a mi hermano Ignacio, por estar ahí para mí por más que vaya, vuelva, cambie, discuta, pelee, cuelgue o me haya desconcentrado.

A mi papá, Adrián, por creer en mi desde chico e impulsarme a dar lo mejor.

A todos los compañeros y amigos del LSC: A David por sus dibujos y sus locuras, a Maxi por sus consejos y por prenderse al nihilismo académico, a Ema, Nacho y Nahuel por no dejarse desconcentrar por mis cualquiereadas, a Ceci por bancarlas tan estoicamente, y a Seba Galimba por ser la figura cuando lo veíamos venir a la oficina.

Al DC, porque con sus gentes y sus pasillos supo ser el lugar donde crecí como programador, como científico y como persona.

A mis amigos de GGA y GARG: Pato, Tebex, Darío, Doc, Juli, Tavo, Marco y Nico, por los tres meses que (¿sobre?)vivimos juntos, las anecdotas, y por las descarriladas que vendrán.

A mis amigos de la facu: Diego, Pablo A., Juli S., Rodra, Luigi, Guille, Marto, Mati B., Lucho, Javo, Nacho, Ivan B., Peter, Marian, Fede L., Ivan S., Agus G., Rober, Leandro L., Lis, Juan H. y a todos y todas los que estuvieron ahí por mí en estos años de educación universitaria.

A mis amigos Brian, Lebe, Nico Bolo y Fede Brunfman, que me conocieron cuando era una persona totalmente distinta y que compartieron ya una vida conmigo.

A Esteban, por permitirnos interrumpirlo una ilimitada cantidad de veces en cada paso de este trabajo y por conseguir todos los recursos para poder hacer investigación de punta.

A Nano, por el valioso tiempo brindado y las reiteradas explicaciones de química hasta que la pudiéramos entender.

A Juan Pablo, por el trabajo conjunto a lo largo de estos años, culminando con un gran esfuerzo satisfactorio.

A Ana y Alfonso, mis padres, por el apoyo emocional, intelectual, económico y moral a lo largo de la carrera. Imposible hubiera sido sin ustedes.

A Mariana, mi hermana, por haber sido un oído de todas las cosas que se me han ocurrido durante la carrera, aunque no hubiese entendido ni medio.

A Gonzalo y Maximilian, por haber convertido la etapa más estresante en la más divertida de mi vida.

A David y Maxi, por la amistad y la enorme cantidad de ideas aportadas directa e indirectamente a este trabajo.

Al DC, por haberme brindado una educación de primera categoría y permitirme colaborar mínimamente con ella haciendo mi parte como ayudante de la mejor materia de la carrera.

A mis amigos, Guillermo, Daniela, Matías, Luciano, Javier, Nacho, Luis, Guille, Fernando, Leandro, Federico L., Marco, Andŕes, Nicolás, Federico T., Laura, Wanda, y a todo los que no nombré mas por desmemoriado que por desagradecido.

A SIASA, Intel y Nvidia por los recursos computacionales brindados, imprescindibles para poder hacer un estudio serio usando tecnologia de punta.

Índice general

1. INTRODUCCIÓN

Con la aparición de las computadoras, han ganado espacio dentro de las ciencias naturales los métodos de simulación. El uso de estas técnicas permite validar modelos teóricos así cómo también brindar información detallada (macro y microscópica) del proceso simulado.

En el ámbito de la química existen diferentes modelos que permiten simular procesos de interés. Dentro de estos modelos hay dos que se destacan por su uso:

- Los métodos basados en la mecánica cuántica, que proporcionan una descripción de la estructura electrónica del sistema.
- Los métodos basados en la mecánica molecular, donde las moléculas son tratadas mediante un campo de fuerza clásico y los electrones no son tenidos en cuenta explícitamente.

1.1. Modelos Cuánticos (QM)

El comportamiento de los fenoemos a pequeña escala (nanométrica) está regido por las leyes de la mecánica cuántica. Este marco teórico desarrollado a comienzos del siglo XX propone que las partículas (cómo electrones y protones) pueden (y deben en algunos casos) ser descriptas cómo ondas. Así, cualquier propiedad de un sistema está determinado por una función llamada *función de onda* (Ψ) que satisface la ecuación de Schrödinger dependiente del tiempo:

$$i\hbar\frac{\partial\Psi}{\partial t}(\mathbf{r},t) = \frac{-\hbar^2}{2m}\nabla^2\Psi(\mathbf{r},t) + V(\mathbf{r},t)\Psi(\mathbf{r},t) \tag{1.1}$$

donde $\mathbf{r} = (r_1, \ldots, r_n)$ es el vector de todas las posiciones de las partículas del sistema, m es la masa de la partícula, V es un potencial que afecta a las partículas y $\hbar$ es la constante de Planck divida por 2π.

Si el campo externo no depende del tiempo esta ecuación se puede simplificar a la de Schrödinger *independiente del tiempo*:

$$\hat{H}\Psi(\mathbf{r}) = E\Psi(\mathbf{r}) \tag{1.2}$$

donde E es la energía asociada a la función de onda Ψ y el operador hamiltoniano $\hat{H}$ se define como

$$\hat{H} = -\frac{\hbar^2}{m}\nabla^2 + \hat{V}$$

Ahora, si bien resolver esta ecuación diferencial sería suficiente para determinar todas las propiedades del sistema, esto no puede hacerse de manera exacta cuando hay más de un electrón en el mismo. Por este motivo, para problemas de mayor tamaño se utilizan aproximaciones para obtener una solución de la ecuación 1.2.

Existen diversos métodos para resolver de forma aproximada esta ecuación con diferente costo computacional y calidad de la respuesta obtenida. Dentro de estos métodos hay

uno que destaca por su excelente relación costo/calidad, el método basado en la Teoría de los Funcionales de la Densidad (DFT, Density Functional Theory) desarrollada por Hohenberg y Kohn en 1964.

En este marco teórico, la *densidad electrónica* ρ que representa la probabilidad de encontrar un electrón en cada región del espacio ocupa un rol destacado. La base de este método consiste en dos teoremas publicados por Hohenberg y Kohn [1]. Estos autores demuestran que ρ y V (y por lo tanto ψ) se encuentran relacionadas biunívocamente, es decir, que una dada densidad electrónica contiene la misma información que la función de onda. De esta manera, cualquier observable (cómo la energía) puede ser representado cómo un funcional de la densidad (de allí el nombre de esta teoría).

Además propusieron la dependencia de la energía del sistema cómo funcional de la densidad de la siguiente forma:

$$E[\rho] = T_s[\rho] + V_{ne}[\rho] + \frac{1}{2}\int\int \frac{\rho(\vec{r}_1)\rho(\vec{r}_2)}{r_{12}} d\vec{r}_1 d\vec{r}_2 + E_{XC}[\rho] \tag{1.3}$$

Donde $T_s[\rho]$ es la energía cinética asociada con la densidad, $V_{ne}[\rho]$ es la energía potencial producto de la interacción entre los electrones (la densidad) y los núcleos, el tercer término es el resultado de la repulsión de Coulomb entre electrones y $E_{XC}[\rho]$ es la energía de intercambio y correlación. Este último término da cuenta de la energía asociada a escribir la función de onda de manera que cumpla con el principio de exclusión de Pauli y de la correlación en la posición instantánea de los electrones.

Esta formulación es exacta si se conociera el término $E_{XC}[\rho]$ dado que los demás tienen solución analítica. En las aproximaciones hechas para calcularlo reside tanto la calidad cómo el costo computacional de este tipo de simulaciones. Por ello, el objetivo central de este trabajo consiste en disminuir el tiempo insumido en el cómputo de el término $E_{XC}[\rho]$.

La forma comúnmente utilizada para calcular este término se basa en definir un funcional local ϵ_{XC} que depende de la densidad (*Local Density Aproximation*, LDA) o de la densidad y su gradiente (*General Gradient Approximation*, GGA) en cada punto del espacio.

De esta manera se puede calcular E_{XC} mediante la integral:

$$E_{XC} = \int \rho(r)\epsilon_{XC}\left(\rho(r)\right) d\vec{r} \tag{1.4}$$

Esta ecuación puede ser aproximada mediante una suma utilizando una grilla de j puntos, con pesos ω_j según:

$$E_{XC} \approx \sum_j \omega_j \rho(r_j)\epsilon_{XC}(\rho(r_j)) \tag{1.5}$$

Otro aspecto importante de esta teoría es que provee una manera de calcular la densidad ρ, mediante el denominado método de Kohn-Sham [2]. Este método se basa en el segundo teorema de Hohenberg y Khon, que establece que para cualquier densidad electrónica de prueba ρ^* que cumpla que $\int \rho^*(\vec{r})d\vec{r} = N$, donde N es la cantidad de electrones del sistema, vale que:

$$E[\rho^*] \geq E[\rho] \tag{1.6}$$

Esto produce un método auto-consistente para calcular ρ. Se empieza con una aproximación inicial ρ^* y se itera el cálculo de la misma hasta alcanzar un mínimo para E.

Este marco teórico es uno de los métodos más populares en problemas de estado sólido, especialmente desde la década del 90 cuando se mejoraron las aproximaciones para modelado de interacciones. El valor de esta teoría para el estudio de las propiedades de la materia le valió a Kohn el Premio Nobel de Química en 1998.

Si bien la relación costo-calidad de este método es muy buena, el costo computacional asociado sigue siendo elevado, lo que limita su aplicación a sistemas pequeños (cientos de átomos como máximo).

Por este motivo, se han desarrollado los métodos conocidos como *mecánica molecular* (MM). En estos métodos los átomos son tratados cómo esferas cargadas y las uniones químicas como resortes (potenciales armónicos). De esta manera, los electrones no son considerados explícitamente, reduciendo drásticamente el costo computacional asociado. Esta técnica es muy poderosa para representar sistemas o procesos en los que no cambia la distribución electrónica.

Sin embargo, en muchos de los problemas de interés en química y bioquímica (por ejemplo una reacción química en solución o en el sitio activo de una proteína), el modelado requiere simultáneamente de la representación de miles de átomos y de un tratamiento explícito de los electrones. Para resolver esto, se han desarrollado técnicas híbridas QM/MM (*Quantum Mechanical / Molecular Mechanics*).

Dentro de este esquema se subdivide el sistema en dos partes:

i) Una parte en la que la estructura electrónica cambia. Se lo modela usando mecánica cuántica (QM).

ii) Para el resto del sistema se aplica un campo de fuerzas clásico (MM).

De esta manera se puede expresar la energía del sistema QM/MM cómo:

$$E = E_{QM} + E_{QM/MM} + E_{MM} \tag{1.7}$$

donde la energía E_{QM} se obtiene mediante el método DFT visto más arriba, la energía E_{MM} proviene de simular el campo de fuerzas clásico y $E_{QM/MM}$ surge de la interacción entre las regiones QM y la regiones MM del modelo.

Esta última se calcula, en este trabajo, mediante la ecuación:

$$E_{QM/MM} = \sum_{l=1}^{N_c} q_l \int \frac{\rho(r)}{\mid r - R_l \mid} + \sum_{l=1}^{N_c} \sum_{\alpha=1}^{N_q} [v_{LJ}(\mid R_l - \tau_\alpha \mid) + \frac{q_l z_\alpha}{\mid R_l - \tau_\alpha \mid}] \tag{1.8}$$

donde el primer término da cuenta de la interacción entre una carga puntual del sistema clásico con la densidad electrónica, y el segundo término representa la interacción entre los núcleos clásicos con los cuánticos mediante un potencial de Lennard-Jones y la interacción Coulombica entre las cargas.

Los métodos *QM/MM*, dentro del marco de métodos multiescala, son ampliamente utilizados en la práctica. Estos modelos han valido a Karplus, Levitt y Warshel el premio Nobel de Química en 2013, por su valor para la simulación de sistemas complejos.

Nuestro trabajo se realiza en base a programas ya existentes. El cálculo de E_{QM} y $E_{QM/MM}$ son realizados por la aplicación LIO [3, 4], el cual fue optimizado en este trabajo para el uso de distintas arquitecturas de CPU y GPU. Este paquete se complementa mediante el uso del programa de dinámica molecular Amber [5], que realiza el cálculo de E_{MM}. Nos concentraremos en las partes computacionalmente más intensivas de LIO,

que corresponden a la implementación de los cálculos de los términos de intercambio y correlación ya introducidos en esta sección.

El diseño de estos algoritmos depende del sistema a tratar. Las simulaciones híbridas requieren la resolución de la estructura electrónica en cada paso de dinámica. Esto restringe el tamaño del problema a resolver a decenas o a lo sumo cientos de átomos. Por estos motivos, el presente trabajo se concentra en la optimización de los algoritmos aplicados a estos sistemas.

1.2. Cómputo de alto rendimiento

Una parte significativa del impacto de las computadoras en los distintos aspectos de las ciencias, incluso de la vida diaria, se debe al crecimiento de su poder de cómputo desde la aparición de los primeros microprocesadores hasta los disponibles hoy en día.

Entre 1986 y 2002, la *performance* de los procesadores creció, en promedio, un 50 % por año [6]. La denominada *Ley de Moore* establece que la densidad de transistores por circuito integrado se duplica cada 18 meses [7]. Esto puede verse en la figura 1.1, que compara la cantidad de transistores de distintos procesadores desde 1971 a la actualidad.

Este crecimiento permitió a los desarrolladores de procesadores incrementar la potencia de cálculo, duplicando la eficiencia cada 24 meses. Esto benefició durante mucho tiempo a los desarrolladores de aplicaciones, quienes escribían programas secuenciales, que solo debían esperar a la aparición de los nuevos modelos de procesadores para ver una reducción sustancial en los tiempos de ejecución de sus aplicaciones.

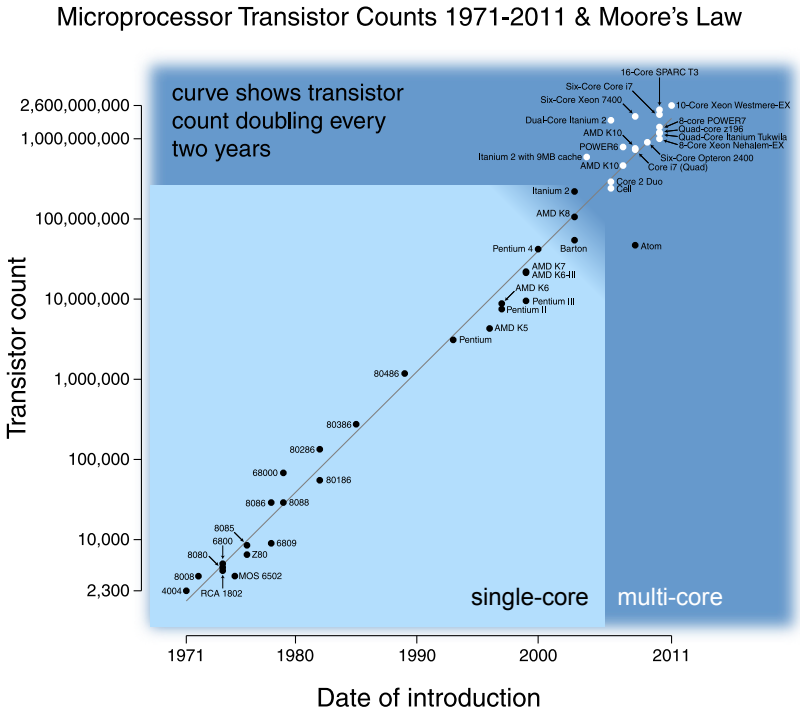


Fig. 1.1: Cantidad de transistores en procesadores emblemáticos desde 1971

Sin embargo, a medida que los transistores disminuyen su tamaño, aumentan su disipación térmica por unidad de superficie. Esto limita la cantidad que se pueden ubicar en

un circuito sin producir que este se comporte de manera errática. El mismo motivo impide continuar el crecimiento de la frecuencia del reloj.

Los problemas térmicos han implicado que desde el 2002, la tasa de crecimiento de la *performance* de los monoprocesadores haya disminuido a un 20 % anual. Consecuentemente, los principales fabricantes de procesadores han modificado el enfoque de investigación y diseño, empezando a hacerse más y más común el uso de múltiples procesadores por chip.

La preocupación por la disipación y el consumo energético insustentables han sido motivadores de diseños con menor frecuencia de reloj, pero aprovechando las aún crecientes densidades de transistores para incrementar las unidades de soporte. Esta estrategia ha resultado en que, en un CPU moderno, menos del 20 % de todos los transistores disponibles se utilicen para realizar cálculos.

Adicionalmente, las mejoras de *performance* debidas al paralelismo a nivel de instrucción mediante técnicas como ejecución fuera de orden, ejecución especulativa, *pipelining*, etc., han sido progresivamente menores. Actualmente, los esfuerzos invertidos en ese área se han concentrado en el paralelismo a nivel de datos (vectorización) y paralelismo a nivel de tareas (multiprocesadores) [7].

Este enfoque en diseño de arquitecturas hacia otros tipos de paralelismo puede verse tanto en nuevos productos en las líneas establecidas (por ejemplo los procesadores Intel i3, i5 e i7) así como también en nuevos desarrollos que apuntan a cómputo de alta *performance*. La revalorización de las placas gráficas (GPUs) para problemas de cómputo intensivo, y los desarrollos nuevos como la arquitectura MIC (*Many Integrated Core Architecture*) de Intel son claros ejemplos de esta tendencia.

El impacto de este enfoque hacia múltiples hilos de ejecución en paralelo en el desarrollo de aplicaciones es significativo. En simulaciones para las áreas de biología, medicina, química o meteorología es de gran interés minimizar los tiempos de ejecución, para permitir realizar predicciones de mayor calidad usando modelos más sofisticados. Aprovechar las nuevas arquitecturas multiprocesador requiere modificaciones en el código que resultan no triviales, a diferencia del crecimiento en la velocidad de reloj que no requería modificaciones en el diseño del programa. Los intentos de escribir programas que conviertan programas seriales (diseñados para un solo procesador) a paralelos, en lenguajes de propósito general como C, C++ o Fortran, han sido relativamente infructuosos [6].

Existen, sin embargo, herramientas que realizan transformaciones de código fuente serial a código fuente con anotaciones de paralelización automáticas (usando bibliotecas de paralelización asistida como OpenMP para las anotaciones). Ejemplos de estas herramientas incluyen Par4All [8] y Cetus [9]. Estas herramientas pretenden, además, generar código para aceleradores especiales como GPUs.

Como consecuencia, resulta necesario el trabajo a nivel de desarrollo de código para utilizar múltiples procesadores. La aparición de nuevas herramientas ayudan al programador en la tarea del uso eficiente de los recursos computacionales existentes. Un ejemplo de esto es Nvidia CUDA (*Compute Unified Device Architecture*), que provee una arquitectura y lenguaje de programación para el desarrollo de aplicaciones que exploten los recursos provistos por tarjetas gráficas, constituyendo la metodología conocida como GPGPU (*General Purpose Graphical Processing Units*). Otros ejemplos se pueden ver en APIs y bibliotecas unificadas de desarrollo como OpenMP o MPI (*Message Passing Interface*), trabajando conjuntamente con compiladores optimizantes como Intel ICC y PGI Fortran.

Estas herramientas, si bien resultan una ayuda muy importante para el programador,

no constituyen una panacea. Todavía la división del trabajo es inherente al problema a resolver en base a las dependencias de las tareas involucradas. Realizar esta división es una labor que, hasta el día de hoy, es responsabilidad del programador especializado.

En este trabajo, se busca comparar distintas arquitecturas de hardware y cómo las características específicas de la simulación química a realizar permiten o impiden la paralelización de trabajo empleando los distintos recursos específicos que cada arquitectura provee.

2. ARQUITECTURAS EN PROFUNDIDAD

2.1. CPU

Los microprocesadores ganan en prevalencia desde principios de los años 70, cuando Intel introduce los modelos `4004` y `8008`. Actualmente, las arquitecturas de procesadores de 64 bits basadas en la línea `x86-64` de Intel dominan no solo el mercado de computadoras personales sino que también el de servidores y *clusters* de cómputo. Dado que utilizaremos el CPU estándar como punto de comparación para las demás arquitecturas, y que éstas están basadas en parte en su diseño, daremos una breve reseña de los aspectos más importantes y desarrollos modernos con respecto a la *performance* dentro de los procesadores actuales. Haremos foco en el concepto de *paralelismo* en las arquitecturas: la posiblidad de ejecutar simultaneamente flujos de instrucciones independientes entre sí.

2.1.1. Tipos de paralelismo

Existen tres categorías de paralelismo que, hoy en día, una arquitectura puede aprovechar para mejorar la *performance* de una aplicación:

- *Instruction Level Parallelism*: Este tipo de optimizaciones buscan ejecutar la mayor cantidad de instrucciones en un mismo hilo de ejecución simultáneamente. Optimizaciones de este estilo incluyen:
  - *pipelines* de procesador: se separan las instrucciones en distintas etapas para ejecutar múltiples instrucciones de manera solapada. Cada etapa es un estadío dentro de una cadena de montaje que empieza decodificando una instrucción y la ejecuta de a partes.
  - ejecución superescalar fuera de orden: ejecutar al mismo tiempo instrucciones que no usan recursos comunes y son independientes entre sí.
  - ejecución especulativa: se basa en la predicción de resultados todavía no finalizados de procesar.
- *Data Level Parallelism*: Consideran las optimizaciones cuyo propósito es lograr aplicar una misma operación a cada elemento de un conjunto datos simultáneamente en un mismo hilo de ejecución. Esta técnica se denomina SIMD (*Single Instruction, Multiple Data*).
- *Thread Level Parallelism*: Concierne al uso de múltiples hilos de ejecución simultáneos, lo cual requiere el uso de procesadores que usualmente comparten la memoria principal (arquitectura SMP, *Symmetric Multiprocessing*). Esto normalmente requiere esfuerzo adicional por parte del programador para mantener consistencia y coherencia.

En base a estos tipos de paralelismo surgieron grandes avances en la arquitectura de procesadores desde 1950 en adelante. Estos se detallan cronológicamente en la tabla 2.1, como también la motivación de su existencia.

A continuación detallamos algunos aspectos de cada una de estas técnicas.

Año	Tecnología	Motivo
1950s	Pipelines	Usar independientemente las distintas unidades del procesador para disminuir los ciclos por instrucción.
1970s	Vectorización	Poder realizar las mismas operaciones sobre muchos datos simultáneamente.
1980s	Cache	Disminuir la latencia de muchos accesos a la memoria aprovechando la localidad de los mismos.
1987	Múltiples ALUs	Poder compensar las ALU que más latencia tienen, se multiplican las unidades para procesar más instrucciones en paralelo.
1988	Predicción de Saltos	Disminuir y hasta evitar los costos de tener que recalcular el pipeline cuando hay saltos.
1995	Fuera de Orden	Solapar la ejecución de instrucciones para compensar por instrucciones lentas y accesos a memoria.
2002	SMT	Poder procesar de a más de un *thread* de OS en un solo core.
2005	Multi-core	Aumentar el poder de cómputo sin aumentar la velocidad de clock.

Tab. 2.1: Cronología de los avances de arquitectura de procesadores, tomado de [7]

2.1.2. Pipeline y Ejecución fuera de orden

Los primeras implementaciones de paralelismo a nivel de un solo procesador fueron a nivel de instrucciones mediante el uso de *pipelines* de múltiples etapas. Por ejemplo, en la arquitectura del Intel Pentium 4 se llegaron a utilizar 20 etapas distintas de pipeline. Cada etapa corresponde a una actividad distinta en el proceso de ejecutar una instrucción. Al tiempo que una instrucción es decodificada, por ejemplo, otra instrucción puede estar siendo leída de memoria ya que, idealmente, las etapas previas no dependen de las posteriores. Este mecanismo funciona bien siempre y cuando una instrucción no dependa de los resultados de otra anterior. Sin embargo, ocurre habitualmente que existe una dependencia entre instrucciones, produciéndose entonces una demora (o *pipe stall*) que requiera ejecutar las instrucciones de manera no solapada (con el costo de *throughput* de instrucciones que ello implica).

Esta técnica llevada a su conclusión lógica se conoce como ejecución fuera de orden (*Out of Order Execution*). Mediante el uso de algoritmos y circuitos dedicados, un procesador puede detectar las dependencias entre las instrucciones y cambiar el orden de ejecución para minimizar la aparición de pipe stalls, manteniendo los mismos resultados. De esta manera, se logra que la mayor parte de las unidades del procesador permanezcan ocupadas el mayor tiempo posible.

Mejoras en este nivel eran usualmente invisibles al programador de un lenguaje de alto nivel como C++, dejando la tarea de aprovechar estas propiedades de la arquitectura a los compiladores optimizantes. Sin embargo, las ventajas de estas técnicas fueron disminuyendo a partir del año 2000 [7].

2.1.3. Extensiones vectoriales

Si bien las técnicas SIMD fueron desarrolladas para las supercomputadoras de los años 70 y 80, su aparición en los microprocesadores x86 modernos ocurre en 1996 con el nombre MMX (*MultiMedia eXtensions*), con mejoras luego en las extensiones SSE y AVX. AVX y AVX2 representan la última versión disponible de las instrucciones de vectorización y están presentes en la línea Intel Xeon de procesadores de alta gama y en las más recientes generaciones de procesadores para consumidores.

El paralelismo de datos puede ser explotado por el compilador, que analiza los ciclos de programa y detecta cuando hay operaciones independientes que pueden ser realizadas en simultáneo, dividiendo la cantidad de instrucciones totales que tiene que realizar un procesador. Un ejemplo de operación de suma vectorial puede verse en la figura 2.1.

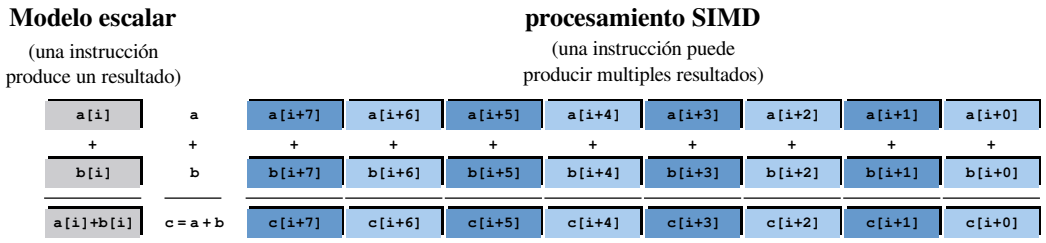


Fig. 2.1: Ejemplo de operación de suma simultánea de ocho elementos. En SSE 4, usando la instrucción `ADDPS`, el procesador puede hacer estas 16 sumas en simultáneo usando un registro de 128 bits. En cambio, un procesador escalar solo podría hacerlas de a una por vez.

El uso de operaciones sobre múltiples valores ha cobrado importancia como uno de los métodos de incrementar la *performance* de ejecución. La longitud de registros SIMD de las extensiones (64 bits para MMX, 128 para SSE, 256 para AVX) se ha duplicado cada seis años, con lo cual es importante para una aplicación que sus operaciones sean lo más vectorizables posible [7]. Para esto, es ideal que las operaciones sean regulares y los ciclos sean claros y con mínimas dependencias, de modo de hacer mejor uso de estas facilidades.

2.1.4. Caches

A diferencia de los procesadores, la velocidad de acceso de las memorias principales no aumentó de una manera tan significativa, como se puede ver en la figura 2.2. Como consecuencia, la memoria empezó a convertirse en un serio cuello de botella a la velocidad de ejecución de los programas.

El concepto de *localidad espacial* corresponde con la observación de que los datos con los que opera una sección de un programa suelen estar *cerca* en memoria. Los diseños de procesadores empezaron a incluir distintos tipos de caches para sacar provecho de esta situación: memorias rápidas, próximas al CPU y de menor tamaño para contener el subconjunto de los datos *cercano* al recientemente usado. Su eficacia impulsó el establecimiento de una jerarquía en orden creciente de tamaño y decreciente en velocidad, empezando por las caches L1 y siguiendo por las L2 y L3.

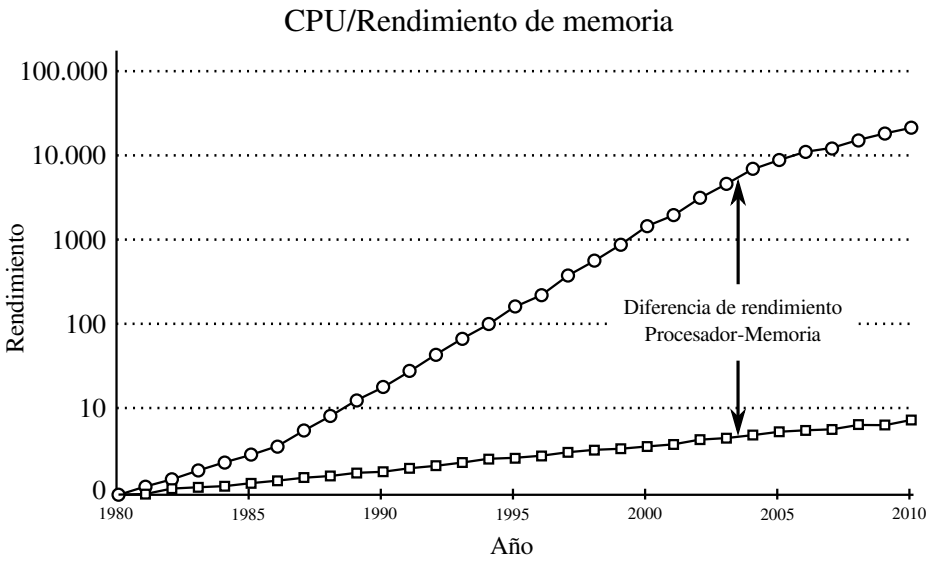


Fig. 2.2: Comparación entre la *performance* de CPU y memoria según el año, la diferencia de entre ellos se denomina *memory gap*. Tomado de [7].

El tamaño de una cache L1 de CPU moderno está en el orden de los 64 KB, una cache L2 en el orden de los 2 MB y una L3 en el orden de 6 MB en adelante.

Si bien la aparición y utilización de caches es transparente al programador en los procesadores de la línea `x86-64`, los accesos irregulares a la memoria pueden producir que la cache se cargue con datos que no volverán a ser utilizados, causando que datos que sí se vayan a reutilizar sean desalojados. El evento en que datos no se encuentren en memoria cache y deban ser buscados en la memoria principal se denomina *cache miss* y tiene un fuerte impacto en el rendimiento del programa. Por esto es que la regularidad de los accesos a memoria para hacer buen uso de caches resulta fundamental para obtener una buena *performance*.

2.1.5. Multiprocesadores

Dentro del área de cómputo de escritorio, servidores y estaciones de trabajo basadas en la línea `x86`, los procesadores MIMD (*Multiple Instruction Multiple Data*) implicaron una revolución en el abordaje de los problema de alta *performance*, pero cada procesador individual continúa las líneas anteriores. Los diseños más utilizados se basan en un arquitectura tipo SMP (*Symmetric Multiprocessing*), en la cual todos los procesadores son iguales y comparten una misma memoria principal. Cada procesador tiene sus propios registros y se comunica con los demás mediante memoria compartida o interrupciones.

Por ejemplo, el procesador Intel Xeon `E7-8800` posee 12 procesadores (núcleos o *cores*) que pueden ejecutar dos hilos simultáneamente cada uno, mediante el uso de la tecnología denominada *Hyper-Threading*.

A diferencia de los otros métodos, las mejoras posibles mediante el procesamiento paralelo en tareas son sustanciales, pero dependen del programador en gran medida. Un programa serial no se beneficiará de múltiples *cores*, incluso siendo recompilado, a menos que este paralelismo se aproveche explícitamente. Otro aspecto importante es la *escalabilidad*, que consiste en que la división de tareas mantenga a todos los procesadores disponibles

ocupados, aunque crezca la cantidad de cores que intervengan.

Un resultado importante a tener en cuenta es la denominada *Ley de Amdahl* [10], que establece una relación entre la aceleración (*speedup*) máximo alcanzable mediante un incremento en la cantidad de procesadores disponibles, el porcentaje de la aplicación que es paralelizable y el porcentaje que no lo es:

$$S(n) = \frac{1}{(1 - B) + \frac{B}{n}} \tag{2.1}$$

donde S es el *speedup* máximo de mejora alcanzable, B es la fracción del algoritmo a ejecutar que se encuentra paralelizada, y n la cantidad de hilos de ejecución paralelos con los que se dispone.

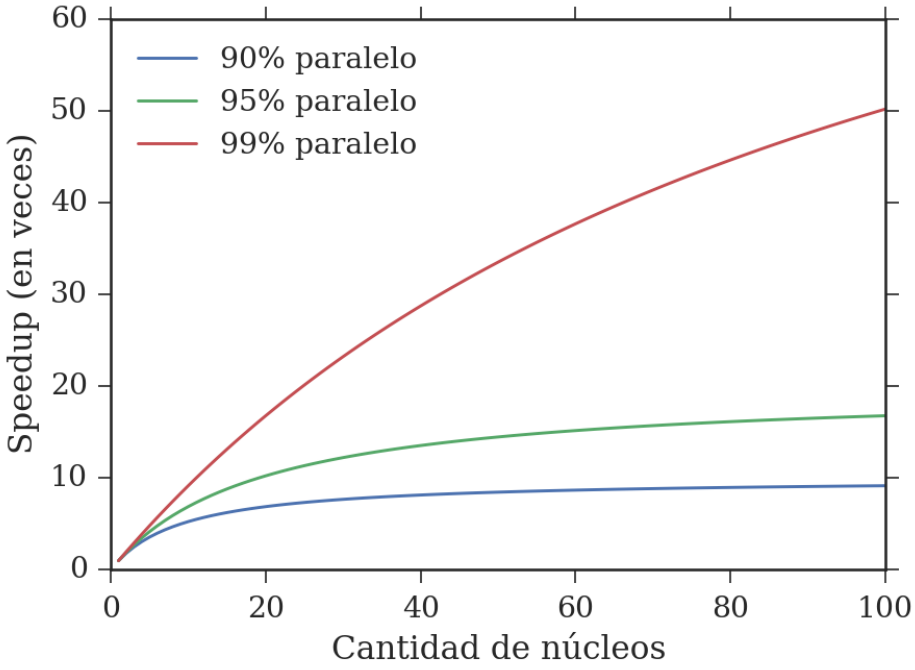


Fig. 2.3: Aceleración teórica máxima (en veces) dada por la ley de Amdahl, según la cantidad de núcleos de procesamiento.

Un ejemplo de esta ley en acción es que si 95 % del problema fuera paralelizable entonces el límite teórico de mejora es de 20 veces (el programa corriendo sobre infinitos cores se ejecutaría en un veinteavo del tiempo que originalmente requería). La figura 2.3 muestra el comportamiento de la ecuación 2.1, que define la ley de Amdahl, con distintas fracciones de código paralelo.

La ley de Amdahl describe el pico teórico de mejora posible. Esta ley es una simplificación, ya que supone que todos los cores tienen trabajo perfectamente distribuido, que no deben comunicarse entre ellos por motivos de sincronización y que no existen otras cargas adicionales introducidas por paralelización misma.

Por otro lado, la presencia de un componente común (la memoria) puede representar cuellos de botella en el acceso a los datos, ya que si el *bus* de memoria es saturado con pedidos, los procesadores deben detener forzosamente su ejecución hasta que los datos estén disponibles, eliminándose entonces el procesamiento paralelo.

Otro punto de conflicto son las caches. Como los procesadores deben tener una visión unificada y consistente de la memoria, a veces es necesario que estos sincronicen los valores de sus caches, especialmente ante una escritura de memoria. Esto se conoce como *coherencia de caches* e involucra una sincronización de alto *overhead*, ya que implica coordinación entre dos o más procesadores a través de un bus de memoria.

Es, en este punto, que el impacto del paralelismo en el comportamiento del programa puede ser tan fuerte como sutil. Un fenómeno que ilustra esto es el de *false sharing* (figura 2.4), que sucede cuando una variable no compartida entre *threads* reside en la misma línea de cache con una que sí. En ese caso, la variable es pasada de lado a lado entre cores aunque nunca fuese necesario, decrementando la escalabilidad del algoritmo y siendo difícil de detectar al depender intrínsecamente del sistema en el que se ejecuta.

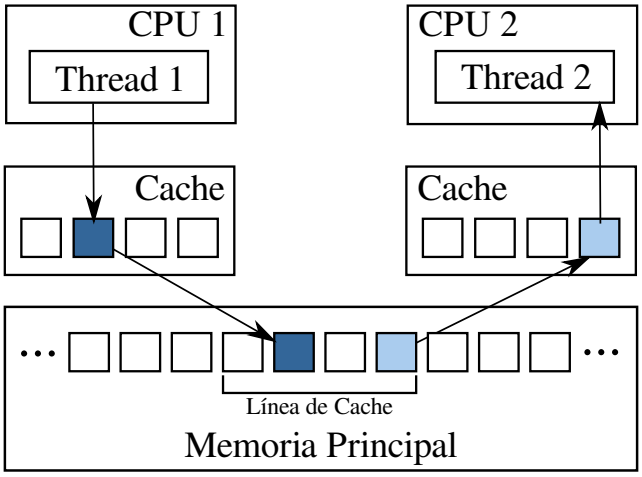


Fig. 2.4: Esquema para ilustrar el problema de *false-sharing* entre caches en un esquema multicore. El *thread* 1 escribe un valor en su cache y el *thread* 2 lee un valor en su cache. Como ambos valores pertenecen a la misma línea de cache la escritura de 1 invalida la línea de donde lee 2, haciendo que tenga que releer toda la línea.

Los desafíos generados por la adición de cores influencian fuertemente los diseños de una arquitectura, no solo en procesadores estándar sino también en aceleradores como las GPUs.

2.2. CUDA

2.2.1. Introducción

Una de las arquitecturas analizadas en este trabajo es la arquitectura GPU desarrollada por Nvidia, conocida como CUDA por las siglas en ingles de *Compute Unified Device Architecture*. CUDA surge naturalmente de la aplicación del hardware desarrollado para problemas gráficos, pero aplicados al cómputo científico.

Las placas de vídeo aparecen en 1978 con la introducción de Intel del chip `iSBX 275`. En 1985, la Commodore Amiga incluía un coprocesador gráfico que podía ejecutar instrucciones independientemente del CPU, un paso importante en la separación y especialización de las tareas. En la década del 90, múltiples avances surgieron en la aceleración 2D para dibujar las interfaces gráficas de los sistemas operativos y, para mediados de la década, muchos fabricantes estaban incursionando en las aceleradoras 3D como agregados a las

placas gráficas tradicionales 2D. A principios de la década del 2000, se agregaron los *shaders* a las placas, pequeños programas independientes que corrían nativo en el GPU, y se podían encadenar entre sí, uno por pixel en la pantalla [11]. Este paralelismo es el desarrollo fundamental que llevó a las GPU a poder procesar operaciones gráficas órdenes de magnitud más rápido que el CPU.

En el 2006, Nvidia introduce la arquitectura G80, que es el primer GPU que deja de resolver únicamente problemas de gráficos para pasar a un motor genérico donde cuenta con un set de instrucciones consistente para todos los tipos de operaciones que realiza (geometría, vertex y pixel shaders) [12]. Como subproducto de esto, la GPU pasa a tener procesadores simétricos más sencillos y fáciles de construir. Esta arquitectura es la que se ha mantenido y mejorado en el tiempo, permitiendo a las GPU escalar masivamente en procesadores simples, de baja frecuencia de reloj y con una disipación térmica manejable.

Los puntos fuertes de las GPU modernas consisten en poder atacar los problemas de paralelismo de manera pseudo-explícita, y con esto poder escalar "fácilmente" si solamente se corre en una placa con más procesadores [13].

Técnicamente, esta arquitectura cuenta con entre cientos y miles de procesadores especializados en cálculo de punto flotante, procesando cada uno un *thread* distinto pero trabajando de manera sincrónica agrupados en bloques. Cada procesador, a su vez, cuenta con entre 63 a 255 registros [14, 15]. Las GPU cuentas con múltiples niveles de cache y memorias especializadas (subproducto de su diseño fundamental para gráficos). Estos no poseen instrucciones SIMD, ya que su diseño primario esta basado en cambio, en SIMT (*Single Instruction Multiple Thread*), las cuales se ejecutan en los bloques sincrónicos de procesadores. De este modo, las placas modernas como la Nvidia Tesla K40 alcanzan poder de cómputo de 4,3 TFLOPs (4300 mil millones de operaciones de punto flotante por segundo) en cálculos de precisión simple, 1,7 TFLOPs en precisión doble y 288 GB/seg de transferencia de memoria, usando 2880 CUDA Cores [16]. Para poner en escala la concentración de poder de cálculo: una computadora usando solo dos de estas placas posee una capacidad de cómputo comparable a la supercomputadora más potente del mundo en Noviembre 2001 [17]. Una comparativa del poder de cómputo teórico entre GPUs y CPUs puede ver en la figura 2.5.

Para poder explotar la arquitectura CUDA, los programas deben ser diseñados de manera de que el problema se pueda particionar usando el modelo de grilla de bloques de *threads*. Para este propósito es que Nvidia desarrolló el lenguaje CUDA.

Hoy en día, poder aprovechar la potencialidad de las GPU requiere una reescritura completa de los códigos ya existentes desarrollados para CPU y un cambio de paradigma importante, al dejar de tener vectorización, paralelización automática y otras técnicas tradicionales de optimización en CPU. Sin embargo, este trabajo ha rendido sus frutos en muchos casos: en los últimos seis años, la literatura de HPC con aplicaciones en GPU ha explotado con desarrollos nuevos basados en la aceleración de algoritmos numéricos (su principal uso). Por este motivo, este trabajo no ahondará en las particularidades del lenguaje CUDA y su modelo de paralelismo, más allá de lo estrictamente necesario para analizar *performance*. Para más información se puede consultar la bibliografia [12, 18, 19].

Además, no todas las aplicaciones deben reescribirse de manera completa. Con la introducción de las bibliotecas `CuBLAS` y `CuFFT`, se ha buscado reemplazar con mínimos cambios las históricas bibliotecas `BLAS` y `FFTw`, piedras fundamentales del cómputo HPC [20, 21].

Nuevas soluciones para la portabilidad se siguen desarrollando: las bibliotecas como `Thrust` [22], `OpenMP` 4.0 [23] y `OpenACC` 2.0 [24] son herramientas que buscan generar

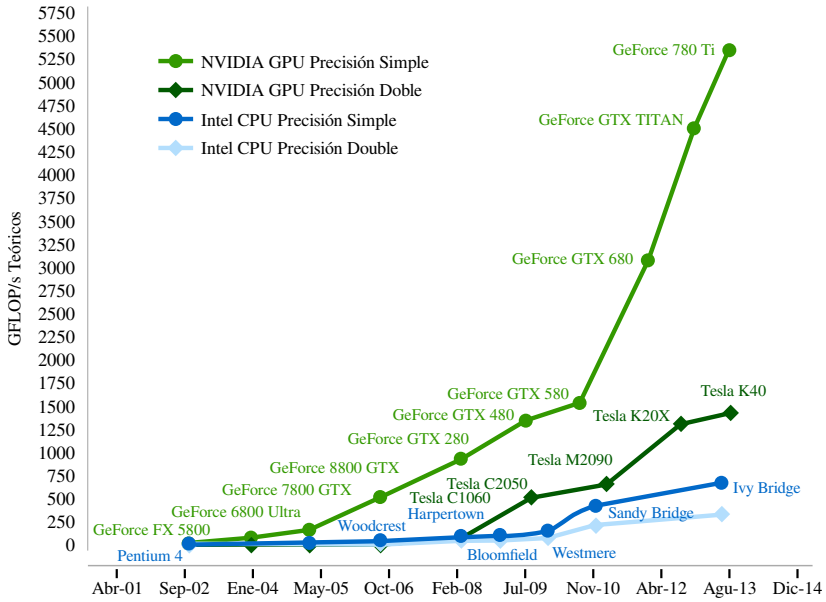


Fig. 2.5: Picos teóricos de *performance* en GFLOPS/s. Tomado de [13].

código que puedan utilizar eficientemente el acelerador de cómputo que se haya disponible. Estas herramientas permiten definir las operaciones de manera genérica y dejan el trabajo pesado al compilador para que subdivida el problema de manera que el acelerador (CPU, GPU, MIC) necesite. Obviamente, los ajustes finos siempre quedan pendiente para el programador especializado, pero estas herramientas representan un avance fundamental al uso masivo de técnicas de paralelización automáticas, necesarias hoy día y potencialmente imprescindibles en el futuro.

2.2.2. Organización de procesadores

Los procesadores GPGPU diseñados por Nvidia han sido reorganizados a lo largo de su existencia múltiples veces pero conservan algunas líneas de diseño a través de su evolución. A continuación se describe la organización definida en la arquitectura `Fermi` y luego analizaremos las diferencias con `Kepler`.

Las arquitecturas de las GPUs se centran en el uso de una cantidad escalable de procesadores *multithreaded* denominados *Streaming Multiprocessors* (SMs). Un multiprocesador esta diseñado para ejecutar cientos de threads concurrentemente, usando sus unidades aritméticas llamadas *Streaming Processors* (SPs). Las instrucciones se encadenan para aprovechar el paralelismo a nivel instrucción dentro de un mismo flujo de ejecución, y funcionando en conjunto con el paralelismo a nivel de *thread*, usado de manera extensa a través del hardware. Todas las instrucciones son ejecutadas en orden y no hay predicción de saltos ni ejecución especulativa, todo se ejecuta solamente cuando se lo necesita [19].

Los SMs (figura 2.6) son unidades completas de ejecución. Cada uno de ellos tiene 32 SPs interconectados entre sí que operan sobre un *register file* de 64 KB común a todos. Los SMs cuentan con múltiples unidades de *Load/Store*, que permiten realizar accesos a memoria independientes. Existen cuatro unidades de SFU (*Special Function Unit*) por SM, para realizar rápidamente operaciones matemáticas trascendentales (trigonométricas, potencias, raíces, etc.). Cada SM ejecuta simultáneamente una cantidad fija de threads,

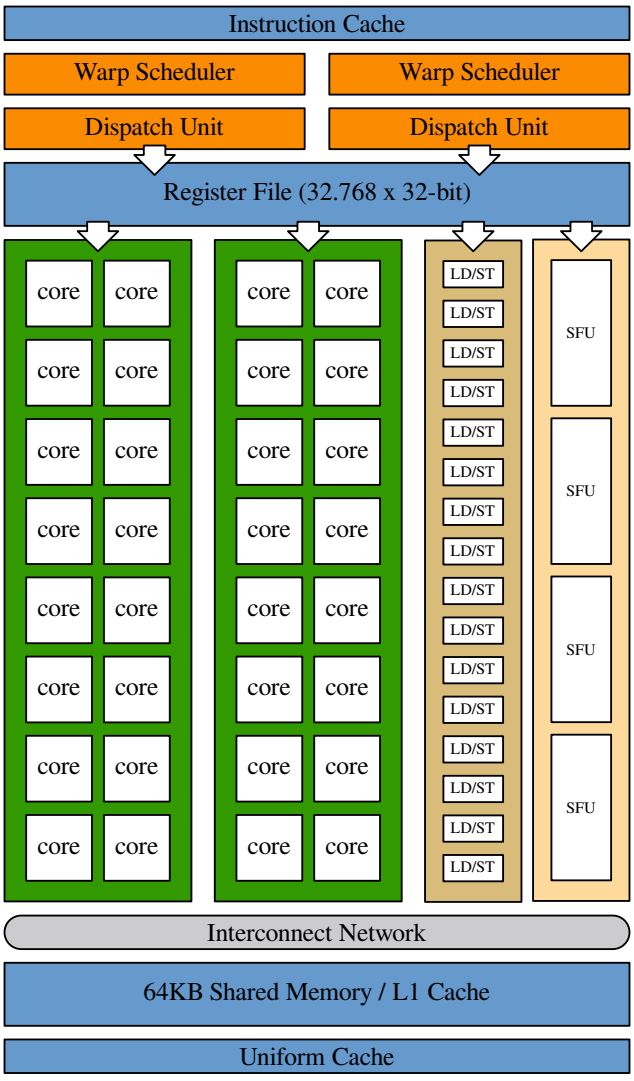


Fig. 2.6: Diagrama de bloques del SM de GF100 Fermi. Basado en [14].

llamado *warp*, con cada uno de estos corriendo en un SP. Las unidades de despacho de warps se encargan de mantener registro de qué *threads* están disponibles para correr en un momento dado y permiten realizar cambios de contexto por hardware eficientemente ($< 25\mu$s) [25]. Con esto, se pueden ejecutar concurrentemente dos warps distintos para esconder la latencia de las operaciones. En precisión doble, esto no es posible, así que hay solamente un warp corriendo a la vez.

Un SM cuenta con una memoria común de 64 KB que se puede usar de forma automática tanto como memoria compartida común a todos los *threads* como cache L1 para todos los accesos a memoria.

Fig. 2.7: Diagrama de bloques de GF100 Fermi. Tomado de [14].

Por como funciona un pipeline gráfico clásico, los SM se agrupan de a cuatro en GPCs (*Graphics Processing Cluster*) y no interactúa con el modelo de cómputo de CUDA. Un esquema de esta división global de los SM y cómo se comunican puede verse en la figura 2.7.

Todos los accesos a memoria global (la memoria por fuera del procesador) se realizan a través de la cache L1 de cada SM y a través de la L2 del todo el procesador. Esta L2 consiste de seis bancos compartidos de 128 KB. Estas caches se comunican de manera directa tanto con la DRAM propia de la placa como con el bus PCI Express por el cual pueden comunicarse dos placas entre sí, sin pasar por CPU, y son *write-through*, es decir cada escritura se hace tanto en la DRAM como en la memoria cache.

Como estos procesadores implementan el estándar IEEE754-2008, cuentan con operaciones de precisión simple y doble acorde al mismo, por lo cual los cálculos intermedios en operaciones como FMA (*Fused Multiply-Add*), que toma tres operandos y devuelve el producto de dos de ellos sumado al tercero, no pierden precisión por redondeo.

2.2.3. Organización de la memoria

La memoria de la GPU es uno de los puntos cruciales de esta arquitectura, un esquema gráfico puede observarse en la figura 2.8. Esta se subdivide entre memorias on-chip y memorias on-board, de acuerdo a su ubicación y latencia de acceso, en cuatro categorías distintas:

- Registros
- Memoria local
- Memoria compartida
- Memoria global

Cada *thread* de ejecución cuenta con una cantidad limitada de registros de punto flotante de 32 bits con latencia de un par de ciclos de clock. A su vez, existe una cantidad finita de registros totales que cuenta un SM (oscila entre 16535 y 65535 registros). Por su baja latencia son la clase principal de almacenamiento temporal.

La memoria local es una memoria propia de cada *thread*, y se encuentra almacenada dentro de la memoria global. Esta memoria es definida automáticamente por el compilador y sirve como área de almacenamiento cuando se acaban los registros: los valores anteriores se escriben a esta memoria, dejando los registros libres para nuevos valores en cálculos, y cuando se terminan estos cálculos se carga los valores originales nuevamente. Cuenta con las mismas desventajas que la memoria global, incluyendo su tiempo de acceso.

La memoria compartida, o *shared*, es una memoria que es visible para todos los *threads* dentro de un mismo SM. Cada *thread* puede escribir en cualquier parte de la memoria compartida dentro de su bloque y puede ser leído por cualquier otro *thread* de este. Es una memoria muy rápida, on-chip, y que tarda aproximadamente 40 ciclos de acceso [26]. Esta memoria es compartida con la cache L1, la cual tiene capacidad de entre 16 KB y 64 KB configurable por software. Esta memoria se encuentra dividida en 32 bancos de 2 KB de tamaño, permitiendo que cada uno de los 32 *threads* acceda independientemente a un float. Si hubiera conflicto, los accesos a ese banco se serializarían, aumentando la latencia de la llamada [18].

La memoria global es la memoria principal fuera del chip de la GPU. Esta es de gran tamaño (de entre 1 GB y 12 GB) y es compartida por todos los SM de la GPU y los CPU que integran el sistema. Es decir, tanto los GPU como los CPU pueden invocar las funciones de CUDA para transferir datos entre la memoria de la placa y la memoria RAM de *host*. La latencia de acceso a la memoria global es de cientos de ciclos [26], sumamente lenta en comparación con el procesador. La memoria global también puede ser mapeada, o *pinneada*, para que exista una copia de esa reserva tanto en la memoria en la placa como en la memoria principal del procesador. El driver de CUDA va a mantener la consistencia entre ambas de manera asíncrona, evitando la necesidad de hacer copias de memoria explícitas. No es ilimitada la cantidad de memoria mapeada posible, por lo que es importante saber elegir qué elementos se van a almacenar de esta manera.

Adicionalmente, la GPU cuenta con múltiples niveles de memorias cache para poder aminorar el hecho de que el principal cuello de botella del cómputo es la latencia en los accesos a memoria global. Estas se dividen en cuatro:

- Cache L1

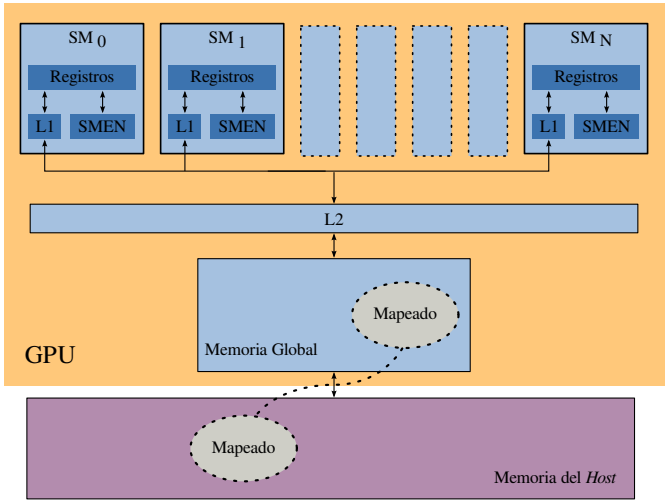


Fig. 2.8: Esquema de la jerarquía de memorias en GPU, detallando de las disponibles en cada SM, la memoria compartida (SMEN), la L2 global, la memoria de la placa (global) y la mapeada entre el *host* y la placa de video. Tomado de [18].

- Cache L2
- Cache constante
- Cache de textura

La cache L1 es dedicada por SM. Esta cache fue introducida en Fermi y su diseño hace que también está dedicada a la memoria compartida, por lo que es posible en tiempo de ejecución darle directivas a la GPU que asigne más memoria cache o más memoria compartida, permitiendo a los bloques tener mayores espacios de memorias compartidas o mayores *hit rates* de caches.

La cache L2 es común a todos los SM de la GPU, donde, a partir de Fermi en Nvidia, todos los accesos de lectura y escritura a memoria global y textura pasan a través de esta [14].

La cache constante es una cache sobre la memoria global dedicada solamente a lecturas de memoria. Esta es muy reducida (solo cuenta con 64 KB) y está optimizada para muchos accesos a la misma dirección. Cuando un *thread* lee esta memoria, se retransmite a los demás *threads* del warp que estén leyendo esa misma dirección, reduciendo el ancho de banda necesario. Si, en cambio, los *threads* leen distintas direcciones, los accesos se serializan. Cuando hay un *miss* de esta memoria, la lectura tiene el costo de una lectura de memoria global.

La cache de textura es una cache sobre la memoria global que presenta no solo localidad espacial, como la mayoría de las caches de procesadores normales (es decir, la cache contiene una porción consecutiva de la memoria principal), sino que se le puede agregar el concepto de dimensiones, para poder modelar datos en más de una dimensión. Esto se adapta de muy bien a los problemas de gráficos en 2D y 3D, y es una herramienta clave a la hora de minimizar los accesos a matrices no solo por filas sino por columnas. Esta cache se debe definir en momento de compilación en el código, ya que tiene límites espaciales (necesarios para poder definir áreas de memoria sobre la cual operar) y a su vez se debe acceder a los datos subyacentes a través de funciones específicas. Una característica

adicional de esta cache es que como necesita resolver estos accesos no convencionales a la memoria, cuenta con una unidad propia de resolución de direcciones. Esta unidad tiene limitantes en cuanto a sus posibilidades, ya que no posee un ancho de banda suficiente como para resolver todos los accesos a memoria globales que podrían surgir, por lo cual su uso debe ser criterioso.

2.2.4. Esquema de paralelismo

Al ser una arquitectura masivamente paralela desde su concepción, CUDA presenta varios niveles de paralelismo, para agrupar lógicamente el cómputo y poder dividir físicamente su distribución. Los principales son:

- Bloques de *threads*
- Grilla de bloques
- Streams
- Múltiples placas

El paralelismo a nivel de bloque instancia una cantidad de *threads*, subdivididos lógicamente en 1D, 2D o 3D. Los *threads* internamente se agrupan de a 32, es decir, un *warp*. Cada uno de estos *threads* va a contar con una manera de identificarlos unívocamente: un `blockId` y, dentro de cada bloque, su propio `threadId`. Además, van a correr simultáneamente en el mismo SM y van a ser puestos y sacados de ejecución de a un warp dinámicamente por el *scheduler* de hardware que cuenta cada SM. Para compartir información entre ellos, se puede utilizar la memoria compartida o las instrucciones de comunicación de *threads* intrawarp (solo disponibles a partir de Kepler [15]).

El paralelismo a nivel de grilla determina una matriz de bloques de ejecución que particiona el dominio del problema. El *GigaThread Scheduler* va a ejecutar cada bloque en un SM hasta el final de la ejecución de todos los *threads* de este. Los bloques no comparten información entre sí. Por esto, no pueden ser sincronizados mediante memoria global ya que no se asegura el orden en el que serán puestos a correr, y un bloque mantiene su SM ocupado hasta que termine de ejecutar, bloqueando a los demás (es decir, no hay *preemption* en los SM).

El paralelismo de stream es una herramienta empleada para hacer trabajos concurrentes usando una sola placa. Esta técnica permite que múltiples kernels (unidades de código en CUDA) o copias de memoria independientes estén encolados, para que el driver pueda ejecutarlas simultáneamente si se están subutilizando los recursos, de forma de minimizar tiempo ocioso del dispositivo. Los streams permiten kernels concurrentes pero cuentan con importantes restricciones que generan sincronización implícita, lo cual hay que tener presente si se desea mantener el trabajo de forma paralela.

El paralelismo a nivel de placa consiste en poder distribuir la carga del problema entre distintas GPUs dispuestas en un mismo sistema compartiendo una memoria RAM común como si fuera un software multithreaded tradicional. CUDA no cuenta con un modelo implícito de paralelismo entre distintas placas, pero es posible hacerlo manualmente eligiendo de manera explícita qué dispositivo usar. Las placas se pueden comunicar asíncronamente entre sí, tanto accediendo a las memorias globales de cada una como ejecutando código remotamente. En las versiones modernas del driver de CUDA, también

pueden comunicarse directamente las placas entre sí a través de la red, permitiendo escalar multinodo fácilmente en un cluster de cómputo [18].

2.2.5. Diferencias entre Tesla, Fermi, Kepler

Hasta ahora se describió la arquitectura vista desde el punto de vista Fermi, que es la segunda arquitectura GPGPU diseñada por Nvidia. Fermi es la evolución de Tesla, construida para desacoplar aún más los conceptos de procesamiento gráfico de modo de lograr un procesador más escalable y de propósito general. La arquitectura sucesora a Fermi es Kepler, presentada en el 2012, con las metas de disminuir el consumo y aumentar la potencia de cálculo [15].

Características	Tesla (GT200)	Fermi (GF100)	Kepler (GK110)
Año introducción	2006	2010	2012
Transistores	1400 millones	3000 millones	3500 millones
Tecnología fabricación	65 nm	40 nm	28 nm
SMs	30	16	15
SP / SM	8	32	192
Caché L1	-	16 - 48 KB	16 - 32 - 48 KB
Caché L2	-	768 KB	1536 KB
Memoria Shared/SM	16 KB	16 - 48 KB	16 - 32 - 48 KB
Registros/Thread	63	63	255
Pico Precisión Simple	240 MAD / clock	512 FMA / clock	2880 FMA / clock
Pico GFLOPS Simple	933	1345	3977
GFLOPS/Watt	3,95	5,38	15,9

Tab. 2.2: Tabla comparativa de las características más prominentes de las tres arquitecturas de CUDA.

En la tabla 2.2 se ve una comparación de las recursos que están más directamente relacionados a la *performance* de un dispositivo GPU. Se puede apreciar el crecimiento notable del poder de cómputo debido a las tecnologías de fabricación, que permitieron aumentar la cantidad de transistores por unidad de superficie. También se puede comprobar que, a diferencia de los CPU, las arquitecturas GPGPU decidieron utilizar esos nuevos transistores disponibles para más núcleos de procesamiento, en vez de dedicarlas a aumentar las memorias cache, que crecieron mínimamente (comparando contra las caches de CPU).

Una de las diferencias más notorias entre Tesla y Fermi es la presencia de FMA contra el MAD (*Multiply - Add*). El MAD realiza la multiplicación y la acumulación en dos pasos, pero más rápidos que hacerlos independientemente por tener hardware dedicado. Debido a que debe redondear entre los pasos, pierde precisión y no respeta completamente el estándar IEEE754-2008. El FMA, en cambio, lo hace en una sola operación, y sin redondeos intermedios.

La métrica usada por Nvidia para publicitar la *performance* de estos dispositivos y poder compararlos entre sí, y contra CPU, son los GFLOPS. Esta unidad mide cuantas operaciones de punto flotante de precisión simple se pueden realizar por segundo. Los GFLOPs son utilizados también por los clusters en el ranking TOP500, donde se ordenan de acuerdo a la *performance* medida usando un software estandarizado, `LINPACK`. No solo

es notable como se cuadruplicó la *performance* (teórica) en solamente seis años, sino que aún más importante es como mejoró la *performance* por Watt. Esto también se ve en que Kepler tiene menos SM que Fermi o Tesla, pero son mucho más poderosos y eficientes. La tecnología de fabricación ha ayudado a la disminución del consumo, un problema que acechaba a los diseños Fermi, ya que sus consumos superiores a 200W por dispositivo los hacían muy difíciles de refrigerar incluso en clusters de HPC. Se puede apreciar entonces la estrategia de mercado de Nvidia de introducirse en las supercomputadoras de todo el mundo, donde el consumo y la refrigeración son factores limitantes (mucho más aún que, por ejemplo, en computadoras de escritorio) [7].

2.2.6. CUDA, Herramientas de desarrollo, profiling, exploración

Para soportar una arquitectura masivamente paralela, se debe usar una ISA (*Instruction Set Architecture*) diseñada especialmente para el problema. En el caso de CUDA, esta ISA se denominada PTX y debe poder soportar conceptos fundamentales del cómputo GPGPU: grandes cantidades de registros, operaciones en punto flotante de precisión simple y doble, y FMA (fused multiply-add). Además, el código compilado para GPU debe ser agnóstico al dispositivo que lo va a correr, por lo cual la paralelización no debe estar demasiado atada a este, sino que el dispatching lo debe poder determinar el driver de la placa en tiempo de ejecución. Un último requerimiento clave de esta ISA es que debe soportar hacer ajustes manuales, para poder construir partes claves de ciertas bibliotecas frecuentemente usadas (como las rutinas de BLAS de álgebra lineal) [14].

El lenguaje CUDA es una extensión de C++, con ciertas características agregadas para poder expresar la subdivisión de las rutinas en *threads* y bloques, junto con mecanismos para especificar qué variables y funciones van a ejecutarse en la GPU y en el CPU. Una característica de CUDA es que todas las llamadas a los kernels de ejecución son asincrónicas, por lo que es relativamente sencillo solapar código en GPU y CPU. A su vez se cuenta con múltiples funciones opcionales, con distinta granularidad, que permiten esperar a que todas las llamadas asíncronas a GPU finalicen, agregando determinismo en forma de barreras de sincronización al lenguaje.

El código CUDA compila usando `nvcc`, una variante del `GNU gcc` que se encarga de generar el código PTX para las funciones que se van a ejecutar en las GPU. Este código objeto después se adosa normalmente con el resto del código que corre en CPU y se genera un binario ejecutable.

Nvidia, además, provee herramientas de profiling para explorar cómo se están utilizando los recursos durante la ejecución. Éstas son esenciales para optimizar, puesto que los limitantes de GPU son sumamente distintos a los de CPU, presentando dificultades conceptuales incluso para programadores experimentados. Las herramientas de profiling no solo muestran *runtime*, sino que sirven para ver dónde hay accesos a memoria excesivos, puntos de sincronización costosos, limitantes en los registros y cómo se superponen las llamadas asincrónicas.

El uso de todas estas herramientas fue vital en este trabajo para poder entender cómo funciona la arquitectura en detalle, cómo medir *performance* y utilización, y cómo los cambios realizados impactaron en las distintas generaciones de dispositivos.

2.2.7. Requerimientos de un problema para GPGPU

Dada la organización de un procesador GPU, un problema debe exhibir al menos las siguientes características para que tenga potencialidad para poder aprovechar las características y recursos disponibles en esta arquitectura:

1. El problema debe tener una gran parte paralelizable.
2. El problema debe consistir, mayormente, de operaciones numéricas.
3. El problema debe poder ser modelado, en su mayor parte, utilizando arreglos o matrices.
4. El tiempo de cómputo debe ser muy superior al tiempo de transferencia de datos.

El ítem 1 se refiere a que debe existir alguna forma de partir el problema en subproblemas que puedan realizarse simultáneamente, sin que haya dependencias de resultados entre sí. Si el problema requiere partes seriales, lo ideal es que se las pueda dividir en partes independientes que sean etapas de una cadena de procesos, donde cada una de éstas exhiban características fuertemente paralelas. Como las arquitecturas masivamente paralelas tienen como desventaja una menor eficiencia por núcleo, si el problema no se puede dividir para maximizar la ocupación de todos los procesadores disponibles, va a resultar muy difícil superar en eficiencia a los procesadores seriales.

El ítem 2 habla acerca de que el método de resolución de los problemas debe provenir de una aplicación numérica o de gran carga aritmética. El set de instrucciones de las arquitecturas GPGPU están fuertemente influenciados por las aplicaciones 3D que las impulsaron en un principio. Éstas consisten mayormente de transformaciones de álgebra lineal para modelar iluminación, hacer renders o mover puntos de vistas. Todos estos problemas son inherentemente de punto flotante, por lo cual el set de instrucciones, las ALUs internas y los registros están optimizados para este caso de uso.

El ítem 3 menciona que los problemas que mejor se pueden tratar en esta arquitectura se pueden representar como operaciones entre arreglos o matrices de dos, tres o cuatro dimensiones. Las estructuras de datos no secuenciales en memoria incurren en múltiples accesos a memoria para recorrerlas y, en las arquitecturas GPGPU, generan un gran cuello de botella. Además, suelen ser difíciles de paralelizar en múltiples subproblemas. Tener como parámetros de entrada matrices o arreglos que se puedan partir fácilmente producen en overheads mínimos de cómputo y permiten aprovechar mejor las memorias caches y las herramientas de prefetching que brinda el hardware.

Item 4 ataca uno de los puntos críticos de esta arquitectura. Para poder operar con datos, se requiere que estén en la memoria de la placa, no en la memoria de propósito general de la computadora. Se debe, entonces, hacer copias explícitas entre las dos memorias, ya que ambas tienen espacios de direcciones independientes. Esta copia se realiza a través de buses que, a pesar de tener un gran *throughput*, también tienen una gran latencia (del orden de milisegundos). Por lo tanto, para minimizar el tiempo de ejecución de un programa usando GPGPUs, se debe considerar también el tiempo de transferencia de datos a la hora de determinar si el beneficio de computar en menor tiempo lo justifica. Las nuevas versiones de CUDA buscan brindar nuevas herramientas para simplificar este requerimiento, proveyendo espacio de direccionamiento único y memoria unificada [18], pero siguen siendo copias de memoria a través de los buses (aunque asincrónicas).

Estas características limitan enormemente la clase de problemas que una GPGPU puede afrontar, y suelen ser una buena heurística para determinar de antemano si vale la pena invertir el tiempo necesario para la implementación y ajuste fino.

2.2.8. Diferencia entre CPU y GPU - Procesadores especulativos

Hasta ahora, solo se consideraron a los GPUs de forma aislada, observando las prestaciones del hardware y una aproximación a la manera en que se escriben los programas para esta arquitectura. La esencia de GPGPU se puede apreciar mejor comparándola contra los motivos de la evolución de CPU, y los problemas que se fueron enfrentando los diseños siguiendo la historia de los componentes que fueron apareciendo en estos. Esto se mostró en la tabla 2.1 (página 8), que detalla algunos de los eventos más importantes que aceleraron la *performance* de los CPU.

Lo clave es observar el siguiente patrón: *"no desechar algo que pudiéramos necesitar pronto"*, *"intentar predecir el futuro de los condicionales"*, *"intentar correr múltiples instrucciones a la vez porque puede llegar a bloquear en alguna de ellas"*.

Todos estos problemas han convertido al CPU en un dispositivo que gira alrededor de la especulación, de los valores futuros que pueden tener las ejecuciones, del probable reutilización de datos. En un CPU moderno (por ej. Intel Xeon E7-8800 [27]) las unidades que verdaderamente realizan las operaciones lógico-aritméticas (las ALU) son muy pocas en comparación con la cantidad utilizadas para las operaciones de soporte.

En contraste, los dispositivos GPU son verdaderos procesadores de cómputo masivo. Están diseñadas para resolver constantemente operaciones muy bien definidas (instrucciones de punto flotante en su mayoría). Comparativamente con un CPU, las ALU de las GPU son bastante pobres y lentas. No funcionan a las mismas velocidades de clock (rara vez superan 1,1 GHz) y sus SP deben estar sincronizados entre sí. Pero la gran ventaja esta en la cantidad.

Un CPU cuenta con pocas ALU por core, dependiendo de la cantidad de cores y del tamaño de sus operaciones SIMD (alrededor de 16 cores por *die* de x86 es el tope de línea ofrecido actualmente, procesando de a 32 bytes simultáneamente). Un GPU cuenta con miles de ALUs en total (más de 2500 CUDA cores en una Tesla K20 [16]). El diseño de esta arquitectura concibe la escalabilidad cuantitativa de las unidades de cómputo como la característica esencial a tener, tanto por su énfasis fundamental, las aplicaciones gráficas, como para su aspecto de coprocesador numérico de propósito general.

Por contrapartida, los GPUs disponen de pocas unidades de soporte del procesamiento. Éstos no disponen de pipelines especulativos, el tamaño de las caches están a órdenes de magnitud de las de CPU, la latencia a las memorias principales de la GPU están a centenas de clocks de distancia, etc. La arquitectura supone que siempre va a tener más trabajo disponible para realizar, por lo cual en vez de intentar solucionar las falencias de un grupo de *threads*, directamente pone al grupo en espera para más adelante y continúa procesando otro warp de *threads*. Se puede notar que durante del diseño de la arquitectura CUDA, buscaron resolver el problema del cómputo masivo pensando en hacer más cuentas a la vez y recalcular datos, si fuera necesario. Esto es una marcada diferencia con respecto a los CPU, que están pensados en rehacer el menor trabajo posible e intentar mantener todos los datos que pueda en las memorias caches masivas.

Nuevamente, en este punto se puede apreciar el legado histórico de los CPU. Al tener que poder soportar cualquier aplicación, no pueden avocarse de lleno a una sola problemática. Para las arquitecturas GPGPU, el hecho de no tener que diseñar un procesador

de propósito general compatible con versiones anteriores, permitió un cambio radical a la hora de concebir una arquitectura de gran throughput auxiliar al procesador, no reemplazándolo sino más bien adicionando poder de cómputo [28].

Las arquitecturas Tesla, Fermi y Kepler conciben el diseño de un procesador de alto desempeño. Su meta principal es poder soportar grandes cantidades de paralelismo, mediante el uso de procesadores simétricos, pero tomando la fuerte restricción de *"no siempre tiene que andar bien"*. Es decir, los diseñadores suponen que el código que van a ejecutar esta bien adaptado a la arquitectura y no disponen casi de mecanismos en el procesador para dar optimizaciones post-compilación. Relajar esta restricción permite romper con el modelo de cómputo de CPU y definir nuevas estrategias de paralelismo, que no siempre se adaptan bien a todos los problemas, pero para el subconjunto de los desafíos que se presentan en el área de HPC y de vídeo juegos han probado ser un cambio paradigmático.

2.2.9. Idoneidad para la tarea

El problema de QM/MM enfrentado en este trabajo cuenta con múltiples operaciones matemáticas de gran volumen de cálculos. En particular, las operaciones matriciales constituyen los principales cuellos de botella en esta aplicación. Estas operaciones se realizan para varios grupos dentro de una grilla de integración (ecuación 1.5), los cuales se pueden realizar de manera independiente (y por lo tanto en paralelo).

Para obtener los valores numéricos de densidad buscados en los puntos, se deben obtener las derivadas primeras y segundas, lo cual implica hacer múltiples operaciones de multiplicación matricial. Este tipo de problemas está estudiado fuertemente en la literatura debido a la multiplicidad de aplicaciones de diferentes campos que requieren de operaciones de álgebra lineal.

En nuestro caso, para un sistema se requieren miles de estas multiplicaciones entre matrices, algunas con matrices de más de 500^2 elementos. Como LIO es un proyecto de resolución numérica de QM/MM, los problemas enfrentados son casi, en su totalidad, operaciones de punto flotante. Luego, dadas las características de contar con un fuerte nivel de paralelismo en los cuellos de botella y de ser operaciones mayormente de punto flotante, se determinó que el uso de GPGPU para este problema era promisorio, en comparación con arquitecturas de propósito general con menos poder de cómputo. La exploración original de esta arquitectura trajo buenos resultados, por lo que se prosiguió su análisis como un camino prometedor [4].

2.3. Xeon Phi

2.3.1. Introducción

La arquitectura Xeon Phi es la culminación de un trabajo iniciado por Intel en 2004, previendo la necesidad de paralelismo masivo para aplicaciones futuras. Saliendo al mercado al final de 2012 y con el propósito de competir en cómputo intensivo con Nvidia CUDA, ha ganado gran tracción dentro de HPC a pesar de ser muy reciente. Por ejemplo, Xeon Phi ha sido implementado en la supercomputadora Tianhe-2 de la Universidad de Sun Yat-Sen en China, listada en Top 500 como la supercomputadora más rápida del mundo en Junio 2013, Noviembre 2013, Junio 2014 y Noviembre 2014 [29, 30, 31]. Los 16000 nodos de esta supercomputadora contienen dos Ivy Bridge Xeon y tres coprocesadores Xeon Phi cada uno, dando un poder total de cómputo teórico de 54,9 PetaFLOPS.

2.3.2. Microarquitectura general

En la concepción del Xeon Phi se tuvieron en cuenta diversos factores, entre los cuales se incluye el consumo energético. Uno de los objetivos fue aumentar la relación de poder de cómputo por Watt de los procesadores Xeon de la época, manteniendo un entorno de desarrollo de propósito general como el de `x86`.

Si bien el consumo no se veía impactado por el set de instrucciones, sí se buscó eliminar diversos componentes del procesador, manteniendo las características que sirvieran al tipo de aplicación a la que se estaba apuntando (programas altamente paralelos a nivel de datos y tareas) [32].

La microarquitectura de este coprocesador se basa en muchos (más de 50) procesadores simétricos que comparten la memoria, lo cual justifica su nombre `MIC` (*Many Integrated Core*). Cada procesador esta basado en el diseño del Intel Pentium, con una ISA (*Instruction Set Architecture*) similar a `IA-32` con soporte para direccionamiento a 64 bits y nuevas instrucciones de vectorización.

Los procesadores tienen un *clock rate* de 1,0 GHz aproximadamente, comparativamente lentos frente a otros procesadores de Intel. Por ejemplo, los cores de un Intel Xeon CPU E5-2620 tienen un *clock rate* de 2,10 GHz, más del doble.

Cada uno de los cores permite hasta cuatro *threads* simultáneos, con el propósito de esconder la latencia de memoria y del tiempo de ejecución de las instrucciones vectoriales. Adicionalmente, el uso de dos *pipelines* (denominados U y V) permite que se ejecuten hasta dos instrucciones por ciclo de clock. Algunas de éstas, sin embargo, solo pueden ser ejecutadas en uno de los dos: por ejemplo las instrucciones de vectorización solo pueden correr en el pipeline U. Para realizar operaciones vectoriales se cuenta con una unidad de vectorización (VPU, *Vector Processing Unit*) con 32 registros SIMD (*Single Instruction Multiple Data*) de 512 bits por *thread*, con lo cual cada una puede, teóricamente, realizar 16 operaciones de punto flotante de 32 bits al mismo tiempo. La latencia de estas instrucciones es de cuatro ciclos de clock. Sin embargo, gracias a su micro arquitectura de *pipeline* se puede lograr que la cantidad de instrucciones efectivamente retiradas (*throughput*) teóricamente llegue a una instrucción vectorial por ciclo, una vez cargado todo el pipeline. Un esquema de la arquitectura puede verse en la figura 2.9.

2.3.3. Pipeline

El *pipeline* de instrucciones cuenta con siete etapas para las instrucciones escalares, y seis más para las instrucciones vectoriales.

La figura 2.10 introduce las diferentes etapas de cada uno de los pipelines. Cada pipeline tiene las fases usuales del ciclo *fetch-decode-execute*: Una instrucción es leída de memoria, luego decodificada en microoperaciones, y luego estas microoperaciones se realizan y se guardan los resultados. Cada etapa del pipeline realiza una fracción de este ciclo.

El *instruction fetch* está dividido en dos fases para elegir el *thread* por hardware a ejecutar: *Prethread picker function* (PPF) y *Thread picker function* (TPF). En la fase PPF se mueve la instrucción a uno de los cuatro buffers de *prefetch* que tiene cada procesador.

La etapa TPF selecciona el *thread* a ejecutar, usando el buffer de prefetch. Cada buffer tiene espacio para dos instrucciones (porque puede ejecutarse una instrucción por el pipeline U y otra por el V). TPF funciona de manera *round robin* entre los buffers de prefetch. Recargar este buffer con instrucciones (por ejemplo cuando hay un *miss* de cache de instrucciones) toma entre cuatro y cinco ciclos.

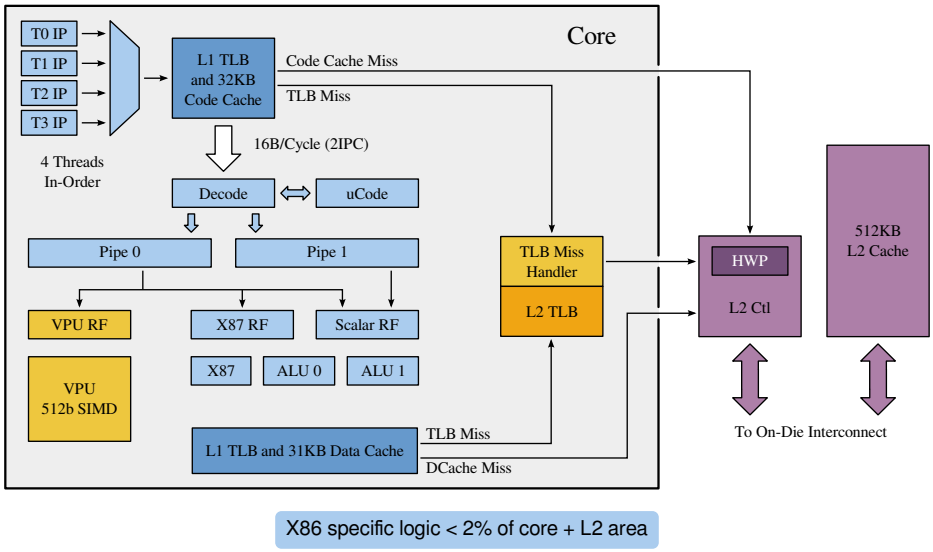


Fig. 2.9: Esquema de un procesador del Xeon Phi, tomado de [32]

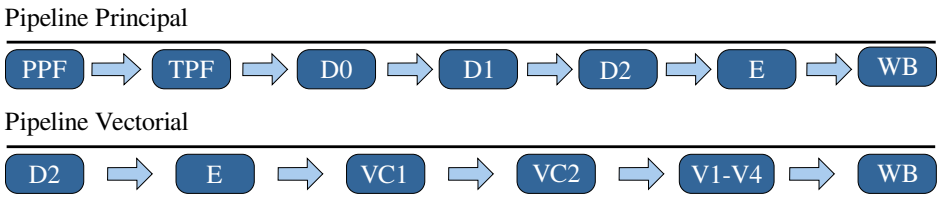


Fig. 2.10: Pipeline del procesador Xeon Phi.

Los cuatro *threads* de un núcleo son usados secuencialmente por el pipeline, de manera de que si uno de los *threads* esta bloqueado se elija a otro que tenga trabajo para hacer. Intel sugiere el uso de al menos dos *threads* por núcleo para esconder latencias de memoria y de instrucciones vectoriales, y así aumentar la *performance*.

Una vez que una instrucción ha sido elegida para decodificarse, esta pasa a las etapas D0 y D1, con una velocidad de decodificación de dos instrucciones por ciclo de *clock*. De ahí son enviadas a cada pipeline para ejecutarse. Por último, se pasa a la etapa de *writeback* (WB), en la cual los resultados se combinan en sus respectivos registros o posiciones de memoria. No necesariamente cuando una instrucción llega a esta fase ha terminado de ejecutarse, puesto que si la operación es vectorial recién finaliza en la unidad vectorial cinco ciclos después.

Este pipeline corto (7 etapas frente a las 20 de la arquitectura Pentium 5 en la que se basa Xeon Phi) contribuye a que los fallos de predicción de saltos tengan menor latencia y a que las instrucciones no vectoriales tengan poca latencia [33].

2.3.4. Estructura de cache

Además de la unidad de vectorización y la unidad escalar, cada procesador cuenta con 32 KB de cache L1 y 512 KB de cache L2 unificada para datos y código. Estas caches son *set associative 8-way* con una línea de cache de 64 bytes. La cache de datos es no bloqueante, de manera que un *miss* de cache de un *thread* en un core no produce un *flush* del pipeline en los demás *threads*.

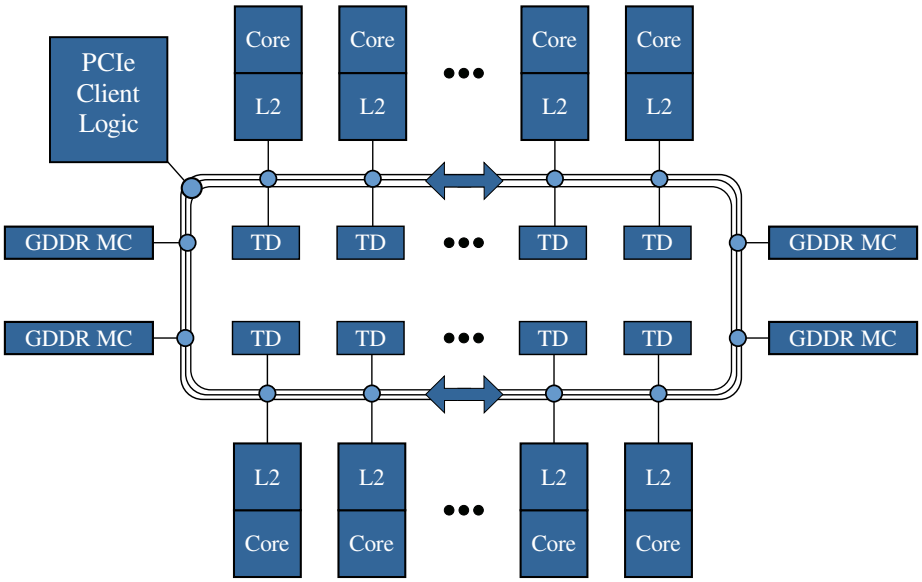


Fig. 2.11: Detalle de la estructura de anillo del Xeon Phi, tomado de [34].

La cache L2 mantiene su coherencia mediante el uso de un directorio distribuido de *tags*. Un esquema del mismo puede verse en la figura 2.11. El mismo está dividido en 64 secciones y une tanto los segmentos del directorio de tags con los cores y los controladores de memoria principal, utilizando una topología de anillo bidireccional, contando con un anillo para datos, uno para direcciones y uno para confirmaciones (*acknowledgements*).

Cuando ocurre un *miss* de cache L2, la dirección es enviada al anillo de direcciones. Si la línea de cache es de otro core, se envía un *request* de *forwarding* y los datos son enviados por el anillo de datos. Si ningún core tiene esa línea de cache, se envía un pedido a uno de los varios controladores de memoria. Cada uno de estos controladores maneja un subconjunto del espacio de direcciones con el propósito de reducir los cuellos de botella y aumentar el ancho de banda.

Cada cache L2 esta dividida en dos bancos, con una latencia de dos ciclos para leer 64 bytes (una línea) y un ciclo para escribir 64 bytes.

2.3.5. Arquitectura del set de instrucciones

Si bien la base del conjunto de instrucciones de la arquitectura del Xeon Phi es la P54C de Pentium (`IA-32`), nuevas instrucciones se han incluido con el propósito de mejorar la capacidad del procesador para cómputo de alta *performance*. Estas operaciones incluyen implementaciones por *hardware* de operaciones comunes en HPC: recíproco de un valor, raíz cuadrada, exponenciación, FMA, etc. También incluyen operaciones más relacionadas con la memoria, como por ejemplo accesos no secuenciales (*scatter and gather*) y stores que no pasen por cache, de manera de aprovechar mejor el ancho de banda de memoria que permite la arquitectura.

El diseño de las instrucciones vectoriales es ternario, con dos operandos fuentes y uno destino codificados en la instrucción. Esta configuración permite una mejora de hasta 20 % sobre la configuración usual binaria de otras arquitecturas SIMD (como por ejemplo `SSE` o `AVX`) [32].

Una expansión adicional para aprovechar los registros amplios son los registros de máscara (*mask registers*). Estos registros de 16 bits se utilizan en varias de las instrucciones vectoriales para habilitar o deshabilitar elementos de los 16 (como máximo) que tiene un registro SIMD del Xeon Phi como parte de un cómputo, contribuyendo a la generalidad de las instrucciones adicionales de la arquitectura. Si un elemento no estuviera en el registro de máscara, el valor resultado de la operación sobre los dos elementos correspondientes de los operandos fuentes es escrito en el lugar correspondiente al operando destino. Esta herramienta permite evitar usar condicionales sobre los operandos y tener que dejar de usar registros vectoriales.

La adición de instrucciones sobre memoria no secuencial (*scatter and gather*) resultan interesantes por su utilización en HPC. También, con el propósito de permitir un control más fino de las caches, los operandos en memoria para operaciones vectoriales permiten la inclusión de un atributo denominado *eviction hint* para indicar que estos datos son importantes y es preferible no retirarlos de la cache para traer otros.

Por último, la unidad vectorial implementa un *prefetcher* por software, tanto para los caches L1 y L2. Estos pueden combinarse con instrucciones de *gather* y *scatter* para disminuir los accesos a memoria y los *stalls* en el retiro de instrucciones.

La mayor parte de estas características son invisibles al programador, pero son aprovechadas por los compiladores para producir código más eficiente y que utilice mejor la arquitectura.

2.3.6. Organización de la memoria

La memoria principal del Xeon Phi consiste de 8 GB de RAM GDDR5 en la placa dividida en bancos. Los cores y la memoria principal se comunican mediante el uso de ocho controladores, conectados con un anillo bidireccional de dos canales a 5,5 GB/s. El tamaño de una transferencia realizada es de 4 bytes. Esto da un límite de ancho de banda teórico de 352 GB/s pero detalles de implementación de los chips limitan este valor a 200 GB/s[32].

Los controladores reciben los pedidos mediante el anillo de direcciones y los convierten en comandos para GDDR5, retornando finalmente los datos en el anillo de datos. También reorganizan los pedidos que se envían de manera que se hagan en un orden conveniente a la memoria principal. Además, los dispositivos GDDR5 tienen el espacio de direcciones interlineado entre distintos bancos de memoria para distribuir mejor la carga de pedidos y aprovechar todo el ancho de banda.

2.3.7. Conexión Host - Coprocesador

El Xeon Phi se conecta con su host mediante el uso de un bus PCI Express 2.0 de 16 líneas. Las transferencias pueden ser mediante I/O (Entrada/Salida) programada o usando DMA (*Direct Memory Access*), que no requiere el procesador. Este bus permite la transferencia no solo al host sino a otros coprocesadores, permitiendo tener múltiples placas en una misma computadora. La velocidad de transferencia alcanzada es de más de 6 GB/s. La DMA ocurre a velocidad de *clock* del procesador, y los ocho canales de DMA pueden transferir en paralelo. Cada transacción de DMA es entre 64 y 256 bytes.

Siendo que el coprocesador dispone de su propio sistema operativo basado en Linux, lo cual incluye una implementación de los protocolos necesarios para utilizar TCP/IP para la transferencia de mensajes en red, el bus PCI se puede utilizar para comunicar el

sistema operativo del host con el del Xeon Phi a través de protocolos como SSH (*Secure SHell*). Esto facilita la administración de la placa, la transferencia de código y datos para ejecución, y el uso del coprocesador como nodo de cómputo adicional en un esquema de cluster.

Como el Xeon Phi es autónomo con respecto al sistema operativo del host, no cuenta con acceso a periféricos y, en particular, a dispositivos de almacenamiento. Para esto se puede utilizar un sistema NFS (*Network File System*) con el propósito de almacenar datos en el host que sean visibles desde el coprocesador.

2.3.8. Modos de ejecución

Existen tres métodos para ejecutar una aplicación en el coprocesador.

1. **Nativo**: El Xeon Phi ejecuta el programa directamente. Esto se debe a la presencia de BusyBox Linux como sistema operativo, lo cual da soporte de sistema de archivos y entorno de ejecución. El programa puede ser enviado al Xeon Phi desde el host y, por defecto, se utiliza un sistema de archivos montado sobre la propia RAM.

2. **Offloading**: El *host* puede delegar la ejecución de ciertas porciones de código al coprocesador. Esto requiere que los datos necesarios para el cómputo sean copiados del *host* a la placa Xeon Phi, pudiendo implicar que el bus puede resultar un cuello de botella importante (puesto que los datos de entrada y la salida deben ser movidos al Xeon Phi y recuperados al finalizar el cómputo).

3. **Simétrico**: En este modo de ejecución se piensa al Xeon Phi y su host como dos nodos en un *cluster* de cómputo. El bus PCIe actúa como una red de alta velocidad. Este modo es especialmente interesante si se dispone de más de un Xeon Phi en un mismo host y se utiliza una interfaz de intercambio de mensajes entre ellos como por ejemplo MPI (*Message Passing Interface*).

2.3.9. Herramientas de desarrollo y profiling

El desarrollo de aplicaciones para el Xeon Phi es, por diseño, muy similar al desarrollo de aplicaciones para las arquitecturas de la linea Xeon de Intel. Esto se debe a la existencia de compiladores de C, C++ y Fortran (los lenguajes más usados en el ámbito de HPC) que pueden producir código para el Xeon Phi, y la presencia de un sistema operativo y *stack* de aplicaciones que facilitan la tarea de ejecución y mantenimiento del código. A diferencia de Nvidia CUDA, no es necesario el uso de un lenguaje de programación diferente.

Los distintos modos de uso para el Xeon Phi reciben soporte de diversas bibliotecas. El uso mediante *offloading*, por ejemplo, es provisto mediante el uso de `pragmas` y opciones de compilación que permiten indicar qué cómputo y cómo deben ser transferidos a la placa. Estos pragmas permiten control fino de cómo enviar los datos a la placa, si deben ser transmitidos de vuelta de la placa, etc.

Asimismo los compiladores de Intel pueden generar código para la arquitectura MIC, el cual se puede ejecutar directamente sobre el Xeon Phi a través del sistema operativo.

Por último, Intel también provee soporte a OpenMP, MPI, Thread Building Blocks, etc. Bibliotecas ya existentes como la `MKL` (*Math Kernel Library*), que implementa las conocidas rutinas de `BLAS` y `LAPACK`, también tienen su versión diseñada para Xeon Phi e, incluso, pueden utilizarlo de manera transparente.

2.3.10. Idoneidad para la tarea

Siendo que el mercado principal para Xeon Phi son las aplicaciones de cómputo científico, esto hace que LIO sea una posible candidata al uso de esta tecnología.

Otros aspectos que hacen interesante la adaptación de una aplicación para utilizar el Xeon Phi son la existencia de una gran cantidad de código ya implementado en C++ y Fortran, el cual ya puede ser ejecutado en el Xeon Phi sin requerir más que una recompilación de los archivos fuente, al menos según lo que indica el fabricante Intel [33].

El código existente en LIO tiene una *performance* fuertemente atada al costo en procesamiento del cómputo siendo esta tarea altamente paralelizable (como se demostró en las implementaciones ya realizadas usando CUDA) y vectorizable. Estas son las características que requiere un problema para que tenga sentido ser considerado para Xeon Phi.

Por último, al tener aspectos muy similares, una implementación que escale tanto en términos de vectorización como en cantidad de procesadores en el caso de una arquitectura como la línea `x86-64` de Intel debería, al menos en teoría, hacer buen uso de las prestaciones de la arquitectura MIC, con lo cual el trabajo invertido en optimizar el código apuntado a CPUs multicore sigue la misma línea que lo necesario para optimizar para Xeon Phi [35]. Esto es ventajoso ya que el esfuerzo no se duplica para desarrollar dos implementaciones diferentes.

3. IMPLEMENTACIÓN

3.1. Implementación existente

Antes de describir las mejoras realizadas sobre la aplicación existente, nos centraremos en los pasos que componen la obtención de la función de onda mediante la construcción de la matriz de Kohn-Sham.

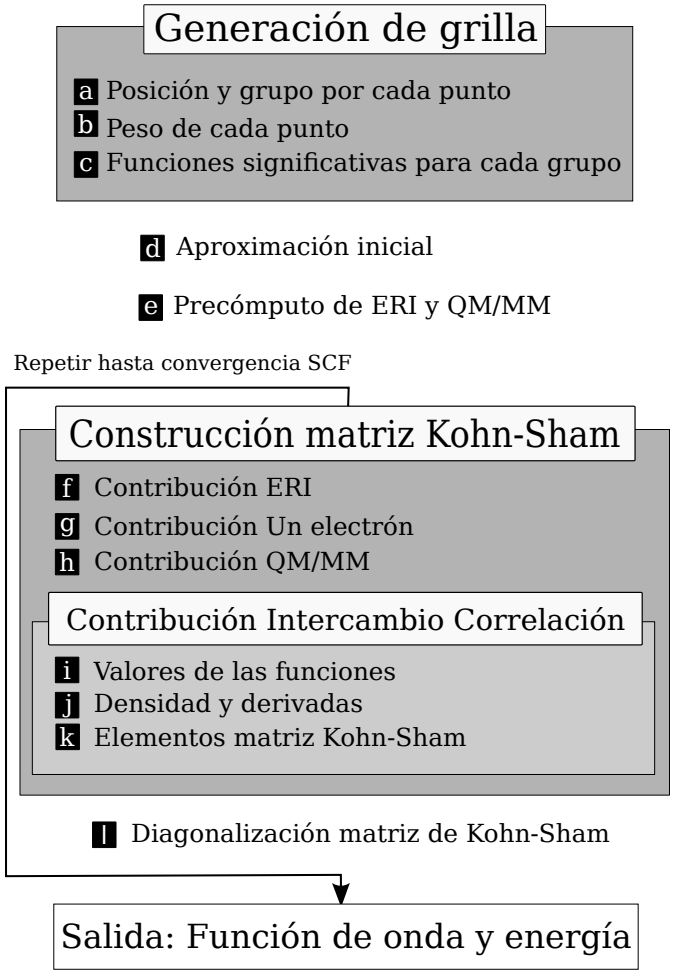


Fig. 3.1: Pasos del cálculo de DFT realizado por LIO.

En la introducción del trabajo se describieron, a grandes rasgos, algunas de las ecuaciones que componen el cálculo de la densidad electrónica. Estas fueron enunciadas sin detallar un método efectivo de resolución del cálculo de la densidad ρ. Para hacerlo se debe definir primero ρ como

$$\rho = |\Psi|^2 = \sum_k^{e^-} |\Phi_k|^2 = \sum_k^{e^-} \left| \sum_i a_i^k F_i \right|^2 \tag{3.1}$$

donde Ψ es la función de onda, Φ_k son las funciones de onda de un electrón (orbitales) y los a_i^k son los coeficientes con los que se expande los orbitales sobre la base de funciones (F_i).

El objetivo principal es obtener los coeficientes a_i^k correspondientes a la mejor densidad posible. El principio variacional (ecuación 1.6) brinda un criterio de selección basado en la minimización de la energía respecto a estos coeficientes.

Alternativamente, la densidad ρ puede definirse a través de una matriz *(matriz densidad)* según:

$$C_{i,j} = \sum_{k}^{e^-} a_i^k a_j^k \tag{3.2}$$

Luego, la densidad ρ puede calcularse como:

$$\rho = \sum_{i=1}^{m} \sum_{j=1}^{i} F_i F_j C_{i,j} \tag{3.3}$$

con m la cantidad de funciones de la base.

Para resolver el término de intercambio-correlación (ecuación 1.5), se discretiza el espacio usando una grilla de puntos. De esta manera se aplican las ecuaciones 3.2 y 3.3 para cada punto de la grilla.

En este trabajo utilizamos las grillas propuestas por Becke [36], compuestas por capas centradas en los núcleos. La distancia entre capas no es uniforme, siendo pequeña cerca de los núcleos (donde la densidad electrónica cambia más rápidamente) y aumentando al alejarse de los mismos. Los puntos cercanos en el espacio se agrupan de modo de calcular los términos de la matriz densidad solamente usando las funciones con contribuciones significativas, de acuerdo a lo propuesto por Stratmann [37]. Podemos aprovechar con esto el hecho de que las funciones Gaussianas (que forman la base utilizada) decaen rápidamente en el espacio.

Como los puntos no se distribuyen homogéneamente en el espacio sino que la mayor cantidad de ellos se concentra cerca de los núcleos, si se usara una partición únicamente basada en cubos de igual tamaño, la cantidad de puntos contenidos en cada grupo diferiría considerablemente. Por ello, además de la utilización de cubos, se usan esferas para contener partes especificas de la grilla. Los grupos esféricos se construyen rodeando los núcleos incluyendo zonas de alta concentración de puntos, de manera que los remanentes tengan una distribución más homogénea. Este esquema de construcción se muestra en la figura 3.2.

Para resolver la ecuación 1.5 es necesario determinar el valor del peso de cada punto de la grilla (ω_i), al no contribuir todos los puntos en la misma proporción a la energía de intercambio-correlación [36].

El detalle algorítmico de la generación de grupos y de cálculo de pesos de puntos fue estudiado previamente [3, 4] y no se han realizado modificaciones en este trabajo.

La obtención de los coeficientes a_i^k que minimizan la energía se hace a través de la matriz de Kohn y Sham. Esta matriz se puede calcular derivando la energía del sistema en función de los coeficientes de la matriz densidad. Para este trabajo, nos interesa la contribución a esta matriz del término de intercambio-correlación, χ:

$$\chi_{i,j} = \frac{\partial E_{XC}}{\partial C_{i,j}} \tag{3.4}$$

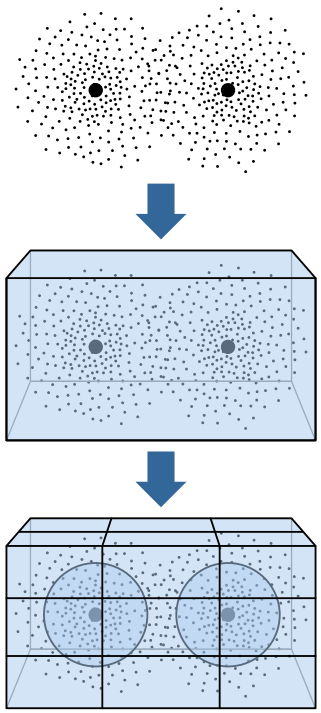


Fig. 3.2: Esquema del mallado de integración, partido en cubos y esferas para un sistema de dos átomos.

Dado que el término ϵ_{XC} depende de (es funcional de) la densidad y su gradiente, para calcular la derivada de este término es necesario calcular las derivadas segundas de la densidad, es decir su Hessiano.

Un diagrama simplificando la estructura del cálculo de DFT se puede ver en la figura 3.1. Los pasos (a) a (e) corresponden a la etapa de inicialización y se calculan una única vez al comienzo de la simulación. La iteración de SCF (*Self-Consistent Field*) se compone de los pasos (f) a (l). Este ciclo se repite mientras la matriz densidad cambie más de una cierta tolerancia previamente establecida [3].

Los pasos de la figura 3.1 computacionalmente más costosos son (i), (j) y (k). En la implementación original de LIO [4], se muestra cómo estos pasos obtienen grandes mejoras en GPU sobre su implementación de referencia en CPU. Sin embargo, estos pasos todavía insumen una gran cantidad de tiempo, incluso en las versiones aceleradas.

A continuación se presentan distintos cambios realizados a las rutinas para minimizar los tiempos, sin cambiar la estructura general del cálculo.

3.2. Implementación en CUDA

El trabajo realizado consistió en un estudio preliminar de los limitantes de desempeño de la implementación en GPU existente y en la aplicación de diversas técnicas para mejorar su rendimiento. Se contemplaron únicamente dos arquitecturas de GPU: Nvidia Fermi y Nvidia Kepler, las predominantes en el mercado GPU de HPC. Las diferencias sustanciales entre ambas, detalladas en la sección anterior hacen que un único análisis unificado sea insuficiente para describir cómo se comportan las diversas mejoras implementadas. Se usaron dos plataformas de prueba para este análisis:

I) 8 procesadores AMD Opteron 6276 de ocho cores cada uno (totalizando 64 cores), 128GB DDR3 ECC de memoria principal y cuatro dispositivos Nvidia Tesla M2090 (Fermi).

II) 2 procesadores AMD Opteron 6320 de ocho cores cada uno (totalizando 16 cores), con 64GB DDR3 ECC y una Nvidia Tesla K40 (Kepler).

Otros nodos de caracteristicas similares se usaron para experimentación, con resultados comparables a los descriptos. Estos sistemas se encuentran detallados en el apéndice A.

Los casos de estudio analizados en esta sección corresponden principalmente al grupo hemo y a una molécula de fullereno C_{60}, ambos descriptos en detalle en el apéndice B.

3.2.1. Limitantes de performance

De las etapas descriptas en la figura 3.1, la más costosa en la implementación existente para GPU es el paso (j). En el cálculo de un sistema de tamaño medio, este paso insumía el 94 % del tiempo total. Esto hacía que esta etapa llevara tanto tiempo como todos los demás cálculos de SCF combinados, incluso en las GPU más veloces del mercado (Tesla K40, GeForce GTX 780). Además, en Fermi el tiempo de cómputo de SCF era menor que en Kepler, yendo en contra de lo esperado considerando las especificaciones de estos dispositivos.

A continuación se detallan los cambios realizados para mejorar el rendimiento de la implementación del cómputo de energía de intercambio-correlación (simplemente *XC* de aquí en adelante).

3.2.2. Subutilización de los SM

La subutilización de los SM (*Streaming Multiprocessor*) se da en los casos donde hay SM que estén listos para ejecutar pero que no puedan hacerlo debido a que se necesitan recursos que no están disponibles (debido a que otro los está utilizando en ese momento). La métrica usada para determinar esta saturación es la *ocupación* de los SP (*Streaming Processor*). Ésta se define como la proporción de *threads* activos sobre el total de *threads* disponibles de un bloque.

Existen en la arquitectura CUDA tres recursos principales que, en un principio, pueden parecer ilimitados pero en realidad son recursos críticos compartidos por los procesadores de la GPU. Éstos son:

- Cantidad total de *threads* por bloque.
- Cantidad total de registros usados por *thread*.
- Cantidad de memoria compartida por bloque.

La asignación de recursos computacionales de los SM funciona corriendo un bloque en cada SM. Este bloque ejecuta sin interrupciones en el SM hasta que terminen todos sus *threads* asignados. Idealmente, cada bloque cuenta con una cantidad de *threads* suficientemente grande de manera de poder esconder la latencia. La arquitectura GPU esta diseñada para este fin, por lo cual se cuenta con un mecanismo de cambio de contexto de costo cero [14] para poder empezar a correr los *threads* de un warp diferente del mismo bloque.

Si un bloque no cuenta con suficiente cantidad de *threads* para ejecutar de manera concurrente, el SM va a tener que esperar a que finalicen las operaciones de alta latencia de estos warps sin nada que hacer mientras tanto. Si, por el contrario, se tuvieran miles de *threads* por bloque, entonces es posible que las operaciones de sincronización de un bloque (por ejemplo un *barrier*) sean excesivamente costosas.

La arquitectura GPU de Nvidia organiza los registros de todos los *threads* en un único *register file* por SM, común a todos los bloques. Como cada *thread* usa decenas de registros para guardar los cómputos intermedios, Nvidia decidió unificarlos, ya que es muy variable la cantidad que va a usar cada kernel de ejecución. Una de las grandes diferencias entre Fermi y Kepler es la cantidad máxima de registros por *thread*. Mientras que Fermi permitía hasta 63 registros, Kepler permite hasta 255. Esto es positivo para poder correr bloques de pocos *threads* pero gran cantidad de registros. Por otro lado, aumenta la presión sobre el register file. Cuando se lanzan muchos *threads* que puedan estar corriendo paralelamente entre todos los SM de la GPU, es posible que se supere la cantidad máxima de registros presentes en el register file. Esta situación fuerza a que el scheduler (el módulo encargado de asignar *threads* de ejecución a los SM) no pueda ejecutar concurrentemente más bloques, dejando SPs ociosos.

Finalmente, al igual que con los registros, la memoria compartida es un recurso limitado. Como solamente se cuenta con hasta 48 KB (Fermi-Kepler) de memoria de este tipo para ser repartida entre todos los bloques que estén corriendo en todos los SM, el scheduler deberá decidir no poner a ejecutar más bloques simultáneamente que los que pueda soportar la cantidad de memoria compartida.

El problema de optimizar el cómputo del término de XC tratado contó con todos estos limitantes. Afortunadamente, las herramientas de profiling usadas reportan estas situaciones, haciéndolas fundamentales a la hora de buscar mejoras a realizar.

3.2.3. Cambios en el threading

El cuello de botella fundamental en la ejecución del cálculo de la densidad electrónica radicaba en cómo se distribuía el trabajo de cómputo entre los kernels. La estrategia de paralelización original determinaba la partición del grupo a resolver instanciando un bloque por cada punto (`blockId.x`<puntos) con una cantidad fija de *threads* (usando `threadId.x`<`BLOCK_SIZE`). Los *threads* servían para reutilizar la memoria compartida; cada *thread* leía un elemento de la matriz densidad ($C_{i,j}$) y luego lo compartía con los demás *threads*.

El cambio más importante para la *performance* provino de reconsiderar la estrategia de paralelización. Asignar un bloque cada punto tiene el problema que en grupos con numerosas funciones, divide el trabajo con una granularidad muy gruesa, generando demsiado trabajo por thread. Esto causa que los SM estén ocupados constantemente por kernels de larga duración. Lo que se busca es que los bloques realicen menos trabajo por llamada, de modo que se puedan despachar a más SM a medida que estos terminen. En definitiva, lo que se busca es incrementar el throughput (cantidad de trabajo por unidad de tiempo) de la placa. Este enfoque se esquematiza como estrategia 1 a la izquierda de la figura 3.3.

La nueva estrategia (a la derecha de la figura 3.3) del cálculo dispone un bloque por cada función, de modo que cada *thread* i calcule $F_i \sum_j^i F_j C_{i,j}$.

La distribución original resultaba natural al problema, pero visto en detalle, implica una cantidad elevada de lecturas a memoria global. Como cada *thread* realiza un punto dentro de un bloque, los F_i que lee son únicos a ese punto. Al modificarse el esquema,

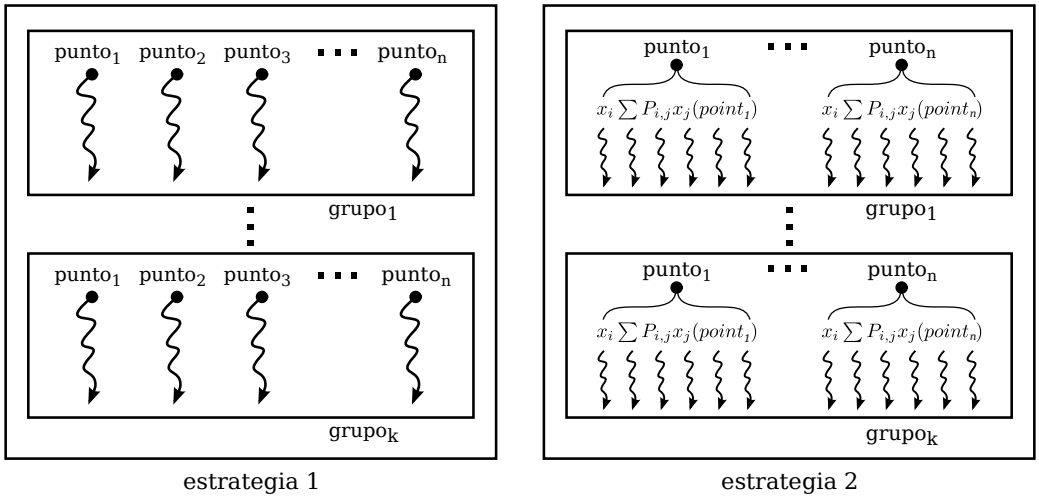


Fig. 3.3: Estrategia 1: corresponde con la implementación original. Estrategia 2: propuesta de modificación de estrategia de paralelismo. Se muestra un esquema de cómputo de la densidad electrónica durante E_{XC}. En lugar de usar un *thread* por punto, se usa un *thread* para cada función en cada punto (que realiza el cálculo detallado matemáticamente en la figura, siendo i el número de función).

el cálculo ahora se realiza parcialmente por cada *thread* y, por lo tanto, cada lectura de F_i se agrega a la memoria compartida. El resto de los *threads* no van a leer las funciones de memoria global, sino de la compartida que ya cargaron los demás. Esto se traduce en una lectura global y `BLOCK_SIZE`-1 lecturas de memoria compartida por *thread*, siendo la latencia de acceso a memoria compartida al menos diez veces menor que la memoria global [26].

Los resultados de este cambio se pueden ver en la figura 3.4, detallando tanto para Fermi como para Kepler la diferencia de *performance*. Observamos que el impacto del cambio es mucho mayor en Kepler que en Fermi: la cache L2 es mayor, la cantidad de registros por *thread* es mayor y se pueden correr más bloques concurrentemente, entre otros.

Este cambio solo es posible gracias al crecimiento de las memorias compartidas (16 KB en la generación de dispositivos existentes en implementación original, actualmente 48 KB). Finalmente, esto también se beneficia del incremento en tanto la cantidad de SM y la de SP por SM. Todos estos factores permitieron una gran aceleración.

La otra modificación importante consistió en partir el problema en más bloques para los grupos que tuvieran mayor cantidad de funciones significativas. Se decidió agregar otra dimensión a la grilla de bloques (`blockId.y`) para determinar cuántos grupos de *threads* son necesarios para procesar todas las funciones de ese punto. Se denomina a este parámetro *altura_bloques* y se calcula para cada partición como *altura_bloques* $= m/$`BLOCK_SIZE`. Para las particiones chicas, este valor no supera a 1. En los cubos y esferas más grandes (de los sistemas probados), *altura_bloques* oscila entre 2 y 6. Esto representa una gran cantidad de bloques adicionales con respecto al método anterior, mejorando el throughput total al tener menos trabajo por bloque y permitiendo un mejor aprovechamiento de los múltiples SM con los que cuentan las placas Fermi y Kepler.

Un punto de intensa discusión durante estos cambios es el valor óptimo de `BLOCK_SIZE`. Para nuestro problema, se decidió utilizar un número de *threads* por bloque múltiplo del

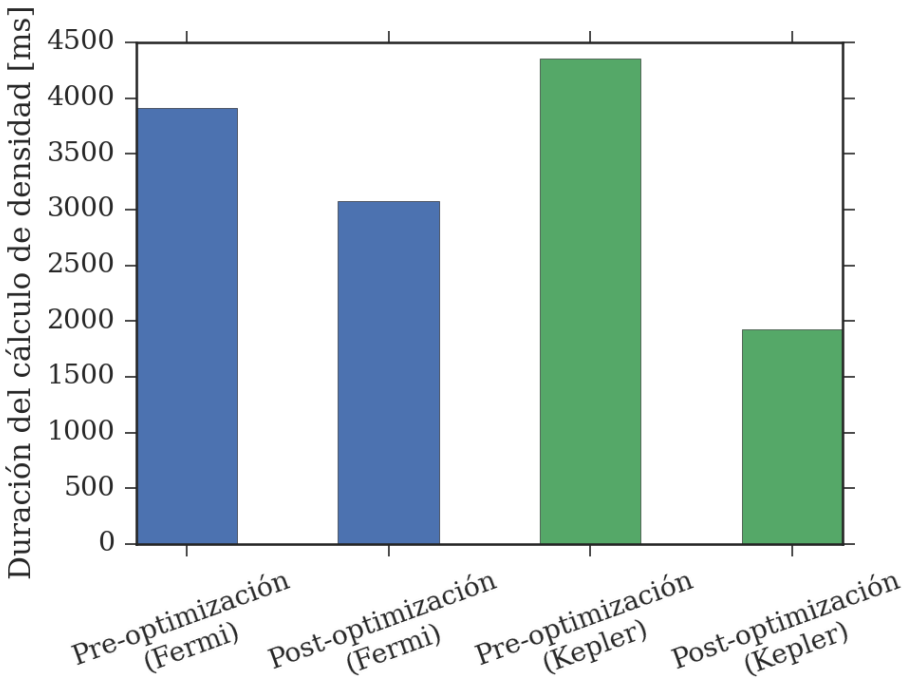


Fig. 3.4: Tiempo del cálculo de la densidad. Se muestran los resultados utilizando el grupo hemo como ejemplo en distintas arquitecturas. Los tiempos antes del cambio de estrategia de paralelización corresponden a *pre-optimización*, en tanto que *post-optimización* refiere los obtenidos con la implementación de la mejora. Otros sistemas ensayados exhiben comportamientos similares.

tamaño de un warp (32 *threads*) y que sean potencias de 2. Esto permite estudiar cómo afectan en el tiempo de procesamiento contar con uno o más warps por bloque. Una ventaja de usar bloques de 32 *threads*, es que el costo de la sincronización es exactamente cero. No se precisa sincronizar puesto que en CUDA los *threads* ya se encuentran sincronizados con sus compañeros de warp. Un bloque chico, además, permite usar más memoria compartida por *thread*, dado que hay una cantidad fija de memoria por bloque (entre 32 KB y 64 KB). Cuando se cuentan con muchos más *threads*, se debe reducir el uso de memoria por thread de modo que todos puedan ejecutar concurrentemente.

Por otra parte, la literatura [18] sugiere que, siempre que sea posible, se usen bloques grandes y con *threads* lo más independientes que se pueda. Una gran cantidad de *threads* por bloque permite tener muchos más warps para ejecutar de modo de esconder las latencias de operaciones y accesos globales. Sin embargo, contar con muchos *threads* hace que las sincronizaciones sean mucho más costosas. Además, como cada SM no cuenta con desalojo de bloques, los recursos están asignados a un mismo bloque por largos períodos.

Inicialmente, `BLOCK_SIZE` se había fijado en 128 *threads* por bloque (cuatro warps). Utilizando más memoria compartida en el esquema de paralelización, este valor resultó demasiado elevado y dificultaba ocupar todos los SM en dispositivos Fermi y Kepler. Además, el costo de sincronización medido usando el profiler de Nvidia resultaba mayor al 75 % del tiempo total bloqueado del código, haciéndolo muy costoso. Con solamente 32 *threads*, se podía maximizar la ocupación de los SM, pero había muchos más bloques y no había tantos SM para poder aprovecharlos. Finalmente, luego de disminuir un poco el uso de

registros por *thread* mediante una reorganización de las operaciones, se fijó este valor en 64 *threads* por bloque. Se muestra en la figura 3.5 que tener solo un warp es bueno, pero mejor aún es tener dos warps ejecutando simultáneamente: cuando uno esta bloqueado por un acceso a memoria, el otro puede ejecutar con costo cero de cambio de contexto.

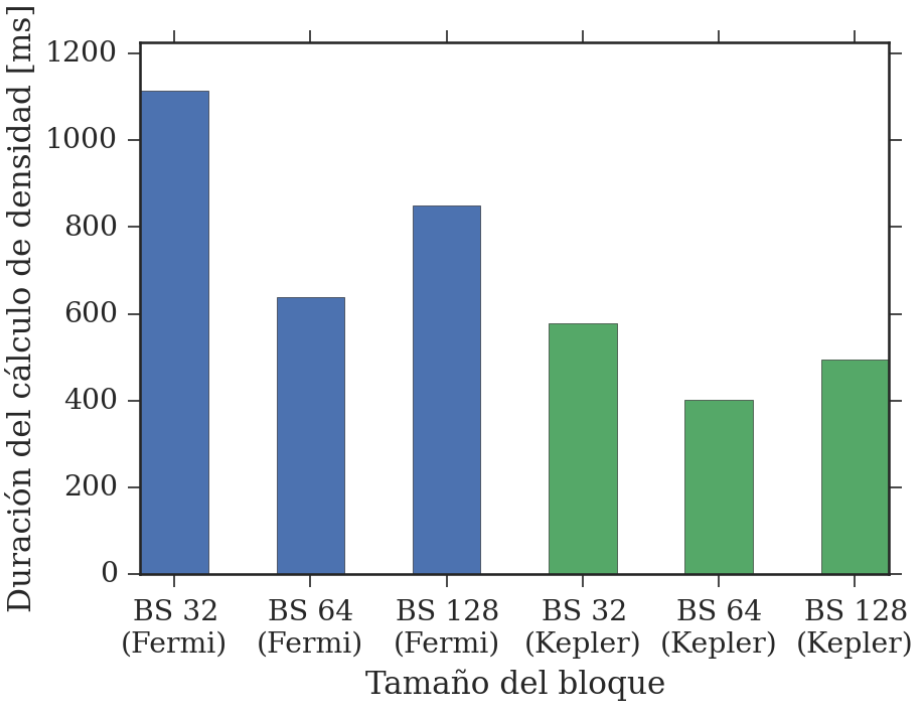


Fig. 3.5: Tiempo de del cálculo de la densidad. Se utiliza el grupo hemo como ejemplo y se varían la arquitectura y el tamaño de `BLOCK_SIZE` (BS).

3.2.4. Cambios en la reducción de los bloques

La reorganización de la paralelización del kernel del cálculo de la densidad creó la necesidad de un paso de unificación de valores que antes se realizaban implícitamente.

Al tener más de un bloque por punto, se deben sumar los resultados de cada uno, es decir, es necesario realizar una *reducción*. Para esto, se reutiliza la memoria compartida que fue empleada en el cálculo de la densidad anterior. Los *threads* ponen su contribución al cómputo realizado en su posición correspondiente de la memoria compartida, evitando tener que realizar operaciones de sincronización entre ellos. Los valores parciales se reducen, luego, con un algoritmo paralelo en árbol: cada *thread* suma el valor de la posición x con el valor en $2 \cdot x$ si este existiera. Esto se repite por la mitad de los *threads*, hasta que todos los valores hayan sido sumados y el *thread* 0 del bloque escribe el resultado a la memoria global.

Esta técnica de reducción es sumamente conocida para arquitecturas distribuidas, generando la respuesta en $O(\log_2 n)$ pasos. La literatura de CUDA [38] sugiere técnicas adicionales para minimizar aún más el tiempo empleado en esta reducción. Sin embargo, al realizarse a lo sumo seis operaciones, no es necesario aplicarlas en nuestro caso particular.

Las cuentas parciales para cada punto se almacenan en memoria global. Para eso se definió una matriz por cada uno de los parámetros que se deben acumular, con un elemento por cada bloque asignado al kernel de densidad electrónica. De este modo, cada bloque tiene un lugar propio para escribir los resultados parciales. Estas matrices son de tamaño $O(\#_{puntos} \cdot altura\_bloques)$, que resulta en menos de 1 MB en el caso más grande que se ha tratado.

El siguiente paso de la reducción consiste en acumular los *altura_bloques* valores generados en un solo por punto. Para lograrlo, se debe recuperar las matrices temporales con las partes de las cuentas, agregarlas y calcular el potencial correcto. Con los resultados obtenidos se generan los coeficientes para calcular la actualización de la matriz de Kohn-Sham y los factores para el cálculo de la matriz de fuerzas.

Esto se refleja en el código como una llamada a un nuevo kernel, posterior a la cuenta de la densidad y con múltiples matrices temporales adicionales. Este kernel es sumamente eficiente porque solo tiene que realizar a lo sumo *altura_bloques* sumas y una llamada a la función que calcula el potencial y con él la contribución a la densidad. Al terminar este kernel, se tiene entonces un resultado por cada punto, lo mismo que se producía anteriormente pero utilizando mucho mejor los recursos del dispositivo.

Lanzar este kernel adicional, al funcionar CUDA de manera asíncrona con respecto al CPU, se puede hacer inmediatamente después de lanzar el kernel de cómputo de la densidad electrónica. Luego, como el cálculo de la densidad demora más que el tiempo necesario para lanzar el kernel de la reducción, este tiene efectivamente un costo despreciable ya que se pueden encolar en la GPU estas dos llamadas independientes. Desafortunadamente, esta estrategia no se puede usar para todos los kernels que componen XC puesto que entre algunos de ellos sí hay operaciones que deben hacerse en forma serial en el CPU.

3.2.5. Cambios en los accesos globales

Las arquitecturas de placas de vídeo están diseñadas en torno al poder de cómputo. Las decisiones tomadas por los diseñadores de las GPUs se concentran alrededor de paralelismo a lo ancho, poniendo un gran énfasis en la cantidad de núcleos. En consecuencia, se dispone de menor cantidad de espacio físico para más memoria.

Esta decisión implica que la amplia mayoría de la memoria de la GPU se encuentra localizada afuera del chip. Por lo tanto, esconder la gran latencia que tiene al acceder a la misma es parte importante del paradigma de programación para GPGPUs.

Es crítico, además, que los accesos a memoria sean a direcciones múltiplos de 16, esto se conoce como *accesos alineados*. El término *coalescencia de memoria* se define en GPU como la organización de los accesos a memoria de manera secuencial, ordenada y predecible. Cuando la GPU accede a memoria de manera alineada, puede traer 16, 32 o 64 elementos de 32 bits en una sola lectura [13], suficiente para cada uno de los *threads* del warp. Si, en cambio, no se respeta la alineación de la memoria o los *threads* tienen un patrón de pedidos impredecible, entonces se deberán serializar los accesos y separar en múltiples transacciones a memoria global para satisfacerlos. Este problema se agrava si se deben hacer accesos frecuentes por cientos de *threads*, como es el caso de los bloques con gran nivel de paralelismo explícito.

Originalmente en la aplicación LIO, todos los *threads* calculaban puntos independientes y no había lecturas compartidas a las matrices. Esto dejó de valer al modificarse el esquema de paralelización a más de un bloque por punto. En este caso, el cálculo de las densidades requiere acceder a las matrices en un orden por columna, el cual es inconveniente: cada

pedido requiere realizar 32 accesos que no pueden realizarse en una sola transacción al no ser coalesciente. Hacer esto para las matrices de funciones, gradientes y Hessianos resulta sumamente costoso. A diferencia del caso de CPU, este problema no es mitigado mediante memoria cache, ya que en GPU las memorias cache L1 y L2 son de mucho menor tamaño.

La solución a este problema consiste en transponer estas matrices. La transposición de matrices es un ejemplo sumamente estudiado por la literatura de CUDA ya que ataca uno de sus puntos débiles: el ancho de banda de transferencia. En la figura 3.6 se puede observar que tanto en Fermi como en Kepler es muy notoria la diferencia de *performance* que se obtiene rediseñando los accesos a memoria. En Kepler, al contar con una cache L2 del doble de tamaño que Fermi, los accesos desalineados tienen un menor costo en *performance*, pero todavía recibe grandes mejoras.

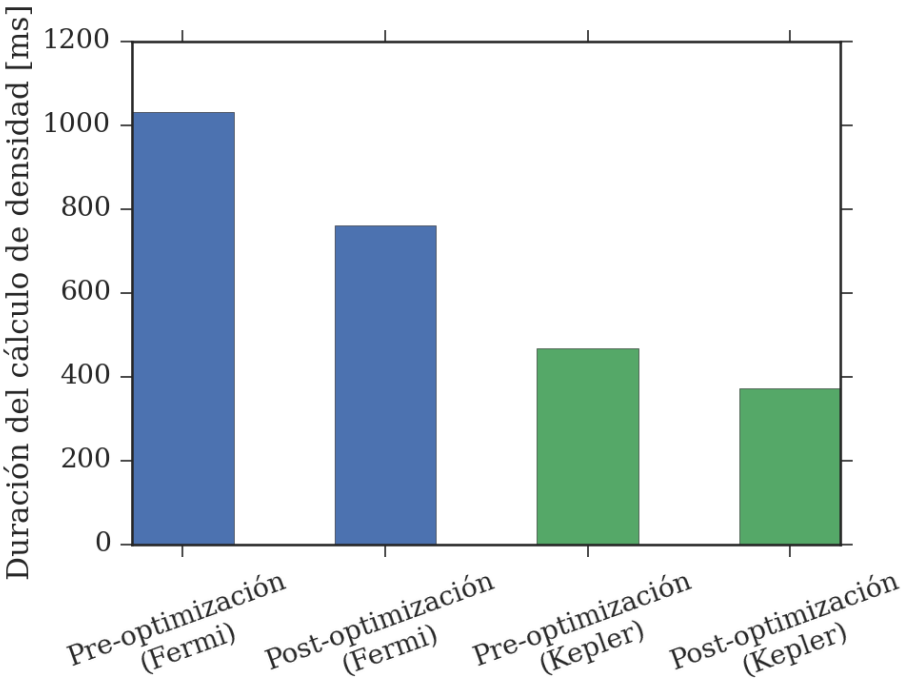


Fig. 3.6: Duración del cálculo de densidad simulando el grupo hemo en distintas arquitecturas antes y después de implementar la transposición de las matrices de función (se incluye el costo de trasponerlas).

Otra de las técnicas con la que cuenta CUDA para mitigar la latencia de acceso es el uso de memorias intermedias entre el procesador y la memoria global. Una de ellas es la cache de textura (la otra es la memoria constante). El detalle crucial de la cache de textura es que un *miss* provoca que se traigan datos no solo contiguos en memoria, como pasa en las caches de CPU normalmente, sino que además se traigan los datos en posiciones lógicas contiguas, es decir, variando las distintas dimensiones de la matriz subyacente.

Las memorias de textura se ajustan bien a los problemas en GPGPU, porque se relacionan íntimamente con mecanismos de paralelismo de CUDA. Como los problemas se pueden dividir en bloques con *threads* en las componentes x, y, z, entonces tiene mucho sentido pensar que las estructuras de datos subyacentes se van a acceder usando índices multidimensionales.

En nuestro problema, la memoria de textura se presenta como una solución para los accesos bidimensionales de la matriz densidad para el grupo de puntos. Esta matriz es utilizada completamente por bloque y, debido a su tamaño, resulta demasiado grande para ser ubicada en memoria constante. Sin embargo, suele entrar casi completamente en la memoria de textura, la cual se ajusta muy bien a las lecturas por filas y columnas.

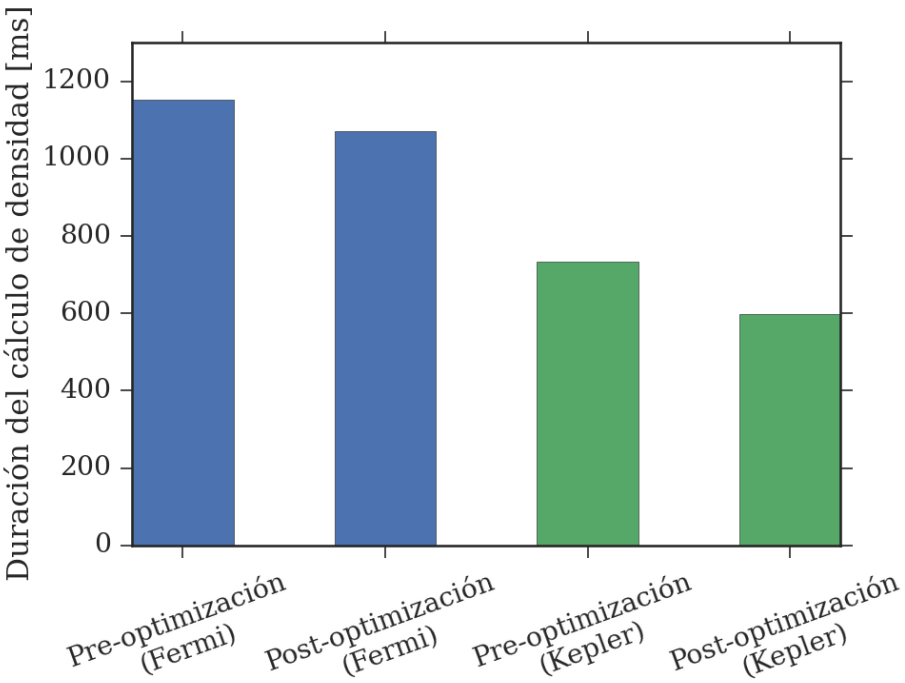


Fig. 3.7: Duración del cálculo de densidad simulando el grupo hemo en Fermi y Kepler usando memoria de textura para leer la matriz densidad.

Como se puede ver en la figura 3.7, esta optimización aprovecha este recurso único de GPU. Sin embargo, agrega un factor más que se debe tener en cuenta a la hora de analizar el código. Para administrar los accesos a la memoria de textura, cada multiprocesador tiene múltiples unidades de textura. Cuando se depende demasiado de esta memoria para esconder la latencia, se presentan contenciones sobre el acceso a estas unidades. Esta cache conviene usarla solamente en los accesos más costosos, ya que el uso excesivo de la misma genera un deterioro completo de *performance*. Esto se observó cuando se intentó aplicar esta técnica para las matrices de funciones.

3.2.6. Cambios en el almacenamiento de matrices temporales

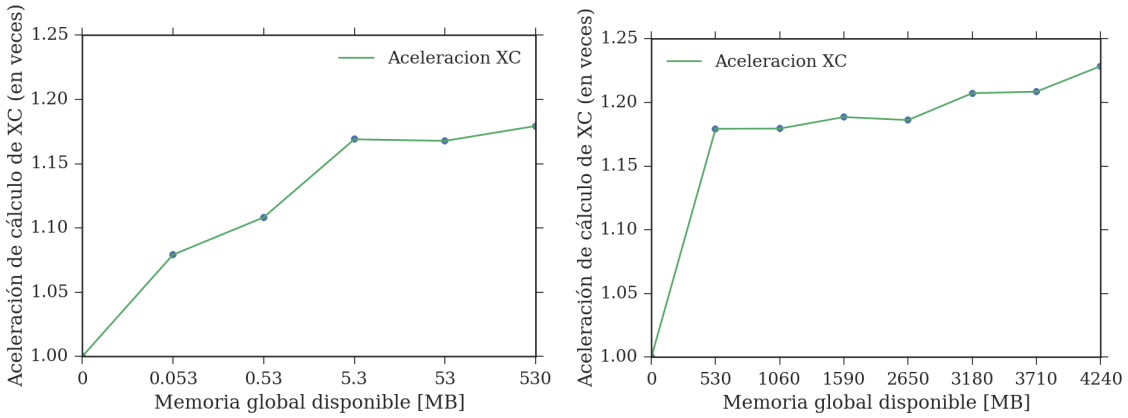
Una de las principales limitaciones de las GPUs es la cantidad fija de memoria, que no se puede expandir al estar soldada a la placa. Esto era especialmente notorio cuando las placas contaban con menos de 1GB de memoria (años 2007-2008). Para problemas de cálculo numérico este limitante es muy serio: los problemas que normalmente entraban en la memoria principal de un CPU, no entraban completos en las GPU. La decisión tomada por muchas aplicaciones de entonces era compensar esto calculando datos intermedios y descartarlos luego; teniendo que ser recalculados en las todas las iteraciones.

Al momento de concebir la implementación original, las placas de vídeo disponibles apuntaban a consumidores y no a HPC, con lo cual disponían de poca memoria. Para aprovechar los recursos disponibles actualmente en estos dispositivos, se desarrolló un método para poder almacenar las matrices de los valores de funciones y sus derivadas en cada punto para cada grupo durante la ejecución de la aplicación, para tantos grupos como fuese posible con la memoria disponible. Esto implica almacenar la matriz de funciones, con una cantidad de elementos $O(puntos \times funciones)$, y si se usa el método GGA (*Generalized Gradient Approximation*) para realizar los cálculos de DFT, también se requieren las matrices de gradientes y Hessianos de las funciones ($O(3 \times puntos \times funciones)$ y $O(6 \times puntos \times funciones)$ respectivamente).

Para poder determinar cuáles grupos son almacenadas en memoria y cuáles no, se ideó una heurística para definir el orden de las particiones a solucionar. Esta heurística estima qué tamaño van a tener las matrices temporales a almacenar y ordena las particiones de menor a mayor. La base de este criterio es que el costo base (sin contar la cantidad de funciones y puntos involucrados en las matrices) para un grupo es muy alto y, por tanto, es más conveniente almacenar muchas matrices temporales de particiones chicas a almacenar las de un pequeño grupo de las particiones grandes.

Para controlar la administración de memoria, se calcula si la placa dispone de suficiente espacio libre para guardar las matrices de funciones y, si puede, se almacenan de manera permanente (hasta que la partición se mueva a otro dispositivo o la liberación de recursos al finalizar las iteraciones de SCF). Este mecanismo además es configurable de modo que una ejecución pueda usar un porcentaje de la memoria con la que cuenta la placa, para poder correr múltiples procesos de simulación concurrentemente.

Un detalle a tener en cuenta es que, incluso si se aumenta el porcentaje de memoria usado para cachear estas matrices, tal vez no alcanza para todo el sistema. Esto se nota principalmente en la línea GeForce, que cuenta con aproximadamente entre un cuarto y la mitad de la memoria global que tienen su equivalente en la línea Tesla. Una estrategia para mitigar el problema es el uso de múltiples placas, que también distribuyen el almacenamiento además del cómputo.



(a) Entre 0 y 530 MB. *(b)* Entre 0 y 4240 MB.

Fig. 3.8: Aceleración del cálculo de una iteración de XC de fullereno en función de la memoria de la placa usada para almacenar las funciones.

Se evaluó el tiempo de ejecución de un sistema modelando un fullereno C_{60}, variando la

cantidad de memoria disponible para el almacenamiento temporal. El fullereno es un caso grande, en el cual se puede apreciar el impacto de almacenar las matrices. En sistemas más reducidos, mucha menos memoria es suficiente para que no se tenga que recalcular nada. Los resultados se pueden ver en la figura 3.8. Se puede observar que el almacenamiento de las matrices de funciones produce una mejora de casi el 25 % en el tiempo de ejecución de toda una iteración de XC. Es interesante notar que son los primeros grupos los que causan mayor diferencia en los tiempos de ejecución. Con 5,3 MB, se pueden guardar los 23 grupos más pequeños del fullereno C_{60}; de ahí en más, el impacto de la mejora decae. Esto sucede porque en grupos chicos el overhead del cálculo usando kernel de funciones es muy elevado en comparación a las cuentas (se desperdician muchos *threads*). Este comportamiento se puede apreciar en la figura 3.9: es notorio el peso importante de los grupos chicos en el tiempo total del cálculo de las funciones.

Dado que, para casi todos los grupos, el costo de calcular las matrices se vuelve cero gracias a las posibilidad en dispositivos modernos de almacenarlas en memoria, se decidió que no era necesario optimizar el cálculo de las mismas.

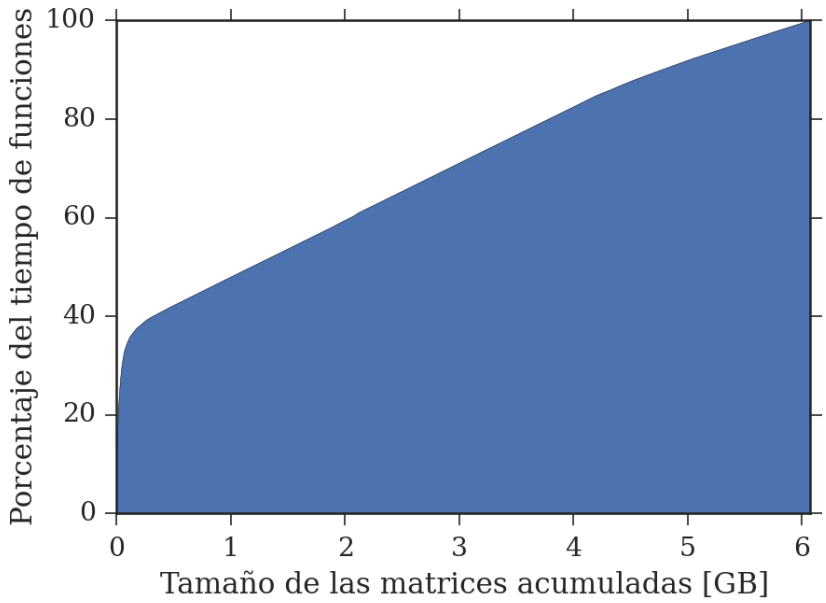


Fig. 3.9: Fracción del total del tiempo de cálculo de las matrices de funciones en una iteración en función del ordenada por el tamaño acumulado de estas en un fullereno C_{60}.

3.2.7. Cambios en las memorias compartidas

Otro camino explorado para mejorar el rendimiento del código es hacer un mejor empleo de las memorias compartidas. Como todos los bloques en ejecución deben utilizar menos de 48KB de esta memoria simultáneamente, es imprescindible minimizar su uso de modo que no se termine subutilizando los SM.

Se experimentó, con un grado de éxito variable, con disminuir el tamaño de los vectores donde se almacenan las derivadas direccionales. El mecanismo utilizado por la implementación inicial correspondía al sugerido por CUDA: agrupar los tres componentes del gradiente y usar un elemento en cero para mantener la memoria alineada. Es decir,

un elemento corresponde a la tira $(\frac{\partial F_i}{\partial x}, \frac{\partial F_i}{\partial y}, \frac{\partial F_i}{\partial z}, 0)$. El método GGA utiliza además las derivadas segundas de las funciones. Al ser funciones Gaussianas de la base, son derivables tanto como se quiera (C^∞), por lo que la matriz Hessiana es simétrica [39]. Entonces, de las nueve derivadas segundas solo es necesario guardar seis. Se usa un esquema similar al del gradiente: un elemento para cada grupo de 3 componentes (siendo en total seis derivadas segundas cada dos elementos), cada uno con un cero de *padding*.

El experimento consistió en remover este ajuste y reducir cada elemento a sus tres componentes. En las memorias globales, traer de a cuatro valores consecutivos fuerza al compilador a alinearlos a 16/32 bytes (simple y doble precisión respectivamente). Esto presenta grandes ventajas a la hora de hacer transferencias de memoria global en los accesos, por lo cual se dejaron como estaban sin modificaciones.

Sin embargo, este mismo criterio no aplica a las memorias compartidas de la GPU. Como los accesos a estas memorias se realizan de a 4 bytes y no de a 16/32, entonces no tiene ninguna ventaja en particular realizar el alineamiento; ya están alineadas porque los elementos de cada punto son flotantes de precisión simple o doble (4 y 8 bytes respectivamente). Adicionalmente, como el cuarto valor no tiene forma de marcarse como algo que no sea padding de alineación, todavía se opera normalmente con él usando operaciones vectoriales definidas para estos tipos, por lo que eliminarlo ahorra una operación de cálculo. Más aún, se observa una disminución del 18 % de la memoria compartida por *thread* (usando el esquema de compartir valores de funciones por punto como fue descripto en la sección 3.2.3), sin ninguna desventaja por alineación a la hora de accederlos. Esta mejora, en teoría, permitiría aumentar la cantidad de bloques ejecutando concurrentemente en los SMs, para poder eliminar la limitación presente debido al uso concurrente de memoria compartida.

Como se puede observar en la figura 3.10, se obtienen muy poca ganancia al usar elementos de tres componentes con respecto a usar elementos de cuatro. Esto muestra que el limitante de concurrencia de este kernel no pasa por falta de memoria compartida. Además, se puede apreciar que los tiempos de acceso a los datos en estas memorias son realmente muy bajos, pudiéndose ver como con incluso un 18 % más de operaciones de lectura-escritura, el tiempo de ejecución casi no varía. Por otra parte, esto muestra también que no hay diferencia en los tiempos de acceso a memoria compartida; buscar tres o cuatro elementos es equivalente. En consecuencia, se debe tener los accesos alineados en la global, pero en la compartida se debe tener la menor cantidad de overhead posible, dadas las diferencias en los buses de acceso.

3.2.8. Escalando más allá de un GPU

Una vez que fueron solucionados muchos limitantes de *performance* en los kernels del cómputo de la densidad electrónica y del cálculo de la matriz de Kohn-Sham, se llegó a un punto donde no fue posible determinar mejoras significativas en el cálculo para reducir aún más los tiempos de ejecución. Se decidió subir un nivel más el paralelismo, de modo de poder solucionar múltiples particiones simultáneamente. Dado que es independiente el cómputo de cada partición (salvo la acumulación en la matriz de Kohn-Sham de salida y en la matriz de fuerzas interatómicas), nos pareció que sería interesante analizar cómo escala distribuir el cómputo en múltiples GPU.

Para dividir el problema entre varios dispositivos se usa el soporte para OpenMP del compilador ICC de Intel para utilizar *threading* de manera sencilla. OpenMP corresponde a un *estándar* de compiladores para paralelismo asistido por el usuario, mediante `pragmas`

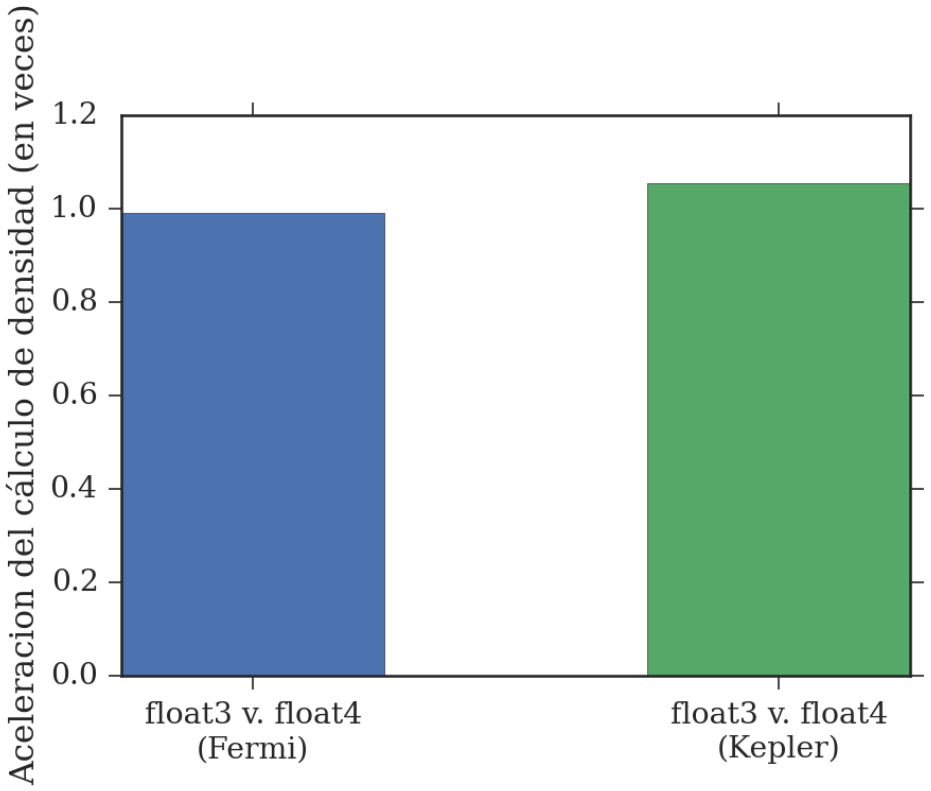


Fig. 3.10: Aceleración obtenida en Fermi y Kepler al reducir a tres componentes los elementos en la memoria compartida.

de compilador y una biblioteca de soporte a estos pragmas sobre primitivas del sistema operativo [23].

"Asistido por el usuario" se refiere a que el programador debe indicarle a OpenMP qué ciclos paralelizar, y debemos asegurar de no introducir efectos colaterales como condiciones de carrera por memoria compartida, *deadlocks*, etc. También se debe indicar a OpenMP qué estrategia usar para dividir las iteraciones de un ciclo entre los hilos de ejecución, cuántos *threads* utilizar, etc [40].

Un ejemplo de como se puede utilizar esta biblioteca para paralelizar un ciclo sencillo se encuentra en el código de C++ de la figura 3.11. En el ejemplo, las iteraciones del ciclo principal serán divididas entre los cuatro *threads* en porciones iguales y consecutivas, y cada uno ejecutará el cuerpo interno del ciclo en sus iteraciones. La cantidad de hilos de ejecución a lanzar y cómo se dividen las iteraciones son controladas mediante los parámetros del `pragma` que el programador explicita en el código. El inicio y fin de un *loop* paralelizado con OpenMP son puntos de sincronización y tienen un *overhead* asociado, por lo cual el costo las operaciones que se hacen en paralelo debe ser significativo para compensar.

Usando esta herramienta, cada uno de los *threads* en el host se configurará para una placa solamente, usando una sección paralela que todos los *threads* ejecuten y donde cada uno usará la placa que le corresponda según su número identificador de *thread* (que OpenMP provee mediante funciones auxiliares). Para que el hilo de ejecución elija la placa que le corresponde, se utiliza una instrucción del driver de CUDA (`CudaSetDevice`) que permite que durante toda la vida del *thread*, todas las llamadas a kernels se realicen

```
#pragma omp parallel for num_threads(4)
for(int i = 0; i < n; i++) {
    int result = 0;
    for(int j = 0; j < m; j++) {
        result += a[i][j] * b[j];
    }
    results[i] = result;
}
```

Fig. 3.11: Ejemplo de programa paralelizado con OpenMP para correr en cuatro *threads*. Cada *thread* tiene asignado una cantidad fija de iteraciones consecutivas.

automáticamente al mismo dispositivo.

CUDA permite que trabajar con múltiples placas de esta manera sea bastante sencillo. Las variables definidas como `__device__`, que residen plenamente en la GPU, son automáticamente instanciadas por cada dispositivo presente. De esta manera, es implícito cual variable usa cada kernel; la que esta definida para su dispositivo actual. Tampoco es necesario, en este caso, comunicar las placas entre sí, por lo tanto el paralelismo tiene un *overhead* mucho menor.

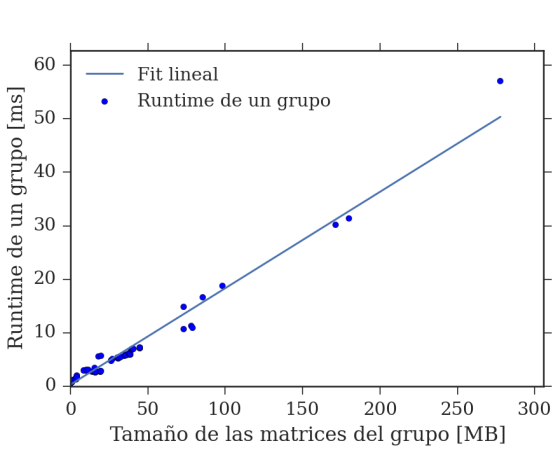
El principal problema que surge del uso de múltiples dispositivos radica en cómo distribuir la carga de los *threads* de modo tal que haya una cantidad de trabajo similar: Todos los *threads* deben sincronizarse al terminar. Por lo tanto, el tiempo de ejecución está determinado por cual de ellos tarde más. Este problema no es serio, siempre y cuando, se utilicen placas idénticas dentro de la configuración del sistema. Cuando se dispone de placas heterogéneas (en configuración de memoria, cantidad de SM, ancho de banda, etc.), se requiere del uso de técnicas de estimación de poder de cómputo para poder hacer una partición de trabajo equitativa.

Para distribuir las tareas se utilizaron dos técnicas combinadas, una para repartir las tareas estáticamente y otra para redistribuirlas dinámicamente de acuerdo al tiempo de ejecución observado de cada tarea.

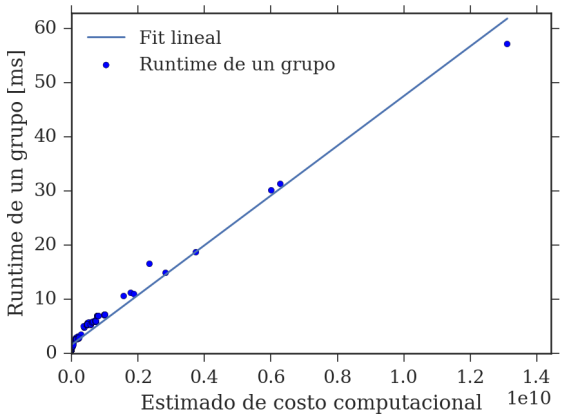
Estimación de cargas estáticas

Para la distribución estática, se usa como estimador del *runtime* el tamaño de las matrices de funciones de un grupo. Esta heurística resulta un acertado predictor del tiempo de cómputo de todos los kernels de un grupo confirmando que el problema actualmente está limitado por la memoria (*memory-bound*). Esto se ve en la figura 3.12: el tiempo de resolución de un kernel escala linealmente con la cantidad de memoria necesaria para operar. Se comparó también contra el predictor usado para estimar tiempo en CPU que se detallará más adelante en la sección 3.3.6.

Como se puede apreciar, hay una relación directa entre el tiempo de resolución con el tamaño de los elementos con los que trabaja. Esto da una pauta de la complejidad computacional del problema que se está resolviendo. Como al menos se necesita cada uno de los elementos de cada matriz de funciones para resolver un punto de cada grupo, se deberá hacer una lectura de cada matriz. Estas lecturas son tan costosas en comparación a las operaciones matemáticas que, incluso solo sabiendo el tamaño de las matrices, se puede predecir aproximadamente el tiempo de ejecución que va a tener un grupo.



(a) Predictor de tiempo usando *size_in_gpu*.



(b) Predictor de tiempo usando función de costo de CPU.

Fig. 3.12: Tiempos de ejecución del grupo hemo en Tesla K40 de acuerdo al costo del mismo según predictor.

El predictor basado en tamaño, si bien es posible observar que tiene una buena correlación con el tiempo de ejecución, no tiene la precisión que tiene el predictor basado en la función de costo en CPU, como se verá en la figura 3.22. No se pudieron encontrar parámetros que permitan ajustarlo para que funcione en GPU. Como Nvidia no publica toda la información interna de los procesadores GPU y, dada la cantidad de lugares adicionales donde se puede incurrir en latencias imprevistas, al funcionar asíncronos al CPU no es posible predecir tiempo de ejecución con excesiva confianza dado únicamente el cómputo a realizar. Si bien ambos predictores obtuvieron resultados similares, se decidió usar el más sencillo, el tamaño de las matrices de funciones, como punto de partida para realizar una partición del sistema en múltiples GPU.

Algoritmo de particionado

El problema de particionado de trabajo en *threads* se puede expresar matemáticamente de la siguiente manera: Siendo la entrada del algoritmo un vector $C = C_1, \cdots, C_n$ de costos y un valor m, la cantidad de *threads* a utilizar, devolver una partición en n cargas $P_1, \ldots, P_m$ tal que

$$\bigcup P_i = C \tag{3.5}$$

$$P_i \bigcap P_j = \emptyset, \qquad \forall 1 \leq i, j \leq m, i \neq j \tag{3.6}$$

y que minimize la carga más pesada en la partición, donde la carga de una partición es la suma de los pesos de todos los elementos que tiene asignados. Es decir, minimiza el valor de:

$$\max_i \sum_{p \in P_i} C_p \tag{3.7}$$

Un aspecto importante a señalar de este problema es que pertenece a la clase de problemas computacionales *NP-hard*. Ésta engloba problemas de similar dificultad: un

problema es *NP-hard* si resolverlo es tan difícil como resolver otro problema *NP-hard* en costo computacional. Estos problemas son difíciles ya que al momento de escribir esta tesis no se conocen algoritmos para ninguno de ellos que tengan costo polinomial en el tamaño de su entrada, y la afirmación o negación de su existencia es una de las preguntas abiertas más conocidas de la computación [41].

La complejidad de los algoritmos exactos que usan programación dinámica, es $O(Nn)$, con N el valor de la suma de P y n la cantidad de elementos de P. Como nuestra métrica de comparación para las particiones es el *runtime* expresado con precisión de microsegundo, el costo computacional de este algoritmo puede ser elevado en sistemas con cientos de particiones de miles de puntos.

La principal desventaja de usar únicamente este algoritmo es que no contempla, al menos en sus versiones más directas, el uso de recursos asimétricos. Es decir, resolver la partición óptima del problema de particiones cuando se sabe que el costo de resolver P_i en el dispositivo A es distinto a resolverlo en el dispositivo B. Por este motivo, no se puede particionarlo completamente de manera estática usando este algoritmo y se necesita adoptar una solución híbrida estática y dinámica, reutilizando la información de runtime para decidir cómo rebalancear las particiones entre iteraciones de XC, luego de haber hecho una aproximación inicial usando el predictor estático.

Resolver el caso de recursos asimétricos es útil para clusters de pequeño y mediano tamaño que incorporan recursos de manera incremental, y contempla también diferencias sutiles pero de fácil omisión en una configuración de GPUs (por ejemplo, el uso de buses de distintas velocidades para comunicar las placas con el *host*).

Una solución aproximada al problema, suponiendo dispositivos de cómputo simétricos, se detalla en la implementación en CPU (para utilizar los procesadores del sistema).

Inicialmente, se definió un orden para realizar toda la resolución del sistema y se las distribuye en orden circular (*round-robin*) entre todos los dispositivos, para generar la partición inicial y cargar a cada placa con una cantidad similar de tareas. Esto no significa que todas vayan a durar lo mismo, resultando posible notarlo en las mediciones realizadas. Luego, utilizando el tamaño de las matrices como indicador del tiempo de ejecución, se aplica una ronda del algoritmo de balanceo dinámico para terminar de balancear estáticamente. Esto se puede hacer ya que, como se mostró en la figura 3.8, existe una fuerte correlación entre el tamaño de las matrices de función y el tiempo de ejecución, la métrica que realmente nos interesa.

El paso de balanceo dinámico es necesario para solventar el problema de la distribución round-robin con grupos grandes, donde el *thread* que tenga que resolver grupos con más puntos va a tardar desproporcionadamente más que los demás. Un caso emblemático es el cubo que contiene al átomo de hierro en el grupo hemo, con más de diez mil puntos y cientos de funciones de la base. Este grupo requiere un esfuerzo computacional al menos dos órdenes de magnitud mayor que el más pequeño para el grupo hemo. La técnica adoptada para este problema consiste en una estrategia *master-slave* entre los dispositivos, donde el CPU genera múltiples *threads* (uno por cada placa) y les asigna un conjuntos de grupos a cada uno para que resuelvan. El tiempo de resolución de cada grupo y el total por *thread* son medidos por el CPU usando el reloj de alta precisión del sistema y al finalizar la iteración, se aplica el algoritmo de balance descripto a continuación. Así, los *threads* que terminen antes recibirán una mayor cantidad de trabajo de otros *threads* para la siguiente iteración del cálculo de SCF. De esta manera, se busca lograr el objetivo de un buen balance de cargas, que terminen de calcular todos los dispositivos al mismo tiempo.

Para hacer el balanceo dinámico, se debe determinar de antemano la *performance* de cada dispositivo con el que contemos. Para esto, se usa la tradicional técnica de definir un caso de prueba representativo del problema, ejecutarlo en cada uno de los dispositivos y anotar cuánto tarda en cada uno de ellos para determinar el poder de cómputo. Una gran ventaja de esto es que permite balancear fácilmente las cargas cuando se realizan operaciones de doble precisión en conjuntos de dispositivos que incluyan placas Tesla y placas GeForce. En simple precisión, los topes de línea de cada una pueden tener *performance* similares (por ejemplo, GTX 580 y M2090, GTX 780 y K20), pero en doble precisión usualmente las Tesla suelen ser entre 2 y 6 veces más veloces. Como las cuentas se realizan o todas en precisión simple, o todas en precisión doble, es sumamente importante particionar el sistema tomando esto en cuenta.

El caso de prueba elegido corresponde con realizar 10 iteraciones del kernel de cómputo de la matriz de Kohn-Sham. Se considera que realizar este cómputo da una evaluación empírica de los recursos necesarios para resolver este problema. Este kernel utiliza muchos de los recursos del dispositivo: realiza gran cantidad de accesos a memoria a través de la unidad de textura, usa principalmente instrucciones `FMA`, tiene una máxima ocupación de los SM y tiene un uso total de la memoria compartida. No se contabilizaron las copias de memoria desde y hacia la placa de los parámetros puesto que, al igual que en el código de la simulación, casi la totalidad de los datos se construyen directamente en la placa y se recuperan las matrices reducidas solamente al final de la operación.

Teniendo las mediciones de tiempo para los d dispositivos disponibles, se construye la matriz de corrección de tiempos de ejecución con los d^2 coeficientes de corrección para poder estimar el tiempo de ejecución en el nuevo dispositivo. Si los dispositivos presentes son idénticos, como suele ser el caso en clusters de cómputo científico, entonces la matriz va a tener, como resulta esperable, solamente valores muy cercanos a 1.

El pseudocódigo del algoritmo de balanceo se detalla a continuación.

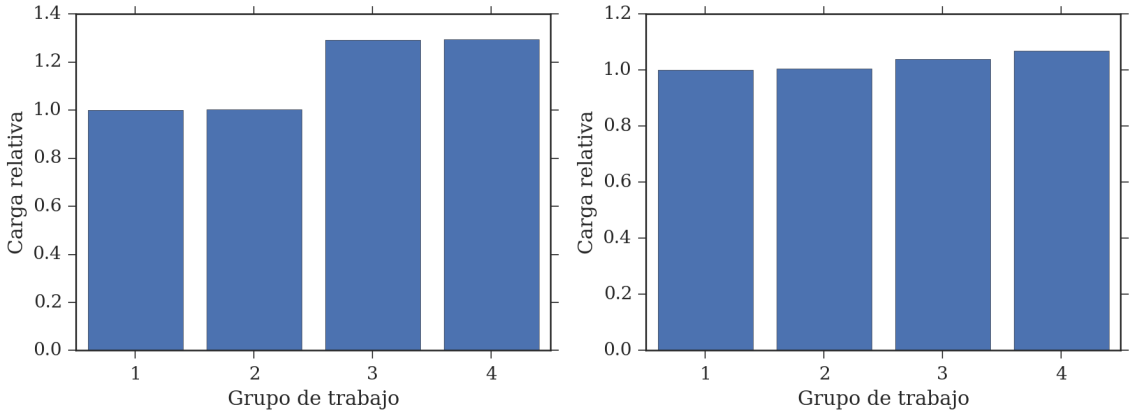
Algoritmo 1: Balance de carga para los distintos *threads*.

$tiempo_{\text{mín}} \leftarrow tiempo[Thread_{\text{mín}}]$
$tiempo_{\text{máx}} \leftarrow tiempo[Thread_{\text{máx}}]$
mientras $tiempo_{\text{máx}}/tiempo_{\text{mín}} > k$ **hacer**
  $\Delta T \leftarrow (tiempo_{\text{máx}} - tiempo_{\text{mín}})/2$
  para cada $G_i \in Trabajos[T_{\text{máx}}]$ **hacer**
    $T_{candidato} \leftarrow tiempo[G_i] * correccion[Thread_{\text{máx}}][Thread_{\text{mín}}] + costo_{migracion}$
    si $|T_{candidato} - \Delta T| < T_{mejor\_candidato}$ **entonces**
      $G_{mejor\_candidato} \leftarrow G_i$
      $T_{mejor\_candidato} \leftarrow T_{candidato}$
    fin
  fin
  Liberar la memoria de las matrices cacheadas de $G_{mejor\_candidato}$
  Mover $G_{mejor\_candidato}$ de $T_{\text{máx}}$ a $T_{\text{mín}}$
  Actualizar la duración total de $T_{\text{máx}}$ y $T_{\text{mín}}$, aplicando la corrección de tiempos
fin

En el algoritmo 1, se definen dos constantes adicionales, k y $costo_{migracion}$. k representa el coeficiente de máxima diferencia de tiempo entre el *thread* de mayor y el de menor duración. Para nuestros casos, una k tal que la diferencia entre *threads* sea menor al 2 %

es aceptable. $costo_{migracion}$ es un coeficiente que agrega un costo al migrar el grupo a otra placa, ya que va a tener que recalcular las matrices de funciones que ya se guardaban en memoria. Cuando el trabajo migre de placa, va a tener que desalojarlas ya que, si bien es posible que entre las placas se accedan mutuamente a memoria global (utilizando el direccionamiento unificado introducido en CUDA 5[13]) no conviene dada la latencia introducida, que se adicionará a lo largo de todas las iteraciones de la resolución. Este coeficiente es equivalente a definir una penalidad por romper la *afinidad* de los cores en CPU.

Otra ventaja de usar este algoritmo, adicionalmente a la partición estática, es que, si bien se considera la matriz de corrección, la finalidad es balancear la carga minimizando los tiempos, por lo que incluso si la estimación original de *performance* relativa era errónea, igualmente se redistribuirán los distintos grupos.



(a) Previo al balance dinámico. *(b)* Posterior al balance dinámico.

Fig. 3.13: Comparación de los tiempos de ejecución para las distintas cargas asignadas a cada placa en el caso del grupo hemo.

El rendimiento de esta optimización se puede apreciar en la figura 3.13. El predictor estático usado genera una partición entre las cuatro placas con aproximadamente un 25 % de desbalance. En solamente dos iteraciones del algoritmo 1, se rebalancean las cargas entre placas moviendo tres trabajos, dos de la placa 4 a la 2 y uno de la placa 3 a la 1. Con eso, las cargas quedan balanceadas a menos del 2 % de diferencia. Este gráfico también permite visualizar la magnitud del error de estimación usando el predictor estático definido anteriormente.

Performance

Resolver un sistema, en principio, debería ser inherentemente paralelo: resolver para un grupo no afecta resolver para otro. Sin embargo, escalar este problema entre múltiples placas hace surgir nuevos problemas que no estaban presentes previamente, como la concurrencia de escrituras sobre el bus de memoria. Como cada dispositivo va a tener que copiarse un fragmento de la matriz global de densidad y acumular en la matriz de Kohn-Sham de salida, va a existir una contención entre los *threads* que manejan cada GPU para leer y escribir en la memoria principal. Esto ocasiona que el *speedup* entre múltiples placas no sea lineal; al haber una cantidad limitada de trabajo que hacer en GPU y un fragmento de resolución que necesariamente tiene que leer y escribir en memoria, resulta

en una situación que no puede evitarse. Se puede ver en la figura 3.14 que la aceleración del código ejecutado en precisión simple no escala linealmente con respecto a la cantidad de dispositivos. Los cuellos de botella se notan principalmente a partir de la tercera placa, haciendo casi inútil el uso de una cuarta.

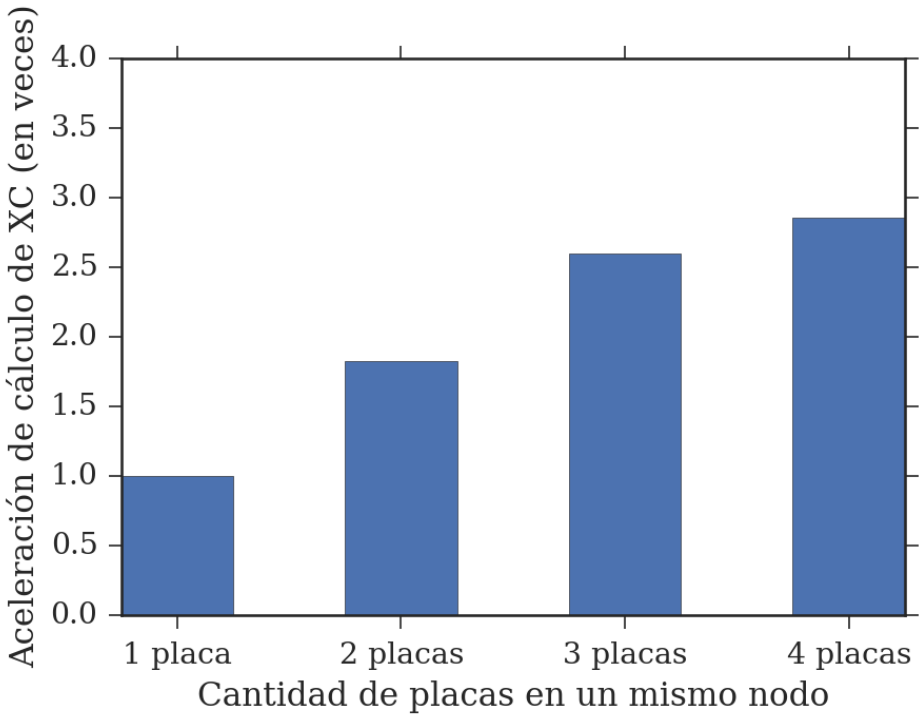


Fig. 3.14: Speedup en veces de correr el grupo hemo en 4 placas Fermi M2090 iguales en precisión simple.

Además de realizar los cálculos en precisión simple, la implementación soporta el uso de doble precisión. Esto minimiza el error numérico de las operaciones a costa de una menor *performance*. El cómputo utilizando precisión doble tiene, sin embargo, ventajas en la escalabilidad entre placas. Como los kernels en doble precisión son *compute-bound*, las copias de memoria son poco costosas en relación al tiempo de resolución de los grupos (el tamaño de las copias se duplica pero la potencia de cálculo se divide por cuatro en Fermi y Kepler). Esto disminuye la presión por el acceso a la memoria principal. Se puede observar en la figura 3.15 una clara mejora en los tiempos de cálculo de los sistemas en doble precisión usando esta técnica de paralelismo. El uso de la versión que emplea unicamente precisión doble es útil para corroborar la exactitud de las cuentas pero no es indispensable; el método iterativo en simple precisión a lo sumo necesitará iteraciones adicionales (menos del 10 % de iteraciones extra en los casos ensayados). Estas iteraciones adicionales no representan tiempo suficiente adicional para que sea meritorio hacerlas asi. Tampoco la calidad numérica adicional es necesaria, al ya estar al borde de la precisión del método DFT usando precisión simple.

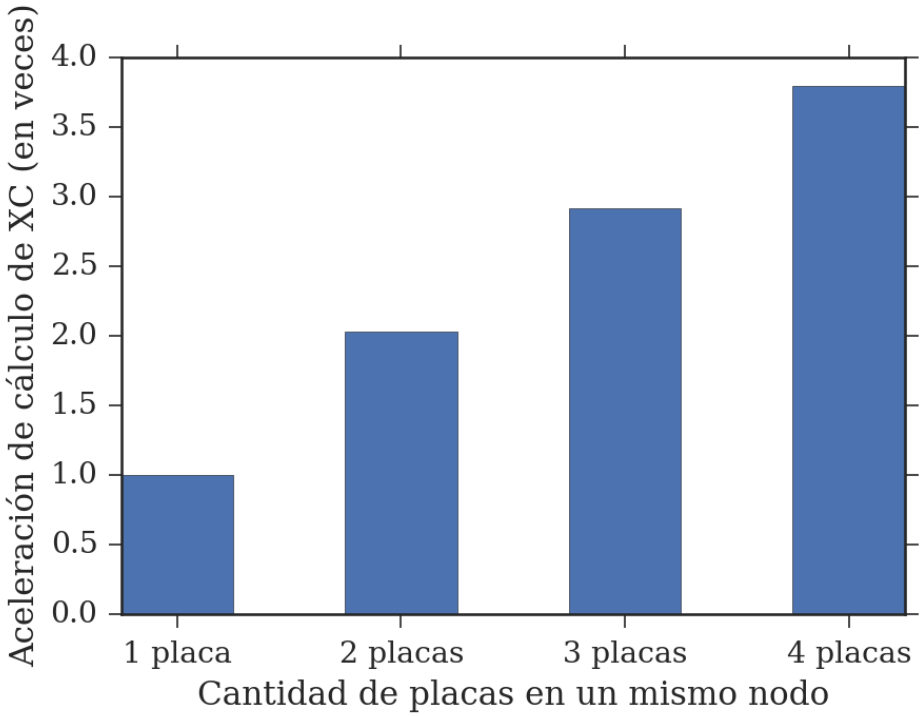


Fig. 3.15: Speedup en veces de correr el grupo hemo en 4 placas Fermi M2090 iguales en precisión doble.

3.3. Implementación en CPU

Así como ya existía una base de código original de CUDA, el trabajo sobre CPU también se realizó sobre el código original existente. El mismo fue diseñado para un uniprocesador, teniendo en mente un conjunto de instrucciones `SIMD` anterior a `AVX`, donde el tamaño de los registros de procesador era de 128 bits.

Con el propósito de adaptar el código para procesadores paralelos y vectoriales, como lo son los de la gama Xeon y Xeon Phi de Intel, se buscó vectorizar y paralelizar el código tanto como fuese posible. En particular, se priorizó lograr una gran escalabilidad en número de procesadores, especialmente considerando la arquitectura del Xeon Phi.

De acuerdo a la bibliografía [35], es necesario lograr que el código no solamente este bien vectorizado sino que, además, escale con la cantidad de procesadores para poder hacer el mejor uso de los recursos del coprocesador Xeon Phi. Por lo tanto, las pruebas iniciales se concentraron en lograr buena escalabilidad y vectorización en CPU únicamente.

Puesto que muchas decisiones arquitectónicas del código se realizaron en base a experimentos con prototipos representativos de las diversas operaciones, incluiremos algunos detalles del dispositivo utilizado para las pruebas.

La computadora utilizada para las pruebas fue un servidor *dual-socket* con 2 Intel Xeon CPU E5-2620 v2, con 6 cores cada uno. Los procesadores corren a una frecuencia de clock de 2,10 GHz, y soportan el set de instrucciones `x86-64` con `AVX1`. Cada procesador cuenta con 64 KB de cache L1, 2 MB de cache L2 para cada par de cores, y 15 MB de cache L3 compartida. El mismo contaba con 32 GB de memoria RAM DDR3 1333 a cuatro canales de memoria, dando una transferencia teórica máxima de 42,6 GB/s [42].

La familia de procesadores Xeon cuenta con tecnología *Turbo Boost*. La misma incluye una funcionalidad en el chip que ajusta dinámicamente la frecuencia de los procesadores de acuerdo a la temperatura y potencia empleadas. Esto, si bien *a priori* es deseable al mejorar la *performance* de los programas, dificulta el análisis de escalabilidad ya que al utilizar más procesadores, aumenta el consumo energético y la temperatura y, por tanto, esta tecnología produce una disminución en la frecuencia de los procesadores. Para evitar este efecto en nuestro análisis, se deshabilitó *Turbo Boost* al realizar las pruebas de escalabilidad. Los resultados finales con *Turbo Boost* están incluidas más adelante.

Adicionalmente, los procesadores Intel Xeon cuentan con *Hyper-Threading*, permitiendo dos procesadores lógicos por cada uno físico. En nuestra plataforma de prueba, esto produce que el sistema reporte un total de 24 procesadores en el sistema. Sin embargo, los dos hilos de ejecución (*hyperthreads*) en un mismo procesador comparten unidades básicas como las ALU. Al no ser totalmente independientes, esto también dificulta el análisis. Para no tomar esto en cuenta se trabajó con *Hyper-Threading* deshabilitado.

La sección de código trabajada corresponde a la parte del procesamiento de LIO optimizada para CUDA. Esta parte del código estaba ya implementada en C++, utilizando bibliotecas del suite de herramientas de Intel para vectorización correspondientes a la última versión disponible al momento de realizar la tesis, Intel C++ Compiler 2013.

3.3.1. Caso de estudio

Para las pruebas de esta sección se utilizó el grupo hemo, el cual es un caso muy utilizado dentro de la literatura. Detalles sobre este sistema y de los otros utilizados se encuentran en el apéndice B de este trabajo.

El tamaño de los cubos utilizado es 3 *a.u.* (unidades atómicas), el valor más apropiado al momento de iniciar el trabajo sobre la implementación para CPU. El mallado en cubos y esferas se introdujo en la sección 3.1 (página 31).

El análisis realizado corresponde a la implementación con precisión simple, aunque aplica también para precisión doble (no se necesita más que recompilar el código para cambiar la precisión empleada).

3.3.2. Estructura original del código

Un esquema de alto nivel del cómputo más intensivo realizado por el módulo XC se encuentra en la el psedocódigo 2. Este código corresponde a una iteración del cálculo de la matriz de Kohn-Sham y se compone de tres partes:

I) cálculo de las matrices relacionadas con las funciones (*funcs*),

II) calcular las densidad electrónica, su gradiente y Hessianos para cada punto (*dens, grad, hess*),

III) utilizarlos para obtener la energía de XC, la contribución a la matriz de Kohn-Sham (*KS*)

Cuando se alcanza el nivel de convergencia seleccionado, se agrega la contribución a la matriz de fuerzas.

Algoritmo 2: Pseudocódigo de la iteración original de LIO

```
solve(PG : PointGroup)
    Leer la matriz de coeficientes inicial.
    Calcular funciones, su gradiente y Hessiano, funcs.
    para cada p ∈ puntos(PG) hacer
        para cada i ∈ functions(PG, p) hacer
            Calcular la contribución a la densidad, dens
            Calcular la contribución al gradiente, grad
            Calcular la contribución a los Hessianos, hess
        fin
        Calcular potencial usando dens y grad.
        Calcular contribución FR_p a KS y Fuerzas usando dens, grad y hess.
        Calcular energía resultado energy con potencial y el peso del punto p.
    fin
    Calcular fuerzas usando los factores FR y los núcleos de los átomos.
    Sumar la contribución de cada punto a la matriz de Kohn-Sham general, KS.
```

Un aspecto importante a señalar es la representación de las matrices en el programa. Las matrices del gradiente y Hessiano tienen como elementos vectores de $\mathbb{R}^3$, y la matriz de valores de funciones es una matriz de escalares. Las operaciones entre vectores y vectores, y entre vectores y escalares tienen la semántica esperable de álgebra lineal. El producto entre dos vectores en el código debe ser interpretado como producto componente a componente, también conocido como *producto Hadamard*, no como producto escalar o vectorial.

La implementación de las operaciones entre vectores merece particular atención. Para aprovechar el set de instrucciones `SSE4`, la última versión disponible al momento de comenzar la implementación, el tipo de datos para un vector de tres componentes (`cvector3`) se adaptó para corresponderse a un registro de `SSE4`. Dado que el ancho de estos registros es de 128 bits, éstos permiten realizar cálculos de a cuatro elementos de punto flotante a la vez. Al ser tres, uno de los elementos del registro `SSE` no se utiliza (fijándole el valor en 0) [3]. Es tarea del compilador, luego, traducir las operaciones usuales a instrucciones vectoriales.

La representación de un vector $\mathbb{R}^3$ de esta manera, si bien da un *speedup* significativo, no es portable ni escalable a otras extensiones SIMD. Al incrementar el ancho de registro un número mayor de los campos deben ser ignorados, malgastando cada vez más espacio útil. Asimismo, se desperdician oportunidades por parte del compilador para optimizar mejor haciendo uso de todos los registros y operaciones que dispone la arquitectura sobre la cual se está compilando, ya que se le indica explícitamente qué operaciones tiene que hacer y cómo.

Los resultados de realizar una ejecución de *profiling* de una iteración de la implementación original se encuentra en la figura 3.16. Como puede verse, el principal dominante es la resolución de la densidad electrónica, que incluye los dos primeros ciclos del pseudocódigo de la figura 2. También se ve la importancia de la actualización de la matriz de Kohn-Sham y el cálculo de las matrices de funciones gradiente y Hessiano, que no son despreciables pero son pequeñas en comparación. Omitimos el cálculo de fuerzas pues éste se realiza solamente en la última iteración, una vez lograda la convergencia del método.

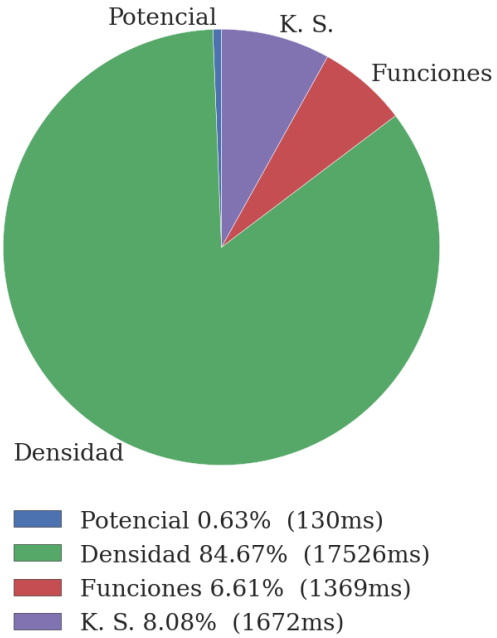


Fig. 3.16: División de los tiempos de iteración para el grupo hemo en la versión original en CPU

3.3.3. Cambios en la vectorización

Dado el peso de los cómputos internos para la densidad electrónica y la energía de XC, se comenzó por eliminar la dependencia explícita de SIMD, eliminando la herencia de esta clase con los tipos de datos de `SSE`.

Esto delegó al compilador la tarea de vectorizar el código apropiadamente. Se produjo, al principio, una degradación importante de *performance*, dado que las operaciones de suma, multiplicación, etc. involucrando arreglos ya no eran vectorizadas por defecto y éstas representan una parte muy significativa del algoritmo de cálculo de la energía.

Esta situación se resolvió con una reorganización en memoria de las matrices. La observación clave para esto se basa en que las operaciones se realizan de a un componente por vez. En particular, el ciclo más intensivo de cómputo (el interior de la iteración, detallado en el pseudocódigo 3) puede dividirse en cada componente para convertirse en diez operaciones de suma sobre un producto de dos vectores.

Algoritmo 3: Pseudocódigo del ciclo principal del cálculo de la energía de XC

`solve(`$PG : PointGroup$`)`
- **para cada** $p \in puntos(PG)$ **hacer**
  - **para cada** $i \in functions(PG, p)$ **hacer**
    - $dens \leftarrow \Sigma_{j=0}^{i} MC_{i,j} \cdot F_{p,j}$
    - $grad \leftarrow \Sigma_{j=0}^{i} MC_{i,j} \cdot F_{p,j}$
    - $hess_1 \leftarrow \Sigma_{j=0}^{i} MC_{i,j} \cdot H_{2p,j}$
    - $hess_2 \leftarrow \Sigma_{j=0}^{i} MC_{i,j} \cdot H_{2p+1,j}$
  - **fin**
  - Calcular la densidad en el punto usando $dens$,$grad$,$hess_1$ y $hess_2$
- **fin**

Una heurística de optimización para estos casos consiste en convertir los arreglos de estructuras (como por ejemplo las matrices de gradiente G y de Hessiano H, que son matrices densas de estructuras de tres valores) en estructuras de arreglos. La diferencia entre estos dos esquemas (conocidos como AOS, *array of structures*, y SOA, *structure of arrays* respectivamente) puede verse en la figura 3.17

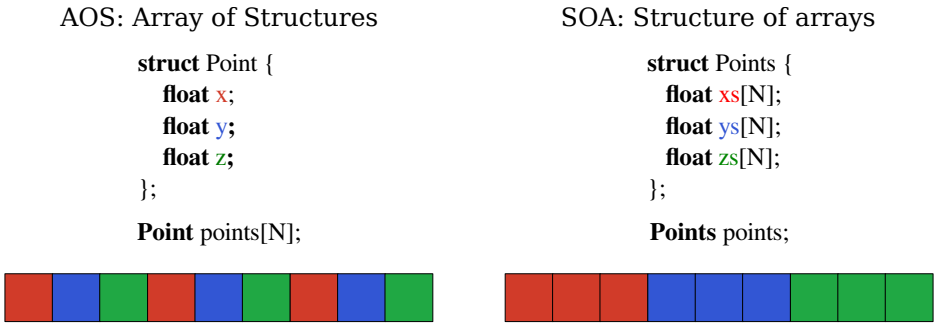


Fig. 3.17: Diferencias entre los esquemas AOS y SOA de organización de matrices, a nivel de código y resultado en memoria.

En este caso, esto implica partir no solo las matrices en tres matrices (una para cada componente), sino también en dividir las seis derivadas segundas (el Hessiano) en matrices separadas. Una descripción de cómo se empaquetan se puede ver en la sección 3.2.7 (página 43), ya que CUDA y CPU compartían la misma manera de empaquetar las derivadas. De esta manera, cada componente es un valor escalar y los ciclos pueden reescribirse en iteraciones independientes que utilizan las operaciones de punto flotante usuales sobre cada elemento.

Un esquema sencillo para visualizar la proyección realizada se muestra en la figura 3.18. Dado que todos los elementos de una misma componente están consecutivos en memoria, pueden ser leídos y procesados de a tantos a la vez como se disponga mediante registros SIMD, en vez de a grupos de a tres como se hacía anteriormente. Además, el cómputo puede aprovechar operaciones como por ejemplo `FMA` para aumentar aún más la cantidad de operaciones matemáticas simultáneas realizables por ciclo de reloj.

En el caso del Hessiano de las funciones, tiene seis componentes en vez de tres, resultando entonces en seis matrices separadas pero que funcionan de manera análoga a la matriz del gradiente.

Debido a la complejidad de los cálculos y tratando de evitar introducir errores sutiles, en un primer momento se decidió que, al calcular las matrices de funciones, se hiciera utilizando el código original pero con las versiones de vectores sin SIMD y, luego, proyectar las componentes a matrices en un último paso. Esto introduce un *overhead* en los cómputos de funciones no despreciable, pero veremos que se puede mitigar con el uso de técnicas adicionales, y en los casos donde resultaba importante se realizó una adaptación del código de funciones también.

La figura 3.19 muestra los resultados de realizar esta optimización en el caso de prueba del grupo hemo en la plataforma de referencia. Como puede verse, la optimización es de aproximadamente un 43 % por sobre la implementación original, incluso considerando que incrementa el tiempo de cálculo de funciones (que todavía se realiza en todas las iteraciones) aún así logra una aceleración significativa sobre la versión original.

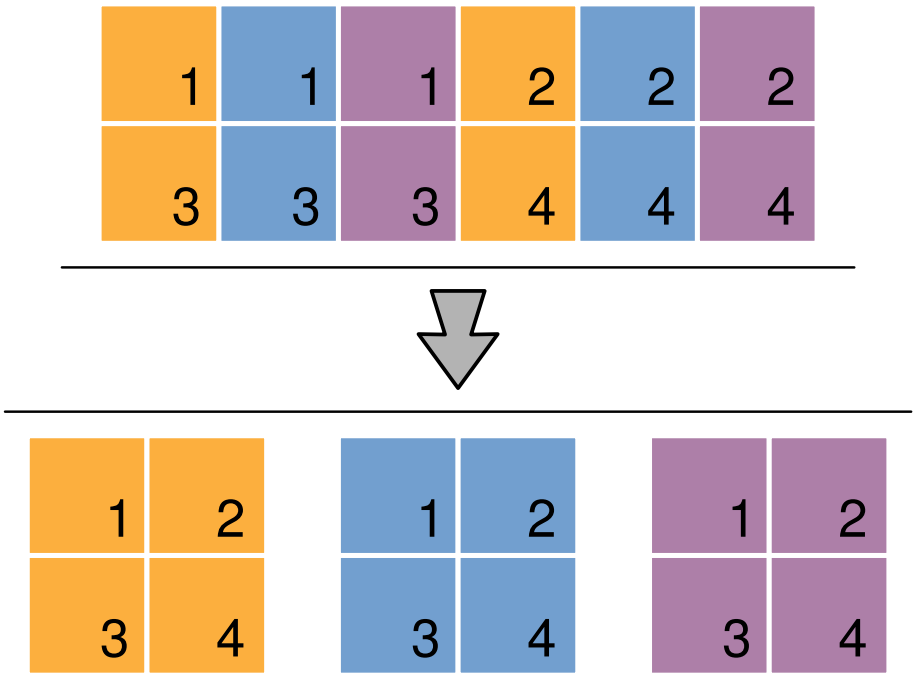


Fig. 3.18: Esquema de la proyección por componente realizada. En la parte superior se muestra la matriz original (en *row major order*). En la inferior se muestran las tres matrices proyectadas.

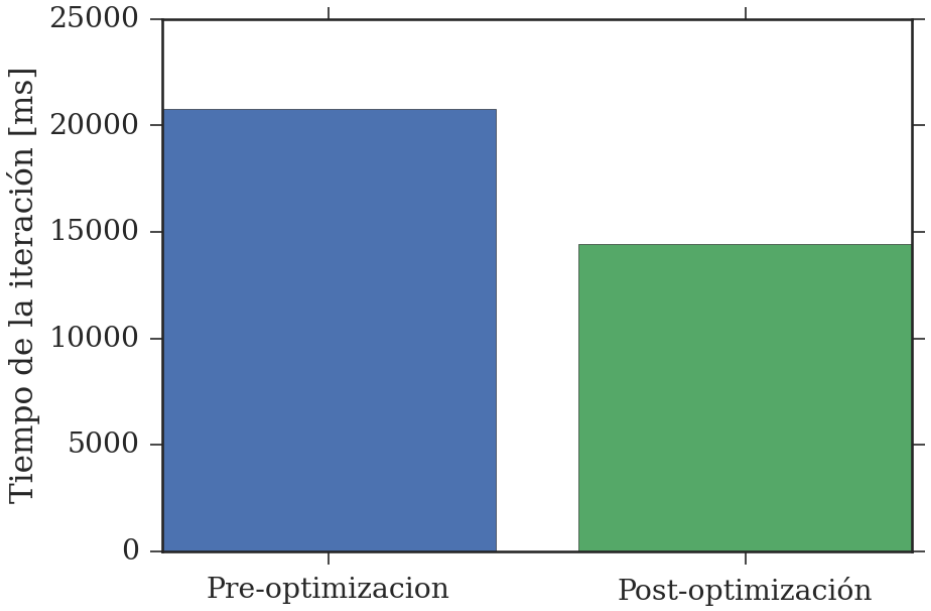


Fig. 3.19: Tiempo en milisegundos para la iteración de XC para el caso de la molécula del grupo hemo, antes y después de proyectar las componentes de las matrices.

3.3.4. Almacenamiento de las matrices

Una mejora considerable surge también de utilizar una estrategia de *caching* similar a la utilizada en el código de GPU para evitar el re-cálculo en cada iteración. A diferencia de GPU, la memoria principal en CPU es de fácil y amplia disponibilidad para procesadores estándar. Esto hace posible calcular, para cada grupo de puntos, las matrices de funciones una única vez antes de empezar las iteraciones. Con esto, el *overhead* introducido por calcular las matrices y luego proyectarlas disminuye considerablemente.

De manera de poder controlar esto, en casos de computadoras que no cuenten con suficiente memoria, se implementó una modificación para almacenar las matrices mientras haya memoria disponible y recalcularlas cuando no la haya.

Esta modificación es sencilla ya que podemos utilizar el mismo *flag* de GPU para marcar que las funciones de un grupo se encuentran ya calculadas y almacenadas en memoria principal. El impacto en el ciclo de iteración es mínimo y solo consiste en remover el cálculo de las funciones y devolver los punteros a las matrices almacenadas.

La figura 3.20 muestra la diferencia entre el programa optimizado en la sección anterior y el actual después de esta modificación. La diferencia es de aproximadamente 27 %.

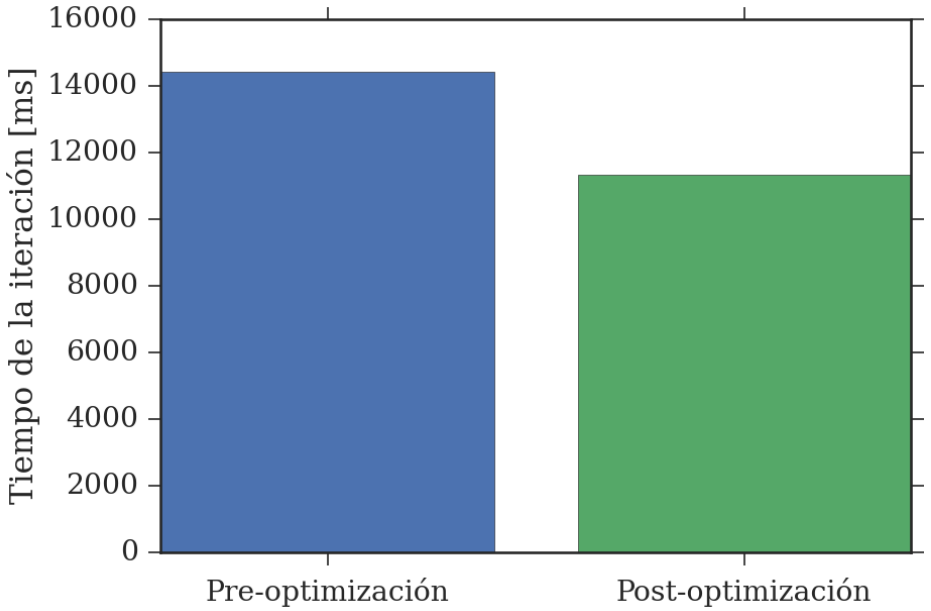


Fig. 3.20: Tiempo en milisegundos para la iteración de XC para el caso del grupo hemo, antes y después de cachear las matrices.

3.3.5. Prototipos de paralelización

Con la optimización anterior, observamos que los ciclos principales del código estaban vectorizados y buscamos, entonces, obtener mejor *performance* haciendo uso de los múltiples procesadores disponibles. Para esto, buscamos una solución que nos permitiera escalar lo mejor posible en cantidad de procesadores: Utilizar n procesadores debería mejorar la *performance* lo más cerca de n-veces posible, ya que sino agregar más procesadores al problema dará cada vez menos dividendos tal cual lo indica ley de Amdahl [10].

Con el propósito de simplificar la tarea de paralelizar el programa, se volvió a hacer uso de las facilidades de OpenMP de ICC, de manera similar a lo realizado en CUDA. OpenMP provee el soporte de asignar código a hilos de ejecución. El próximo paso es determinar una división de trabajo. En un primer momento se plantearon dos posibles rutas hacia paralelizar el código: dividir los grupos entre *threads* de procesamiento o utilizar múltiples procesadores para dividir el trabajo correspondiente a los puntos y funciones dentro de un mismo grupo. Esta segunda posibilidad corresponde a una estrategia similar a la estrategia implementada para GPU.

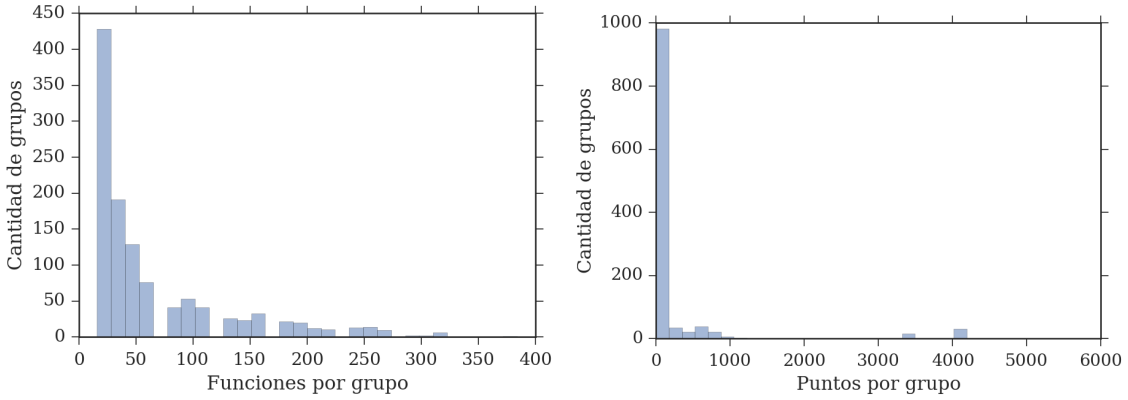
La decisión no es sencilla debido a las características del trabajo a realizar y la arquitectura de los procesadores Xeon y Xeon Phi. A diferencia de las GPUs, éstos no cuentan con una gran cantidad de procesadores y exhiben un costo alto para lanzar un hilo de ejecución. Los hilos de ejecución comparten memorias cache y principal. Si la cantidad de hilos es mayor que la cantidad de procesadores disponibles, deben ser asignados rotativamente a procesadores por parte del *scheduler* del sistema operativo. El *overhead* de decidir qué hilos de ejecución corren en cada momento puede afectar seriamente la *performance*.

Esto hace inviable, al menos en una primer instancia, realizar una partición idéntica a la de los *kernels* implementados en CUDA cuando resuelve para grupos chicos de puntos: no es claro si el costo de orquestar los procesadores para realizar los cómputos es comparable al trabajo mismo a realizar.

Por otro lado, tampoco es trivial partir los grupos entre los procesadores disponibles ni dividir los puntos de un grupo entre procesadores para resolverlos. El motivo de esto es la forma que tiene la grilla de integración con la que se trabaja.

La grilla, como ya se detalló anteriormente, se divide en cubos y esferas. Las esferas corresponden a los núcleos de los átomos del sistema, mientras que los cubos son determinados por la grilla de integración y tienen tamaño fijo. En líneas generales, las esferas suelen tener muchos puntos para procesar, mientras los cubos suelen tener una pequeña o mediana cantidad de puntos.

Tomamos como ejemplo el caso del grupo hemo, el empleado para ajustar los parámetros de nuestra implementación. Un histograma de la cantidad de funciones por grupo, y la cantidad de puntos por grupo, puede verse en la figura 3.21.



(a) Cantidad de funciones. *(b)* Cantidad de puntos.

Fig. 3.21: Histogramas para la cantidad de puntos y funciones para los grupos correspondientes al caso de prueba del grupo hemo.

Como puede verse en el sistema de prueba, la cantidad de grupos y de funciones para

un grupo es altamente variable. Los grupos más pesados, como las esferas, tienen una cantidad del orden de los miles de puntos y cientos de funciones, mientras que hay 748 grupos (65 % del total) con menos de 50 funciones a calcular por punto, y 927 (80 % del total) que poseen menos de 100 puntos. Adicionalmente, el grupo más grande tiene 5238 veces más puntos y 24 veces más funciones a calcular que el más chico.

Por lo tanto, se identificaron las siguientes dificultades:

- Dividir los grupos entre hilos de ejecución tiene la dificultad de que los grupos más grandes dominan a los más chicos, pudiendo producir un serio desbalance de carga de trabajo, especialmente cuando se incrementa la cantidad de procesadores en juego.

- Recorrer cada grupo y dividir los puntos a resolver entre procesadores tiene como desventaja que muchos grupos tienen una muy pequeña cantidad de puntos, con lo cual incrementar la cantidad de procesadores no sería eficiente, porque no habría trabajo para asignar a los procesadores adicionales. Este desbalance también es indeseable.

  Además, como ya señalamos, el costo de sincronización de hilos de ejecución al terminar un ciclo paralelizado con OpenMP no es menor, por lo tanto si la cantidad de trabajo para cada *thread* no resulta suficiente para compensar este *overhead*, no vale la pena incrementar la cantidad de *threads*.

La solución propuesta a este problema consiste en un híbrido entre estas dos estrategias: Los grupos que son demasiado chicos en cantidad de trabajo son agrupados y divididos entre los procesadores disponibles y los que sí sean lo suficientemente grandes son procesados secuencialmente, pero los procesadores se dividen el trabajo de los puntos a procesar para cada uno de los grupos grandes. Determinar si un grupo es demasiado "chico" o "grande"

3.3.6. Análisis de costo computacional de grupos

Siguiendo la estrategia planteada de paralelización, el trabajo de resolución de los grupos se distribuye entre los procesadores disponibles y se resuelven, entonces, en paralelo. Al ser necesario que todos los trabajos asignados estén finalizados para poder avanzar con la siguiente etapa de cálculo, el tiempo total de ejecución corresponde al tiempo máximo que alguno de los procesadores tome en resolver su parte.

Puesto que antes de empezar las iteraciones de resolución ya se saben cuáles son los grupos a procesar, se puede usar estar información para hacer una partición en cargas una vez antes de empezar a iterar. Para esto se necesita un estimativo del costo computacional de un grupo y un algoritmo que, dados los costos de los grupos y la cantidad de *threads* a utilizar, asigne cada grupo a cada *thread*.

Para tener una idea del costo de cada grupo, se usa el estimador de operaciones introducido en [3] junto con un ajuste constante para considerar *overheads* fijos a cada grupo. Matemáticamente el estimador utilizado es:

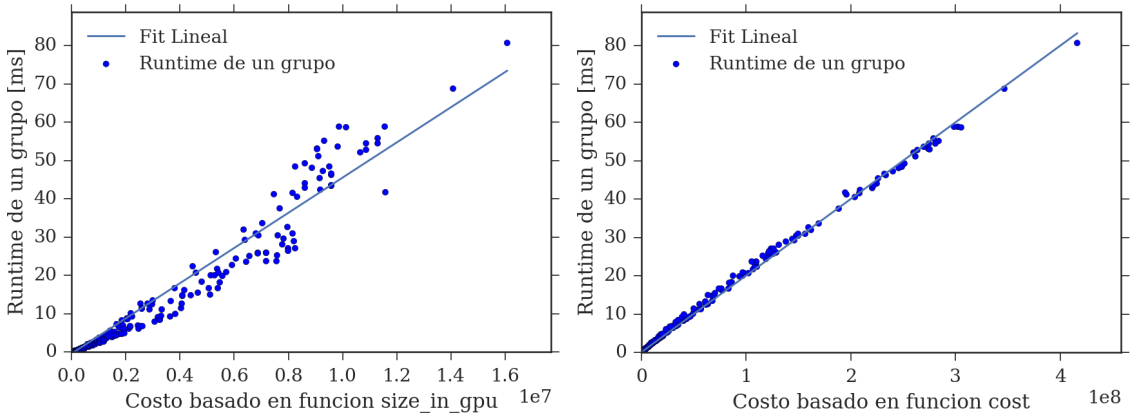
$$Costo(PG) = \frac{\#f(PG) \cdot \#p(PG) \cdot (\#p(PG) + 1)}{2} + C \tag{3.8}$$

donde $f(PG)$ son las funciones del grupo y $p(PG)$ los puntos del grupo.

La constante fue ajustada experimentalmente de acuerdo a los ejemplos disponibles, en este caso particular del grupo Hemo, y se decidió utilizar el valor $C = 250000$ para experimentos futuros. Si bien se utilizó este experimento como ajuste, veremos posteriormente el comportamiento del algoritmo en otros casos **??**.

Un interrogante planteado es acerca de la conveniencia en CPU utilizar el estimador de trabajo que ya se desarrolló para GPU. Sin embargo, debido a la diferencia entre ambas arquitecturas, los estimadores que dan buenos resultados en una no lo hacen en la otra.

En la figura 3.22 puede verse una comparación entre ambos predictores en pruebas realizadas para CPU. Cada gráfico muestra la calidad relativa del estimador correspondiente con respecto al tiempo de ejecución (en milisegundos) para los grupos examinados. Se puede ver que el tiempo de ejecución, a diferencia de lo que ocurre en la implementación en GPU, es mejor predicho por el estimador de costo que por el tamaño de cada grupo en memoria.



(a) Predictor con *size_in_gpu*. *(b)* Predictor con función de costo.

Fig. 3.22: Tiempos de ejecución de cada grupo considerado pequeño de acuerdo al costo del mismo según predictor, para el ejemplo del grupo hemo.

3.3.7. Algoritmo de particionado

Para dividir los trabajos de manera equitativa entre los procesadores disponibles, es necesario revisitar el problema de particionado. Esta vez, a diferencia de GPU, contamos con procesadores simétricos y tenemos tanto muchos trabajos como potencialmente muchos procesadores.

En este caso, podemos usar un nuevo algoritmo basado en otro problema similar conocido como *Bin packing* [41]. En este problema, se tienen contenedores de capacidad C y se desea saber cuantos de ellos hay que usar como mínimo para poder ubicar n trabajos cuyos pesos son $V_1, \ldots, V_n$.

La diferencia vemos es en el grado de libertad: En el problema de particionado, tenemos una cantidad limitada de contenedores (*threads*) pero podemos cargarlos tanto como queramos, mientras en *Bin Packing* tenemos una capacidad máxima pero tantos contenedores como necesitemos. Sin embargo, en ambos casos queremos minimizar este grado de libertad.

Si conociéramos, para el problema de dividir el trabajo entre hilos de ejecución, el valor C de carga máxima que uno de estos tiene que procesar, se podría usar un algoritmo

que resuelva *Bin packing* para obtener la mínima cantidad de hilos de ejecución que se necesitaría para distribuir el trabajo y que ninguno de ellos este cargado más que C. Si estos son menos que la cantidad máxima de procesadores que tenemos, sabemos que podemos hacer la distribución dentro de los recursos disponibles.

La analogía es que cada contenedor es un *thread* y cada objeto es un grupo o trabajo a resolver.

Queda entonces dar un algoritmo para determinar C.

Este algoritmo es fácil de obtener en base a las siguientes tres observaciones:

a) Es imposible que M sea más chica que el menor de los costos de los trabajos a procesar. Esto es trivialmente demostrable.

b) Dado que siempre que se disponga de al menos un procesador, M es como mucho el total de todos los trabajos. Esto también es trivialmente cierto.

c) Si tenemos una solución tal que cada *thread* está cargado hasta M unidades de trabajo, tenemos una solución para toda carga máxima $M', M' \geq M$. Análogamente, si no podemos encontrar una solución para M, es imposible que obtengamos una para $M', M' \leq M$.

Las dos primeras afirman que M está en un rango acotado posible de valores y lo que hay que hacer es buscarlo dentro de ese rango. Si bien esto ya es suficiente desde un punto de vista de correctitud (el algoritmo funciona), no es eficiente si el rango de búsqueda es grande. Al agregar el tercer ítem, se puede utilizar el algoritmo de búsqueda binaria.

El algoritmo de búsqueda binaria es uno de los más conocidos dentro de las ciencias de la computación: Dada una propiedad p y un rango de valores del dominio de p, $[L, R]$, se busca L' y R' tal que L' sea el último valor que cumpla la propiedad p y R' el primero que no la cumpla. Además, si p cumple las tres condiciones anteriores: L cumple la propiedad, R no, $\forall V \leq L', p(V)$, $\forall V \geq R', \neg p(V)$. Para encontrar L' y R' se puede ir probando el valor del medio del intervalo que tenemos: Si en este elemento vale p entonces se debe seguir buscando a la izquierda, y si no vale se debe seguir buscando a la derecha. Cuando el rango a buscar sea indivisible se ha entonces encontrado los dos valores L' y R' buscados.

De este modo, se utiliza búsqueda binaria para encontrar el M. En cada paso de la búsqueda, para un M candidato, se usa un algoritmo de *Bin packing* para obtener cuántos hilos de ejecución necesitaríamos para dividir el trabajo de manera que ningún procesador reciba más trabajo que M. Si esta cantidad es mayor que la cantidad de procesadores disponibles, entonces se necesita que al menos uno de los procesadores esté más cargado. Sino, se puede intentar cargarlo menos.

Un pseudocódigo para esta sección del algoritmo esta disponible en el algoritmo 4.

Un problema de esta estrategia, al igual que el problema de particionado, *Bin packing* es un problema *NP-hard*. Sin embargo, existen buenos algoritmos aproximados para resolver el problema. Éstos pueden devolver una respuesta que no sea óptima, pero razonablemente cerca de serlo.

Una estrategia posible de este estilo es la estrategia denominada *First fit decreasing*, que es la utilizada en este trabajo.

Esta estrategia consiste en, iterativamente, ubicar los objetos en orden de mayor a menor en costo en el primer contenedor en el que se pueda. De no disponerse ninguno, se utiliza uno nuevo y se repite el algoritmo. Se sabe que este algoritmo da una respuesta que como máximo es $\frac{11}{9}O + 1$, donde O es el óptimo para el problema a resolver [43].

La razón por la cual esta heurística fue seleccionada es que es simple de implementar, tiene una complejidad $O(n \log n)$ con n la cantidad de trabajos con lo cual es ligeramente peor que lineal, y se saben buenas cotas de error como vimos anteriormente. Adicionalmente, el uso de un algoritmo más costoso computacionalmente puede no valer la pena frente a los errores del predictor, pudiendo resolverse el desbalance dinámicamente con otros métodos.

Esta sección del algoritmo también puede verse en pseudocódigo en 4. La complejidad resultante es $O(n \log n + n \log M)$ con n la cantidad de elementos y M la suma de todos los costos. Dado que M y n son de tamaño razonablemente pequeño y el algoritmo solo debe ser ejecutado una única vez antes de la primer iteración, se determinó aceptable para utilizarlo en la implementación.

Algoritmo 4: Pseudocódigo del algoritmo para particionar trabajo entre *threads*.

```
partition(C = {C_1, ..., C_n}, m)
    sort(C)
    L = min(C) − 1, R = sum(C)
    mientras R − L > 1 hacer
        // Invariante: (L, ..., R] contiene la capacidad máxima.
        Capacity ← (L+R)/2
        Partition ← splitbins(C, Capacity)
        si #Partition ≤ m entonces
            R ← Capacity
        en otro caso
            L ← Capacity
        fin
    fin
    devolver splitbins(C, R)
splitbins(C = {C_1, ..., C_n}, m)
    Bins ← ∅
    para cada c ∈ C hacer
        Sea Fits todos los contenedores donde c entra
        Si Fits es vacío agregar un contenedor nuevo.
        Tomar el primer contenedor, Bin, de Fits.
        Agregar c a Bin
    fin
    devolver Bins
```

3.3.8. Cambios en paralelismo

Además de implementar la división de grupos entre los hilos de ejecución, todavía es necesario modificar la paralelización interna de la versión original de LIO con dos propósitos:

i) Acceso a matrices globales: esto es una causa de degradación en la *performance* porque requiere que los accesos entre *threads* a posiciones comunes sean coordinados.

ii) Ciclos internos: estos ciclos no estaban paralelizados. Mediante el uso de OpenMP se avanzó en la paralelización logrando que para los grupos grandes también se aproveche el multiprocesamiento.

Para resolver el primer punto, se utilizó una matriz de Kohn-Sham y una matriz de fuerzas distintas para cada hilo de ejecución, de manera que sus grupos asignados usaran estas matrices para hacer los almacenamientos temporales en lugar de acceder directamente a las globales.

Al finalizar la iteración, todas las contribuciones de estas matrices son acumuladas en la matriz global una a la vez. Dado que este paso solo se hace una vez por iteración, su tiempo de ejecución resulta despreciable en CPU.

Para el segundo punto, se realizó un análisis del código de la iteración para reestructurarlo de manera de que se adapte bien a la paralelización mediante OpenMP.

Como ya se vio anteriormente, se identificaron en el código tres cómputos clave:

1. El cálculo de la energía de XC, que también calcula valores usados por los demás.
2. El cálculo de fuerzas.
3. El cálculo de la matriz de Kohn-Sham.

El segundo de estos ciclos solo se realiza en la última iteración si se busca calcular las fuerzas, por lo que no fue el principal foco de nuestra optimización.

El primero de los ciclos, correspondiente al algoritmo 3, es paralelizable en base a que no hay dependencias entre las iteraciones más allá de que se debe calcular la energía total sumando las contribuciones de cada punto. Resolver esto es sencillo de realizar utilizando un *feature* de OpenMP denominada reducción que permite especificar que cada *thread* debe tener una copia local de la variable y al finalizar todos los hilos de ejecución, el resultado debe combinarse. Esto es posible solo si los valores a combinar pueden hacerse asociativamente. En este caso, al tratarse de sumas entre escalares, se pueden realizar trivialmente.

Por lo tanto, asignamos una cantidad de iteraciones a cada *thread*. Para asegurar que las mismas sean consecutivas y lo más parecidas posible usamos un *scheduler* estático.

Además de calcular la contribución a la energía, los valores de aporte a la matriz de fuerzas y de Kohn-Sham de cada punto se almacenan en arreglos separados para posterior uso.

El segundo ciclo, esquematizado en el esquema del algoritmo 5, no es tan sencillo de paralelizar. Dado que los resultados se almacenan sobre una misma matriz, una solución posible sería tener una matriz para cada *thread* y que cada una aporte sus resultados parciales. Si bien se eligió una estrategia similar para la paralelización externa, en este caso no resulta una opción atractiva por los siguientes motivos:

1. La cantidad de matrices crece linealmente con la cantidad de *threads* y, a diferencia del caso anterior en el cual la operación de reducción de todas las matrices a la global se realizaba una vez por iteración, en este caso debería realizarse para cada grupo en cada iteración. Esto perjudica las ventajas obtenidas por dividir las iteraciones entre los múltiples procesadores.
2. La operación de reducir las matrices a una matriz global es una operación fuertemente limitada por el acceso a memoria (*memory-bound*), con lo cual está limitada por la capacidad del ancho de bus de memoria y no por el cómputo a realizar.

El segundo punto enumerado anteriormente amerita un análisis especial. Para ello se diseño un *benchmark* con el objetivo de ilustrar la falta de escalabilidad de este problema

(la reducción de matrices) con respecto a la cantidad de procesadores del sistema, al menos en la arquitectura Xeon.

El cuerpo principal para el programa utilizado como *benchmark* puede verse en la figura 3.23. La manera en que se reducen las matrices es en forma de árbol invertido: primero se reducen todas las matrices consecutivas de a pares, luego solo las que contienen el resultado de la etapa anterior y así hasta llegar tener una única matriz con el resultado general. Un ejemplo de este procedimiento se encuentra en la figura 3.24.

```
for(int step = 1; step <= mats; step = step * 2) {
    #pragma omp parallel for schedule(static) num_threads(threads)
    for(int acum = 0; acum < mats - step; acum = acum + step * 2) {
        for(int i = 0; i < n; i++) {
            matrices[acum].sum(matrices[acum+step]);
        }
    }
}
```

Fig. 3.23: Ciclo principal de la implementación de un algoritmo paralelo de suma de matrices.

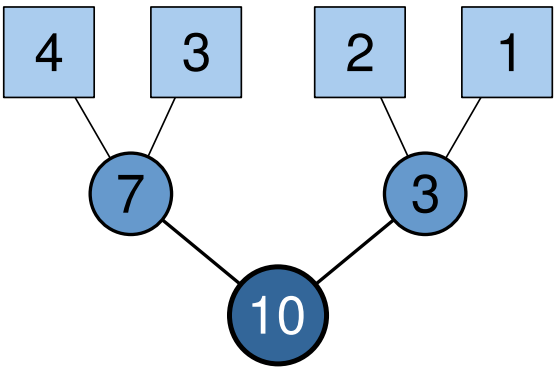


Fig. 3.24: Esquema de las reducciones realizadas en cada iteración del ciclo de la figura 3.23, esquematizadas usando enteros para claridad. Las iteraciones a cada nivel del árbol pueden hacerse en paralelo.

En teoría, este algoritmo debería producir un mejor tiempo de ejecución utilizando múltiples procesadores, pues los cálculos de una reducción en el mismo nivel del árbol pueden hacerse en paralelo.

Para evaluarlo, se midieron los tiempos de reducir 128 matrices de 1024×1024 elementos de punto flotante de precisión simple, variando la cantidad de hilos de ejecución a utilizar. Este caso es representativo de los tamaños y cantidad de matrices a reducir en LIO en base a los ejemplos estudiados.

Para compensar por primeras lecturas y otros factores, se tomó un promedio de los resultados después de realizar 20 mediciones sucesivas con la misma cantidad de *threads*.

Como puede verse en la figura 3.25, los resultados son desalentadores: el uso de multiprocesamiento no resulta útil para este problema, más allá del uso de dos procesadores.

Observando el problema en sí, se puede notar lo siguiente: Cada elemento de las matrices a sumar es accedido en memoria una única vez y el único cómputo realizado con

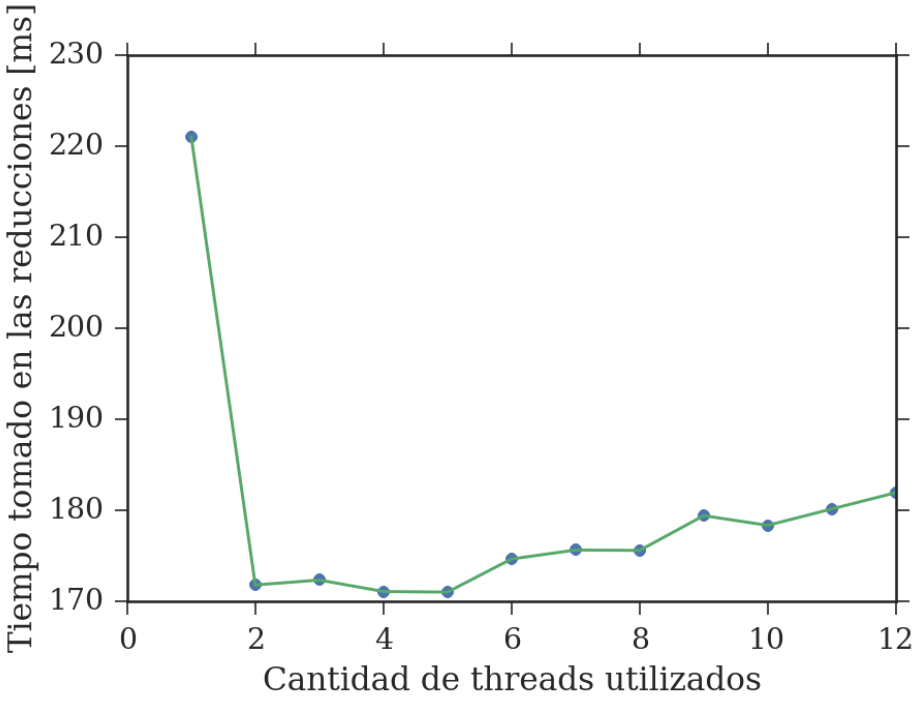


Fig. 3.25: Tiempo en segundos para sumar 128 matrices de 1024×1024 elementos, según la cantidad de *threads* usados. Los resultados corresponden a un promedio de 20 ejecuciones consecutivas para cada cantidad de *threads*.

este elemento es una suma con los demás. La proporción de lecturas de memoria por cada operación con los datos leídos es muy desfavorable.

Además, al no utilizarse los datos leídos múltiples veces, cada línea de datos tiene un *cache miss* compulsivo asociado (el de la línea donde reside) que no se ve compensado por más de un uso del dato mientras reside en cache.

Este tipo de problemas se denomina *memory bound*, al ser la velocidad del *bus* de memoria y la organización de la misma el factor limitante, no la capacidad de cómputo o el tamaño de los caches.

Esto implica que, pasada una cierta velocidad de reloj de procesador y una cantidad de núcleos, incrementar estos recursos no produce mejoras apreciables. Esto es lo que se puede ver en la figura 3.25. Es más, el *overhead* de introducir nuevos hilos de ejecución termina incrementando el tiempo de ejecución a partir de cinco *threads*.

Determinar que ese es el caso en la prueba de concepto realizada requiere herramientas de análisis más sofisticadas. Para corroborarlo, se utilizó la herramienta de código abierto *perf*. Este programa usa los contadores de *performance* del procesador, que permiten llevar la cuenta de eventos de procesador importantes como *cache-misses* o *stalls* de procesador, para dar estadísticas a nivel arquitectural de la aplicación y de muy fino detalle.

A continuación se muestran los resultados de ejecutar un análisis de *perf* para 1 y 12 hilos simultáneos en la máquina de prueba. Resulta de especial interés el valor de `stalled-cycles-frontend`, que corresponde al contador de *performance* `UOPS_ISSUED.ANY`. Este contador registra los eventos en los que el procesador está detenido, es decir, el *pipeline* de operaciones no avanza pues se encuentra a la espera de datos que lleguen de memoria

principal [44]. En este caso, parece ser que este factor es un limitante extremadamente fuerte.

```
$ perf stat -B ./benchmark 1024 128 1
1 0.180418953613844

 Performance counter stats for './benchmark 1024 128 1':

       2076,031257 task-clock                #    0,998 CPUs utilized
               225 context-switches          #    0,108 K/sec
                 0 cpu-migrations            #    0,000 K/sec
            67.583 page-faults               #    0,033 M/sec
     4.325.757.818 cycles                    #    2,084 GHz
     3.685.447.979 stalled-cycles-frontend   #   85,20% frontend cycles idle
   <not supported> stalled-cycles-backend
     1.801.562.845 instructions              #    0,42  insns per cycle
                                             #    2,05  stalled cycles per insn
       220.639.658 branches                  #  106,280 M/sec
           191.368 branch-misses             #    0,09% of all branches

       2,080571933 seconds time elapsed

$ perf stat -B ./benchmark 1024 128 12
12 0.182727314613294

 Performance counter stats for './benchmark 1024 128 12':

      22160,476291 task-clock                #   10,531 CPUs utilized
             2.402 context-switches          #    0,108 K/sec
                17 cpu-migrations            #    0,001 K/sec
            66.704 page-faults               #    0,003 M/sec
    46.310.011.626 cycles                    #    2,090 GHz
    42.797.073.792 stalled-cycles-frontend   #   92,41% frontend cycles idle
   <not supported> stalled-cycles-backend
     9.369.677.336 instructions              #    0,20  insns per cycle
                                             #    4,57  stalled cycles per insn
     2.657.329.767 branches                  #  119,913 M/sec
           687.016 branch-misses             #    0,03% of all branches

       2,104317579 seconds time elapsed
```

Esto lleva a concluir que la reducción impactaría muy negativamente en la escalabilidad de la implementación y que a medida que la cantidad de procesadores se incremente, esta sección del código se haría consecuentemente más pesada. Entonces, se buscó reorganizar este ciclo para evitar esta situación, es decir, dejar de tener una matriz separada para cada *thread*.

El ciclo a modificar está en la figura 5. Es posible dejar de tener que tener una matriz por *thread* simplemente mediante la inversión de los ciclos internos y externos en el algoritmo. Esto nos permite poder dividir los índices de la matriz global entre múltiples hilos

de ejecución.

Algoritmo 5: Cálculo original de la matriz de Kohn-Sham

$R \leftarrow 0_{m,n}$
$F \leftarrow funciones(PG)$
para cada $p \in puntos(PG)$ **hacer**
  para cada $i, j \in m \times n$ **hacer**
    $R_{i,j} \leftarrow R_{i,j} + F_{p,i} \cdot F_{p,j} \cdot factores_p$
  fin
fin
para cada $i, j \in indices\_kohn\_sham(PG)$ **hacer**
  $KS_{i,j} \leftarrow KS_{i,j} + R_{i,j}$
fin

La versión modificada puede verse en la figura 6. Lo que se hace para este caso es recorrer los índices a actualizar de la matriz total y para cada índice obtener la contribución de todos los puntos. Dado que cada índice debe ser actualizado por, a lo sumo, un solo hilo de ejecución, no hay necesidad de replicar matrices ni de reducirlas.

Algoritmo 6: Cálculo de la matriz de Kohn-Sham reestructurado para paralelismo

$R \leftarrow 0_{m,n}$
$F \leftarrow functions(PG)^T$
para cada $i, j \in indices$ **hacer**
  $KS_{i,j} \leftarrow KS_{i,j} + \Sigma_{p \in points(PG)} F_{p,i} \cdot F_{p,j} \cdot factors_p$
fin

Un problema provocado por la implementación es que se recorre la matriz de funciones por columnas y no por filas. Este orden es poco apropiado para los caches ya que cada fila de la matriz probablemente resida en líneas de cache diferentes. Esto puede disminuir apreciablemente la *performance* del algoritmo. Además, afecta negativamente la escalabilidad en procesadores, al incrementarse la cantidad de invalidaciones de cache y por lo tanto el *overhead* del algoritmo de coherencia de caches inter-procesadores.

La solución a este problema es, sin embargo, sencilla. Alcanza con trasponer la matriz de valores de funciones. La misma puede trasponerse una vez y almacenarse en memoria al iniciar las iteraciones, teniendo un relativo bajo costo de creación. Con esta modificación, el orden de iteración es por filas, el cual es *cache-friendly*.

Esta modificación del algoritmo introduce un nuevo aspecto a considerar, similar a las consideraciones hechas para la cantidad de puntos y funciones: una poca cantidad de índices a actualizar en un mismo grupo eclipsa el *overhead* que introduce el uso de OpenMP. La cantidad de índices de la matriz de Kohn-Sham a actualizar también tiene una distribución bastante dispar, como puede verse en la figura 3.26.

Esto enfatiza aún más la necesidad de utilizar una paralelización híbrida para grupos chicos y grandes, al contar ahora con un factor más que determina cuánto trabajo se le asigna a los *threads*.

Estos dos factores se tienen en cuenta en la función que determina si un grupo es considerado grande o no. Para considerarlo grande, el mismo debe tener una cantidad de puntos y una cantidad de índices en la matriz de Kohn-Sham mayor que un umbral, multiplicado por la cantidad de hilos de ejecución a realizar.

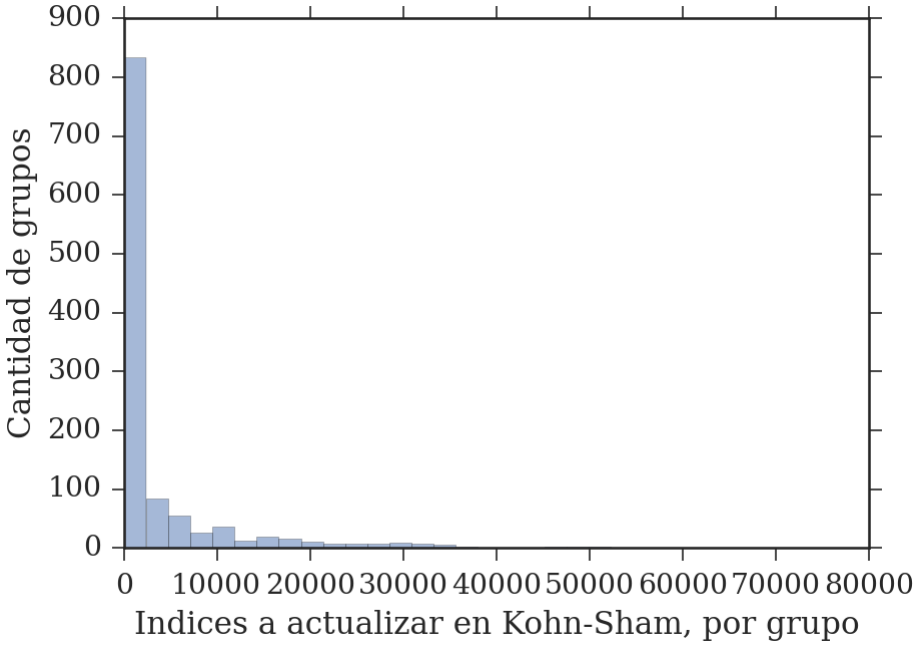


Fig. 3.26: Distribución de la cantidad de índices a actualizar de la matriz de Kohn-Sham para el grupo hemo.

3.3.9. Algoritmo de segregación

Ya paralelizados los ciclos internos y armada la partición, queda determinar cuándo un grupo es grande o chico. Como lo que se busca es evitar resolver un grupo con insuficiente cantidad de puntos y/o funciones en relación a la cantidad de núcleos del sistema, se usa un proceso de decisión sencillo: si el grupo tiene más puntos e índices a actualizar que un cierto umbral multiplicado por la cantidad de *threads* a utilizar, se lo considera grande, sino es chico.

Se busca que cada *thread* esté ocupado la mayor cantidad de tiempo posible: menos trabajo para cada procesador implica que los *overheads* de sincronización se hacen más notorios. Sabiendo que la cantidad de trabajo depende de la cantidad de puntos e índices de la matriz de Kohn-Sham, se busca que cada a *thread* le sea asignado una cantidad de suficiente de trabajo.

Para decidir el valor del umbral, se realizaron diversas pruebas utilizando el caso del grupo hemo. De los valores resultantes se ven algunos de los resultados en la figura 3.27. En base a estos resultados se decidió un valor de umbral de 80 para pruebas posteriores.

Si bien parece que la diferencia es de poca importancia, es importante señalar que, siendo que lo que se busca lograr es la mayor escalabilidad posible, diferencias pequeñas afectan considerablemente al *speedup* obtenido.

3.3.10. Algoritmo de balanceo

Por último, si bien el algoritmo de particionado y la función de costo son buenas para los casos estudiados, no son perfectas. Como puede verse en la figura 3.28a, hay una diferencia entre costo que resulta importante en algunos casos. Esto abre la posibilidad de obtener alguna mejora adicional en *performance* utilizando la misma técnica de balanceo

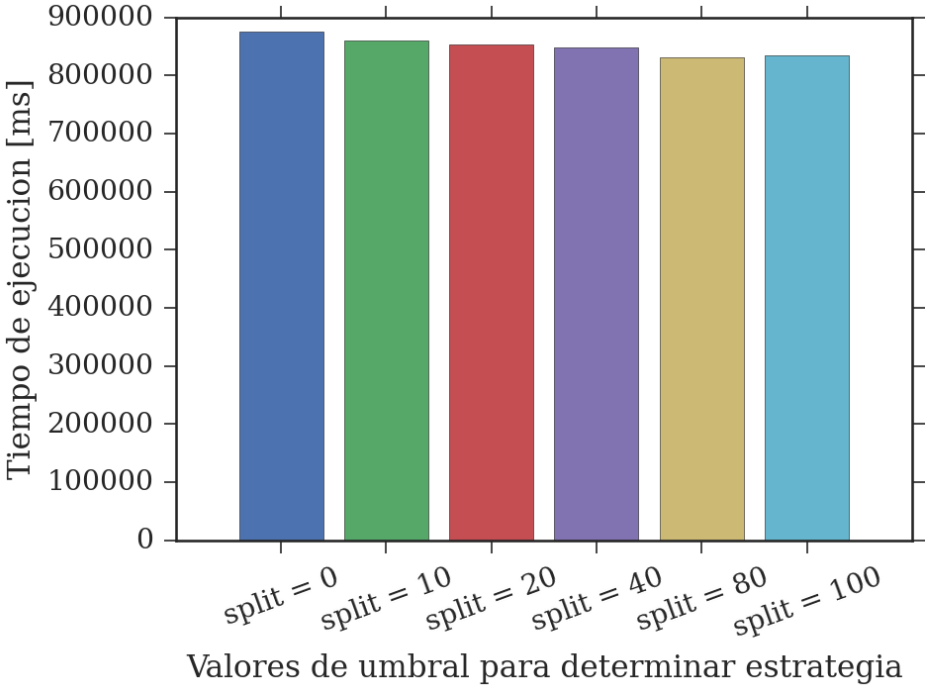


Fig. 3.27: Valor del tiempo de iteración de XC en milisegundos, para distintos valores del umbral de cantidad de puntos y matriz de Kohn-Sham a asignar a cada *thread*

de cargas que obtuvo buenos resultados en la implementación para GPU.

El algoritmo en CPU es equivalente al algoritmo 1 de GPU. Se mide el tiempo de resolución para cada grupo y cuánto dura cada hilo de ejecución en total. Luego, se mueven grupos desde el *thread* de mayor duración al de menor eligiendo los grupos cuya duración mejor se ajuste a la diferencia de tiempos totales. Para más detalles del algoritmo puede verse la sección 3.2.8.

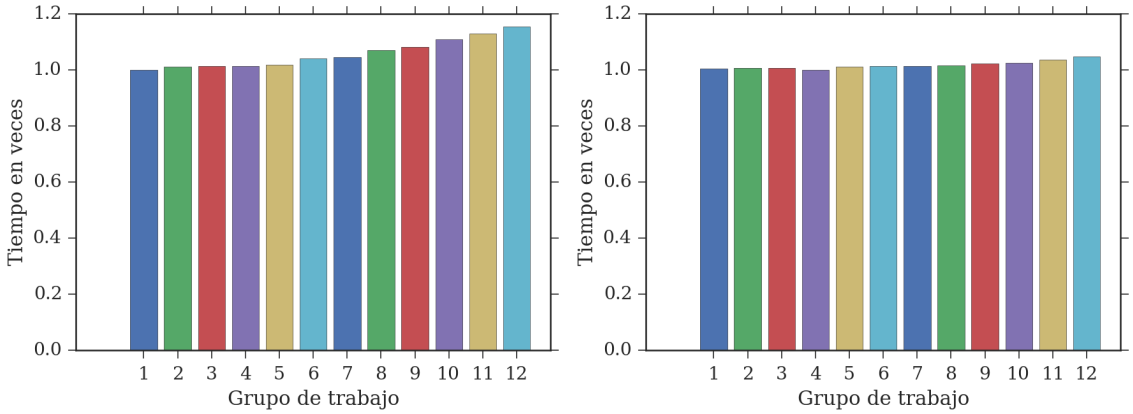
Ya se vio anteriormente que el costo computacional resulta bajo. En el caso del grupo hemo, se consigue finalmente menos del 5 % de diferencia entre cargas con solo una iteración de rebalanceo, obteniéndose los resultados de la figura 3.28b.

3.3.11. Paralelización a funciones y pesos

Aunque el trabajo se concentró en mejorar la iteración del cómputo de la energía de XC, al ser ésta la parte más pesada y que se ejecuta muchas veces (51 en el caso del grupo hemo), se realizaron unas mejoras al cálculo de los pesos de cada punto de la grilla y al cómputo de las funciones por grupo. Estos cambios se detallan en esta sección.

En el caso del cálculo de pesos por grupo, el cómputo consiste de un único ciclo sobre todos los puntos y luego filtrar los puntos que resultan despreciables. Separando este ciclo en dos partes, se pudo paralelizar el primero con OpenMP de manera análoga a lo realizado anteriormente.

Los resultados son buenos, aunque no demasiado: el ciclo escala 10 veces con 12 *threads* para el caso del grupo hemo, como puede verse en la figura 3.29. El ciclo de filtrado de puntos no es fácilmente paralelizable puesto que se deben agregar los elementos en orden a un arreglo de grupos.



(a) Previo al balance. *(b)* Posterior al balance.

Fig. 3.28: Comparación de los tiempos de ejecución para las distintas cargas asignadas a cada hilo de ejecución en el caso del grupo hemo.

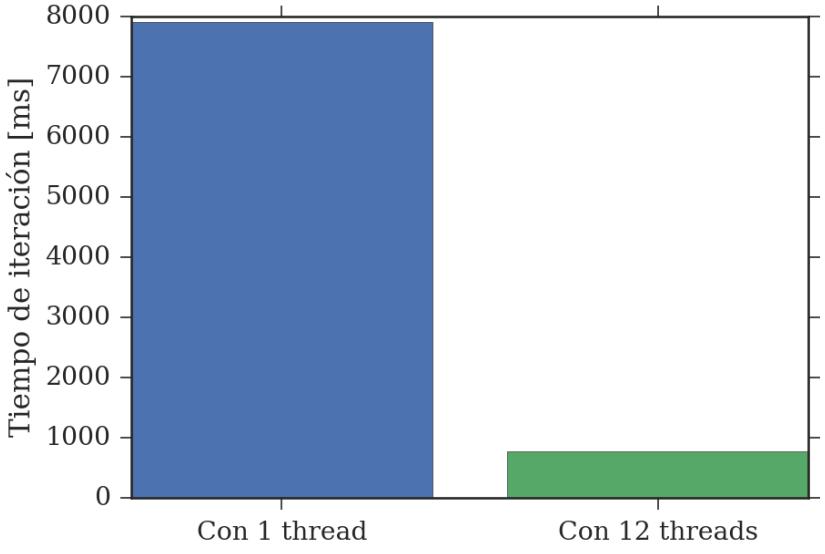


Fig. 3.29: Tiempo en milisegundos para la paralelización del cómputo de los pesos de la grilla de integración, según cantidad de *threads*

El caso de funciones es más interesante, porque los resultados obtenidos son bastante malos: Si bien calcular las funciones para cada grupo es una tarea muy paralelizable en teoría debido a que no tienen dependencia entre ellos, a diferencia de los cálculos de la iteración, los resultados obtenidos no fueron satisfactorios.

Por un lado, se logró evitar las proyecciones de matrices realizando todos los cálculos usando las matrices de componentes. Esto aumentó la *performance* de manera significativa, como puede verse en la figura 3.30. Los resultados se condicen con los obtenidos para la iteración de XC.

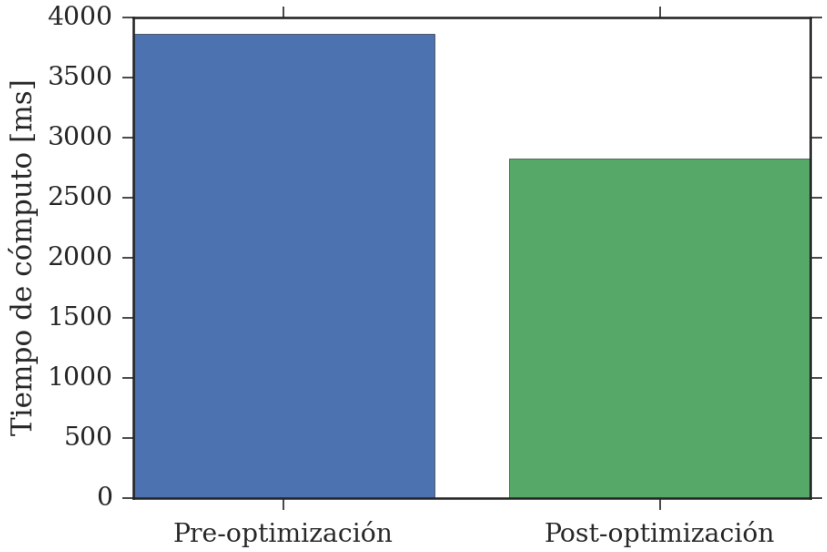


Fig. 3.30: Tiempo en milisegundos del cómputo de todas las matrices para todos los grupos en el caso del grupo hemo, antes y después de usar las matrices por componente.

Nuevamente, la causa es una baja relación entre cómputo y accesos a memoria, y la disparidad de costo entre grupos. La primera dificulta la división de los puntos del grupo entre hilos de ejecución, mucho más que en el caso de la iteración: en esta última los valores de las matrices eran reutilizados. La otra, al igual que antes, complica la paralelización externa, en la que subconjuntos de los grupos son resueltos en paralelo por *threads* distintos.

Para tratar de contrarrestar este efecto en la paralelización externa, se utilizó un *scheduler* guiado, en el cual cada *thread* va recibiendo porciones fijas de las iteraciones de manera dinámica. De esta manera un *thread* muy cargado recibirá menos iteraciones mientras que los ociosos reciben más.

El funcionamiento de esta estrategia no está asegurado puesto que el *runtime* de OpenMP no tiene conocimiento *a priori* de los costos.

Los resultados pueden verse en el esquema del algoritmo 3.31. La paralelización externa es mejor que la interna, pero ambas escalan mal, menos de seis veces para 12 procesadores. Aunque la importancia no es tanta para estos cálculos porque se realizan una vez, sirven también de punto de comparación con el trabajo realizado para paralelizar la iteración.

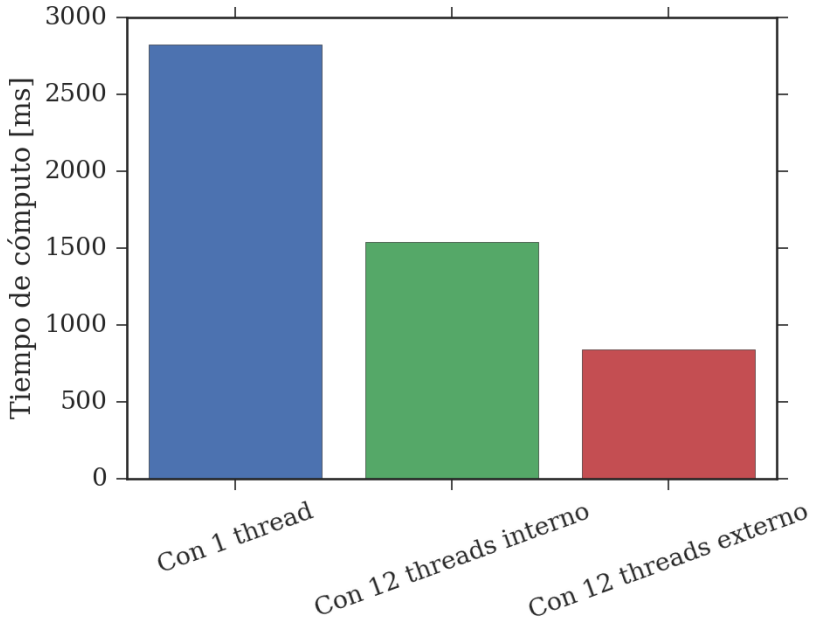


Fig. 3.31: Tiempo en milisegundos para la paralelización del cómputo de matrices para los grupos, según tipo de paralelización y cantidad de *threads*

3.4. Implementación en Xeon Phi

Se realizaron pruebas utilizando el código mejorado para CPU en una plataforma que incluye un Xeon Phi. Dado que el interés es estudiar la factibilidad de su utilización para esta aplicación y, siendo que el trabajo se concentraba en la iteración, se decidió estudiar cómo se comportaba el Xeon Phi en el modo nativo. Este modo corresponde con la ejecución de toda la aplicación directamente en el coprocesador.

Este enfoque produce una disminución en el tiempo de transferencia de las matrices del host al coprocesador, comparado con el modo *offloading*, aunque tiene limitaciones en el resto del código que no fue específicamente optimizado (como ocurre en la solución CUDA con el código que se ejecuta en el host), y limita el uso de memoria (ya que otras matrices de cálculos de SCF se almacenan también en el Xeon Phi). Como nos interesa realizar un primer acercamiento a esta arquitectura para analizar su comportamiento, consideramos que analizar el modo nativo constituye un punto de partida de interés.

El Xeon Phi utilizado para las pruebas corresponde al modelo `5110P` de 61 procesadores, con 8GB de memoria RAM GDDR5 y versión de MPSS (*Multicore Platform Software Stack*) 3.2.3.

Considerando los buenos resultados obtenidos con la implementación en CPU en términos de vectorización del código y de paralelización, se empleó esta misma versión del código fuente como punto de partida. De acuerdo a [35], el diseño del Xeon Phi debería aprovechar la implementación realizada, suponiendo que el código escala bien en CPU.

3.4.1. Resultados preliminares

Como se hizo en la sección anterior, se utilizó como ejemplo el caso de estudio del grupo hemo, con tamaño de cubos igual a 3 a.u.

No se pueden hacer comparaciones contra el código original de CPU recompilado para Xeon Phi, por las siguientes razones:

- La versión original utiliza la biblioteca GSL (GNU Scientific Library) para ciertas operaciones numéricas en CPU. El uso de esta biblioteca fue eliminado debido a que requería ser recompilada especialmente para Xeon Phi y su utilización en el código era mínima. El resto del código de la aplicación que usaba BLAS, utilizaba la implementación de Intel en la MKL que sí cuenta con una versión disponible en Xeon Phi.
- La implementación de vectores $\mathbb{R}^3$, como ya fue explicado en la sección 3.3.2, utilizaba extensiones vectoriales especiales del compilador apuntando a SSE 4. Dado que este set de instrucciones es incompatible con el del Xeon Phi fue necesario remover esta optimización.

Por lo tanto, las comparaciones se realizan contra la versión optimizada para CPU como se ha descripto en la sección anterior. El código fuente inicial utilizado es exactamente el mismo que en la versión definitiva en CPU, con la diferencia de que es compilado para la arquitectura MIC.

En primera instancia se obtuvieron resultados muy desalentadores en la versión *monoprocesador* del código, como puede verse en la figura 3.32. Para el caso particular del cálculo de XC, pueden verse los resultados en la figura 3.33.

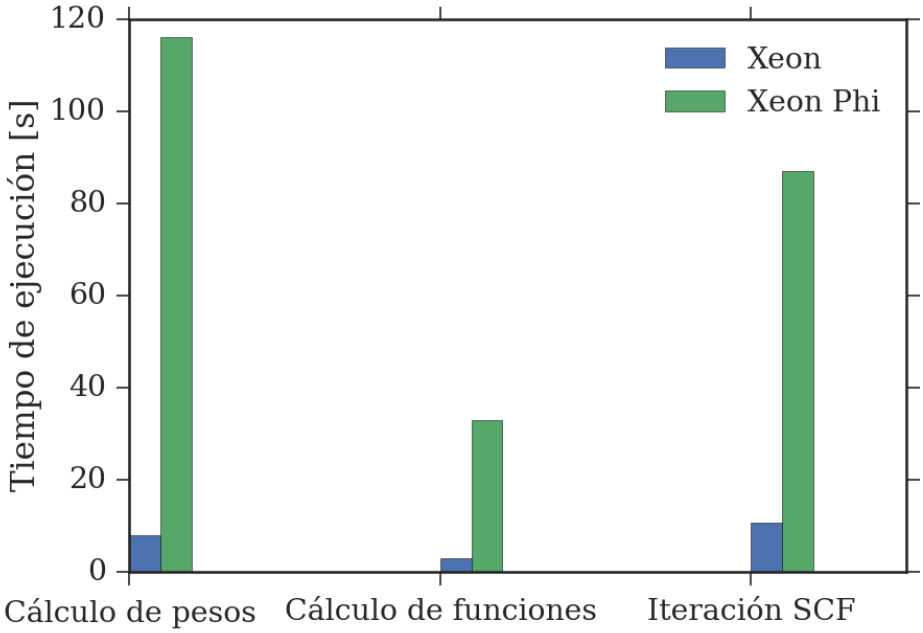


Fig. 3.32: Tiempo de ejecución para las distintas partes del paquete LIO, comparativamente para Xeon y Xeon Phi, en el caso del grupo hemo con un solo procesador

Tanto el cómputo de pesos, funciones y la iteración en sí para el caso del grupo hemo muestran una ralentización muy fuerte, incluso considerando que todos los ciclos fueron vectorizados por el compilador, según los reportes generados durante el proceso de compilación.

En el caso de XC, se ve que esta ralentización sucede en las dos operaciones principales: tanto el cálculo de la densidad electrónica como la actualización de la matriz de Kohn-Sham sufren considerables desaceleraciones.

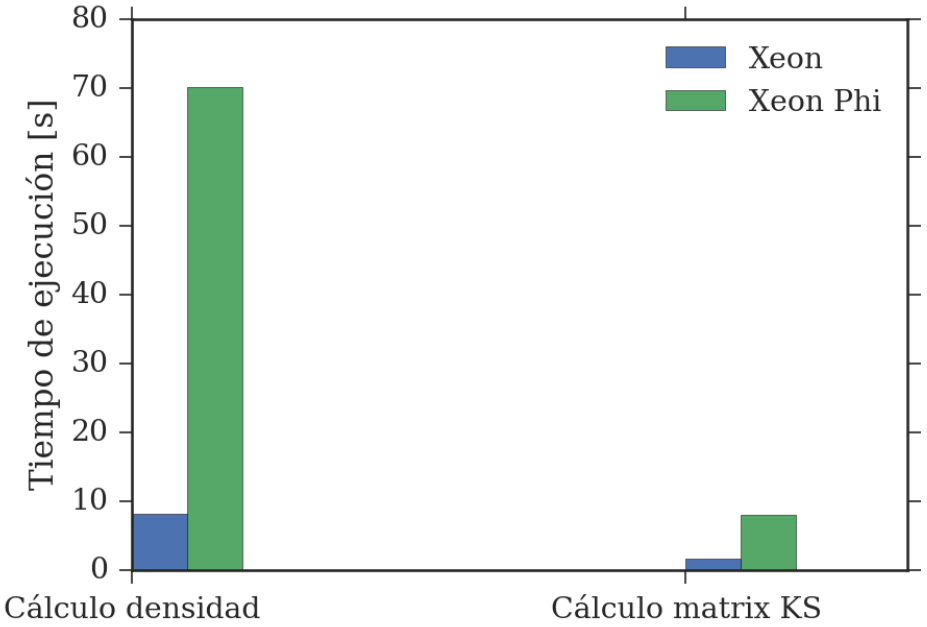


Fig. 3.33: Tiempo de ejecución para las distintas partes de la iteración XC, comparativamente para Xeon y Xeon Phi, en el caso del grupo hemo con un solo procesador

Se corrobora, entonces, la pobre *performance* del Xeon Phi en código serial, incluso en secciones sencillas y vectorizados por el compilador.

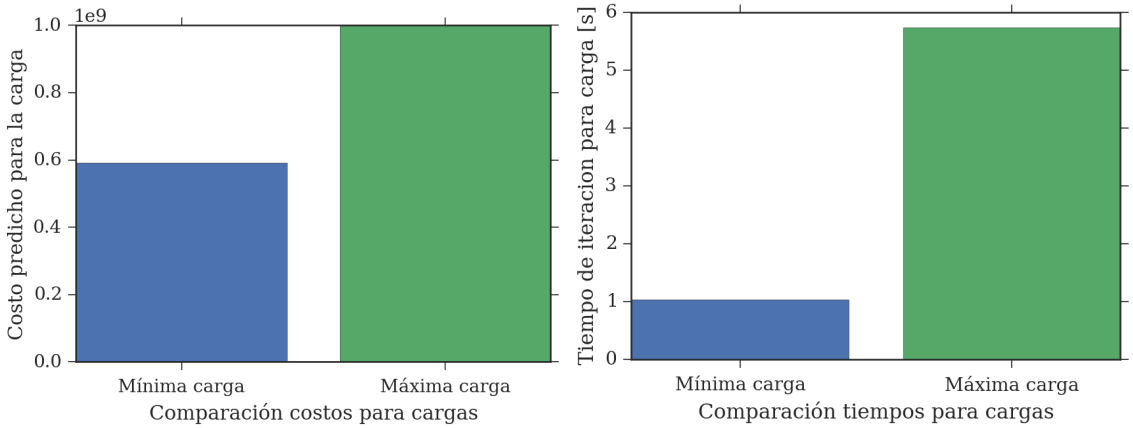
La disminución del desempeño por núcleo con respecto al Xeon impacta fuertemente en toda la aplicación, incluso en la parte más paralelizada. Al poner más presión en el uso efectivo de los núcleos, se visualizan notoriamente las pérdidas predichas por la ley de Amdahl, resultando en una baja *performance* y en una baja escalabilidad.

Un ejemplo de estos efectos puede verse en la paralelización de los grupos chicos en cargas para cada procesador. Al ser el código serial en el Xeon Phi muy poco eficiente, se vuelve necesario utilizar más procesadores para resolver un mismo grupo. Esto conduce a usar un umbral de separación menor para considerar más grupos como grandes, pero esto impacta negativamente en la escalabilidad dado que:

- Utilizar más *threads* de los que pueden ser aprovechados por un grupo disminuye la escalabilidad, al dejar procesadores ociosos.
- El algoritmo de partición cuenta con menos grupos para manejar, con lo cual tiene menos capacidad de balanceo entre los procesadores. El desbalance también produce que haya procesadores ociosos.

Por ejemplo, en la figura 3.34 se puede ver el balance de la partición resultante de utilizar 60 procesadores y la función de costo y valores de división calculados en la sección anterior. Se detalla tanto el desbalance teórico (predicho por la partición) como el balance de tiempo. Este último es especialmente pronunciado y puede verse un incremento muy

notable del tiempo de ejecución también para grupos chicos. Al ser los mismos resueltos serialmente, este incremento es consistente con los resultados anteriores, pero desfavorece la división en cargas de los grupos chicos en el Xeon Phi.



(a) Costo total predicho para la partición *(b)* Tiempo de ejecución efectivo

Fig. 3.34: Comparación entre la menor y mayor de las cargas para *threads*, según el tipo de parámetro, para una iteración del grupo hemo en Xeon Phi, usando la función de costo y división en grupos obtenida para CPU, en 60 *threads*.

De lo anterior se desprende también la necesidad de modificar la función de costo para el Xeon Phi. En esta arquitectura, todo cómputo serial tiene serias limitaciones con respecto a un CPU convencional, con lo cual ahora no se tiene un *overhead* por cubo tan pesado con relación al cómputo. Por esto, la función de costo anterior no resulta útil, tal cual puede verse en la figura 3.35. En esta figura, también se encuentran los motivos por los cuales se acrecentó el desbalance tan notoriamente en la figura 3.34.

Adicionalmente, tampoco resulta útil el balance de cargas en este caso: La baja *performance* serial provoca que cada cálculo lleve más tiempo. Por lo tanto, las cargas más pesadas (incluso para distintos valores del umbral) dominan un procesador entero y demoran a todos los demás y, como la carga suele consistir de un solo grupo (en las pruebas realizadas con el grupo hemo al momento de realizar las optimizaciones), no es posible hacer un balance de cargas.

Por otro lado, el aumento de *threads* y su comportamiento serial (de movimiento de memoria principalmente) hace más costosas las operaciones de reducción de resultados intermedios. A diferencia de CPU, en el cual este paso era despreciable, en el Xeon Phi pasa a tener un valor considerable, como puede verse en la figura 3.36. Esto disminuye aún más las ventajas obtenidas por la paralelización externa, ya que más procesadores involucrados producen más matrices que se deben reducir. Si bien se puede mejorar esta reducción aprovechando el incremento en la capacidad del bus de memoria del coprocesador, es un factor adicional a tener en cuenta contra la partición en grupos.

Como conclusión, entonces, se puede ver que en el caso de Xeon Phi, no tiene sentido realizar una paralelización de cubos y esferas chicos y grandes como la realizada en la implementación CPU. Para hacer corresponder esto, empleamos un umbral que permita que todos los grupos sean considerados grandes y sean calculados con todos los *threads* disponibles. Es decir, utilizamos solamente paralelización dentro de cada grupo para el Xeon Phi en base a los *trade-offs* presentados.

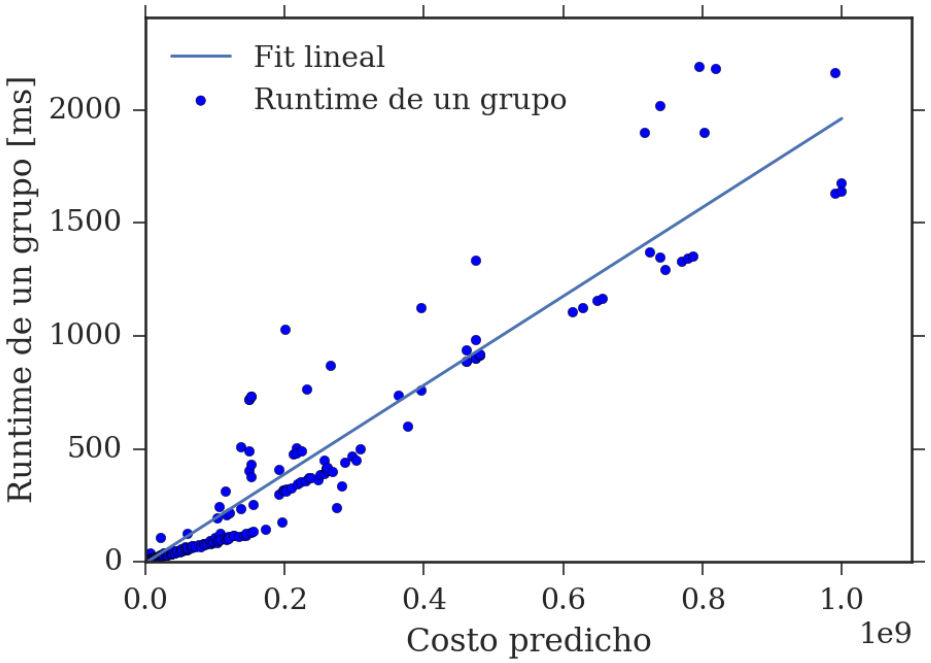


Fig. 3.35: Tiempo de resolución en milisegundos en función del costo predicho por el algoritmo para CPU, para cada grupo, en el caso del grupo hemo en Xeon Phi. Como puede verse, no hay una correlación fuerte entre estos valores.

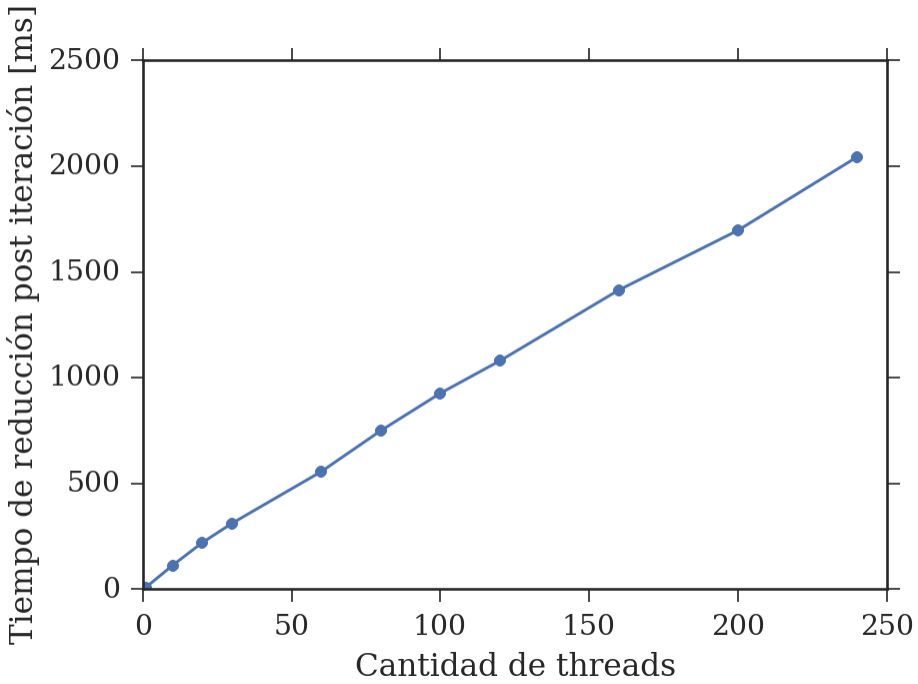


Fig. 3.36: Tiempo de reducción de las matrices de fuerzas y Khon-Sham para la iteración del grupo hemo en Xeon Phi, según la cantidad de *threads*.

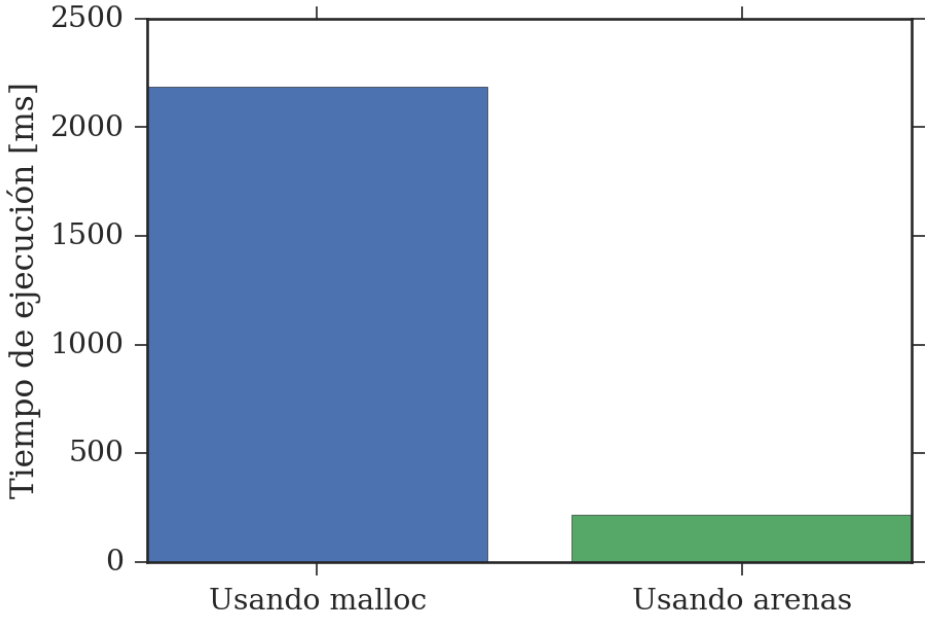


Fig. 3.37: Comparativa entre el tiempo de cálculo de funciones para el grupo hemo, usando el esquema de arenas de memoria presentado frente al uso de *malloc* estandar.

Adicionalmente, secciones despreciables en CPU deben ser paralelizadas para poder hacer uso del Xeon Phi. Un factor de importancia discutido es el ancho de banda de memoria disponible en esta arquitectura, debido a su memoria GDDR5 de alta velocidad. Ésto, junto con el *hardware* para asignar hilos a procesadores, le permite tener un techo más alto en términos de cuántos *threads* pueden ejecutar antes de saturar el bus de memoria. Un ejemplo de este comportamiento es la lectura de la matriz de Kohm Sham de cada grupo. En Xeon Phi este ciclo ejecutado serialmente era comparable con el resto del cómputo de XC paralelizado. Al paralelizarlo, logramos una mejora de cuatro veces en esta sección.

Por otra parte, un problema que requiere solución en este dispositivo es la limitada cantidad de RAM. Teniendo solamente 8 GB disponibles y no pudiendo expandirse (al contrario del *host*), es necesario tener en cuenta que todas las matrices utilizadas para todos los grupos pueden no entrar en la memoria principal simultáneamente.

En el caso particular de grupo hemo, todas las matrices de funciones pueden ser almacenadas efectivamente para todos los grupos. Sin embargo, en sistemas más grandes (como el fullereno C_{60}, que analizamos en los resultados) no siempre es así.

Para resolver este problema, se usa una estrategia similar a la de GPU, utilizando un *pool* ajustable de memoria, según la cantidad disponible en el sistema. También se ordenan los cubos y esferas por su costo en memoria, que no es el mismo que en GPU por como se almacenan las matrices y por el uso de una matriz de funciones transpuesta.

A diferencia de GPU, como se vio en la sección 2.3, es muy importante lograr un buen uso de las caches por su limitado tamaño (aunque no despreciable) y por como está diseñado el sistema de memoria. Reservar temporariamente en cada iteración produce que salga de un rango potencialmente distinto de la memoria física, contribuyendo a generar *cache-misses* y causando mucha fragmentación de la memoria.

Para alivianar este problema, utilizamos una estrategia de reserva de memoria según la estructura de la aplicación. La memoria se pide al principio en una sección fija y continua

(de esta manera sabemos también un límite superior de cuánta memoria se necesita para correr). Esta memoria se divide en dos secciones contiguas de manera que todas las matrices y arreglos estén consecutivos en memoria. Las dos secciones (también conocidas como *arenas* en la jerga) son:

- *Memoria permanente*: En esta memoria se almacenan todas las matrices que persisten entre iteraciones. Adicionalmente, todas las matrices de grupos que se puedan mantener forman parte de esta sección.
- *Memoria efímera (transient)*: Esta memoria es para las matrices temporales en las iteraciones. Al iniciar el procesamiento de un grupo se la sobrescribe con lo que necesite el nuevo grupo, con un costo muy bajo de reinicialización.

Esta estrategia tiene consecuencias muy impactantes en el costo de calcular las funciones, incluso ante la baja disponibilidad de memoria. Como puede verse en la figura 3.37, la primera iteración insume mucho tiempo y las iteraciones posteriores insumen comparativamente muy poco tiempo, incluso sin realmacenar matrices. Esto es peor que en el caso ideal que se logra en Xeon (donde podemos almacenar todas las funciones sin volver a calcularlas), pero dado que el cómputo de las matrices está fuertemente limitado por la memoria (porque solo se usa cada posición una vez), usar siempre el mismo bloque de memoria entre iteraciones contribuye a un mejor comportamiento del *cache*.

4. RESULTADOS

A continuación presentamos algunos de los resultados obtenidos para las distintas arquitecturas analizadas, tanto de tiempo de cómputo como de escalabilidad para los casos de prueba usados. Las máquinas y opciones de compilación utilizadas se detallan en el Apéndice A.

Los tiempos de ejecución se midieron utilizando el reloj estándar de alta precisión para Linux (`clock_gettime` [45]), al igual que se hizo en las secciones anteriores. Debido al balance de cargas utilizado en la división de los grupos pequeños, se consideraron las mediciones en la que las cargas para cada procesador se encontraban balanceadas (siendo éstas la mayoría de las iteraciones de SCF). Tampoco se consideró la primera iteración, ya que incluye un costo de cálculo de funciones que se aprovecha en las siguientes iteraciones.

Siendo que las máquinas utilizadas dedicadas a estas simulaciones únicamente y habiéndose obtenido una varianza despreciable entre las mediciones de las distintas corridas, no se incluyen estos datos en los resultados presentados (al no proveer información adicional).

Los casos de prueba utilizados corresponden a moléculas del grupo hemo, Fullereno C_{60} y Caroteno. El grupo hemo fue elegido por ser un representante de tamaño medio de los sistemas normalmente simulados. El fullereno C_{60} y el caroteno fueron elegidos por tener geometrías diferentes y ser tambien grupos de tamaño medio. El detalle químico está incluido en el Apéndice B.

4.1. Resultados en CPU

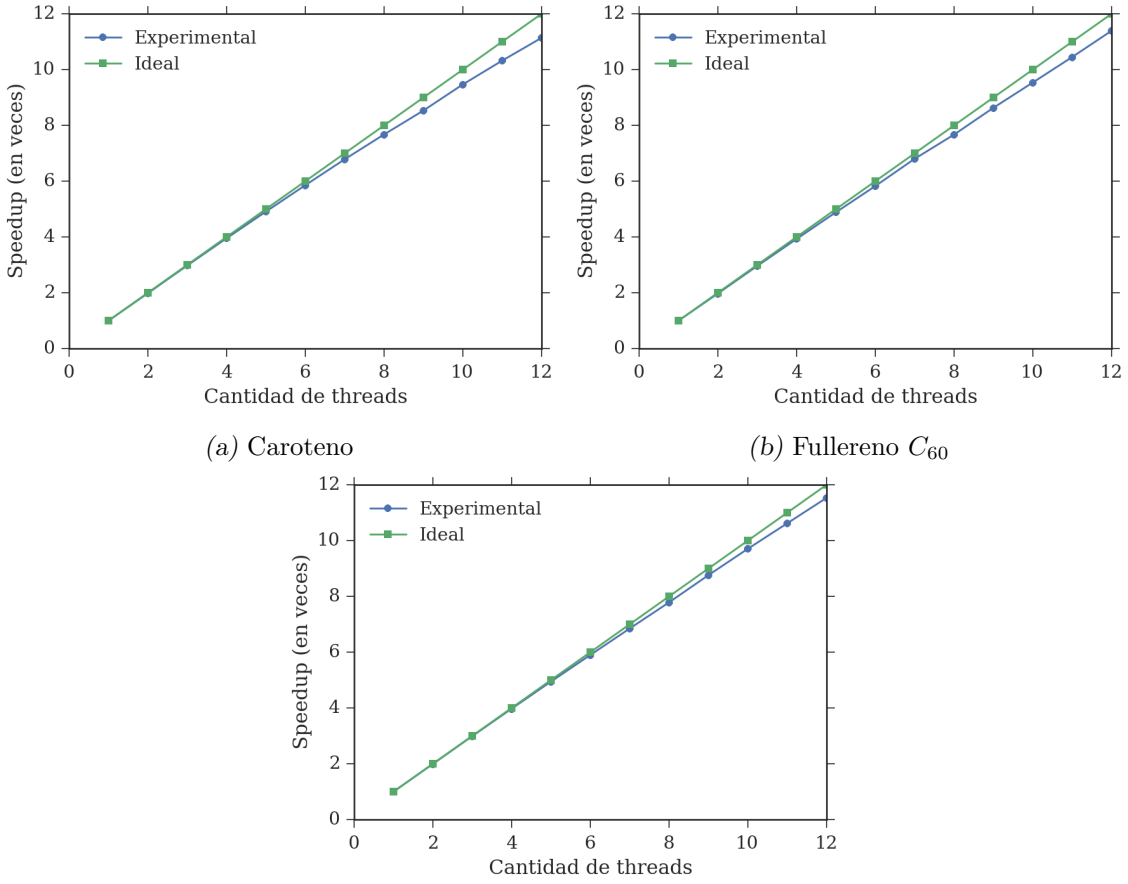
Las figuras 4.1a, 4.1b, y 4.1c muestran el *speedup* conseguido para la versión de CPU en las pruebas realizadas en el servidor Xeon para los casos del grupo hemo, Caroteno y Fullereno C_{60} respectivamente. Se detalla el *speedup* para cada valor de threads intermedios entre 1 y 12 inclusive (la cantidad de núcleos de procesador del sistema de prueba).

Siendo éste un análisis de escalabilidad, se trató de homogeneizar los procesadores tanto como fue posible. Por esta razón, los tiempos se midieron deshabilitando *Hyper-Threading* y *Turbo Boost* para estos experimentos.

Los resultados obtenidos son muy cercanos al óptimo teórico, lográndose una aceleración de 11,39, 11,14 y 11,53 en Fullereno C_{60}, Caroteno y el grupo hemo respectivamente. Una comparación de estos tres casos en cuanto a tiempo efectivo de ejecución (donde sí se consideran todas las prestaciones del procesador) puede verse en la figura 4.8.

También se realizó un análisis del uso de vectorización y su impacto en el resultado final. Para eso usamos los casos anteriormente mostrados con 12 *threads* y habilitando todas las características del procesador, y comparamos un código explícitamente no vectorizado (usando la opción `no-vec` de Intel ICC) con el código vectorizado. El resultado, que se puede ver en la figura 4.2, muestra el impacto de la vectorización utilizada, que llega a un *speedup* de cuatro veces aproximadamente.

Dada la heterogeneidad del conjunto de casos elegido, se puede concluir que la versión de código presentada hace un excelente uso de las prestaciones de los procesadores multicore tradicionales de la arquitectura Xeon en una interesante variedad de pruebas.



(a) Caroteno

(b) Fullereno C_{60}

(c) Grupo hemo

Fig. 4.1: Speedup conseguido para una iteración XC en Xeon, según la clase de problema examinado.

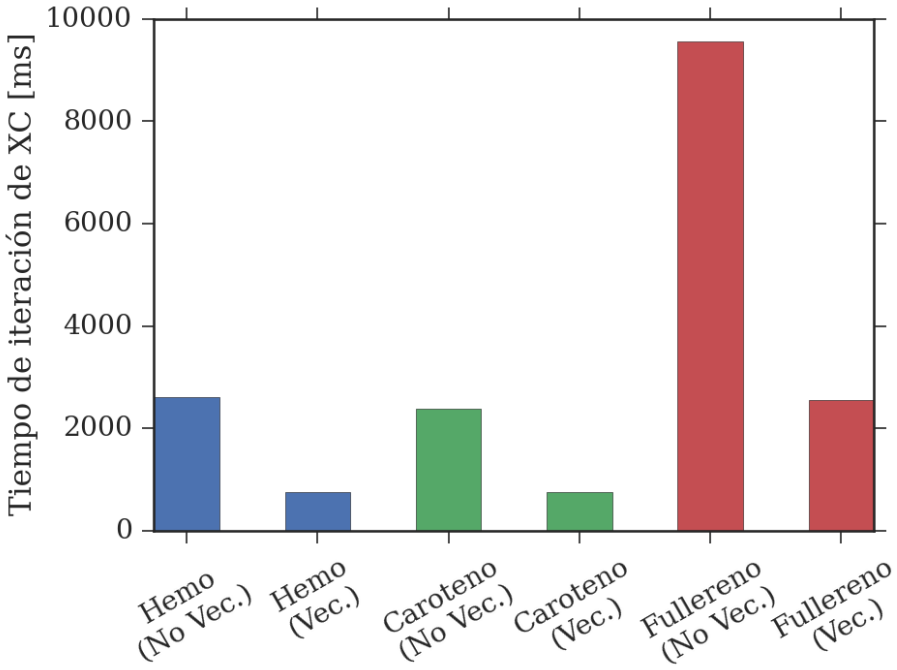


Fig. 4.2: Comparación para los tres casos de prueba estudiados del código final en CPU con vectorización deshabilitada y habilitada, en 12 *threads* del Xeon usado con *Turbo Boost*.

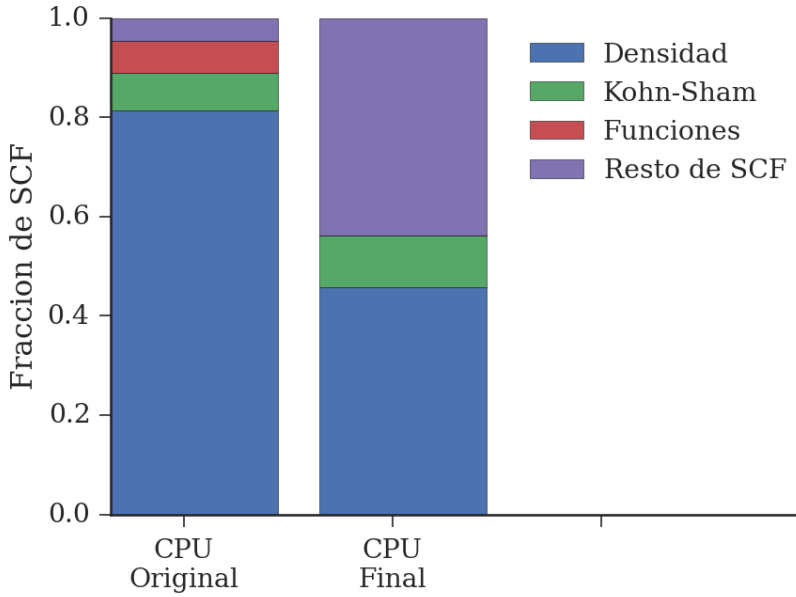


Fig. 4.3: Porcentajes de tiempo de los pasos de XC dentro de SCF corriendo el caso de grupo hemo en Xeon E5-2620 antes y después de las optimizaciones realizadas.

Los tiempos parciales de XC de una iteración de SCF se encuentra en la figura 4.3. El caso usado corresponde al del grupo hemo, ejecutado en el servidor Xeon. El *speedup* final alcanzado por XC es del orden de 22 veces en la plataforma de prueba utilizada. La proporción de tiempo entre XC y SCF mejoró, de modo que los esfuerzos adicionales de optimización deberían guiarse en primera instancia en el resto de la iteración de SCF y luego, de vuelta, en el cálculo de la densidad electrónica.

4.2. Resultados en GPU

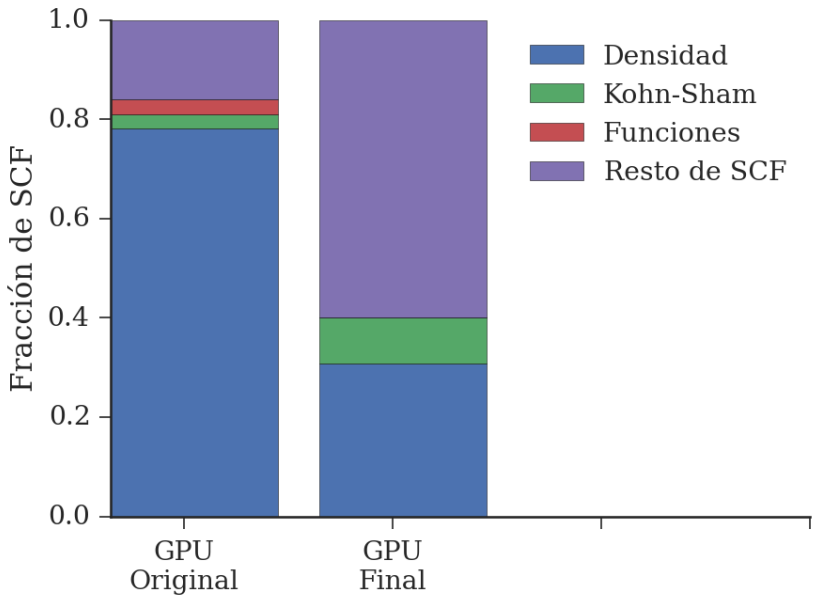


Fig. 4.4: Porcentajes de tiempo de los pasos de XC dentro de SCF corriendo el caso de grupo hemo en una Tesla K40 antes y después de las optimizaciones realizadas.

La figura 4.4 muestra la fracción de tiempo que insume el cálculo de XC dentro de SCF antes y después de realizar las optimizaciones. Estos resultados fueron obtenidos con una plataforma utilizando una Nvidia Tesla K40. Dentro del cálculo del término de intercambio correlación, el cálculo de la densidad electrónica representaba, por mucho, la porción que mayor cantidad de tiempo utilizaba. Con la reducción en los tiempos de cómputo de XC y, al guardar las funciones en memoria para no tener que recalcularlas constantemente, se obtuvo una aceleración de alrededor de 8 veces para la arquitectura Kepler y de 5 veces para Fermi, como se ve en la figura 4.5.

La implementación de técnicas de multi-GPU también obtuvieron grandes mejoras, como se vio en la sección 3.2.8. Incluso, no teniendo una mejora lineal en el cálculo de simple precisión, esta técnica permite explotar todos los recursos disponibles en un sistema de cómputo con múltiples coprocesadores numéricos GPGPU.

4.3. Resultados en Xeon Phi

Los resultados de la escalabilidad del código conseguido en Xeon Phi se pueden ver en la figura 4.6 para los tres casos de estudio vistos. La escalabilidad obtenida no es tan

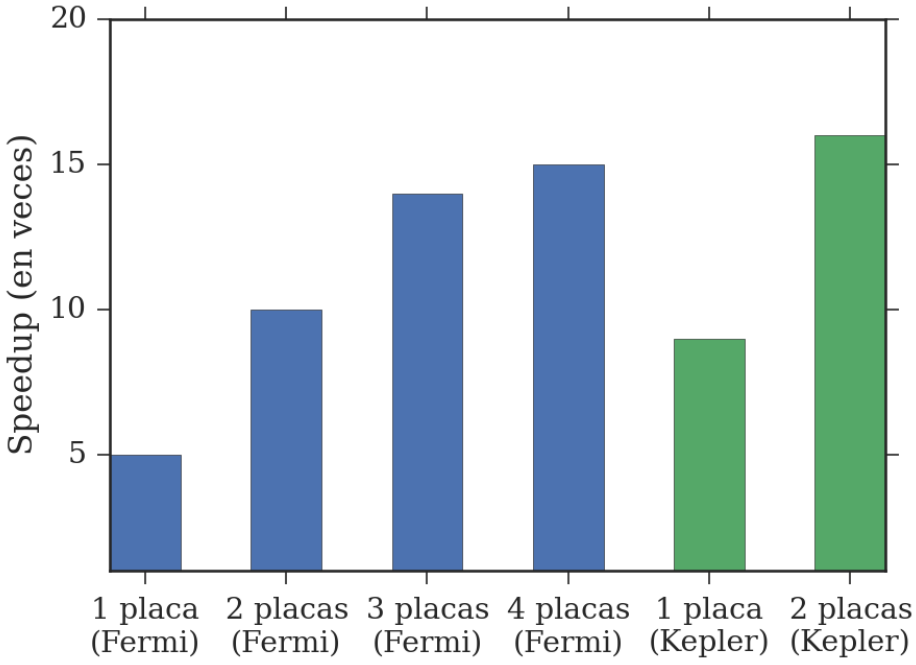


Fig. 4.5: Aceleración en veces del cálculo de XC de correr el grupo hemo comparando implementación original en CUDA contra las versiones optimizadas finales.

buena como en CPU, entre otras razones por utilizar solamente la paralelización interna (que no es tan buena para cubos chicos). Como puede verse, la paralelización consigue una escalabilidad en el orden del 98 % - 99 % según el caso.

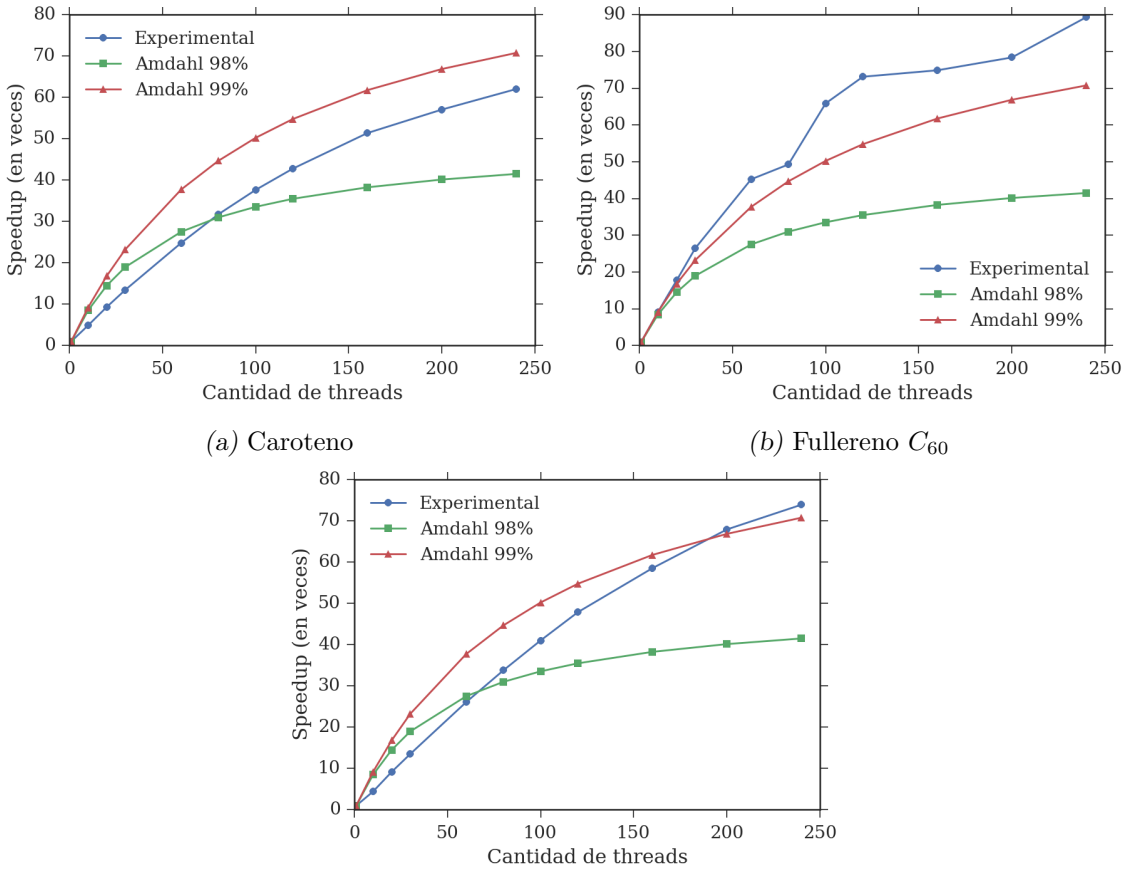
Por último, en la figura 4.7, se muestran los resultados de las distintas partes de la iteración SCF en Xeon Phi. Como se puede ver, la baja *performance* serial hace que las demás secciones dominen muy fuertemente, lo cual enfatiza aún más la necesidad de adaptar el código serial para utilizar las nuevas prestaciones vectoriales y paralelas de las nuevas arquitecturas.

4.4. Comparación entre arquitecturas

A continuación detallamos la comparación entre arquitecturas para los tres casos estudiados: grupo hemo, caroteno y fullereno. Se analiza la iteración XC usando los mejores parámetros para cada arquitectura (tamaño 8 a.u. de los cubos para GPU, tamaño 3 a.u. de los cubos para CPU y Xeon Phi).

El CPU utilizado es el servidor Xeon con el que se realizaron los experimentos anteriores y la GPU utilizada es una Tesla K40. En el caso de GPU, todas las matrices pueden ser almacenadas en la memoria global, con lo cual no hay tiempo de re-cómputo de funciones por iteración. En el Xeon Phi esto no ocurre, por lo que se usó toda la memoria que estuviese disponible para las arenas sin provocar que el programa dejara de funcionar por falta de RAM *on device*.

Los resultados se encuentran en la figura 4.8. Como puede verse la implementación en GPU es la más rápida, aunque la versión de CPU resulta comparable (menos que dos veces). El Xeon Phi está dentro de un rango similar a la implementación de CPU, también



(a) Caroteno

(b) Fullereno C_{60}

(c) Grupo hemo

Fig. 4.6: Speedup conseguidos para una iteración XC en Xeon Phi, según la clase de problema examinado.

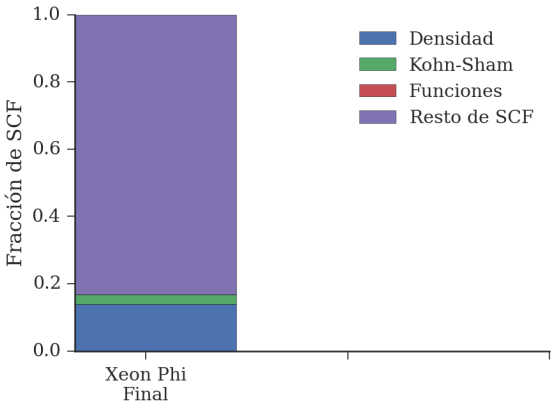


Fig. 4.7: Tiempos parciales para la iteración SCF separado entre la iteración de XC y el resto.

a menos de dos veces que la de CUDA, pero resulta la más lenta de las tres.

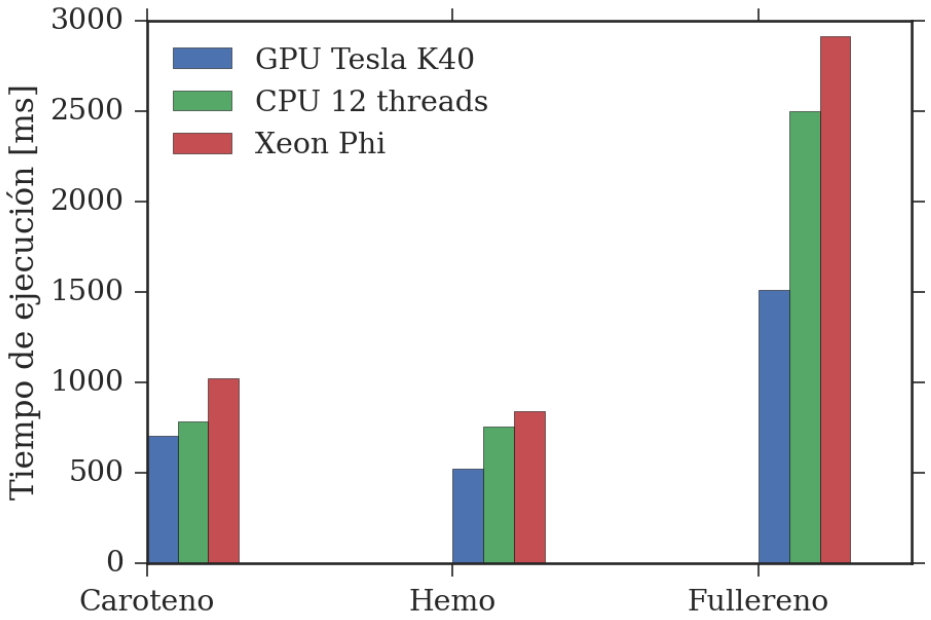
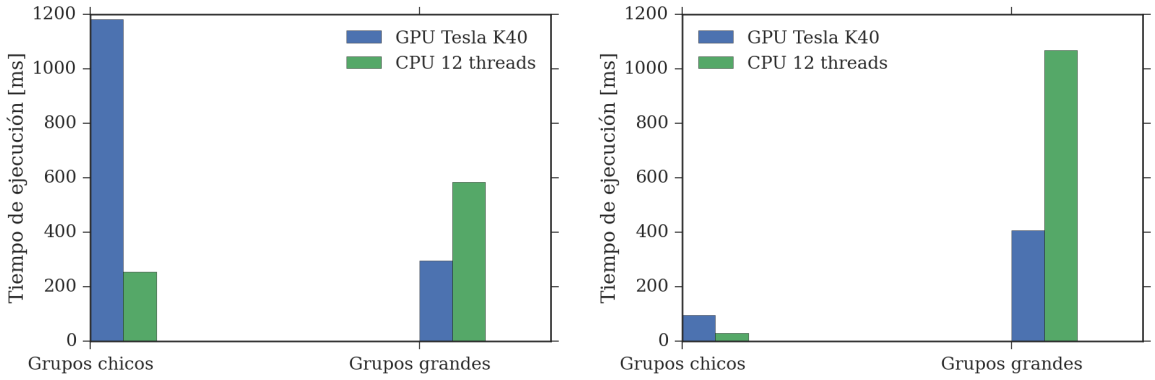


Fig. 4.8: Comparación entre arquitecturas con respecto al tiempo de iteración de XC para los tres casos de estudio analizados.

4.5. Tamaño de los grupos

Un parámetro con el que se experimentó durante el desarrollo del trabajo fue el tamaño de las particiones. Los cubos y las esferas que agrupan los puntos fueron descriptos en la sección 3.1. El parámetro del tamaño de los cubos siempre se consideró que debía ser "grande" para GPU mientras que debía ser "chico" para CPU. Para comprobar el impacto, analizamos para cada arquitectura el tiempo de ejecución de cada grupo en función del costo computacional, definido en la sección 3.3.6. Éstos los clasificamos como grupos "chicos" y "grandes", el mismo criterio usado en la implementación en CPU para decidir qué estrategia de paralelismo usar. Luego, evaluamos el tiempo de ejecución acumulado para cada una de estos grupos en cada arquitectura.

En la figura 4.9 se muestran los tiempos de ejecución para el grupo hemo discriminando grupos chicos y grandes, para las implementaciones de CPU y GPU usando cubos de tamaños 3 y 7 a.u. Estos resultados muestran por qué usar tamaños de cubos chicos para CPU y grandes para GPU es conveniente. La causa es que el overhead que se ve en GPU al hacer grupos chicos es sumamente elevado, a pesar de que el costo computacional sea comparativamente bajo. Éste, sin embargo, se vuelve despreciable cuando se analiza el costo de resolver los grupos de mayor tamaño. Consistentemente, la GPU obtiene un menor tiempo de ejecución con respecto al CPU usando la paralelización interna. En consecuencia, si la partición genera demasiados grupos, la GPU no va a poder superar el overhead de resolver cada uno, haciendo que el tiempo acumulado sea mayor al de CPU a pesar de que tenga una peor *performance* en particiones grandes.



(a) Grupos de tamaño 3 a.u. *(b)* Grupos de tamaño 7 a.u.

Fig. 4.9: Tiempos acumulados de ejecutar el grupo hemo, separados por tamaño de grupos y caracterización.

4.6. ¿Qué me conviene comprar?

Luego de todas las mejoras realizadas a la aplicación LIO, queda este interrogante pendiente: para ejecutar simulaciones de QM/MM, ¿Qué versión me conviene usar: CPU o GPU?

Esta pregunta es siempre capciosa y susceptible a cambios tecnológicos que vuelvan obsoleta cualquier discusión en muy corto tiempo. Intentaremos responderla utilizando dos clases de dispositivos en los cuales ejecutar LIO, uno para una estación de trabajo y el otro para un servidor de cómputo. Definimos, para cada configuración, una versión para GPU y otra para CPU de modo de priorizar los recursos invertidos. No se evaluó Xeon Phi en este análisis al ser una plataforma nueva con poco *hardware* disponible que la soporte y con un costo sumamente elevado en comparación a la performance obtenida. Se buscó que los costos de ambos niveles sean parejos, de modo de hacer la comparación realista en términos de *hardware* a usar. El costo promedio del la estación de trabajo definida en la la tabla **??** es de aproximadamente US$1500, mientras que el del servidor de computo ronda los US$5000 (precios estimativos a Enero de 2015).

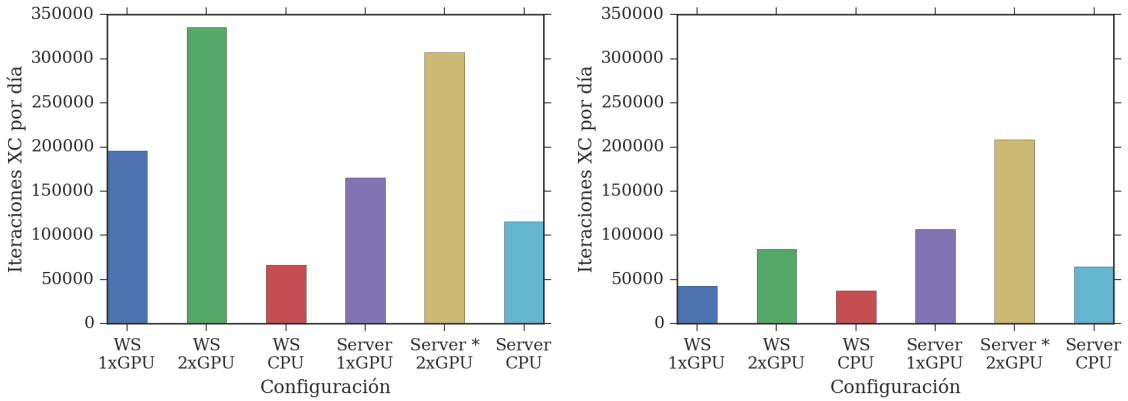
Configuración	Estación de trabajo	Servidor
GPU	Intel Core i5-4460 @ 3.2GHz 8GB RAM DDR3 1 o 2 x GeForce GTX 780 3GB	Intel Xeon E5-2609 @ 2.4GHz 16GB RAM DDR3 1 o 2 x NVIDIA Tesla K40 12GB
CPU	Intel Core i7-3770 CPU @ 3.40GHz 16GB RAM DDR3	2 x Intel Xeon E5-2620 v2 @ 2.10GHz 32GB RAM

Tab. 4.1: Distintas configuraciones de estaciones de trabajo y servidores para cómputo de QM/MM usando LIO

Lo que primero se debe notar en GPU es que se usan placas que tienen potencia de cálculo casi equivalente cuando se utiliza precisión simple. Sin embargo, elegimos una Tesla para configuración de servidor porque son las placas mejor preparadas para HPC. Éstas cuentan con cuatro veces más memoria en la placa, con ECC (*Error Correcting Code*) y

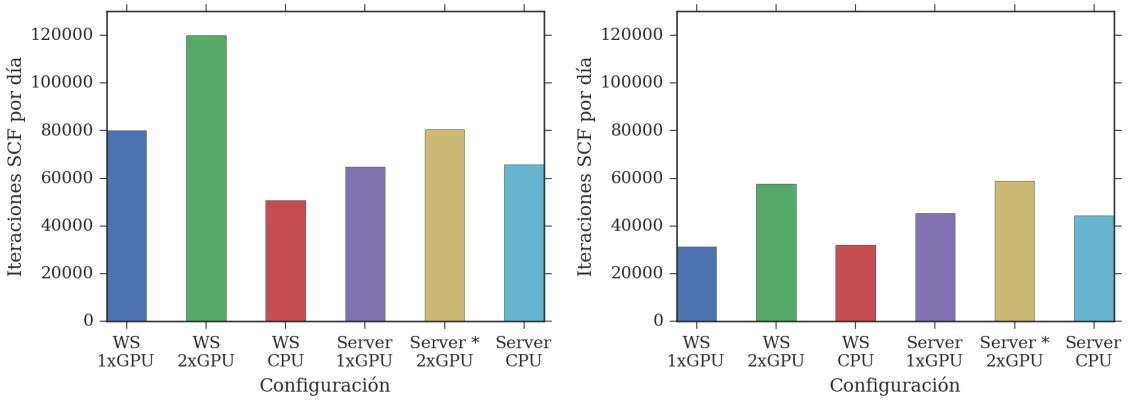
con mayor MTBF (*Mean Time Between Failures*), factor vital en servidores que deben correr de manera confiable por largos períodos de tiempo.

La métrica que usaremos para comparar los distintos sistemas es cuántas iteraciones del cálculo de XC y SCF se pueden ejecutar por día. Ésto lo hacemos para medir estrictamente la *performance* de QM, importante en simulaciones de *Time-Dependant Density Functional Theory* por ejemplo, para no hablar de las implementaciones de los sistemas de QM/MM que utilizan LIO. Esta métrica es similar a las que se usan en el área de MM. Programas como Amber[5] la usan para comparar distintas configuraciones de hardware sobre la cual correr eficientemente.



(a) Precisión simple. *(b)* Precisión doble.

Fig. 4.10: Cantidad de iteraciones de XC por día usando distintas configuraciones. Se usa el grupo hemo con los parámetros óptimos una vez aplicadas todas las optimizaciones. El (*) marca resultados teóricos extrapolando las aceleraciones alcanzadas en la sección 3.2.8.



(a) Precisión simple. *(b)* Precisión doble.

Fig. 4.11: Cantidad de iteraciones de SCF por día usando distintas configuraciones. Se usa el grupo hemo con los parámetros óptimos una vez aplicadas todas las optimizaciones. El (*) marca resultados teóricos extrapolando las aceleraciones alcanzadas en la sección 3.2.8.

Un factor a tomar en cuenta para analizar la *performance* de diversas configuraciones, como en la figura 4.11, es que el cálculo de SCF incluye contribuciones que no son solamente las relativas a intercambio-correlación (las estudiadas en esta tesis), por lo cual los tiempos

de ejecución totales van a presentar menores aceleraciones que las presentadas en secciones anteriores.

La cantidad de iteraciones de XC por día se ven en la figura 4.10. Mostramos de manera independiente el cálculo de XC porque creemos que es posible conseguir mejoras similares a las obtenidas en este trabajo en el resto de las contribuciones de SCF.

La principal ventaja que se obtiene de correr en las estaciones de trabajo con GPU es la potencia de cálculo concentrado en las placas de vídeo. La mayor aceleración del cálculo de XC se obtiene usando dos GeForce GTX 780, en comparación con únicamente usando el procesador. Esto se condice con las hipótesis discutidas anteriormente sobre que los núcleos del procesador se encuentran completamente ocupados. En cambio, las estaciones de trabajo con CPU cuentan con la ventaja de que se acelera también el resto de las operaciones que componen una iteración de SCF. Estos cálculos son de menor costo computacional comparada con las de XC, pero luego de las aceleraciones alcanzadas por este trabajo, su peso es comparativamente grande. Como estas otras contribuciones de SCF no están aceleradas con GPU, ese recurso se subutiliza fuera de XC.

Cuando se comparan las configuraciones de servidores, el panorama cambia sustancialmente. Las aceleraciones obtenidas en la versión GPU provienen principalmente de un solo factor, la mayor cantidad de memoria global disponible en la línea de GPU para HPC. Esta memoria adicional permite mantener las matrices de funciones en memoria sin tener que recalcularlas, optimización detallada en la sección 3.2.5. Como la GeForce GTX 780 cuenta con una velocidad de clock mayor a la K40 (probablemente porque en la línea de GPU para vídeo juegos sea más importante la *performance* que la estabilidad) las operaciones se realizan incluso más rápido en la versión de consumidor. Sin embargo, la Tesla K40 tiene un costo alrededor de ocho veces mayor que la GeForce GTX 780 al momento de realizar esta tesis.

En la configuración de CPU la *performance* escala linealmente en la cantidad de núcleos. Dadas las optimizaciones realizadas con objetivo en mejorar la escalabilidad, es de notar la gran diferencia de *performance* con respecto a la de estación de trabajo. Priorizar CPU favorece también a todo el resto del cálculo de SCF. Las contribuciones que no son de intercambio-correlación se resuelven, o bien a través de bibliotecas BLAS en CPU, o bien aprovechando las técnicas de paralelización automática que brindan los compiladores usados.

Comparando ambas configuraciones de servidores, hoy en día es mucho más fácil encontrar nodos de cómputo de HPC usando múltiples procesadores que nodos con múltiples GPU por varios motivos, especialmente en clusters de pequeño y mediano tamaño. Primero, porque son más generales: los nodos se pueden usar para múltiples aplicaciones de HPC, todavía mayormente basadas en CPU. Segundo, por cuestiones energéticas: los requerimientos de electricidad de un cluster HPC son muy elevados y las GPUs pueden tener un consumo que hace muy difícil la disipación térmica. Finalmente, el costo: las placas GPU de la línea de servidores cuestan de seis veces más que los procesadores de muy alta gama que se usan en HPC. Si las aplicaciones no van a hacer uso constante de éstas y tener aceleraciones proporcionales al costo, pueden no ser rentables.

Con la aplicación ya optimizada tanto para CPU como para GPU, si uno dispone de una máquina con dos GPU potentes, esa sería la configuracíon más apropiada para maximizar la velocidad del cómputo. Sin embargo, si se cuentan con procesadores con muchos cores, tambien se puede llegar a desempeños similares sin la necesidad de adquirir nuevo *hardware* específico.

5. CONCLUSIONES

La simuluación numérica de problemas QM/MM es una herramienta moderna que ayuda a descubrir fenómenos químicos desde los principios más básicos de la física. Estas técnicas, sin embargo, son de tan alto costo computacional que no es posible visualizar comportamientos que duren más que fracciones de segundo de tiempo simulado. Todas las optimizaciones que se puedan hacer en este cómputo, entonces, pueden ser de gran utilidad, permitiendo extender la frontera del tiempo simulado y permitiendo analizar fenómenos no posibles de observar previamente.

En este trabajo, se estudiaron múltiples optimizaciones apuntadas a maximizar el uso de los recursos de cómputo. Se pudieron observar grandes mejoras al tiempo de procesamiento rediseñando los algoritmos de cálculo de DFT para aprovechar todas las prestaciones disponibles en las arquitecturas paralelas estudiadas.

En CPU, se pudieron obtener mejoras de hasta 22 veces sobre la *performance* original en el cálculo de intercambio-correlación, consiguiendo una escalabilidad lineal en la cantidad de procesadores disponibles. La velocidad final alcanzada es cercana a las de las GPU de alta gama de la generación anterior y está a menos de 1.5x del rendimiento en la placa más rápida de la generación actual. Esto se logró en base a reestructuraciones algorítmicas, al uso de bibliotecas estándar de paralelización, a reorganización de los datos en la memoria para facilitar la vectorización y a un análisis de partición de trabajos. Es, además, importante señalar que la implementación realizada solo requiere herramientas incorporadas al compilador, sin uso de bibliotecas externas, favoreciendo la portabilidad del código a múltiples plataformas y abriendo la posibilidad de incorporar el uso de bibliotecas como ATLAS y MAGMA para acelerar cálculos puntuales.

En GPU se consiguieron aceleraciones de hasta 8 veces en este cálculo con respecto a la versión original que ya estaba adaptada para GPGPU. Las mejoras pasaron por encontrar los limitantes de las distintas funciones del cómputo, reestructurando las paralelizaciones, almacenando más resultados temporales y poder aprovechando los cambios en la arquitectura de las GPU con respecto a las memorias *on-chip*. Finalmente, se obtuvieran mayores aceleraciones aplicando paralelismo entre múltiples placas GPU, donde, a pesar de no escalar linealmente con respecto a la cantidad de dispositivos en precisión simple, resultó una gran mejora si se cuentan con los recursos ociosos.

En Xeon Phi no se consiguieron resultados que superaran los obtenidos en CPU o GPU, pero se lograron rendimientos comparables. La baja *performance* de código serial resultó un fuerte limitante que se logró mitigar mediante el uso de paralelismo y un uso de arenas de memoria para mejorar la localidad de cache. Se determinaron características de la arquitectura MIC claves para atacar el problema de optimizar una aplicación de esta índole. Se identificó la necesidad de lograr una escalabilidad muy alta en cantidad de procesadores y la importancia de vectorizar apropiadamente el código. También se tuvo en consideración el comportamiento de las caches y la memoria, y cómo aprovecharlas efectivamente logra mejorar el rendimiento de maneras que no son viables para otras arquitecturas. Los resultados coinciden con los obtenidos en otros estudios de la literatura.

5.1. Trabajo a futuro

El trabajo pendiente inmediato que se observa y brindaría, inicialmente, una gran aceleración, es la realización de una versión híbrida CPU-GPU de LIO. Como pudimos observar, la resolución de grupos es totalmente independiente entre sí (salvo una reducción final). Esto se explotó en forma de paralelismo de múltiples placas GPU y en múltiples núcleos de procesador. Nada, *a priori*, previene que se puedan usar simultáneamente como recursos asimétricos para resolver de forma híbrida los sistemas. Los resultados vistos en la sección 4.5 muestran muy claramente que existe algún criterio para poder separar grupos en CPU y GPU y que, al poder realizarse en paralelo, podrían explotar todos los recursos disponibles y lograr grandes aceleraciones.

La exploración preliminar que se realizó con el Xeon Phi en este trabajo trajo resultados que lo ponen en una posición, en primera instancia, desventajosa con respecto a CPUs y GPUs de alta gama. El desempeño serial de los núcleos resultó un cuello de botella fundamental que, al menos durante este trabajo, no se pudo resolver completamente. Una versión híbrida CPU-Xeon Phi usando *offloading* resulta prometedora, pero las aceleraciones máximas alcanzables estarán fuertemente influenciadas por los costos de transferencia de memoria entre CPU y el Xeon Phi.

Un área de interés para estudiar en la aplicación es la construcción de las particiones con los mejores parámetros para cada arquitectura. Si la aplicación pudiera descubrirlos sin tener que cargarlos a mano, sería un gran progreso para evitar mucho tiempo de ajuste fino manual. Esto también se puede expandir para muchos de los parámetros que se optimizan en GPU (el tamaño de los bloques de todos los kernels CUDA) y en CPU (el estimador de costos de grupos). Esto permitiría que, simplemente con una recompilación del código, se pueda ejecutar la aplicación en generaciones próximas de GPU y CPU sin tener que reescribir estas secciones de código cada dos o tres años.

Además, pudimos observar al menos cuatro categorías de trabajos futuros que se desprenden para poder extender LIO y estudiar posibles mejoras de *performance* considerando las arquitecturas disponibles en el mercado.

Versiones híbridas

1. Hacer una versión híbrida CPU-GPU-Xeon Phi
2. Intentar otros modos de ejecución para el Xeon Phi, especialmente *offloading*.
3. Probar implementar en FPGA los cálculos de SCF para poder resolverlos por hardware en procesadores Atom.
4. Implementar una versión MPI para poder resolver una iteración distribuyendo a múltiples CPU/GPU/Xeon Phi en distintos nodos.
5. Explotar paralelismo de etapas a nivel más granular, como realizar las densidades en CPU y las matrices de Kohn-Sham en GPU.

Balance de cargas

1. Volver a analizar el algoritmo de partición de trabajos para hacer más equitativas las cargas sin tener que recurrir al balanceo durante la iteración.

2. Modificar el algoritmo de generación de grilla para que genere grupos de tamaños variable y así distribuir mejor las cargas.

3. Estudiar el problema de partición y sus funciones de costos para que sean acertadas tanto en CPU como en GPU.

Explotar más paralelismo

1. Mejorar la estrategia de paralelización para que emplee una cantidad de hilos de ejecución de acuerdo al problema (por ej. cubos medianos reciben más de un *thread*, pueden correrse menos al mismo tiempo pero más rápido).

2. Portear LIO a OpenCL para unificar el código entre arquitecturas.

3. Investigar el uso de bibliotecas BLAS (Magma, MKL, CUBLAS, ATLAS, etc.) para simplificar fragmentos del cálculo de SCF.

4. Analizar la posibilidad de usar CUDA Streams para intentar lograr kernels concurrentes y maximizar el uso de una placa.

5. Acelerar los pasos de SCF actualmente single-core y que no tienen implementaciones en GPU.

6. Acelerar el cálculo de las contribuciones de Coulomb para las fuerzas ínter atómicas, la sección de SCF más intensiva en cómputo luego de XC.

7. Analizar otras estrategias de paralelismo en sistemas distribuidos como MapReduce para simulaciones muy grandes.

Nuevas aplicaciones químicas

1. Experimentar el comportamiento de sistemas químicos muy grandes que no entren en memoria; como fraccionarlos.

2. Integrar LIO a otros sistemas de MM como GROMACS y CHARMM.

Bibliografía

[1] P. Hohenberg, W. Kohn, Inhomogeneous electron gas, Phys. Rev. 136 (1964) B864–B871.
URL http://link.aps.org/doi/10.1103/PhysRev.136.B864

[2] W. Kohn, L. J. Sham, Self-consistent equations including exchange and correlation effects, Phys. Rev. 140 (1965) A1133–A1138.
URL http://link.aps.org/doi/10.1103/PhysRev.140.A1133

[3] M. A. Nitsche, M. Ferreria, E. E. Mocskos, M. C. González Lebrero, GPU Accelerated Implementation of Density Functional Theory for Hybrid QM/MM Simulations, Journal of Chemical Theory and Computation 10 (3) (2014) 959–967.
URL http://pubs.acs.org/doi/abs/10.1021/ct400308n

[4] M. A. Nitsche, Aceleración de cálculos de estructura electrónica mediante el uso de procesadores gráficos programables., Master's thesis, Universidad de Buenos Aires (2009).

[5] R. Salomon-Ferrer, D. A. Case, R. C. Walker, An overview of the amber biomolecular simulation package, Wiley Interdisciplinary Reviews: Computational Molecular Science 3 (2) (2013) 198–210.
URL http://dx.doi.org/10.1002/wcms.1121

[6] P. S. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann Publishers Inc., 2011.

[7] J. L. Hennessy, D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, 5th Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[8] M. Amini, Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators, Theses, Ecole Nationale Supérieure des Mines de Paris (Dec. 2012).
URL https://pastel.archives-ouvertes.fr/pastel-00958033

[9] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, S. Midkiff, Cetus: A source-to-source compiler infrastructure for multicores, Computer 42 (12) (2009) 36–42.

[10] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), ACM, New York, NY, USA, 1967, pp. 483–485.
URL http://dx.doi.org/10.1145/1465482.1465560

[11] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, Cg: A system for programming graphics hardware in a c-like language, ACM Trans. Graph. 22 (3) (2003) 896–907.

[12] N. Wilt, The CUDA Handbook: A comprehensive guide to GPU Programming, Pearson Education, 2013.

[13] Nvidia, Cuda programming guide, Tech. rep., Nvidia (2015).

[14] Nvidia Corporation, NVIDIA's Next Generation CUDA Compute Architecture: Fermi, Tech. rep., Nvidia Corporation (2009).
URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[15] NVIDIA, Kepler GK110 whitepaper (2012).
URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[16] NVIDIA, Kepler GK110 family (2013).
URL http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf

[17] Top500, November 2001 list, http://www.top500.org/lists/2001/11/, accessed: 2014-08-22.

[18] R. Farber, CUDA application design and development, Elsevier, 2011.

[19] A. Tatourian, Nvidia gpu architecture and cuda programming environment (2013).
URL http://tatourian.com/2013/09/03/nvidia-gpu-architecture-cuda-programming-environ

[20] Nvidia, Cuda cublaslibrary, NVIDIA Corporation, Santa Clara, California.

[21] Nvidia, Cuda cufft library, NVIDIA Corporation, Santa Clara, California.

[22] N. Bell, J. Hoberock, Thrust: A productivity-oriented library for cuda, GPU Computing Gems 7.

[23] OpenMP 4.0 Specification.
URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[24] OpenACC 2.0 Specification.
URL http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf

[25] D. Patterson, The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges, Tech. rep., NVIDIA (2009).
URL http://www.nvidia.com/content/pdf/fermi_white_papers/d.patterson_top10innovationsinnvidiafermi.pdf

[26] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos, Demystifying gpu microarchitecture through microbenchmarking, in: Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on, IEEE, 2010, pp. 235–246.

[27] Intel, Intel xeon e7 8800 specifications, http://ark.intel.com/products/53580/Intel-Xeon-Processor-E7-8870-30M-Cache-2_40-GHz-6_40-GTs-Intel-QPI, accessed: 2014-08-22.

[28] Peter Glaskowsky, NVIDIA's Fermi: The First Complete GPU Computing Architecture, Tech. rep., Nvidia Corporation (2009).
URL http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIAFermi-TheFirstCompleteGPUComputingArchitecture.pdf

[29] Top500, June 2014 list, http://www.top500.org/lists/2013/06/, accessed: 2014-08-22.

[30] Top500, June 2013 list, http://www.top500.org/lists/2013/11/, accessed: 2014-08-22.

[31] Top500, November 2014 list, http://www.top500.org/lists/2014/06/, accessed: 2014-08-22.

[32] h. Rahman, Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers, 1st Edition, Apress, Berkely, CA, USA, 2013.

[33] S. U. Thiagarajan, C. Congdon, N. S, N. L. G, Intel xeon phi coprocessor quick start developers guide, Tech. rep., Intel Corporation (2013).
URL https://software.intel.com/sites/default/files/article/335818/intel-xeon-phi-coprocessor-quick-start-developers-guide.pdf

[34] G. Chrysos, Intel xeon phi coprocessor - the architecture, https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, accessed: 2014-10-14.

[35] J. Jeffers, J. Reinders, Intel Xeon Phi Coprocessor High Performance Programming, Newnes, 2013.

[36] A. D. Becke, A multicenter numerical integration scheme for polyatomic molecules, The Journal of Chemical Physics 88 (4) (1988) 2547–2553.
URL http://scitation.aip.org/content/aip/journal/jcp/88/4/10.1063/1.454033

[37] R. Stratmann, G. E. Scuseria, M. J. Frisch, Achieving linear scaling in exchange-correlation density functional quadratures, Chemical Physics Letters 257 (3–4) (1996) 213 – 223.
URL http://www.sciencedirect.com/science/article/pii/0009261496006008

[38] M. Harris, et al., Optimizing parallel reduction in cuda, NVIDIA Developer Technology 2 (4).

[39] J. Marsden, A. Tromba, Vector Calculus, W. H. Freeman, 2003.
URL http://books.google.com.ar/books?id=LiRLJf2m_dwC

[40] R. Gerber, Getting started with openmp*, Tech. rep., Intel Corporation (2012).

[41] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 2nd Edition, The MIT Press, 2001.

[42] Intel, Intel Xeon Processor E5-2620.
URL http://ark.intel.com/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI

[43] M. Yue, A simple proof of the inequality $ffd(l) \leq 11/9opt(l) + 1, \forall l$ for the ffd bin-packing algorithm, Acta Mathematicae Applicatae Sinica 7 (4) (1991) 321–331.
URL http://dx.doi.org/10.1007/BF02009683

[44] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, Vol. 3, Intel Corporation, 2014.

[45] OpenGroup, `clock_getres()` manual documentation (2004).
URL http://pubs.opengroup.org/onlinepubs/009695399/functions/clock_getres.html

Apéndice

A. EQUIPAMIENTO USADO PARA CORRER LAS PRUEBAS

Para realizar las pruebas de *performance* se utilizaron diversos nodos de cómputo, de diversos lugares que gentilmente proveyeron tiempo de CPU para poder correr estos trabajos.

Las pruebas de GPU se realizaron en las siguientes maquinas.

- CECAR - GPU2: 2 x AMD Opteron 6320 - 64GB DDR3 - Nvidia Tesla K40
- CECAR - GPU0: 2 x AMD Opteron 6320 - 64GB DDR3 - 2 x Nvidia Tesla M2090
- GIOL - GNODE01: 8 x AMD Opteron 6276 - 128GB DDR3 - 4 x Nvidia Tesla M2090
- FFYB - LAB8: Intel Core i5-3330 @ 3.0 GHz - 8GB DDR3 - 2 x GeForce GTX 780
- FFYB - LAB7: Intel Core i5-3330 @ 3.0 GHz - 8GB DDR3 - GeForce GTX 580 - GeForce GTX 780

Estas maquinas contaron con el siguiente software:

- Intel C/C++ Compiler 2013
- Intel Fortran Compiler 2013
- Intel MKL 2013
- Nvidia CUDA 6.0

Las pruebas de CPU se realizaron en las siguientes maquinas:

- SIASA - Host: 2 x Intel Xeon E5-2620 v2 - 32GB DDR3 - Intel Xeon Phi 5110P

Estas maquinas contaron con el siguiente software:

- Intel C/C++ Compiler 2015
- Intel Fortran Compiler 2015
- Intel MKL 2015

B. DESCRIPCIÓN DE MODELOS QUÍMICOS PROBADOS

A continuación detallamos los grupos moleculares que fueron usados como caso de estudio de la implementación realizada.

Grupo hemo

Este sistema esta compuesto por el grupo funcional Hemo incluido en las hemoproteínas, formado por una porfirina (con las cadenas laterales abreviadas), una prolina, un átomo de hierro Fe y una molecula de monóxido de carbono (CO) unida a este. Es un sistema muy utilizado en simulaciones QM/MM, aunque por su elevado costo computacional no se pueden realizar simulaciones de dinámica molecular cuánticas hasta el momento.

Contiene en total 48 átomos, donde su mayoría son relativamente pocos costosos de simular pero en particular el átomo de hierro requiere muchas funciones y puntos para describirlo apropiadamente dada su configuración electrónica. Un esquema de la molécula puede verse en la figura B.1.

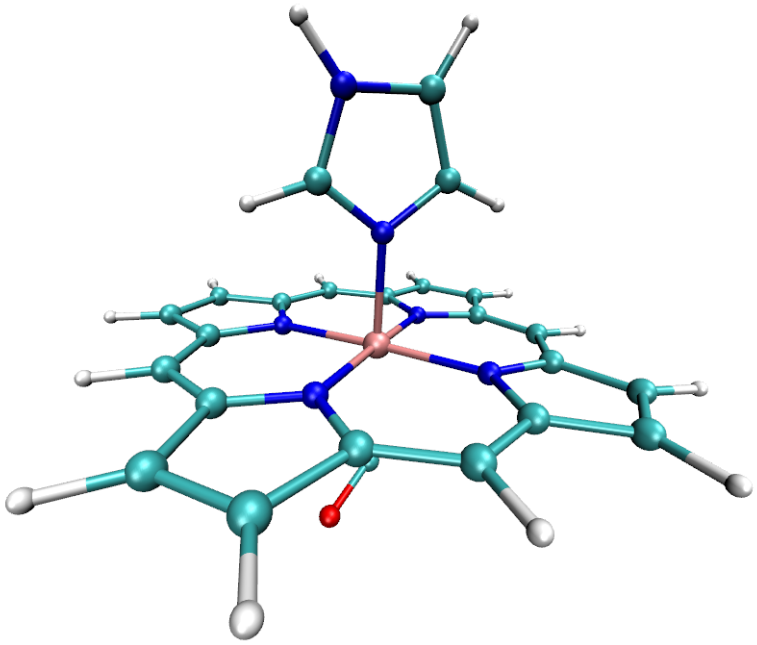


Fig. B.1: Render de hemo

Caroteno

El β-caroteno es el carotenoide más abundante en la naturaleza, por lo que da su nombre a todo un grupo de compuestos bioquímicos. Contiene 92 átomos, más que los que posee fullereno, pero los mismos son en su mayor parte hidrógenos, que al poseer un solo

electrón tienen grillas pequeñas y pueden ser simulados con poco esfuerzo computacional. El costo en funciones y puntos es intermedio entre el grupo hemo y fullereno. Un esquema puede verse en la figura B.2.

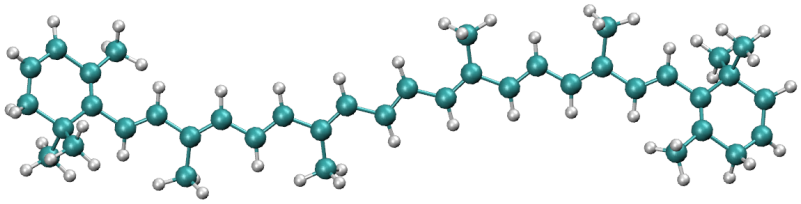


Fig. B.2: Render de caroteno

Fullereno C_{60}

El sistema Fullereno está compuesto por 60 átomos de carbono unidos en estructura de icosaedro truncado. Esta relacionado fuertemente con los nanotubos de carbono, compuestos de gran interés en la ciencia de los materiales. Es el sistema de mayor costo computacional estudiado, dada la proximidad de los átomos y los enlaces en los orbitales p. Se modela usando la base de funciones DZVP. Un render de la molécula puede verse en la figura B.3.

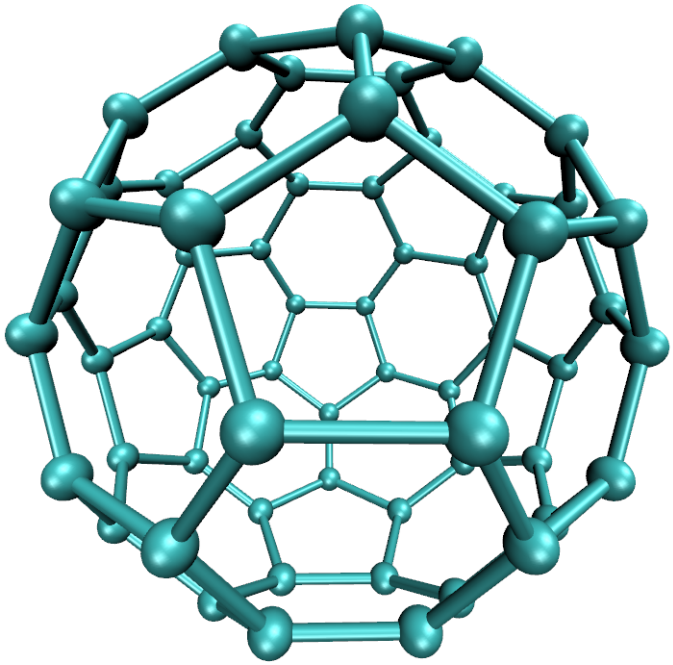


Fig. B.3: Render de fullereno C_{60}