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

Ampliando las capacidades de una aplicación de cómputo QM/MM utilizando GPU

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

Eduardo Andrés Perez Leale

Director: Dr. Esteban Mocskos
Codirector: Dr. Mariano Camilo González Lebrero
Buenos Aires, 2019

RESUMEN

Hoy en día, la simulación numérica es una disciplina fundamental dentro de una gran cantidad de áreas de la ciencia y la tecnología. El uso de estas técnicas permite validar modelos teóricos así como también brindar información detallada (macro y microscópica) del proceso simulado, complementando el uso de las técnicas experimentales tradicionales. Este trabajo se enfoca en modificar el software de estructura electrónica `LIO`. Este código se basa en la teoría de los funcionales de la densidad (DFT) y permite estimar propiedades de sistemas moleculares de manera muy eficiente mediante el empleo de tarjetas gráficas (GPUs) en las partes más demandantes del cálculo. En el esquema de DFT la calidad (así cómo el mayor costo) está dada por el cálculo de lo que se llama "energía de intercambio y correlación"(EXC), por ello LIO realiza los calculos asociados a estos terminos en GPU. Para el cálculo de EXC existen múltiples "recetas" (llamadas funcionales de intercambio y correlación) que pueden ser convenientes para el tratamiento de diferentes sistemas o propiedades. La versión original de `LIO` incluía un único funcional (llamado PBE) lo que limitaba sus capacidades.

`Libxc` es una biblioteca de funcionales de intercambio y correlación para DFT diseñada para realizar simulaciones utilizando únicamente procesadores generales por lo que su implementación en GPU puede no resultar eficiente.

El desafío planteado en esta tesis es lograr utilizar `Libxc` desde `LIO` sin que impacte en su performance. Para lograrlo se realizaron los siguientes pasos:

1. Se integró `LIO` con `Libxc` sin modificar la estructura general de `Libxc` para poder realizar simulaciones con mayor diversidad de funcionales.

2. Se seleccionó un subconjunto de funcionales de `Libxc` y se realizó su implementación en GPU, para lo cual se realizó una modificación en la arquitectura y los algoritmos de `Libxc`. Se comenzó por el funcional PBE para poder validar el correcto funcionamiento de la aplicación. Luego, se implementó el resto de los funcionales seleccionados y se obtuvo una versión de `LIO` integrada con `Libxc` para GPU.

3. Se analizó el uso de los recursos de la tarjetas gráficas con el fin de poder aprovechar al máximo la arquitectura de las GPUs. Se buscó cambiar la estrategia de paralelización en la estructura del cómputo de `Libxc` para aprovechar la mayor cantidad de memoria que poseen las tarjetas gráficas para almacenar más resultados intermedios.

4. Se exploró el uso de diversos compiladores provistos por el proyecto `LLVM` en un intento de realizar una traducción automática del código fuente de la biblioteca `Libxc` para GPUS.

Como resultado de este trabajo se pudo lograr una ampliación de las capacidades del cómputo del software `LIO` al vincularlo con la biblioteca `Libxc` con un impacto menor en la performance que va de 19 % en la primer etapa, al 4 % en la etapa final. También se lograron identificar los puntos clave en la implementación que permitirían continuar con las tareas de optimización. Las mejoras permiten que `LIO` pueda ser utilizada con una amplia gamma de algoritmos que realizan los cálculos de intercambio de energía y correlación lo que consecuentemente permite realizar simulaciones en condiciones más variadas.

Como corolario de este trabajo se pudo estandarizar (aunque no automatizar) un procedimiento que permite realizar una traducción de los funcionales de `Libxc` para que puedan ser ejecutados en GPU.

Palabras claves: QM/MM, DFT, `LIO`, `Libxc`, GPGPU, CUDA, LLVM, *scheduling*.

Índice general

1. INTRODUCCIÓN

Hoy en día, la simulación numérica es una disciplina fundamental dentro de una gran cantidad de áreas de desarrollo científico y tecnológico. El uso de estas técnicas permite validar modelos teóricos así como también brindar información detallada (macro y microscópica) del proceso simulado, complementando el uso de las técnicas experimentales tradicionales.

En el ámbito de la química existen diferentes metodologías que permiten simular procesos de interés. Hay dos que se destacan por su uso:

I) Los métodos basados en la mecánica cuántica, que proporcionan una descripción detallada de la estructura electrónica del sistema.

II) Los métodos basados en la mecánica molecular, en los cuales 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 fenómenos 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 (como electrones y protones) pueden (y deben en algunos casos) ser descriptas como 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, 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 (como la energía) puede ser representado como un funcional de la densidad (de allí el nombre de esta teoría).

Además, propusieron la dependencia de la energía del sistema como 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 (Electron Repulsion Integral - ERI) 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 como el costo computacional de este tipo de simulaciones. Uno de los puntos principales en los cuales se han centrado diversos trabajos, consiste en disminuir el tiempo insumido en el cómputo del término $E_{XC}[\rho]$ [2].

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 [3]. 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 como 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 como:

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

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, 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 [2, 4], el cual está optimizado 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}.

1.2. Libxc

`Libxc` [6] es una biblioteca de funcionales de intercambio y correlación en el marco de la teoría funcional de la densidad. El objetivo es proporcionar un conjunto portátil, bien probado y confiable de funcionales de intercambio y correlación que puedan ser utilizadas por otros programas.

En `Libxc` se pueden encontrar diferentes tipos de funcionales: LDA, GGA, híbridos y mGGA. Estos funcionales dependen de la información local, en el sentido de que el valor del potencial en un punto dado depende únicamente de los valores de la densidad (y el gradiente de la densidad y la densidad de energía cinética, para los casos de GGA y mGGA), en un punto dado puede calcular la energía de intercambio y correlación:

$$
\begin{aligned}
E_{\mathrm{XC}}^{\mathrm{LDA}} &= E_{\mathrm{XC}}^{\mathrm{LDA}}[n(\vec{r})] \\
E_{\mathrm{XC}}^{\mathrm{GGA}} &= E_{\mathrm{XC}}^{\mathrm{GGA}}[n(\vec{r}), \vec{\nabla} n(\vec{r})] \\
E_{\mathrm{XC}}^{\mathrm{Hyb}} &= a_x E^{\mathrm{EXX}} + E_{\mathrm{XC}}^{\mathrm{GGA}}[n(\vec{r}), \vec{\nabla} n(\vec{r})] \\
E_{\mathrm{XC}}^{\mathrm{mGGA}} &= E_{\mathrm{XC}}^{\mathrm{mGGA}}[n(\vec{r}), \vec{\nabla} n(\vec{r}), \nabla^2 n(\vec{r}), \tau(\vec{r})]
\end{aligned}
$$

Donde $n(\vec{r})$ es el valor de la densidad en las coordenadas $\vec{r}$, $\tau(\vec{r})$ es la energía cinética en el mismo, $\vec{\nabla}$ denota el gradiente, y $\nabla^2 n(\vec{r})$ el laplaciano.

También puede calcular los gradientes de esta energía y para la mayoría de los funcionales también derivadas de orden superior.

El código fuente de los funcionales de `Libxc` se encuentra implementado en `C` generado a través de un programa de manipulación de objetos matemáticos llamado `Maple`.

Un factor importante a tener en cuenta es que `Libxc` está diseñada para ser ejecutada en CPU y utiliza números de doble precisión para realizar los cálculos de los funcionales. Esto implica que el mayor esfuerzo del presente trabajo estará centrado en, no solo realizar una traducción automática del código de los 180 funcionales para que funcionen en GPU, sino también que esa traducción utilice de manera eficiente los recursos que proveen las arquitecturas de cómputo paralelo de las placas de video.

En resumen, el objetivo del presente trabajo es extender la funcionalidad de `LIO` poniendo a disposición los 180 funcionales que provee `Libxc`, modificados de modo que puedan ser ejecutados eficientemente en tarjetas gráficas (GPUs).

2. ARQUITECTURAS EN PROFUNDIDAD

2.1. GPGPU con CUDA

Una de las arquitecturas analizadas en este trabajo es la arquitectura GPU desarrollada por Nvidia, conocida como `CUDA` por las siglas en inglés de *Compute Unified Device Architecture*. `CUDA` surge naturalmente de la aplicación del hardware desarrollado para aplicaciones de uso intensivo de 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 nativamente en la GPU, y se podían encadenar entre sí, uno por pixel en la pantalla [7]. Este paralelismo es el desarrollo fundamental que llevó a las GPU a poder procesar operaciones gráficas órdenes de magnitud más rápido que los CPU normales.

En el 2006, Nvidia introduce la arquitectura `G80`, que es la primer GPU que deja de resolver únicamente problemas de gráficos para pasar a ser un motor genérico. Contaba con un set de instrucciones consistente para todos los tipos de operaciones que realizaba (geometría, vertex y pixel shaders) [8]. Como subproducto de esto, la GPU pasa a tener procesadores simétricos más simples 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 [9].

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 [10, 11] de memoria para almacenar resultados. Las GPUs cuentan 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 un 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 [12]. 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. Una comparativa del poder de cómputo teórico entre GPUs y CPUs puede verse en la figura 2.1.

Sin embargo, para poder explotar la arquitectura CUDA, los programas deben ser

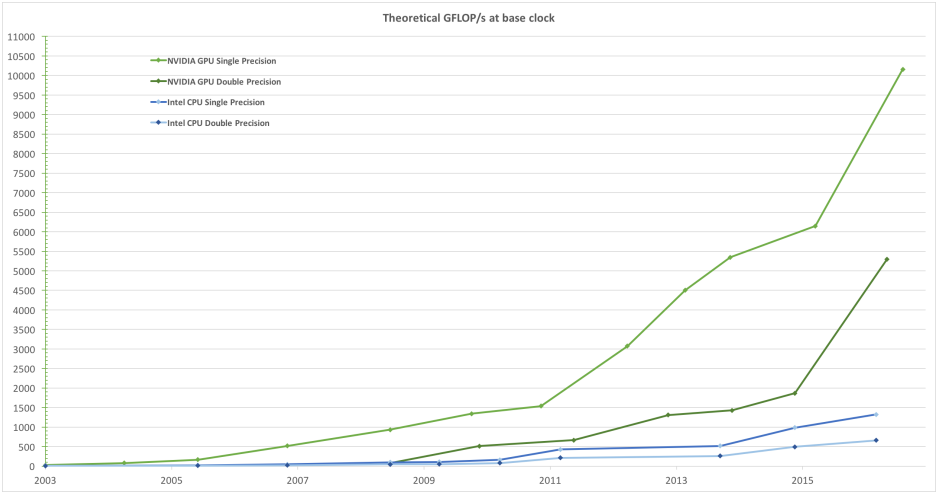

Fig. 2.1: Picos teóricos de *performance* en GFLOPS/s. Reproducido de [9].

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 la arquitectura `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 [8, 13, 14].

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 científico [15, 16].

Además, se siguen desarrollando nuevas soluciones que consideran la portabilidad entre plataformas: las bibliotecas como `Thrust` [17], `OpenMP` 4.0 [18] y `OpenACC` 2.0 [19] son herramientas que buscan generar 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 pendientes 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.

2.1.1. 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 en detalle la organización definida en la arquitectura `Fermi` y en las secciones subsiguientes daremos los detalles más importantes de las arqui-

tecturas `Kepler`, `Maxwell` y `Pascal`. Por último realizaremos una breve comparación de las características principales de cada arquitectura.

Las arquitecturas de las GPUs se centran en el uso de una cantidad escalable de procesadores *multithreaded* denominados *Streaming Multiprocessors* (SMs). Un multiprocesador está 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 [14].

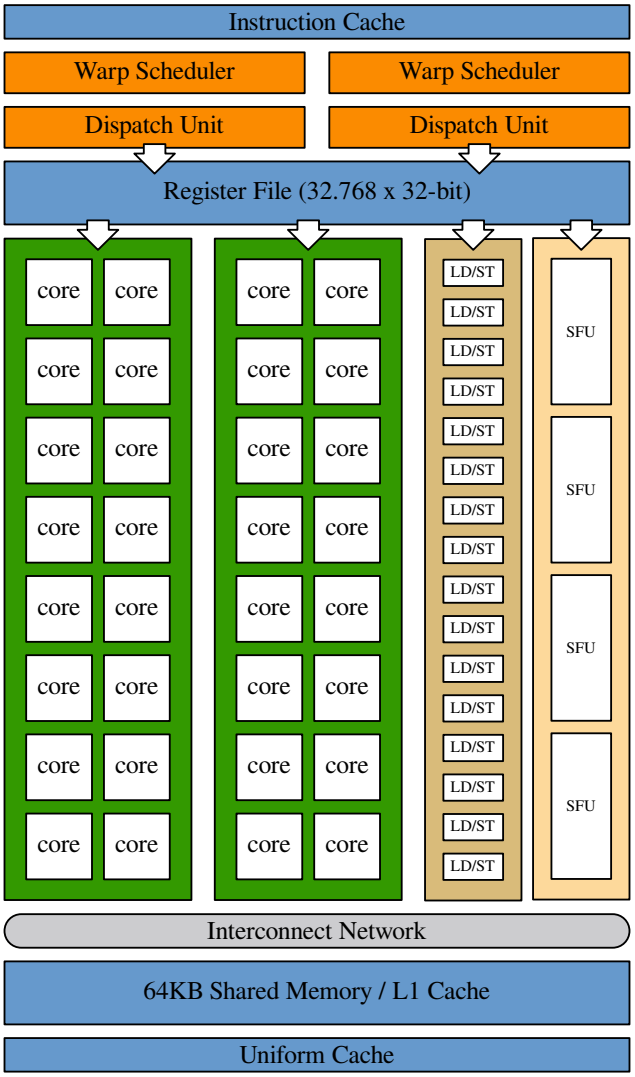

Fig. 2.2: Diagrama de bloques del SM de GF100 Fermi. Basado en [10].

Los SMs (figura 2.2) 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, 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) [20]. 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. 2.3: Diagrama de bloques de GF100 Fermi. Tomado de [10].

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 2.3.

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.

2.1.2. 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 2.4. 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 con los que cuenta un SM (oscila entre 16535 y 65535 registros). Debido a la baja latencia de acceso, 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 éste. Es una memoria muy rápida, on-chip, y que tarda aproximadamente 40 ciclos de acceso [21]. Esta memoria es compartida con la cache L1, la cual tiene capacidad de entre 16 KB y 64 KB configurable por software. 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 acceso a la memoria [13].

La memoria global es la memoria principal fuera del chip de la GPU. Ésta 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 [21], 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 de la placa como en la memoria principal del procesador. El driver de CUDA mantiene 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.

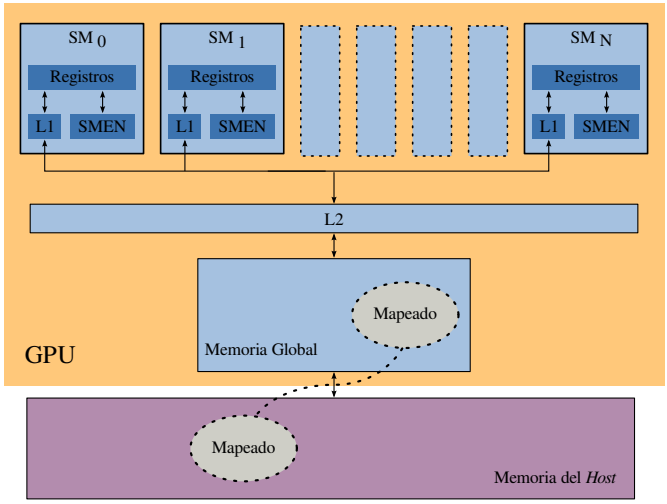

Fig. 2.4: Esquema de la jerarquía de memorias en GPU, detallando de las disponibles en cada SM, la memoria compartida (SMEN), la L2 global, la memoria de la placa (global) y la mapeada entre el *host* y la placa de video. Tomado de [13].

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
- Cache L2
- Cache constante
- Cache de textura

La cache L1 es dedicada por SM. Esta cache fue introducida en la arquitectura Fermi y su diseño hace que también esté 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 ésta [10].

La cache constante es una cache sobre la memoria global dedicada solamente a lecturas de memoria. Ésta 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 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.

2.1.3. 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 con que cuenta cada SM. Para compartir información entre ellos, se puede utilizar la memoria compartida o las instrucciones de comunicación de *threads intra-warp* (solo disponibles a partir de Kepler [11]).

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 éste. 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 [13].

2.1.4. Arquitectura Kepler

La arquitectura Kepler [11] fue presentada a fines del año 2012 y posee más del doble de transistores que su arquitectura predecesora (Fermi). Uno de los principales objetivos del diseño de la arquitectura Kepler fue el de mejorar el consumo de energía. Su procesador de 28 nm juega un rol fundamental en la reducción del consumo de energía, pero a su vez se realizaron varias modificaciones a la arquitectura de la GPU para lograr reducir aún más el consumo y lograr mantener un buen rendimiento, dando como resultado una mejora de 3x por watt con respecto a la arquitectura Fermi.

Otro de los objetivos del diseño fue el de incrementar de forma significativa el rendimiento de la aritmética de doble precisión. En efecto, la arquitectura Kepler proporciona más de 1 TFLOP en el rendimiento de operaciones de doble precisión con más del 80 % de eficiencia en las operaciones de multiplicaciones matriciales de doble precisión (DGEEM) contra $60 - 65$ % en la arquitectura Fermi.

La arquitectura Kepler [11] es bastante parecida a la de Fermi, pero con una diferencia importante: La cantidad de SPs por Streaming Multiprocessor (llamados SMX en Kepler) se incrementan de 32 a 192. La frecuencia del reloj se decrementa de 1.5 GHz a 1 GHz continuando la tendencia de incrementar el poder de cómputo al utilizar mas núcleos que corren a menor frecuencia de reloj. Los puntos claves de la arquitectura incluyen:

- Un nuevo procesador SMX.
- Un subsistema de memoria mejorado, el cual ofrece capacidades adicionales de cache, más ancho de banda en la jerarquía y un sistema de I/O a memoria totalmente rediseñado y sustancialmente más rápido.
- Soporte de hardware en todo el diseño el cual permite nuevas capacidades en el modelo de programación.

Cada una de las unidades SMX contiene 192 núcleos CUDA de simple precisión, y cada núcleo tiene incorporadas unidades lógico-aritméticas. La arquitectura también respeta la norma IEEE 754-2008 respecto a la aritmética de simple y doble precisión que fue introducida en Fermi, incluyendo además las operaciones conocidas como multiply-add (FMA), que pueden resolver una multiplicación y una suma como si fuera una única instrucción. Las unidades SMX de Kepler también mantienen lo que se conoce como unidades de función especial (SFUs) las cuales sirven para aproximar de manera rápida operaciones transcendentales, con un total de 32 SFUs (8 veces más que en Fermi).

El warp scheduler de los SMX también presenta modificaciones. Bajo este esquema se puede programar un warp completo (32 threads) cada ciclo, cuando en Fermi el scheduler solo se podía programar 16 threads. Cada SMX ofrece cuatro schedulers para los warps y ocho unidades de despacho, permitiendo de esta manera que cuatro warps sean ejecutados simultáneamente. Por lo tanto, se puede programar la ejecución de 128 threads en

Fig. 2.5: Esquema de organizacion de los SMX en la arquitectura Kepler, aquí podemos ver los 192 núcleos de simple presición. Tomado de [11].

cada ciclo para los 192 núcleos. A diferencia de la arquitectura Fermi, la cual no permite que instrucciones de doble precisión sean emparejadas con otras instrucciones, la arquitectura Kepler permite que las instrucciones de doble precisión se emparejen con otras instrucciones.

El número de registros de memoria a los cuales puede acceder un thread se cuadriplicó, permitiendo que cada thread tenga acceso hasta 255 registros. Los kernels de CUDA que utilizan una gran cantidad de registros ahora pueden lograr aceleraciones sustanciales como resultado de la mayor cantidad de registros disponibles por subproceso. Para mejorar aún más el rendimiento, Kepler implementa un nuevo mezclador de instrucciones, el cual permite compartir datos entre los threads de un warp. El mezclador soporta referencias indexadas y ofrece una mejora en el uso de la memoria compartida ya que las operaciones de *load* y *store* son llevadas a cabo en un solo paso.

Subsistema de memoria

La jerarquía de memoria en la arquitectura Kepler está organizada de manera similar a la de la arquitectura Fermi. La arquitectura soporta accesos a memoria unificados para *loads* y *stores* a través de una memoria cache L1 por cada multiprocesador SMX. La arquitectura Kepler también permite mediante directivas de compilador el uso de una memoria cache adicional para lectura de datos. Así como en la arquitectura Fermi, cada SMX tiene 64 kB de memoria *on-chip* la cual puede ser configurada como 48 kB de memoria compartida con 16 kB de cache L1, o como 16 kB de memoria compartida con 48 kB de memoria cache L1. Además, la arquitectura Kepler permite 32 kB / 32 kB divididos entre

memoria compartida y memoria cache L1.

El ancho de banda de la memoria compartida para operaciones que superan los 64 B fue duplicado en comparación con los SM de la arquitectura Fermi. Además de la memoria cache L1, la arquitectura Kepler introduce una memoria cache de 48 kB para datos de solo lectura. En la generación Fermi, la memoria cache de solo lectura era accesible por la unidad de texturas. En Kepler, la cache es accesible directamente por el SM para operaciones generales de carga de datos. Por otro lado, también dispone de una memoria cache L2 dedicada de 1536 kB, el doble que en las arquitecturas Fermi. El ancho de banda de la memoria cache L2 también se duplicó al doble con respecto a la arquitectura Fermi.

La eficiencia de la memoria de textura fue mejorada de forma significativa, cada una de las unidades SMX contiene 16 unidades de filtrado de texturas (cuatro veces más que la arquitectura Fermi). Además, la arquitectura Kepler modifica la forma en que se maneja el estado de las texturas. En la arquitectura Fermi, para que la GPU referencie a una textura, había que asignar un espacio de memoria de tamaño fijo previo a la ejecución de un kernel. Dicho espacio de memoria limita la cantidad de texturas sobre las cuales un kernel puede leer en tiempo de ejecución. En la arquitectura Kepler, se elimina este paso de asignación ya que el estado de la textura se almacena como un objeto en memoria y es el hardware el que se encarga de buscar los objetos bajo demanda. Esto elimina cualquier limitación en el número de texturas que pueden ser referenciadas por un kernel.

Mejoras principales de la arquitectura Kepler

Las siguientes características de la arquitectura Kepler permiten un incremento en la utilización de la GPU, simplifican el diseño de programas que realizan cómputos en paralelo y ayuda a simplificar la creación de programas para que sean compatibles tanto en computadoras de escritorio como en supercomputadoras.

Paralelismo dinámico

El paralelismo dinámico es una nueva característica de la arquitectura Kepler el cual permite a la GPU generar trabajos nuevos por su cuenta, sincronizar los resultado y controlar la programación de esos trabajos a través de hardware dedicado. Todo sin tener que incluir el uso del CPU del host. En arquitecturas anteriores, todo el trabajo que realizaban las GPUs era invocado desde el CPU del host, se esperaba a su finalización y por último se copiaban los resultados nuevamente al host. En las arquitecturas Kepler, un kernel puede realizar llamadas a otros kernels, crear streams, eventos y manejar las dependencias necesarias para procesar trabajo adicional sin necesidad de interactuar con la CPU del host. Esta innovación ayuda a los desarrolladores a crear y optimizar programas que tienen una gran dependencia de datos, también permite que una mayor cantidad de programas sean ejecutados directamente en la GPU. Esto permite que el CPU del host se encuentre libre para realizar tareas adicionales, o para que los sistemas se puedan configurar con CPU menos potentes para realizar la misma cantidad de trabajo. Esta nueva característica de los kernel de poder realiazar trabajos adicionales le permite a los programadores realizar un mejor balance de los recursos en el diseño de sus programas y también poder concentrarse en las áreas del problema que requieren mayor poder de cómputo.

Hyper-Q

Hyper-Q permite que múltiples núcleos de CPU del host puedan utilizar simultáneamente la misma GPU para realizar trabajos, de este modo se incrementa de forma significativa la utilización de la GPU y al mismo tiempo se reduce tiempo en el que el CPU del host se encuentra ocioso. Con Hyper-Q se incrementa el número total de conexiones (conocidas como *work queues*) entre el host y el distribuidor de trabajos de CUDA (CUDA Work Distributor o CWD) permitiendo un total de 32 conexiones en paralelo (en la arquitectura Fermi solo existe una sola conexión disponible).

NVIDIA GPUDirect

NVIDIA GPUDirect se refiere a la capacidad que tienen las GPUs dentro de una computadora, o las GPUs en diferentes servidores en una red, de intercambiar datos sin tener que utilizar la memoria principal del host. Esta nueva característica mejora el rendimiento y reduce la latencia en la transferencia de datos, por otro lado reduce el ancho de banda permitiendo que la GPU se encuentre libre para realizar otras tareas.

2.1.5. Arquitectura Maxwell

La primera generación de placas de video con arquitectura Maxwell fueron presentadas a principios de el año 2014 y contiene un conjunto de mejoras diseñadas para obtener un mayor rendimiento energético por watt. La primera GPU basada en la arquitectura Maxwell fue diseñada para trabajar en ambientes en donde el recurso de energía es limitado como en computadoras portátiles y computadoras personales pequeñas [22].

La arquitectura Maxwell presenta un nuevo diseño para los Streaming Multiprocessors (SM) el cual mejora de manera drástica el rendimiento por Watt. Mejoras para controlar la lógica de particionado, el balance de los trabajos, los schedulers, el número de instrucciones emitidas por ciclo de reloj entre otras permite que los SM de la arquitecura Maxwel (llamados SMM) superen ampliamante el rendimiento de los SM de la arquitectura Kepler.

La organización de los SM también sufrió modificaciones (ver figura 2.6). La cantidad de núcleos por SM fue reducida a una potencia de dos, dejando un total de 128 núcleos. Cada uno de los SMM se encuentra dividido en cuatro bloques de procesamiento los cuales a su vez tienen su propio buffer de instrucciones, su propio scheduler y cuenta con 32 núcleos. Este nuevo esquema de división simplifica el diseño, la lógica del scheduler, ahorra energía y reduce la latencia de los cómputos. Estos nuevos bloques de procesamiento comparten cuatro filtros de texturas y una memoria cache de texturas. Se combinaron las funciones de memoria cache L1 con las de memoria cache de texturas, y la memoria compartida se encuentra en una unidad separada la cual puede ser accedida por cada uno de los bloques.

Aunque el número total de núcleos por SM disminuyó de 192 en la arquitectura Kepler a 128 en la arquitectura Maxwell, gracias a las mejoras introducidas en esta nueva generación, el rendimiento por SMM es generalmente mayor al 10 % con respecto a los SMX de la arquitectura Kepler. Los SMM mantienen el mismo número de instrucciones por ciclo y reducen la latencia en las operaciones aritméticas con respecto a la arquitectura Kepler.

Así como los SMX, cada SMM contiene cuatro schedulers para los warps, pero a diferencia de los SMX, a cada uno de los núcleos SMM se les asigna un scheduler en particular. Dado que la cantidad de núcleos es una potencia de dos, el esquema de asignación de schedulers se ve simplificado ya que cada uno de los schedulers de los warps son asignados a

Fig. 2.6: Esquema de organización de los SMM en la arquitectura Maxwell. Tomado de [22].

un conjunto de núcleos de tamaño igual a la longitud de un warp.

Las capacidades de cómputo de los SMM de la arquitectura Maxwell son similares en muchos aspectos a las capacidades de cómputo de los SMX de la arquitectura Kepler, pero con algunas mejoras clave que apuntan a mejorar la eficiencia. El número de registros de memoria y la cantidad máxima de *warps* en un SMM son los mismas que en un SMX (64K con registros de 32-bits y 64 *warps*), así como el número de registros de memoria on-chip por thread (255). Sin embargo, el número máximo de threads activos por MP fue duplicado a 32 lo que da como resultado una mejora para los kernels que utilizan bloques con pocos threads (64 o menos). Otra mejora que presentan los SMM es que la latencia en la dependencia de instrucciones aritméticas fue reducida significativamente.

Los SMM tienen 64 kB de memoria compartida por SM, a diferencia de las arquitecturas Fermi y Kepler en las cuales los 64 kB de memoria son compartidos entre la memoria cache L1 y la memoria compartida. Esto es posible gracias a que la arquitectura Maxwell combina la funcionalidad de la memoria cache L1 y la de la memoria cache de texturas en una sola unidad.

La arquitectura Maxwell mantiene el *Paralelismo Dinámico* que fue introducido en la arquitectura Kepler que permite que los threads que se ejecutan en la GPU puedan ejecutar kernels sin necesidad de pasar por el CPU del host.

El tamaño de la memoria cache L2 fue incrementado a 2048 kB(contra 256 kB en la arquitectura Kepler) lo que permite tener una capacidad adicional de almacenamiento que puede ser utilizada para texturas y operaciones atómicas entre otras cosas. Al tener más memoria cache se reducen los accesos a la memoria principal y por lo tanto se reducen los cuellos de botella provocados por los accesos a memoria.

En resumen, toda la arquitectura fue rediseñada, los flujos de datos fueron optimizados, el ahorro de energía mejoró significativamante, por solo nombrar algunas de las modificaciones. Como resultado de todo este esfuerzo, la arquitectura Maxwell es capaz de brindar un rendimiento dos veces mayor al de sus predecesoras.

2.1.6. Arquitectura Pascal

En el año 2016 NVIDIA presenta la arquitectura Pascal la cual incluye tres nuevas características claves con respecto a las generaciones anteriores: Stacked DRAM, Memoria Unificada y NVLink. Stacked DRAM o Memoria 3D es una nueva característica por la cual se incluyen chips de memoria en el mismo módulo en donde se encuentra la GPU. Esto le permite a la GPU obtener datos de memoria mucho más rápido, incrementando el rendimiento y la performance. De esta manera es posible desarrollar una GPU más compacta y de mayor potencia para dispositivos pequeños. Como resultado, el ancho de banda de la memoria y la capacidad de cómputo es más de dos veces mayor, por otro lado el ahorro de energía es cuatro veces mayor que el de las arquitecturas predecesoras.

La *Memoria Unificada* consiste en permitirle al CPU del host acceder directamente a la memoria de la GPU y viceversa, de esta forma los desarrolladores se ahorran el paso de tener que reservar memoria entre ambos a la hora de escribir sus programas.

Con `NVlink` se incrementa el ancho de banda entre el CPU del host y la GPU, permitiendo que la transferencia de datos se dé a velocidades mayores a los 80 GB/s, comparado con los 16 GB/s de las arquitecturas predecesoras. De esta forma, la velocidad de las aplicaciones se encuentra menos restringida por la velocidad en la transferencia de los datos entre el CPU del host y la GPU.

2.1.7. Diferencias entre Tesla, Fermi, Kepler, Maxwell y Pascal

En la tabla 2.1 se presenta una comparación de los recursos que están más directamente relacionados a la *performance* de una 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 las CPU, las arquitecturas GPU 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).

Características	Tesla (GT200)	Fermi (GF100)	Kepler (GK110)	Maxwell (GM107)	Pascal (GP100)
Año introducción	2006	2010	2012	2014	2016
Transistores	1400 millones	3000 millones	3500 millones	7100 millones	153000 millones
Tecnología fabricación	65 nm	40 nm	28 nm	28 nm	16 nm (FinFET)
SMs	30	16	15	32	64
SP / SM	8	32	192	128	64
Caché L1	-	16 kB and 48 kB	16 kB, 32 kB and 48 kB	64 kB	64 kB
Caché L2	-	768 kB	1536 kB	2048 kB	4096 kB
Memoria Shared/SM	16 kB	16 kB and 48 kB	16 kB, 32 kB and 48 kB	96 kB	64 kB
Registros/Thread	63	63	255	255	255
Pico GFLOPS Simple	933	1345	3977	4612	10600
GFLOPS/Watt	3,95	5,38	15,9	28,7	57,3

Tab. 2.1: Tabla comparativa de las características más prominentes de las diferentes versiones de arquitecturas CUDA.

La medida usada por Nvidia para publicitar la *performance* de estos dispositivos y poder compararlos entre sí, y contra CPU, son los GFLOPS. Ésta mide cuántas operaciones de punto flotante de precisión simple se pueden realizar por segundo. Los GFLOPs son utilizados también por los clusters en el ranking TOP500, en donde se ordenan de acuerdo a la *performance* medida usando un benchmark pre-establecido (i.e. `LINPACK`). No solo es notable cómo se cuadruplicó la *performance* (teórica) en solamente seis años, sino que aún más importante es como mejoró la *performance* por Watt. 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 200 W 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) [23].

2.1.8. 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 ejecutar, por lo cual la paralelización no debe estar demasiado atada a éste, 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 BLAS de álgebra lineal [10]).

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 como 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, como medir *performance* y utilización, y cómo los cambios realizados impactaron en las distintas generaciones de dispositivos.

2.1.9. 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. La intensidad aritmética 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 datos 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 GPU 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 estos casos de uso.

El ítem 3 menciona que los problemas que mejor se pueden tratar en esta arquitectura se pueden representar como operaciones entre vectores 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 GPU, pueden generar un importante 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.

Mientras tanto, el ítem 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 del host. 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 GPUs, 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 [13], pero siguen siendo copias de memoria a través de los buses (aunque asincrónicas).

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

2.1.10. Diferencia entre CPU y GPU, procesadores especulativos

Hasta ahora, solo se consideraron a las 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 la GPU se puede apreciar mejor comparándola contra los motivos de la evolución de CPU, y analizando los problemas que se fueron enfrentando los diseños siguiendo la historia de los componentes que fueron apareciendo en éstos.

Lo clave es observar el siguiente patrón que se da en el universo CPU: *"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 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 y de la probable reutilización de datos. En un CPU moderno (por ejemplo, el Intel Xeon E7-8800 [24]) 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 diseñados para resolver constantemente operaciones muy bien definidas (instruccio-

nes de punto flotante con precisión simple). Comparativamente con una 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 SPs deben estar sincronizados entre sí. Pero la gran ventaja se encuentra en la cantidad.

Una CPU cuenta con pocas ALU por core, dependiendo de la cantidad de cores y del tamaño de sus operaciones SIMD, suele ser alrededor de 16 cores por *die* de `x86` es el tope de línea ofrecido actualmente, procesando de a 32 bytes simultáneamente. Una GPU cuenta con miles de ALUs en total (más de 2500 CUDA cores en una `Tesla K20` [12]). 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, las GPUs disponen de pocas unidades de soporte del procesamiento. Éstas no disponen de pipelines especulativos, el tamaño de las caches son mucho más pequeñas que las de una 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 el diseño de la arquitectura CUDA, buscaron resolver el problema del cómputo masivo pensando en hacer más cuentas a la vez recalculando 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 enfocarse de lleno en una sola problemática. Pero con las arquitecturas GPU, 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 [25].

Las arquitecturas Tesla, Fermi y Kepler conciben el diseño de un procesador de alto desempeño. Su meta principal es poder soportar un alto nivel 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 está 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.

2.1.11. Idoneidad para la tarea

El problema enfrentado en este trabajo cuenta con múltiples operaciones matemáticas y gran volumen de cálculo. En particular, las operaciones del cálculo de funcionales constituyen los principales cuellos de botella en la aplicación analizada.

Para obtener los valores numéricos de densidad buscados en los distintos puntos, se deben obtener las derivadas primera y segunda, 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.

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 [4], por lo que se prosiguió su análisis como un camino prometedor [26].

3. HERRAMIENTAS

3.1. Herramientas

En esta sección vamos a mencionar las distintas herramientas que se utilizaron para intentar realizar la traducción automática del código de los funcionales de `Libxc` para que se puedan ejecutar en GPU. Dado que estamos intentando realizar una traducción automática de código, necesitamos herramientas que generen código para arquitecturas de hardware específicas (en nuestro caso para `CUDA`). Una de las más conocidas es el conjunto de herramientas provistas por `LLVM`. Se optó por utilizar `LLVM` por varias razones, las cuales vamos a describir más adelante. Se va a explicar qué es el compilador `clang` y su conjunto de bibliotecas, que son de especial interés para este trabajo. Por último, vamos a profundizar en la herramienta utilizada para generar el código de máquina específico para la arquitectura `CUDA` a través del uso del compilador `llc`.

3.1.1. LLVM

La infraestructura del compilador `LLVM` [27] tiene sus orígenes en la Universidad de Illinois y fue desarrollado con el objetivo de tener un compilador moderno que soporte simultáneamente compilación estática y dinámica para lenguajes de programación arbitrarios. El projecto `LLVM` comenzó en el año 2000 bajo la supervisión de Chris Lattner y Vikram Adve. El nombre `LLVM` proviene de *low level virtual machine*, pero a medida que crecía el alcance del proyecto, fue modificado para no generar confusión dado que el proyecto tenía muy poco que ver con las máquinas virtuales tradicionales. Hoy en día, `LLVM` es el nombre que se utiliza para denotar todo el conjunto de productos relacionados con el proyecto.

`LLVM` es un proyecto de código abierto, implementado en `C++`, que se puede utilizar bajo la licencia "UIUC" BSD-style. Es multiplataforma y funciona tanto en Windows, Mac, Linux, como en Solaris. Dentro de la familia de proyectos de `LLVM`, podemos encontrar varios subproyectos derivados de éste que son utilizados tanto en aplicaciones de código abierto como así también en muchos proyectos cerrados. Un ejemplo es el compilador `nvcc` de Nvidia, el cual está basado en `LLVM`.

Fig. 3.1: Ejemplo de un diseño de tres fases tradicional.

Desde sus comienzos en el año 2000, `LLVM` fue diseñado como un conjunto de bibliotecas, enfatizando el desarrollo de interfaces bien definidas. `LLVM` utiliza el tradicional diseño

de compilación de tres fases, las cuales consisten en un *front-end*, un *optimizador* y un *backend*, tal como se muestra en el diagrama de la figura 3.1.

La representación interna (RI) de `LLVM` es un conjunto de instrucciones virtuales de bajo nivel similares a `RISC` (que a su vez es muy similar al código assembler). La RI del código fuente de un lenguaje de programación compilado con `LLVM` está diseñada para proveer optimizaciones y análisis de código a un nivel intermedio. En el código 3.1, se puede ver un pequeño ejemplo de código `LLVM` en formato de RI correspondiente al código `C` del listado 3.2.

```
define i32 @add(i32 %a, i32 %b) {
entry:
        %tmp1 = add i32 %a, %b
        ret i32 %tmp1
}
```

Listing 3.1: Ejemplo de código RI

```
unsigned add(unsigned a, unsigned b) {
        return a+b;
}
```

Listing 3.2: Ejemplo de código C

El diseño de `LLVM` hace posible definir las tareas de las tres fases de manera clara. El *front-end* realiza la tarea de pre-procesamiento (como en el caso de la familia de lenguajes `C`), el análisis léxico, el parseo y las validaciones semánticas. Luego de este proceso, convierte el código fuente a código RI (en esta etapa el código no tiene por qué ser optimizado). En el siguiente paso del proceso, el optimizador recibe el código RI, le aplica optimizaciones y envía el resultado al *back-end*. El *back-end* convierte el código RI a código de máquina o a *byte-code* dependiendo de la arquitectura para la cual se quiere generar el código final.

Un efecto de este diseño es que cuando se crea un nuevo *front-end* en `LLVM` para un nuevo lenguaje, puede que ya exista un optimizador y uno o varios *back-ends* disponibles. Por ejemplo, se puede compilar un nuevo lenguaje A para las arquitecturas de destino B, C y D, si existen *back-ends* en `LLVM` para B, C y D. Esto se ilustra en la figura 3.2, los únicos requisitos para crear un nuevo *front-end* son que el lenguaje de entrada pueda ser representado en código RI. Este proceso también funciona a la inversa: cuando se crea un nuevo *back-end*, cada lenguaje de programación que cuenta con una implementación de `LLVM` puede ser compilado para ese *back-end*. El diseño también implica que una implementación de una nueva optimización afectará a todos los lenguajes de programación que tengan implementado un *front-end* y un *back-end* en `LLVM`, lo cual resulta en un alto grado de reutilización de código. Este diseño reduce drásticamente el esfuerzo de desarrollo requerido para crear un nuevo compilador, dado que la mayoría del trabajo ya se encuentra realizado.

Optimizaciones

Las arquitecturas de la CPU como de las GPU presentan diferencias sustanciales debido a que realizan trabajos diferentes. Por ejemplo, las CPU puede realizar operaciones típicas como ser, predicciones de saltos o ejecuciones fuera de orden, pero en cambio, las GPUs no realizan ninguna de estas operaciones. Debido a estas diferencias, las optimizaciones pensadas para una CPU pueden no ser adecuadas para una GPU.

`LLVM` provee varias optimizaciones generales para GPU y también realiza optimizaciones específicas para las GPUs de arquitectura CUDA. A continuación describimos las optimizaciones más importantes que presenta `LLVM` para GPU,

- Optimizaciones escalares directas: esta optimizaciones eliminan la redundancia en el código.
- Inferir espacios de memoria: esta optimización ayuda a inferir los espacios de memoria de una dirección de memoria, de esta manera el *backend* puede emitir *loads* y *stores* de manera más rápida.
- Eliminación de ciclos y funciones en línea: la eliminación de ciclos y el reemplazo de funciones en línea es mucho más importante en la optimización de código para GPU que para CPU, esto se debe a que los saltos en el código de la GPU tienden a ser más costosos.
- Análisis de alias en espacios de memoria: el análisis de alias asegura que no existan dos punteros apuntando a la misma dirección de memoria.
- Divisiones de 64 bits: la división de enteros de 64 bits es mucho más lenta que la división de enteros de 32 bits para las GPUs de NVidia. Esta optimización intenta determinar si el divisor y el dividendo pueden, siempre que sea posible, representarse con un número de 32 bits en tiempo de ejecución.

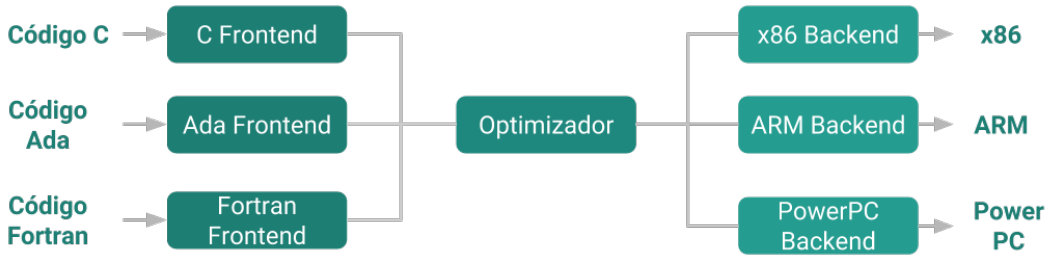

Fig. 3.2: Módulos en el diseño de tres fases.

3.1.2. Clang

`Clang` es uno de los subproyectos de `LLVM` que fue creado para ser un *front-end* para varios de los lenguajes de la familia `C`. Es compatible con los lenguajes `C`, `C++`, `objective C` y `objetive C++`. Está diseñado para ser compatible con `gcc` y también puede ser utilizado como su reemplazo directo, lo que simplifica una migración de `gcc` a `Clang`. La arquitectura basada en bibliotecas propuesta para `Clang`, permite que los nuevos desarrolladores puedan contribuir sin que deban tener conocimiento alguno de la base de código completa, solo necesitan tener conocimiento de la parte en la que están trabajando.

`Clang` provee soporte para varios frameworks como `OpenMP`, `OpenCL` y provee extensiones de lenguajes para `CUDA` de Nvidia. Este último soporte es el que vamos a utilizar para poder realizar una traducción del código de los funcionales de `Libxc` a código que se ejecute directamente en la placa de video.

3.1.3. LLC

Dentro del conjunto de herramientas que trae `LLVM` se encuentra el compilador `llc`, que se encarga de compilar código intermedio generado con `Clang` a código assembler de un arquitectura específica. El código assembler generado con esta herramienta puede ser compilado nuevamente con herramientas de assembler y linkers nativos del lenguaje para generar un archivo ejecutable.

Para el presente trabajo, vamos a utilizar el compilador `llc` en la etapa final de la traducción del código de los funcionales de `Libxc` para generar el código assembler para `CUDA` (`nvptx`), linkearlo y generar los archivos ejecutables. De esta manera vamos a intentar definir un procedimiento automático para la traducción de código que se ejecuta en CPU a código listo para ser ejecutado en GPU.

4. IMPLEMENTACIÓN

4.1. Implementación existente

Antes de describir las modificaciones 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.

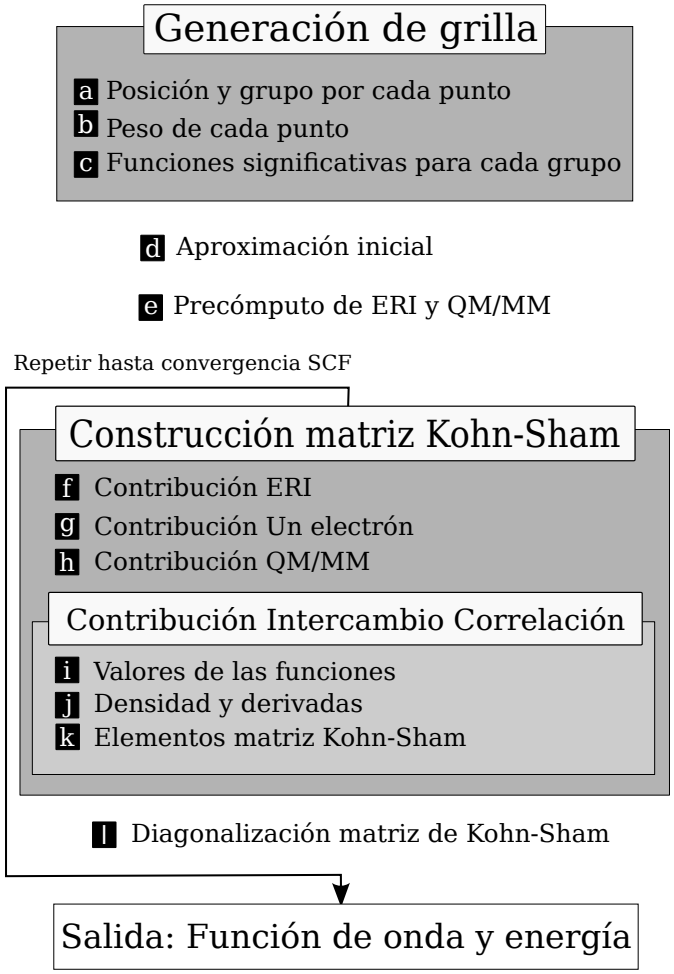

Fig. 4.1: Pasos del cálculo de DFT realizado por `LIO`.

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 según indica la teoría cuántica como,

$$\rho = |\Psi|^2 = \sum_k^{e^-} |\Phi_k|^2 = \sum_k^{e^-} \left| \sum_i a_i^k F_i \right|^2 \tag{4.1}$$

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 expanden los orbitales sobre un conjunto (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:

$$C_{i,j} = \sum_{k}^{e^-} a_i^k a_j^k \tag{4.2}$$

Luego, la densidad ρ puede calcularse como:

$$\rho = \sum_{i=1}^{m} \sum_{j=1}^{i} F_i F_j C_{i,j} \tag{4.3}$$

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 4.2 y 4.3 para cada punto de la grilla.

En este trabajo utilizamos las grillas propuestas por Becke [28], 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 [29]. Podemos aprovechar con esto el hecho de que las funciones Gaussianas (que forman la base utilizada) decaen rápidamente en el espacio.

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 [28].

El detalle algorítmico de la generación de grupos y de cálculo de pesos de puntos fue estudiado previamente [2, 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:

$$\chi_{i,j} = \frac{\partial E_{XC}}{\partial C_{i,j}} \tag{4.4}$$

Dado que el término E_{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 4.1. 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 [2].

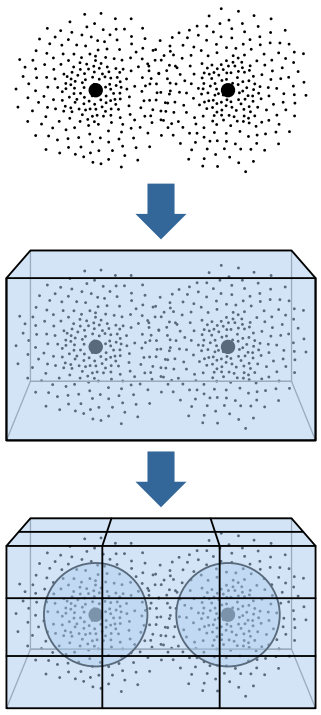

Fig. 4.2: Esquema del mallado de integración, partido en cubos y esferas para un sistema de dos átomos.

Los pasos de la figura 4.1 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 en el mencionado paso (j) de la figura 4.1 para la integración de la aplicación con la biblioteca `Libxc`.

4.2. Integración LIO - Libxc versión cpu

La primera parte del trabajo consistió en aprender a utilizar la biblioteca `Libxc`, implementada para ser ejecutada en CPU. Se buscó determinar las transformaciones de datos que se necesitan realizar desde `LIO` para poder utilizar los funcionales provistos por `Libxc`.

Luego de implementar el código que se encarga de transformar los datos para que sean utilizados por `Libxc`, realizamos pruebas para determinar si el funcional que se encuentra implementado en `LIO` [30] obtiene los mismos resultados que el mismo funcional implementado en `Libxc`. Esto resultaba un punto crítico ya que todos los funcionales implementados en `Libxc` utilizan números de doble precisión (`double` en `C`). En contrapartida, con `LIO` se utiliza precisión mixta (`float` y `double`) para realizar las operaciones. Luego realizamos un test de stress sobre la integración para determinar si los resultados eran numéricamente estables.

Esta primera etapa del trabajo se centra en buscar la correctitud de la implementación para asegurarse que el reemplazo de los funcionales en `LIO` por los de `Libxc` no afecta los resultados. Luego de obtener una confirmación, se procede a proponer modificaciones a la arquitectura de `LIO` para que la integración con `Libxc` sea escalable y se encuentre basada

en patrones de arquitectura de software ya probados.

4.2.1. Estructura original del código LIO

En primer lugar, se procede a analizar cómo es el módulo principal de `LIO` y, luego, cómo se lo modifica para que utilice los funcionales de `Libxc`. Un esquema de alto nivel del cómputo más intensivo realizado por el módulo que realiza el cálculo de los funcionales de intercambio de energía y correlación (de ahora en más XC) se encuentra en el pseudocódigo 1. Este código corresponde a una iteración del cálculo de la matriz de Kohn-Sham y se compone de tres partes:

1. Cálculo de las matrices relacionadas con el valor de las funciones base en los puntos (`funcs`),

2. Calcular las densidad electrónica, su gradiente y Hessianos para cada punto (`dens, grad, hess`),

3. 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 1: 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.
```

4.2.2. Cálculo de energía modificado

La integración de `Libxc` a `LIO` resulta en el pseudocódigo del algoritmo 2, en el cual se puede ver que `LIO` tiene que copiar los resultados parciales que tiene en memoria de la placa de video a memoria RAM para que `Libxc` pueda realizar los cálculos correspondientes. Luego de que `Libxc` realiza las cuentas, `LIO` se encarga nuevamente de dejar los datos en memoria de la placa de video, siendo todo este proceso uno de los más costosos de toda

la simulación.

Algoritmo 2: Pseudocódigo de la iteración de `LIO` integrado con `Libxc`

```
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
        Copiar dens, grad, hess de GPU a CPU
        Realizar transformaciones de dens, grad, hess para LIBXC
        // Integración con LIBXC
        Calcular potencial usando LIBXC
        Copiar resultados LIBXC a GPU
        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.
```

Como resultado de esta primera implementación se incrementa la posibilidad que tiene `LIO` de realizar simulaciones con todas la familia de funcionales que provee `Libxc`. Si bien el costo (en horas hombre) de esta implementación es bajo, el costo computacional de utilizar `LIO` con la versión CPU de `Libxc` motiva a realizar un esfuerzo para traducir el código de `Libxc` para que pueda ser ejecutado directamente en GPU.

4.3. Traducción de Libxc a CUDA

Para resolver este problema, se optó por la siguiente metodología:

1. Analizar en profundidad la implementación actual de `Libxc`.
2. Determinar en qué partes del código se encuentra el cálculo de los funcionales.
3. Elegir el mismo funcional que utiliza `LIO` para realizar la simulaciones y traducirlo a `CUDA`.
4. Determinar qué porciones del código del cálculo del funcional pueden ser paralelizables.
5. Aplicar una traducción de las funciones potencialmente paralelizables a código `CUDA`.
6. Realizar pruebas de estabilidad numérica para determinar la calidad de la solución.

4.3.1. Arquitectura de Libxc

`Libxc` posee más de 180 funcionales, éstos se agrupan en familias de funcionales: LDA, GGA, híbridos y MGGA. Cada una de estas familias tiene una forma distinta de realizar el cálculo de intercambio de energía y correlación. Actualmente `LIO` sólo tiene implementado un funcional de la familia GGA (Perdew, Burke & Ernzerhof), por lo que vamos a comenzar a realizar las modificaciones sobre `Libxc` comenzando con el mismo funcional ya que contamos con un punto de comparación para realizar las validaciones posteriores.

Análisis de componentes

El primer paso previo a la modificación del código es determinar cómo está organizada la arquitectura de `Libxc`. El gráfico de la figura 4.3 muestra cómo es la arquitectura original de `Libxc`. En ella podemos ver cómo está organizada la implementación de las familias de funcionales. Los componentes principales se llaman `work`, los cuales, para cada familia de funcionales, son los encargados de realizar la inicialización de parámetros y de las estructuras de datos que el funcional necesita para realizar los cálculos.

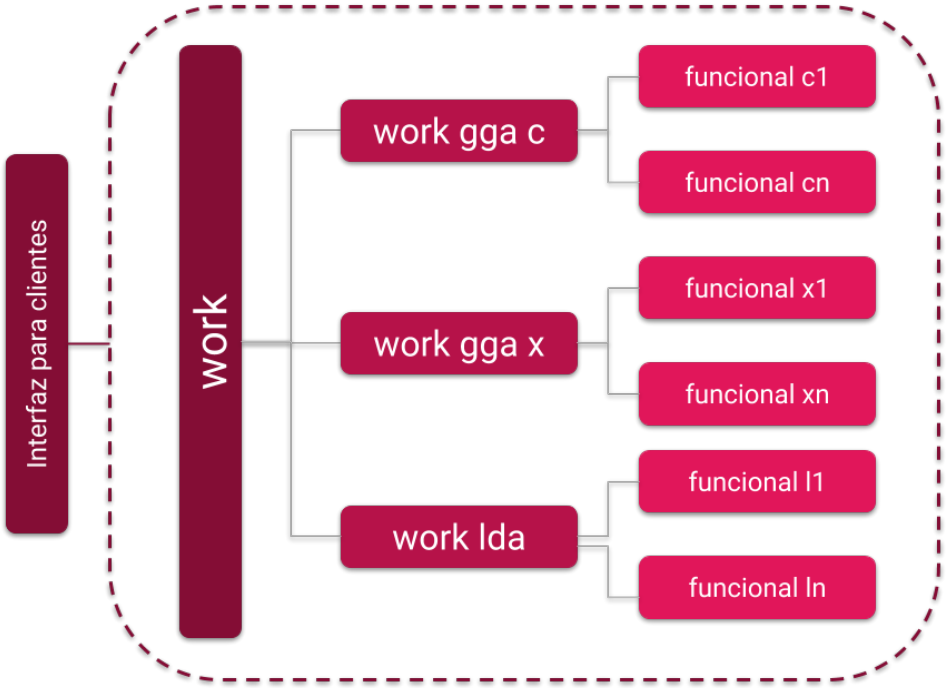

Fig. 4.3: Arquitectura de `Libxc`.

Dentro de cada componente se encuentra el ciclo principal que itera sobre cada uno de los datos de entrada. Por lo tanto, la primera parte de la traducción consiste en determinar cómo podemos modificar el código de los componentes `work` para poder realizar los cálculos en paralelo. También se debe considerar que cada uno de los `work` exporta sus interfaces para los clientes de `Libxc`. Esta interface no puede ser modificada para poder asegurarnos

que cualquier programa que sea cliente de `Libxc` pueda utilizar tanto la versión original como la versión modificada para GPU.

Cálculo de los funcionales

Luego de analizar en profundidad el código de `Libxc`, se determinó que la parte del cálculo numérico de cada funcional se encuentra en archivos separados. Esto resulta conveniente ya que, en principio, facilita la traducción a `CUDA` de cada funcional. Si bien `Libxc` cuenta con la implementación en `C` del código de cada funcional, la traducción no se puede hacerse de forma directa (o sea traducir directamente de código `C` a código `CUDA`). Esto se debe a que el código de los funcionales de `Libxc` está generado utilizando una herramienta de manipulación simbólica llamada `Maple`. Esta generación de código automático de archivos de `Maple` a archivos de `C` está diseñada y orientada a funcionar únicamente en CPU. Esto desemboca en que una traducción directa del código `C` generado por `Maple` a `CUDA` no resulte eficiente para esta arquitectura.

Por lo tanto, la primera parte del trabajo está enfocado en realizar la traducción de los funcionales de forma directa. El resultado de la traducción tiene como consecuencia que el tiempo de ejecución de las simulaciones es significativamente mayor que la implementación original de `LIO`, por lo que se procede a realizar un análisis detallado del código de los funcionales para determinar en dónde se encuentran las causas que producen un bajo rendimiento y, en caso de ser posible, aplicar optimizaciones.

Queda fuera del foco de este trabajo modificar la forma en que `Libxc` realiza sus cuentas internamente, ya que el objetivo es crear un traductor automático que realice la cantidad mínima de modificaciones al código original de `Libxc` para que este funcione en GPU. De esta manera, se espera poder utilizar los nuevos funcionales que la comunidad genere en el proyecto `Libxc` a medida que vayan apareciendo con solo ejecutar una herramienta de traducción.

Copia de datos a memoria de la placa de video

La primer versión de la traducción de `Libxc` a `CUDA` supone que todos los datos necesarios para realizar los cálculos de los funcionales están en memoria RAM, por lo tanto deben ser copiados a la memoria de la placa de video. Esta operación agrega un *overhead* importante al tiempo que insume la simulación. Por lo tanto, y para optimizar este pasaje de parámetros, se modificaron las interfaces en la traducción de `Libxc` en la cual se supone que todos los datos que necesita para realizar los cálculos de los funcionales ya se encuentran copiados en la memoria de la placa de video. De esta forma los clientes de la biblioteca se encargan de reservar y limpiar la memoria para almacenar los datos que necesiten.

Cantidad de threads y tamaño de los bloques

Otro punto de sumo interés es la cantidad de threads que van a interactuar en la ejecución paralela. Si bien se podría pensar, de manera inocente, que agregar más threads puede producir un incremento en la performance, esto no resulta totalmente cierto. La performance también depende de la disponibilidad de los distintos recursos (cantidad de memoria de registros, memoria compartida) para mantener activos los SM. Todo esto está directamente relacionado con el hardware sobre el cuál se realiza la implementacíon.

Para las primeras versiones de la integracíon se utilizaron placas Nvidia `Tesla C2070`, las cuales cuentan con un valor máximo de 1024 threads por bloque. Se decidió utilizar como valor inicial 256 threads por bloque. Si bien esto parece poco, dado las capacidades de la placa, la idea es iterar sobre la traducción de los funcionales para determinar cuál es el valor ideal para la cantidad de threads que deben ser utilizados en esta nueva versión de `Libxc`.

4.4. Integración LIO - Libxc versión GPU

En esta sección se explican las decisiones tomadas para realizar la implementación de la solución que integra la biblioteca `Libxc` con la aplicación `LIO`.

4.4.1. Arquitectura de la solución

Para resolver la comunicación entre `LIO` y `Libxc` se decidió utilizar soluciones basadas en patrones de diseño que permitan encapsular la comunicación entre ambas piezas de software. Por otro lado, se necesita encapsular el pasaje y transformación de datos ya que `LIO` y `Libxc` realizan los cálculos de maneras diferentes. Para reducir el impacto y tratar que los cambios en `LIO` sean solo los necesarios, se decidió implementar el patrón de diseño conocido como *Proxy* para que `LIO` se comunique con `Libxc` para realizar el cálculo de los funcionales. Este patrón de diseño permite encapsular el uso de `Libxc` y delegar en una clase la funcionalidad de la biblioteca. Básicamente, el *Proxy* a `Libxc` exporta interfaces que utilizan los datos de las simulaciones de `LIO`, los convierten a datos que necesita `Libxc`, realiza las llamadas a las funciones de `Libxc`, obtiene los resultados, los convierte y formatea para que sigan siendo utilizados por la simulación de `LIO`. En el diagrama de la figura 4.4, se puede tener una vista global de los tres componentes principales de la solución. La interacción entre el *Proxy*, `LIO` y `Libxc` se muestra en el diagrama de la figura 4.5, aquí se puede ver cómo se va a realizar el pasaje y transformación de datos en las simulaciones.

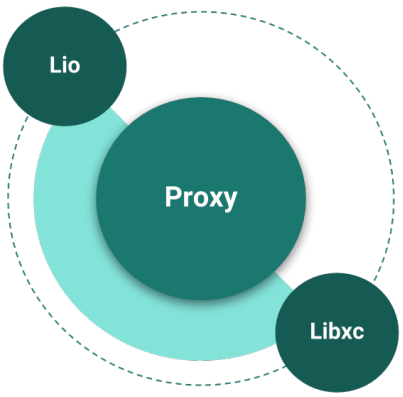

Fig. 4.4: Vista global de los componentes de la arquitectura de la solución propuesta para integrar `LIO` con `Libxc`.

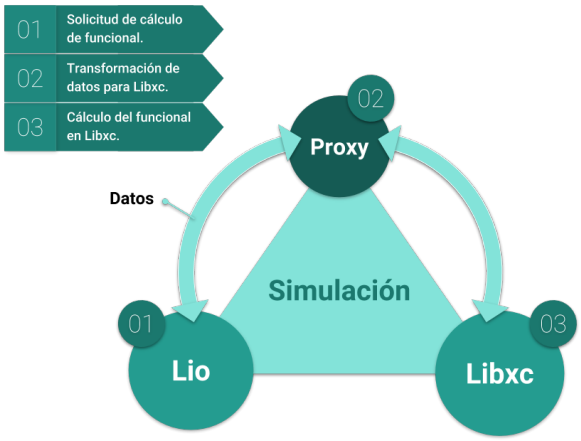

Fig. 4.5: Funcionamiento del Proxy entre `LIO` y `Libxc`.

4.4.2. Modificaciones a los componentes

En la sección 4.3.1 se mostró cómo es la arquitectura original de `Libxc` y también que el cálculo de los funcionales se encuentra agrupado por familias de funcionales en componentes llamados *work*. Los componentes *work* contienen las interfaces para los clientes de `Libxc` y, dado que el objetivo de la traducción es minimizar los cambios en el código, no se van a realizar modificaciones sobre la interfaces. Solo se modifican las partes que realizan el cálculo de los funcionales, de esta manera la interfaz para los clientes no se modifica y permite encapsular la rutina principal que realiza el cálculo del funcional en `CUDA`. En el diagrama de la figura 4.6 se ve cómo queda la nueva arquitectura de cada componente *work* con respecto a la arquitectura original. Se incluye cómo queda la interfaz, la inicialización de parámetros que se necesitan para realizar los cálculos de los funcionales y qué partes se ejecutan en CPU y cuáles en GPU.

4.4.3. Precisión numérica

Con la biblioteca `Libxc` funcionando en GPU, el próximo paso consiste en realizar la integración con `LIO`. Recordemos que uno de los puntos clave de esta integración es el tipo de datos que maneja `LIO` y `Libxc`. Dado que `Libxc` utiliza números de doble precisión (`double`) para realizar los cálculos de los funcionales, se tuvo que implementar una serie

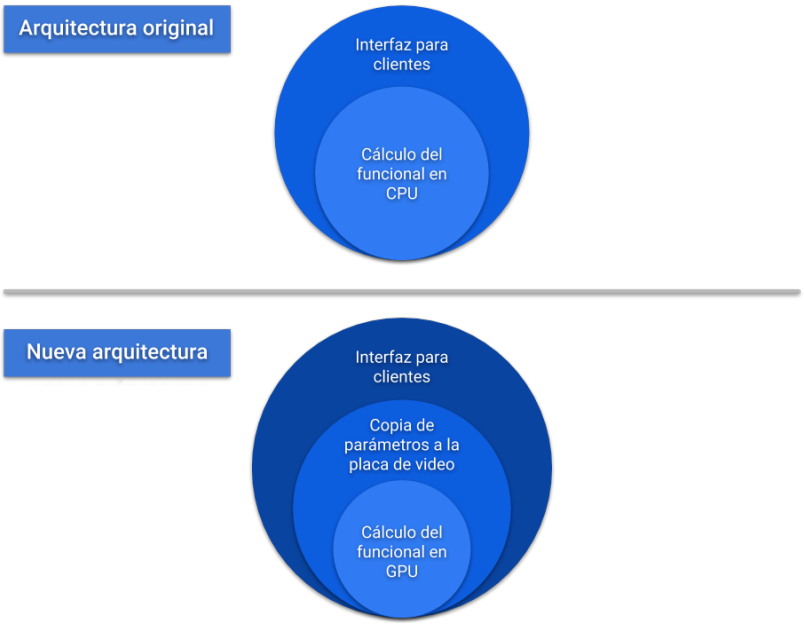

Fig. 4.6: Modificaciones a la arquitectura interna de los componentes *work* de `Libxc`.

de rutinas adicionales que complementan la primera versión de la aplicación.

Algoritmo 3: Pseudocódigo de la iteración de `LIO` integrado con `Libxc` en GPU.

```
solve(PG : PointGroup)
    Leer la matriz de coeficientes inicial.
    ...
    Transformar los datos para LIBXC-GPU
    Calcular potencial usando LIBXC-GPU
    Transformar los datos desde LIBXC-GPU
    ...
    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.
```

En el algoritmo 3 se muestra cómo queda la nueva implementación de `LIO` con `Libxc` funcionando en GPU. Si bien en la integración se ahorra el pasaje de parámetros de CPU a GPU, todavía existen pasos intermedios en el cálculo del funcional de `Libxc` que requieren ejecutar código en CPU. En el gráfico de secuencia de la figura 4.7, se puede ver cómo es la ejecución en CPU y GPU del algoritmo 3.

4.5. Resultados preliminares

Con la primera versión del programa funcionando, el siguiente paso consiste en realizar las mediciones de los tiempos de ejecución de las simulaciones. Para realizarlas, se utilizaron sistemas provistos por `LIO` que corresponden a moléculas del grupo *Hemo* y *Caroteno*. El detalle químico de ambos sistemas se incluye en el Apéndice B (pág. 81). Para los casos de prueba se utiliza el mismo funcional que tiene actualmente implementado `LIO` (Perdew, Burke & Ernzerhof [30]) ya que sirve como punto de referencia para realizar comparaciones y validar la solución. En los gráficos de las figuras 4.8 y 4.9 se ve la comparación de los

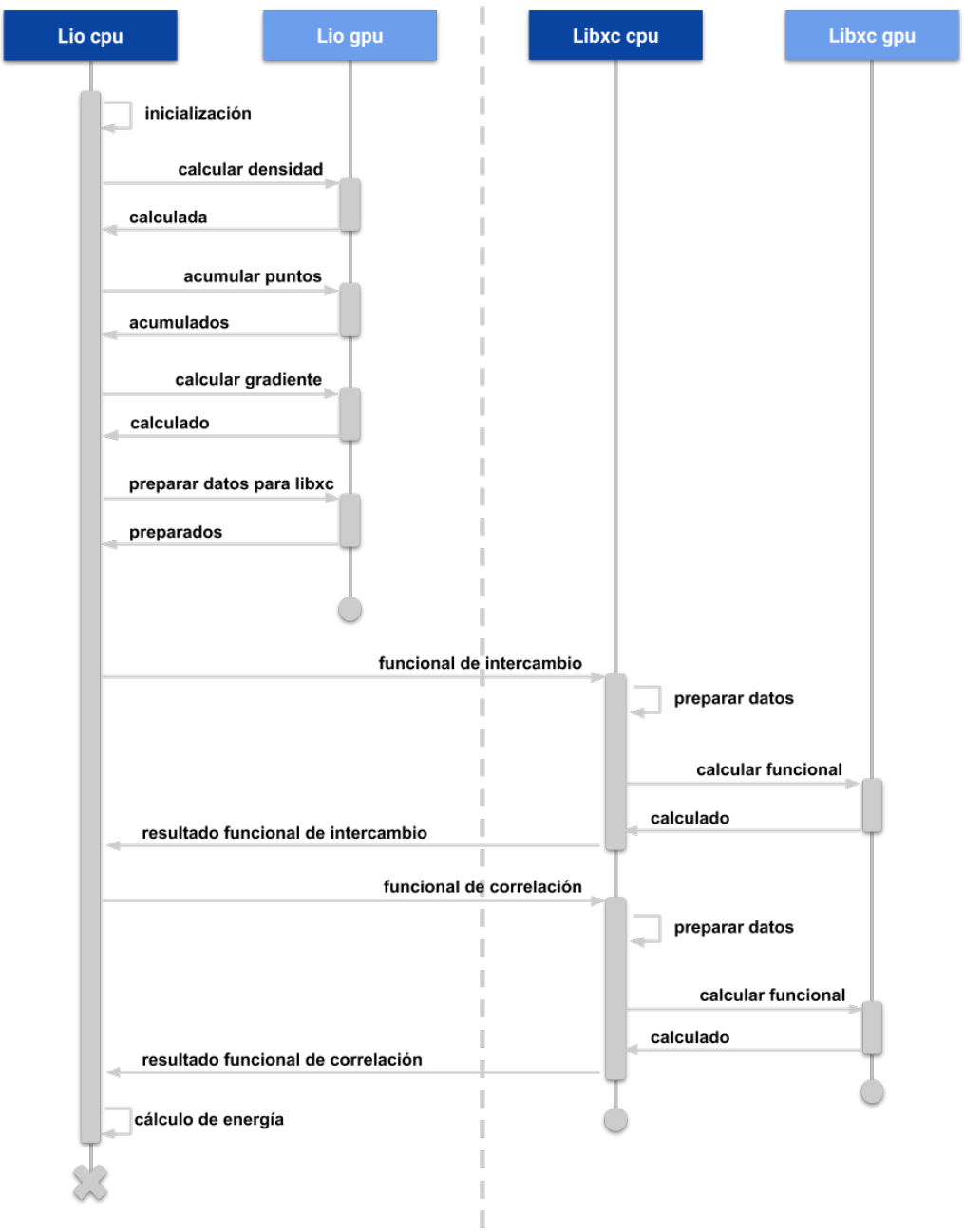

Fig. 4.7: Cálculo del funcional en `LIO` con `Libxc`.

tiempos de ejecutar 100 simulaciones con la versión original de `LIO` y con la nueva versión que usa los funcionales de `Libxc` implementados en GPU.

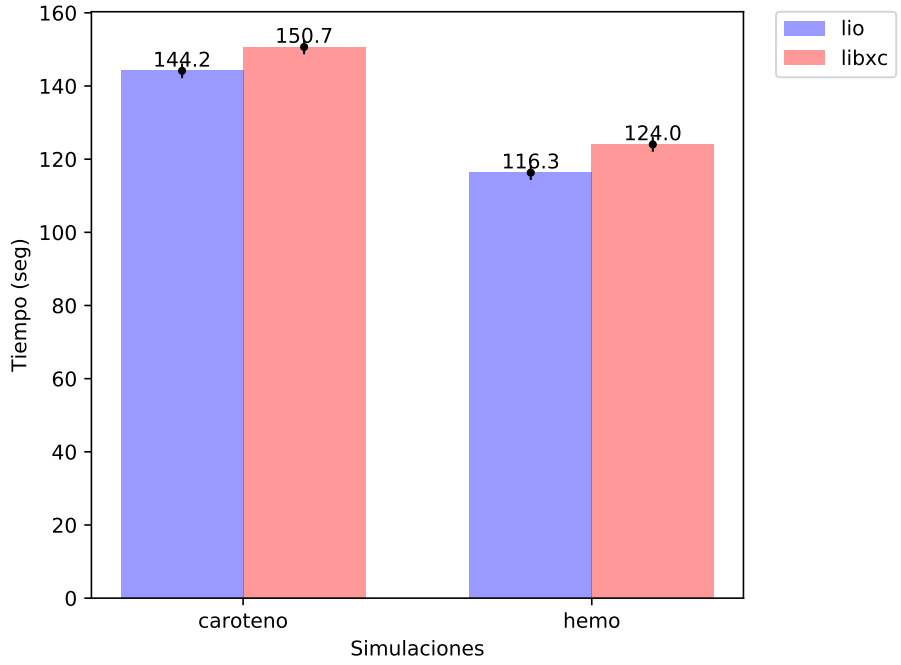

Fig. 4.8: Tiempo total de las simulaciones para los sistemas de hemo y caroteno, obtenidos luego de 100 corridas.

Estos resultados preliminares muestran que los tiempos de ejecución de la implementación de `LIO` con `Libxc` son ligeramente mayores que los tiempos de la implementación original de `LIO`. En vista de esto, se realiza un análisis detallado de la implementación de `Libxc` a GPU para determinar dónde se encuentran los cuellos de botella e intentar eliminarlos.

4.6. Optimizaciones a Libxc

Luego de realizar varios test de stress y ejecutar diversas simulaciones, se realiza un estudio más detallado del código resultante de la traducción de `Libxc` a GPU con el fin de poder determinar en donde se encuentran los cuellos de botella y, en los casos que se pueda, realizar optimizaciones que permitan mejorar los tiempos de las simulaciones. Para llevar a cabo esta tarea, se utilizan las herramientas de profiling que provee Nvidia, éstas son el *Visual Profiler* y la herramienta de profiling por línea de comando *nvprof*, también se utiliza el *debugger* `CUDAdbg`, las herramientas para monitoreo de memoria `CUDA-memcheck` y una herramienta adicional que permite conocer cómo es la asignación de recursos de la placa de video para los kernels `CUDA-ocuppancy-calculator`.

4.6.1. Detectando cuellos de botella

Una de las principales preocupaciones al realizar este trabajo fue la de cómo se podrían realizar optimizaciones a la traducción de `Libxc` sin tener que modificar los algoritmos que

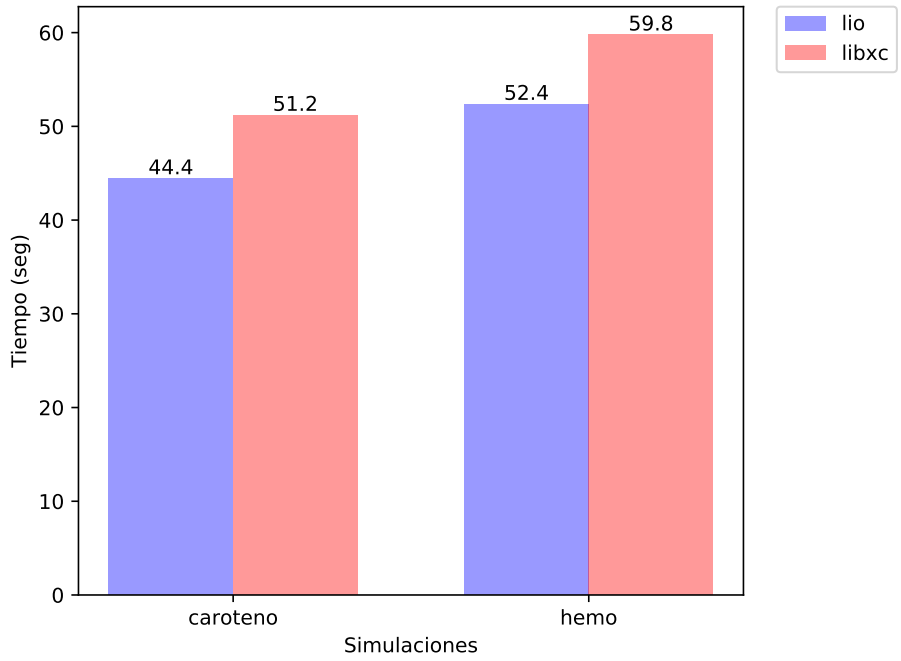

Fig. 4.9: Tiempo que insume el cálculo del funcional en las simulaciones de los sistemas de hemo y caroteno, obtenidos luego de 100 corridas. Se omiten las barra de errores ya que los mismos son menores a $1x10^{-6}$.

realizan el cálculo de los funcionales.

Se escribieron test de unidad, específicos para ejecutar el código que realiza el cálculo de los de los funcionales, cuyo objetivo es el de poder determinar en qué partes del código se producen los cuellos de botella. Se ejecutaron cientos de corridas sobre los casos de prueba complementado éstos con las herramientas de profiling que provee Nvidia (*nvprof*). Como resultado, obtuvimos métricas de la GPU que nos muestran, entre otras cosas:

1. Cantidad de lecturas y escrituras para los distintos tipos de Memoria.
2. Rendimiento, en Gb/s, en los accesos a los distintos tipos de Memoria.
3. Rendimiento, en Gb/s, en la copia de datos de Memoria de la GPU a CPU.
4. Eficiencia de saltos en el código.
5. Actividad del multiprocesador.
6. Cantidad de instrucciones por Warp.
7. Cantidad de Warps utilizados por ciclo.
8. Cantidad total de operaciones de simple y doble precisión.
9. Porcentaje de búsqueda de instrucciones.
10. Porcentaje de dependencia de instrucciones.

11. Porcentaje de búsqueda de datos en memoria.

El siguiente paso consistió en realizar un analisis de qué porciones del código afectan de manera negativa estas métricas. El diagrama de secuencia de la figura 4.10 muestra en dónde se producen los cuellos de botella en la simulación, en "rojo" se encuentran marcadas las partes del código que se ejecuta en GPU, que es el lugar en donde se detecta la mayor cantidad de problemas de performance.

4.6.2. Posibles causas de los cuellos de botella

Nos encontramos en un escenario en el cual se está ejecutando código diseñado originalmente para CPU pero en GPU. Esto tiene como desventaja que no se está utilizando de manera eficiente la memoria de la placa de video. Dado que una gran mayoría de los cálculos que realiza `Libxc` utilizan una gran cantidad variables auxiliares, al realizar la traducción directa a GPU se está malgastando dicho espacio en memoria.

Para tener una mejor panorama de la situación, se incluye un ejemplo concreto: si una función de `Libxc` usa 100 variables auxiliares, entonces para cada thread de `CUDA` se termina definiendo la misma cantidad de variables auxiliares al momento de ejecución. Si se considera la cantidad de threads que se utiliza para paralelizar, debido a la falta de recursos disponibles se llega rápidamente a una disminución de la cantidad de *warps* disponibles en los bloques de memoria en la placa de video para ejecutar la aplicación. Por lo tanto se procedió a determinar cuál es la cantidad máxima de registros que utilizan los funcionales de `Libxc`, en un intento de determinar como estos valores afectan la performance.

El compilador de Nvidia permite agregar varias opciones que permiten mostrar información extra cuando realiza la compilación del código de un kernel. Uno de estos datos es la cantidad de registros que el compilador asigna a cada uno de los kernels. Luego de compilar la biblioteca, se recolecta la información notando que la cantidad máxima ronda en los 63 registros para algunos funcionales. En el gráfico de la figura 4.11 se puede ver cómo varía la cantidad de *warps* que se utilizan a medida que aumentan las cantidad de registros que utilizan los kernels. Como es de esperar, mientras más registros utilizan los kernels hay menos *warps* disponibles. Esta es una de las causas principales por la que se produce una subutilización de recursos de la memoria de la placa de video.

Por otro lado también resulta de interés saber qué sucede con la cantidad de *warps* por bloques de memoria cuando se modifica la cantidad de threads en los kernels que realizan los cálculos de los funcionales cuando estos utilizan 63 registros de memoria. El gráfico de la figura 4.12 muestra cómo es la relación entre la cantidad de threads y el tamaño *warps*. Se puede ver que la mayor cantidad de *warps* se alcanza cuando la cantidad de threads por bloque de memoria que se asigna a un kernel es de 64, 128, 256 y 512 threads.

También es interesante ver cómo se comporta la performance a medida que se modifica la cantidad de registros que se utilizan en los kernels de los funcionales. Para esto, se realizan varias pruebas con el código traducido de los funcionales en los cuales se modifica la cantidad máxima de registros de memoria de la placa de video que pueden utilizar para realizar los cálculos. El objetivo es intentar determinar cuál es la configuración óptima de esta variable. El gráfico de la figura 4.13 muestra cómo varía el tiempo de ejecución del funcional `gga_c_pbe` a medida que se incrementa la cantidad máxima de registros que utiliza para realizar los cálculos. El mismo resultado para el funcional *gga_x_pbe* se puede ver en el gráfico de la figura 4.14.

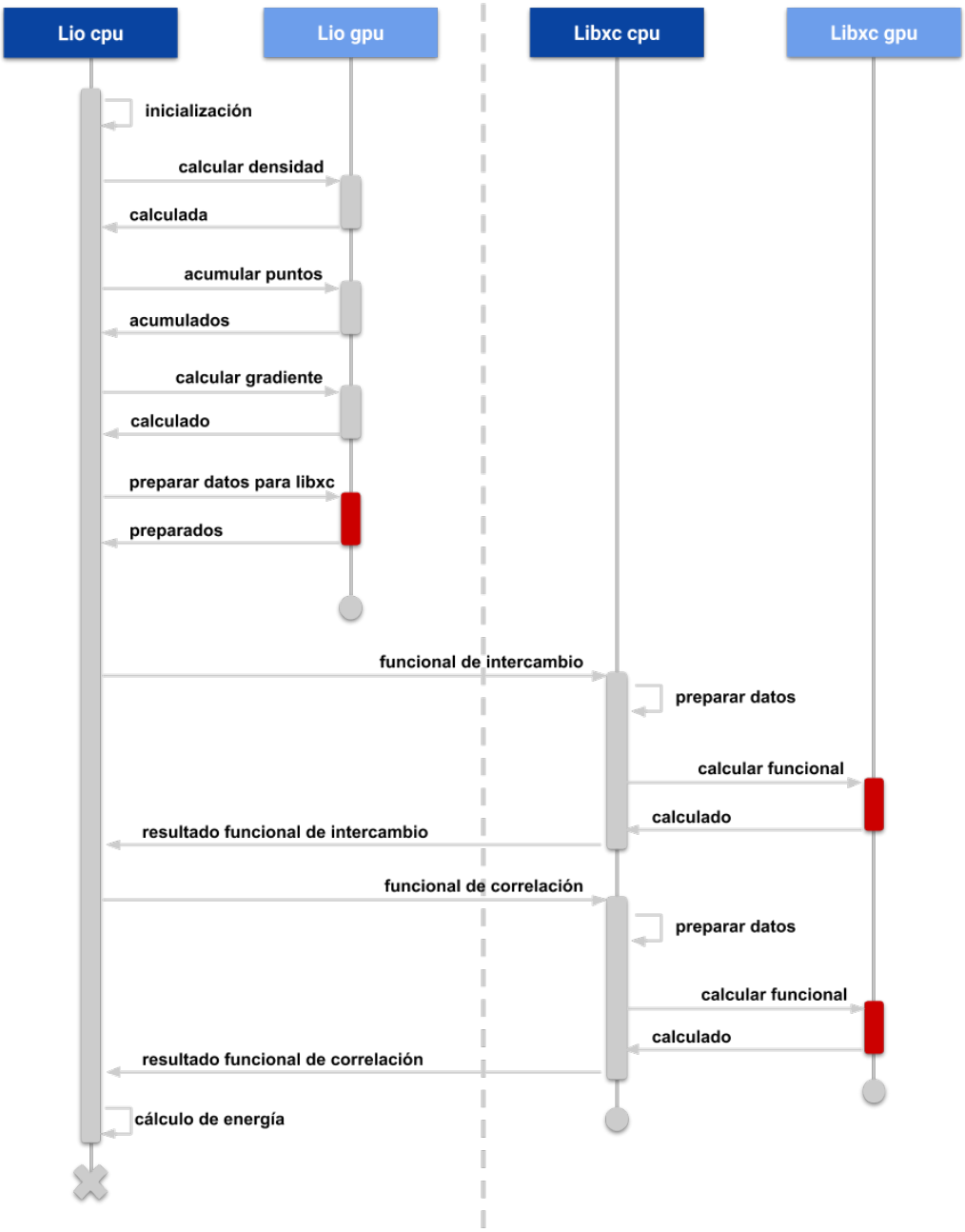

Fig. 4.10: Cuellos de botella en las simulaciones de `LIO` con `Libxc`.

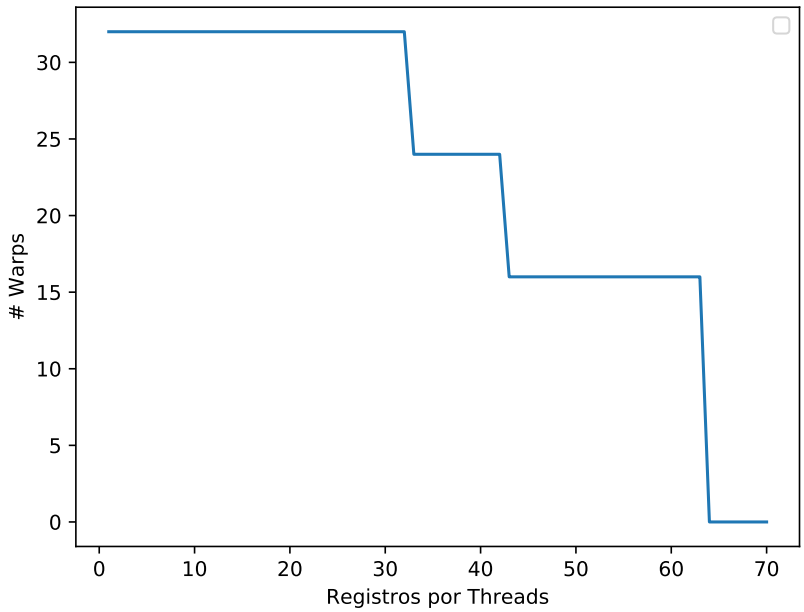

Fig. 4.11: Cantidad de *warps* por bloques de memoria en función de la cantidad de registros asignada a un kernel de `Libxc`.

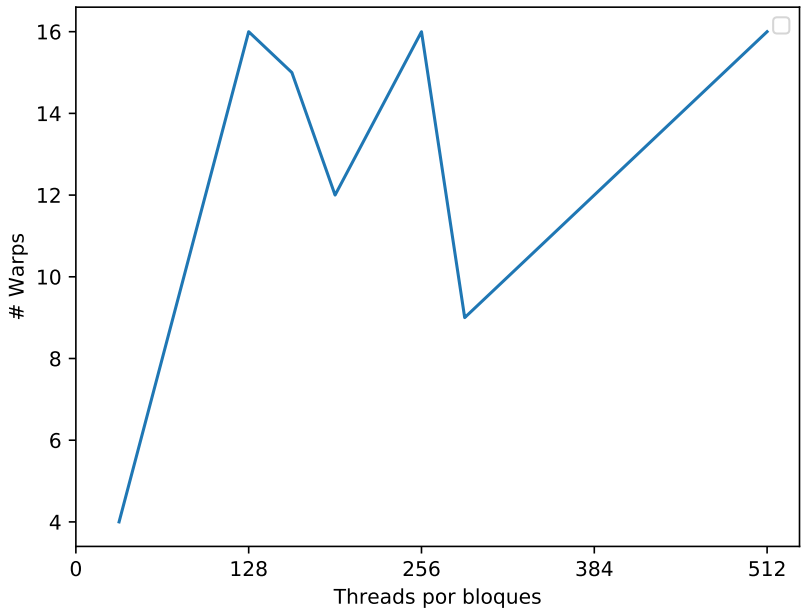

Fig. 4.12: Cantidad de *warps* en función de la cantidad de threads para kernels de GPU que están configurados para utilizar 63 registros de memoria como máximo.

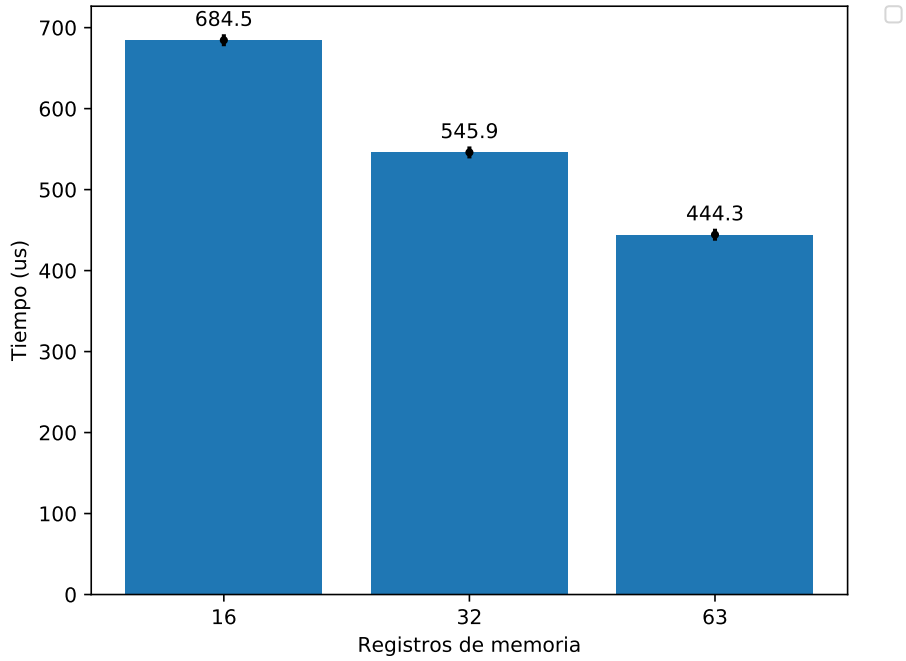

Fig. 4.13: Tiempo de ejecución del funcional `gga_c_pbe` en función de la cantidad máxima de registros de memoria de la placa de video con el cual fue configurado, obtenidos luego de 100 corridas.

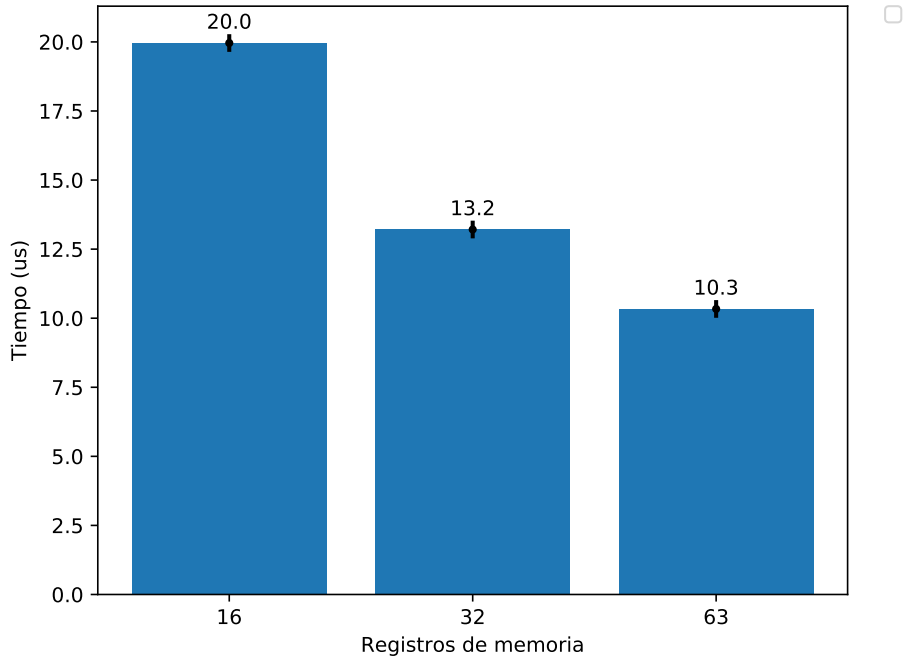

Fig. 4.14: Tiempo de ejecución del funcional *gga_x_pbe* en función de la cantidad máxima de registros de memoria de la placa de video con el cual fue configurado, obtenidos luego de 100 corridas.

Estos resultados parciales muestran que se obtienen mejoras a medida que se incrementan la cantidad máxima de registros de memoria disponibles para cada kernel. Por lo tanto, parece razonable fijar la cantidad de registros en 63 para cada kernel y continuar realizando las siguientes pruebas modificando otras variables para intentar determinar otros posibles cuellos de botellas en la traducción.

4.6.3. Buscando soluciones a los cuellos de botella

Hasta ahora se pudo detectar que el factor limitante de la performance es la cantidad de registros que utilizan las funciones de `Libxc` que realizan el cálculo de los funcionales, por lo tanto se buscarán herramientas que permitan mejorar o alivianar esta limitación. Luego de buscar en la literatura sobre cómo mejorar el rendimiento de los kernels de `CUDA`, encontramos que existe la posibilidad de compilar el código parametrizando la cantidad máxima de registros que utilizan las funciones que se ejecutan en la placa de video.

Generalmente, al incrementar la cantidad de registros se incrementa la performance de los kernels. Sin embargo, dado que los registros se asignan desde un *pool* de registros globales en cada GPU, un valor más alto de esta opción también reducirá la cantidad máxima de bloques, afectando así el paralelismo. Por lo tanto, se debe determinar el valor de la cantidad de registros máximos por kernel que produzca una mejora en la performance.

4.6.4. Variando la cantidad de threads

En las secciones anteriores se estableció, sin demasiada justificación, un valor constante para la cantidad de threads que se utiliza para configurar los algoritmos que fueron paralelizados en la traducción de `Libxc`. Luego de realizar un análisis detallado del código y teniendo en cuenta la gran cantidad de registros de memoria que utilizan los funcionales de `Libxc`, se realizaron varias pruebas variando la cantidad de threads que usan los kernels. Previo a estas pruebas, se utilizó la herramienta de Nvidia `CUDA-occupancy-calculator`, que permite realizar una estimación de cómo varía la performance de los kernels en función de la cantidad de threads que utiliza. En el gráfico de la figura 4.12 se ve cómo varía la cantidad de *warps* por bloques de memoria que utiliza un kernel en función de la cantidad de threads. La cantidad de bloques de memoria llega a su máximo para los valores de 64, 128, 256 y 512 threads. Se comprobó que la cantidad de *warps* por bloques de memoria se mantiene constante para la cantidad de memoria compartida por bloque que utilizan los kernels.

Si bien utilizar la máxima cantidad de *warps* por bloques de memoria no asegura una buena performance, resulta de interés medir la performance de las traducciones de los funcionales variando la cantidad de threads. Para realizar estos experimentos se utiliza uno de los funcionales que mayor cantidad de registros utiliza. En el gráfico de la figura 4.15 se ve cómo varía la performance del cálculo del funcional `gga_c_pbe` a medida que se incrementa la cantidad de threads que se utilizan para configurar el kernel que realizan el cáculo de energía. Por otro lado, en el gráfico de la figura 4.16 se presenta la performance en función de la cantidad de threads para el kernel que realiza el cálculo de correlación.

Estos resultados indican que la traducción de los funcionales no escala de la manera esperada. Al aumentar la cantidad de threads que se ejecutan en paralelo, se espera una mejora en la performance, pero no es lo que ocurre. Esto vuelve a motivar un análisis más profundo de la traducción del código de los funcionales.

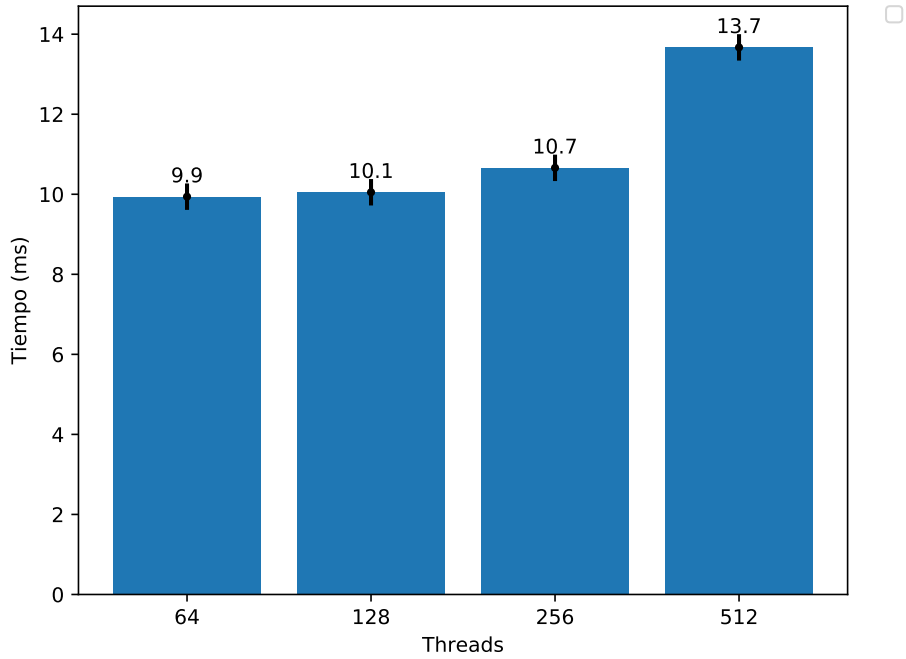

Fig. 4.15: Tiempo de ejecución del funcional *gga_x_pbe* en función de la cantidad de threads que se ejecutan en paralelo, obtenidos luego de 100 corridas.

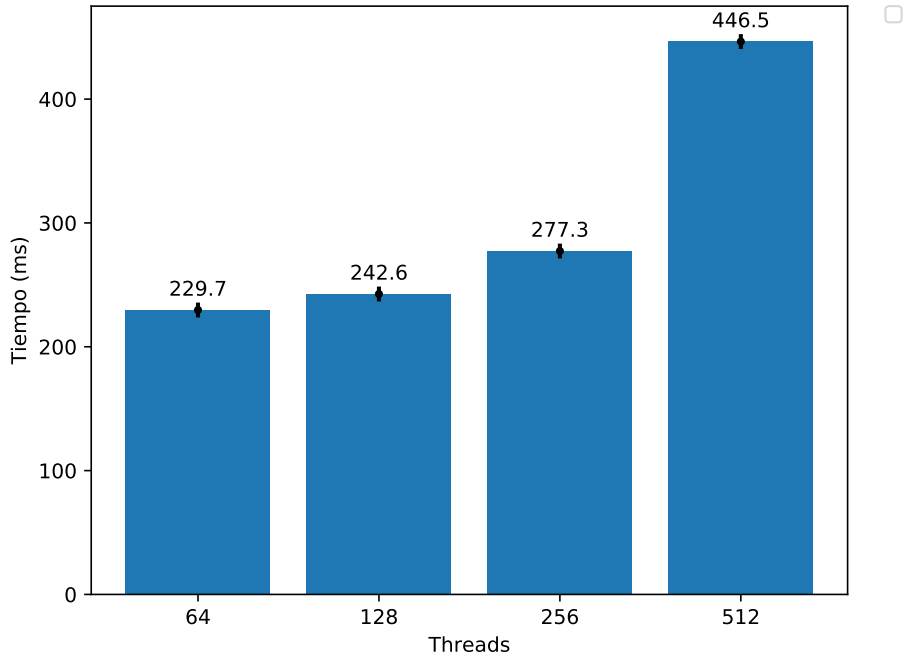

Fig. 4.16: Tiempo de ejecución del funcional `gga_c_pbe` en función de la cantidad de threads que se ejecutan en paralelo, obtenidos luego de 100 corridas.

4.6.5. Deep profiling

Siguiendo con el análisis de la traducción del código de los funcionales, se van a intentar determinar cuáles son las causas que hacen que los funcionales `gga_c_pbe` y `gga_x_pbe` no escalen al incrementar la cantidad de threads asignados a trabajar en paralelo. La idea es analizar la mayor cantidad de variables posible durante la ejecución de los funcionales para intentar detectar en qué parte del código están los problemas.

Atascamientos (stalls)

Analizando en detalle las métricas obtenidas luego de la ejecución de varias pruebas, se nota que la mayor cantidad de problemas (atascamientos/stalls) se producen debido a la gran cantidad de operaciones de memoria involucradas en los cálculos. El gráfico de la figura 4.17 muestra en detalle las causas de los problemas que encuentra el *profiler* de CUDA cuando ejecuta el código del funcional `gga_c_pbe`.

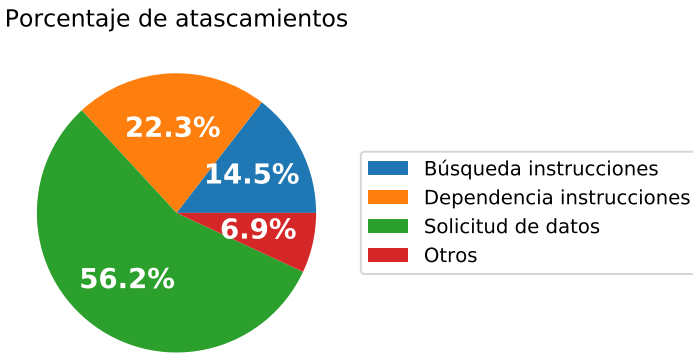

Fig. 4.17: Porcentaje de atascamiento (stalls) en la ejecución del funcional `gga_c_pbe`, obtenido luego de 100 corridas.

Como ya se había notado con los resultados preliminares de las secciones anteriores, la mayor cantidad de problemas radica en los accesos a memoria y en la gran cantidad de instrucciones que realiza el funcional. Por lo tanto, se debe enfocar los esfuerzos en reducir estos dos problemas.

Causas de los atascamientos

Realizando un análisis detallado del código, se llega a la conclusión de que el problema se encuentra en la gran cantidad de variables que utilizan los funcionales de `Libxc` para hacer las cuentas. En la tabla 4.1 se ve la cantidad de variables que utilizan los funcionales que fueron utilizados para las pruebas. En la misma podemos ver que el funcional `gga_c_pbe` es el que más variables tiene definidas para realizar los cálculos. Esto trae como consecuencia que el compilador deba reservar espacio para esta cantidad de variables en la memoria global de la placa de video. Dado que la mayoría de estos datos tampoco se encuentran en la memoria cache L1 o L2, la latencia que se produce por cargar y almacenar los datos impacta fuertemente en la performance.

Basados en la literatura sobre el tema [31], se encuentran varias técnicas para resolver o reducir estos problemas. Una de ellas es definir el tamaño máximo de la cantidad de

funcional	variables
gga_x_pbe	17
gga_c_pbe	544

Tab. 4.1: Cantidad de variables de los funcionales gga_c_pbe y *gga_x_pbe* del código original de Libxc. En la tabla C.1 puede verse una lista completa de variables por funcional, para todos los funcionales de Libxc.

registros que utilizan los kernels. Como se vio en las secciones anteriores de este trabajo, esta técnica ya está siendo utilizada y aún así la performance solo mejora un poco.

Otra forma de intentar reducir los accesos a memoria es modificando el código. El kit de desarrollo de Nvidia provee unas funciones que permiten configurar cómo se prefiere realizar los accessos a memoria global de la placa de video. Estos cambios pueden programarse para cada función por separado o directamente configurarlo para todas las funciones. Las opciones disponibles son las siguientes:

1) Disponer de memoria compartida más grande y de memoria cache L1 pequeña.

2) Disponer de una memoria cache L1 de mayor tamaño y memoria compartida pequeña.

3) Igual tamaño de memoria compartida y memoria cache L1.

En la tabla 4.2 se muestran los porcentajes de las causas de los atascamientos de memoria al variar la configuración para el funcional gga_c_pbe.

Para realizar una mejora significativa en los resultados, se debe modificar la forma en que Libxc realiza el cálculo de los funcionales realizando optimizaciones en el código fuente.

Configuración de Cache	Búsqueda de instrucciones	Dependencia de instrucciones	Solicitud de datos	Otros
Default	14,54 %	22,33 %	56,19 %	6,94 %
CUDAFuncCachePreferShared	13,69 %	23,56 %	55,12 %	7,13 %
CUDAFuncCachePreferL1	10,8 %	24 %	57,19 %	8,01 %
CUDAFuncCachePreferEqual	13,06 %	25,53 %	53 %	8,41 %

Tab. 4.2: Porcentaje de atascamientos (stalls) del funcional gga_c_pbe para las distintas configuraciones de memoria cache.

Mejoras al código

Se realiza, entonces, un análisis de los funcionales que se utilizan (gga_c_pbe y gga_x_pbe) para determinar si es posible mejorarlos. Para ello, la bibliografía de Nvidia propone varias estrategias, entre ellas están:

- Compilar los kernels de CUDA con la opción *–ptxas-options=-v* para obtener el detalle de la cantidad de memoria que usa el kernel e intentar reducirla.

- Eliminar la mayor cantidad de "`if`'s" de los kernels, ya que esto provoca ramificaciones en el código que degradan la performance.
- Si el kernel tiene ciclos, intentar eliminarlos (*loop unrolling*).
- Si el kernel utiliza arreglos estáticos, acceder a los mismos utilizando constantes y no variables.

Teniendo esto en cuenta, se realiza nuevamente una compilación de los códigos traducidos de los funcionales en la cual se detecta que se produce el fenómeno de *spills* de memoria. Esto significa que, dada la gran cantidad de variables que utilizan los funcionales de `Libxc` para realizar las cuentas, la memoria de registros de la placa es insuficiente y se produce una gran cantidad de lecturas y escrituras a memoria global, que es la utilizada para suplir esa falta.

Los resultados de la tabla 4.3 son la prueba de que, en efecto, el código de los funcionales de `Libxc` utiliza una gran cantidad de variables para realizar el cálculo. Esto no sorprende ya que la biblioteca está diseñada para ejecutarse en CPU y no en GPU.

funcional	variables	load (bytes)	stores (bytes)
`gga_x_pbe`	17	400	448
`gga_c_pbe`	544	9240	4112

Tab. 4.3: Cantidad de variables y spills de memoria de los funcionales `gga_c_pbe` y `gga_x_pbe` del código original.

Se procede a realizar un análisis detallado de las variables que utilizan los funcionales para intentar determinar si todas esas variables son utilizadas, si pueden reutilizarse, si hay variables que pueden ser globales, o si hay variables que pueden ser compartidas por todos los threads. En el cálculo del funcional, `Libxc` permite determinar qué orden de derivadas se va a utilizar (orden 1, 2 o 3). Por otro lado, `LIO` solo necesita los datos de las derivadas de orden 1 y orden 2. Esto significa que se puede modificar el código de los funcionales eliminando todas la variables que se utilizan para calcular las derivadas de tercer orden en adelante.

`Libxc` utiliza varias constantes para realizar los cálculos, estas constantes pueden ser compartidas por todos los threads. Del mismo modo, se pudo determinar que un cierto conjunto de variables pueden ser declaradas como *shared*. Pero esto no es todo, también se realizaron varias modificaciones en las funciones que realizan las configuraciones de los funcionales. Se eliminaron ciclos dentro de los kernels, se eliminó el uso de variables auxiliares, y se clasificaron las variables restantes en: *shared*, *constant* y *global*.

Como resultado de estas técnicas se obtuvieron varios kernels con menos líneas de código que comparten y reutilizan variables y que, además, realizan solo los cálculos necesarios.

En la tabla 4.4 se presenta la cantidad de variables y spills de memoria que se obtuvieron luego de aplicar las optimizaciones.

Alineando las estructuras de datos

Otro de los problemas encontrados fue que `Libxc` tiene una gran cantidad de estructuras de datos implementadas con *structs* de `C` que utiliza para configurar los parámetros

funcional	variables	load (bytes)	stores (bytes)
gga_x_pbe	5	12	8
`gga_c_pbe`	200	1152	748

Tab. 4.4: Cantidad de variables y spills de memoria de los funcionales `gga_c_pbe` y `gga_x_pbe` luego de aplicar optimizaciones.

que de los funcionales. Muchas de estas estructuras son pasadas como párametros de las funciones y en varios casos las estructuras de datos contienen, a su vez, punteros a otras estructuras de datos. Por esta razón, se tuvo que tener especial cuidado al momento de realizar la copia de los datos que necesita el kernel en su ejecución. Además, muchas de estas estructuras de datos tienen una gran cantidad de variables que no son utilizadas, lo que provoca que se ocupe la memoria con datos que nunca se utilizan. La consecuencia de esto es que los kernels disponen de menos memoria para realizar su ejecución.

Para resolver este problema se tuvieron que definir estructuras de datos que utilizan solo la cantidad de variables necesarias para el cálculo de cada funcional. Para mejorar el acceso a los datos de estas estructuras, se utilizan directivas del compilador que provee el toolkit de `CUDA` que permite alinear los *structs* en memoria al momento de compilación. De esta manera, se busca mejorar aún más el acceso a los datos en los kernels. La tabla 4.5 contiene una descripción de las estructura de datos más importante de `Libxc` para los funcionales que se utilizan en las simulaciones.

El ejemplo de código que se muestra en la lista 4.1 contiene un ejemplo de una estructura de datos de `Libxc` en `C` desalineada y el código de la lista 4.2 muestra cómo se utilizan las directivas del compilador de `CUDA` para alinear las estructuras de datos en memoria. En los gráficos de las figuras 4.18 y 4.19 podemos ver cómo varía el acceso a los datos cuando las estructuras se encuentran alineadas en memoria y cuando no lo están.
Lo primero que notamos del gráfico 4.18 fue que el tiempo de copia se reduce drásticamente para las estructuras de datos de mayor tamaño luego de que estas fueran alineadas en memoria. El tiempo de copia de la estructura $xc\_gga\_work\_c\_t$ (cuyo tamaño es de 552 *bytes*) es de 4 veces mayor cuando se alinean los datos en la memoria de la placa de video. Un comportamiento similar puede verse para la estructura xc_func_type. Pero para estructuras de datos pequeñas (menores a los 100 *bytes*) como $xc\_gga\_work\_x\_t$ y $xc\_func\_info\_type$ se observa una mejora marginal en los tiempos de copia. Cabe aclarar que si las estructuras de datos son potencia de 2, como el caso $xc\_gga\_work\_x\_t$, no es necesario utilizar las directivas del compilador para realizar alineamientos. El gráfico de la figura 4.19 muestra como el rendimiento, expresado en GB/s, mejora cuando las estructuras de datos se encuentran alineadas en memoria que cuando no lo están. En este caso, sí podemos vemos una diferencia notable en el rendimiento para las estructuras xc_func_type y $xc\_work\_gga\_c\_t$, el rendimiento para la estructura $xc\_func\_info\_type$ es apenas mejor y finalmente no se detectan mejoras en el rendimiento de la copia de datos para la estructura $xc\_gga\_work\_x\_t$.

Listing 4.1: Ejemplo de una estructura de datos de `Libxc`

```
struct xc_func_type {
        const void *info;
        int nspin;
        ...
```

Estructura	Descripción	Tamaño (bytes)
xc_func_type	contiene la configuración básica del funcional	192
xc_func_info_type	contiene la información específica de cada funcional	96
xc_gga_work_c_t	contiene los parámetros comunes para los funcionales de la familia gga que realizan el intercambio de energía	552
xc_gga_work_x_t	contiene los parámetros comunes para los funcionales de la familia gga que realizan el calculo de correlacion	64

Tab. 4.5: Estructura de datos utilizada por los funcionales de `Libxc`.

};

Listing 4.2: Ejemplo de una estructura de datos de `Libxc` alineada utilizando notacion de `CUDA`

```
struct __align__(256) xc_func_type {
        const void *info;
        int nspin;
        ...
};
```

Configuración automática del tamaño de bloques

Todas las técnicas hasta ahora aplicadas ayudaron a reducir el consumo de memoria por parte de los funcionales, pero aún no se pudo obtener una mejora en el uso de los bloques de memoria de la placa de video. Si bien en las secciones anteriores se muestran los resultados obtenidos al modificar la cantidad de threads asignados para el cómputo en paralelo, el objetivo es buscar una forma automática de realizar la asignación de recursos en memoria para la ejecución de cada kernel en función del tamaño de los datos de entrada. La sospecha es que si realizamos una configuración dinámica del tamaño de bloques, la performance podría mejorar.

El framework de desarrollo de `CUDA` provee varias funciones que asisten para realizar esta tarea. Mediante el uso de estas funciones es posible calcular cuál es el mejor tamaño de bloque que maximiza el uso de la memoria para los kernels.

El siguiente paso consiste en modificar `Libxc` para que, previo a la ejecución del cálculo de los funcionales y en función del tamaño de los datos de entrada, realice de manera automática el cálculo del tamaño de bloque que necesita la placa de video para ejecutar el kernel y así buscar la máxima ocupación.

En los gráficos de la figura 4.17 y 4.20 se ven los porcentajes de los cuellos de botella encontrados por el *profiler* de Nvidia en la ejecución del funcional `gga_c_pbe`. El primer gráfico contiene los datos de la traducción original mientras que el segundo contiene los datos que se obtuvieron luego de aplicar las optimizaciones descriptas en las secciones anteriores además de la configuración automática de la memoria asignada a la ejecución de los kernels.

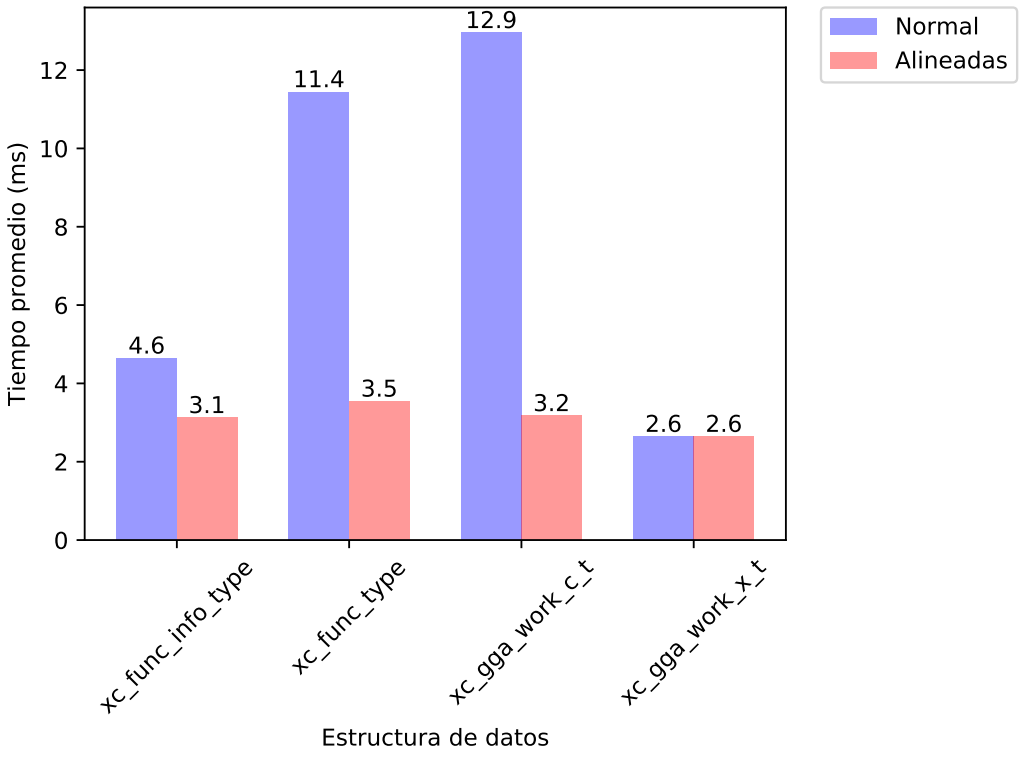

Fig. 4.18: Tiempo promedio de copia de los *structs* a memoria de la placa de video cuando los datos se encuentran alineados y cuando no, obtenidos luego de 100 corridas. Se omiten las barra de errores ya que los mismos son menores a $1x10^{-3}$.

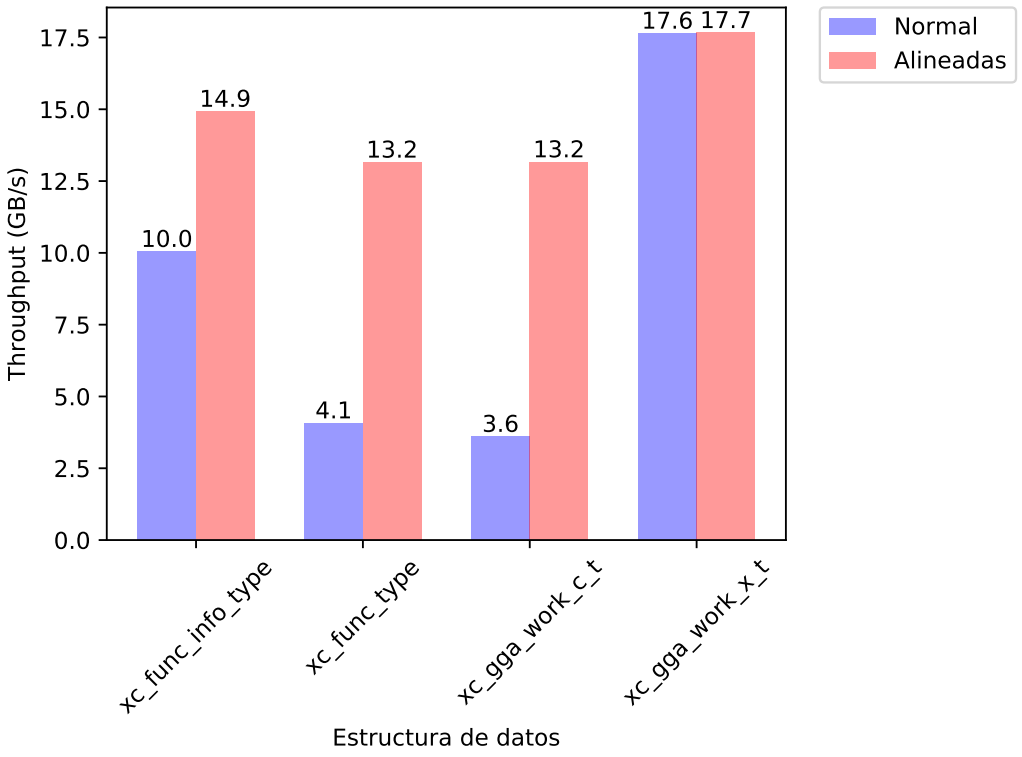

Fig. 4.19: Rendimiento de copia de los *structs* a memoria de la placa de video cuando los datos se encuentran alineados y cuando no, obtenidos luego de 100 corridas. Se omiten las barra de errores ya que los mismos son menores a $1x10^{-2}$.

El gráfico de la figura 4.21 muestra la evolución del porcentaje de bloques ocupados del código original con respecto al porcentaje de ocupación de bloques obtenido luego de las optimizaciones aplicadas. Si bien se obtuvieron mejoras, se busca que tengan un impacto mayor. Idealmente, para obtener un buen resultado se necesitaría obtener un porcentaje de ocupación mayor al 60 % para asegurarse de que está utilizando correctamente los recursos de la placa de video.

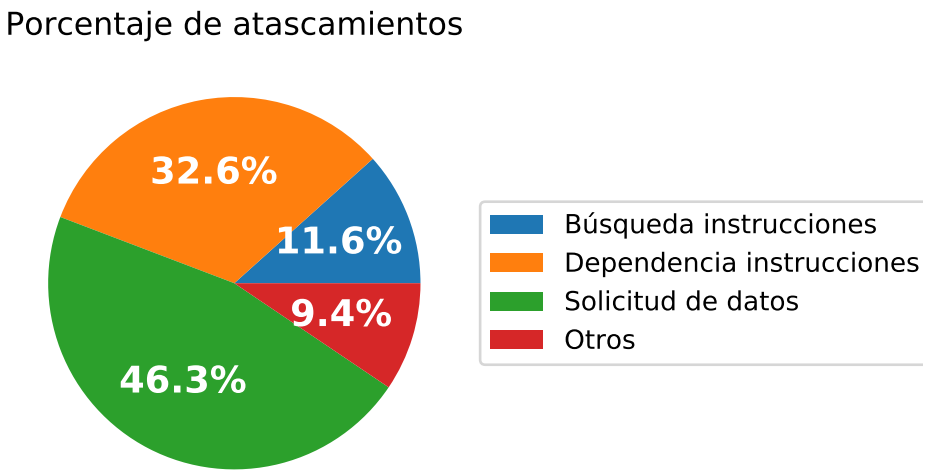

Fig. 4.20: Porcentajes de causas de cuellos de botella (obtenido de 100 corridas) luego de aplicar configuración automática de memoria para el funcional *gga_c_pbe* de `Libxc`.

4.6.6. Modificando la forma de compilación

El próximo paso en el intento de mejorar el rendimiento del código de los funcionales de `Libxc` consiste en realizar pruebas con distintos compiladores. La familia de compiladores que se utiliza para realizar estas pruebas pertenecen al proyecto `LLVM`. Particularmente, se utiliza el compilador `clang` que provee soporte para compilar código `CUDA`. Dado que el funcional que más problemas de performance presenta es el funcional `gga_c_pbe`, se centran los esfuerzos en él.

Se espera que con alguna de estas herramientas y, utilizando las distintas opciones de optimizaciones disponibles, se logre obtener un código optimizado para `CUDA`. En esta sección se omiten las combinaciones de parámetros que se probaron sobre las opciones de compilación ya que son demasiadas para ser incluídas directamente. En el apéndice E se encuentran detalladas todas las opciones de compilación que se utilizaron con las herramientas de `LLVM` para intentar optimizar el código de los funcionales.

Compilando código CUDA con clang

La primera de las pruebas es reemplazar directamente el compilador `nvcc` por el compilador `clang` y realizar las pruebas de performance para el funcional `gga_c_pbe`. Luego, se busca obtener resultados de los tiempos de ejecución y comparar con que compilador obtenemos mejores resultados.

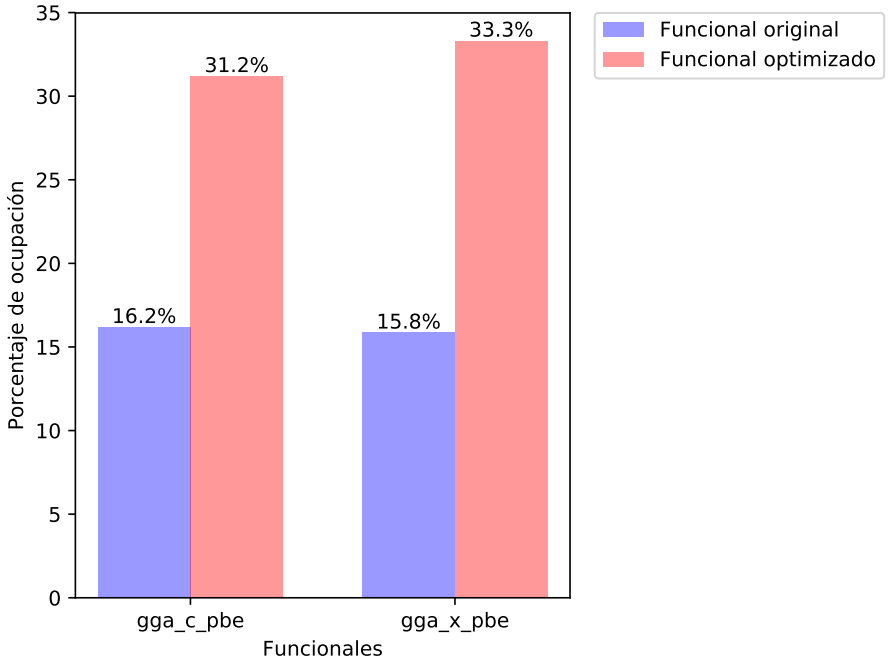

Fig. 4.21: Porcentaje de ocupación de bloques del código de los funcionales de `Libxc`.

Assembler de CUDA

La próxima prueba consiste en generar código *assembler* de `CUDA` a partir del código del funcional. El código *assembler* de `CUDA` se conoce con el nombre de `nvptx` y se genera directamente con el compilador `nvcc` indicándole que la salida de la compilación sea directamente código *assembler*. Una vez que se dispone del código *assembler*, se debe modificar la forma en que se realizan las llamadas a los funcionales que se ejecutan en la placa de video. Para esto se debe utilizar una API de `CUDA` que permite cargar en tiempo de ejecución el código *assembler* del funcional. Al realizar las pruebas con este procedimiento, se dispone de un manejo profundo de los recursos de la placa de video. Por ejemplo, en cada llamada podemos configurar manualmente el tamaño de los bloques de memoria, la cantidad de threads que utilizan las funciones y el tamaño máximo de memoria compartida. El procedimiento para realizar estas pruebas es el siguiente:

1) Compilar el código del funcional con `clang` y generar código intermedio.

2) Aplicar optimizaciones al código intermedio.

3) Generar código `nvptx` con el compilador `nvcc`.

4) Realizar las pruebas variando las configuraciones de memoria.

Clang y LLC

La siguiente prueba consiste en generar código `nvptx` utilizando las herramientas provistar por `LLVM`. La razón detrás de utilizar este conjunto de herramientas se debe a que el compilador `nvcc` está desarrollado sobre `LLVM`. Para realizar estas pruebas se instaló `LLVM` versión 6.0.

Luego, se realizó la compilación del código del funcional `gga_c_pbe` con `clang` para generar código intermedio sobre el cual podemos generar código *assembler* específico para la arquitectura de Nvidia. Con el código *assembler* de `CUDA` se volvieron a realizar las pruebas de stress y se compararon los resultados de la ejecución con los resultados anteriores.

El gráfico de la figura 4.22 muestra los tiempos de ejecución del funcional `gga_c_pbe`, luego de realizar 100 corridas, que se obtienen al generar el código fuente con los compiladores que fueron previamente mencionados. En el mismo podemos ver que el mayor tiempo de ejecución se dá cuando se convierte el código del funcional a assembler de cuda. Como contrapartida podemos ver que con el compilador *clang* se obtienen los mejor tiempos, esto era de esperarse ya que el compilador *nvcc* de Nvida está basado en esta tecnlogía. Otro de los problema que podemos ver es que a medida que se incrementa la cantidad de *threads*, también aumenta el tiempo de ejecución, al incrementar la cantidad de *threads* en la ejecución del funcional esperabamos que el resultado sea el inverso, pero podemos ver que esto no es asi. La razon detrás de este fenomeno es la gran cantidad de variables auxiliares que utiliza el funcional `gga_c_pbe`, lo que produce un incremento de los accesos a memoria global de la placa de video ya que la cantidad de registros asignados a la ejecución de cada *thread* resulta insuficiente.
Aparentemente ninguna de las optimizaciones automáticas aplicadas al utilizar los distintos compiladores resuelve el problema de la gran cantidad de registros que utiliza el funcional `gga_c_pbe` para realizar las cuentas.

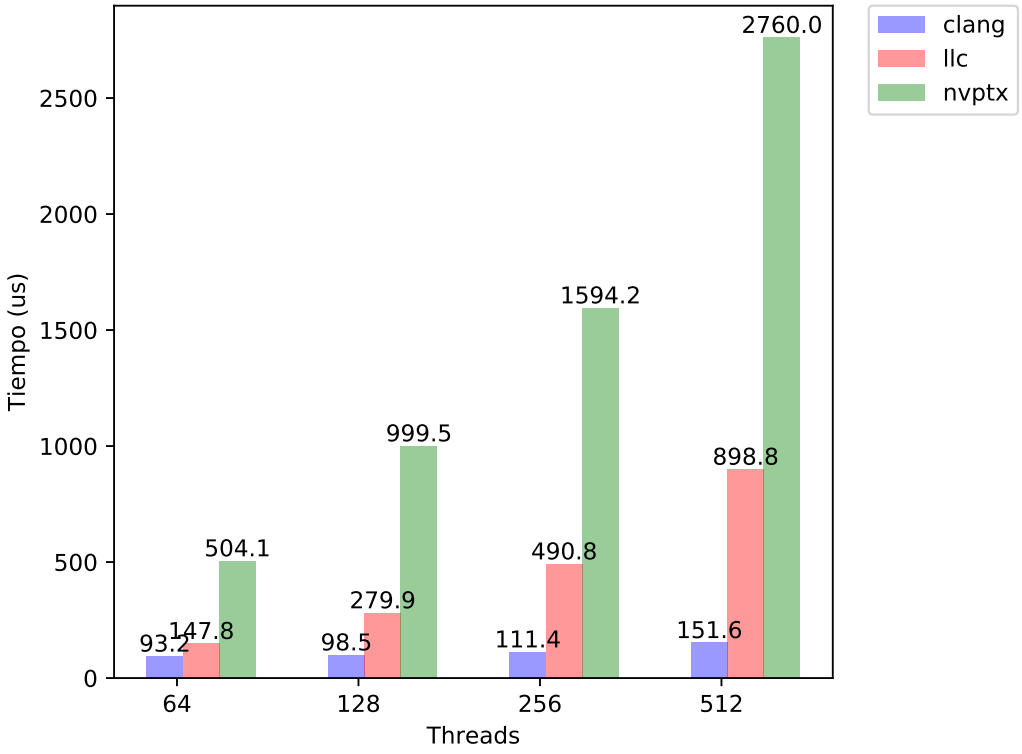

Fig. 4.22: Tiempo de ejecución del funcional `gga_c_pbe` para las distintas formas de compilacion, obtenidos luego de 100 corridas. Se omiten las barra de errores ya que los mismos son menores a $1x10^{-6}$.

Rendimiendo según compiladores

La tabla 4.6 muestra el tiempo de ejecución del funcional `gga_c_pbe` que se obtuvo luego de realizar las pruebas de stress utilizando diversos compiladores.

Compilador	Threads	Tiempo(us)
clang - llc	64	147,79
clang - llc	128	279,93
clang - llc	256	490,77
clang - llc	512	898,82
nvcc - ptx	64	504,12
nvcc - ptx	128	999,49
nvcc - ptx	256	1594,2
nvcc - ptx	512	2760
clang	64	93,249
clang	128	98,5
clang	256	111,37
clang	512	151,56
nvcc	64	46,16
nvcc	128	47,18
nvcc	256	51,52
nvcc	512	68,96

Tab. 4.6: Tiempo de ejecución del funcional `gga_c_pbe` (obtenido de 100 corridas) en función de la cantidad de *threads* asignados al *kernel* para los distintos compiladores que fueron utilizados.

4.7. Completando la traducción

Se procedió a realizar la traducción del resto de las familias de los funcionales que implementa `Libxc` (LDA y MGGA). Dado que `LIO` solo implementa funcionales de la familia GGA, únicamente se realizaron pruebas de integración y no simulaciones, dado que no existen datos para constrastar los resultados.

Como resultado adicional de la traducción realizada, se generó un procedimiento para hacer traducciones de funcionales de `Libxc` escritos para CPU a GPU para cada una de las familias de funcionales existentes. Debido a que las implementaciones de los funcionales son muy diversas y, dado que el código varía según el funcional y según la famila a la cual pertenece, no es posible generar un procedimiento automatizado que realice las traducciones directamente sin intervención del programador.

5. RESULTADOS

A continuación se presentan los resultados finales obtenidos en la ejecución de las simulaciones.

5.1. Tiempos de ejecución de las simulaciones

Los tiempos de ejecución de las simulaciones se midieron utilizando la implementación de *timers* que provee `LIO`. Para medir la cantidad de llamadas a las funciones de `CUDA` se utilizó la herramienta `nvprof`, al igual que se hizo en las secciones anteriores.

Los casos de prueba utilizados corresponden a moléculas del grupo *Hemo* y *Caroteno*. Ambos grupos fueron elegidos por ser representativos de tamaño medio de los sistemas normalmente simulados. El detalle químico está incluido en el Apéndice B.

El funcional utilizado para todas las simulaciones es el funcional definido por Perdew, Burke & Ernzerhof, que es el único funcional que se encuentra implementado en `LIO` y con el cual se pueden comparar los resultados.

5.1.1. Resultados obtenidos utilizando la versión original de Libxc

En el gráfico de la figura 5.1 se ve la comparación entre los tiempos de las simulaciones cuando se utiliza la versión original de `LIO` contra la que que utiliza la biblioteca de `Libxc`. Dichos resultados corresponden a la primera implementación de la solución en la que se utiliza la versión original de `Libxc`.

El gráfico de la figura 5.2 muestra los tiempos que insume la simulación en realizar solo el cálculo de los funcionales.

Los resultados obtenidos al utilizar `LIO` con `Libxc` en su versión original no son buenos. Esto era de esperarse ya que para estos experimentos se utiliza la implementación original de `Libxc` que realiza los cálculos en CPU.

Por lo tanto, para que `Libxc` pueda realizar el cálculo de los funcionales, los datos deben ser copiados desde la memoria de la placa de video a memoria de la máquina donde está instalada la placa. Luego de que `Libxc` realiza los cálculos, los resultados deben ser copiados nuevamente a la placa de video para que `LIO` continúe con la simulación. No obstante, si bien los tiempos de las simulaciones son más altos, esta versión de `LIO` cuenta con más de 180 nuevos funcionales para realizar simulaciones, ampliando notoriamente las posibilidades de esta aplicación.

5.1.2. Resultados obtenidos utilizando la primera versión de Libxc en GPU

En el gráfico de la figura 5.3 podemos ver la comparación entre los tiempos de las simulaciones cuando utilizamos la versión original de `LIO` contra la versión que utiliza la implementación en GPU de Libxc.

El gráfico de la figura 5.4 muestra los tiempos que insume la simulación en realizar sólo el cálculo de los funcionales.

En este caso se comienza a notar una mejora significativa en los tiempos de las simulaciones entre la versión original de `Libxc` y la versión que realiza los cálculos de los

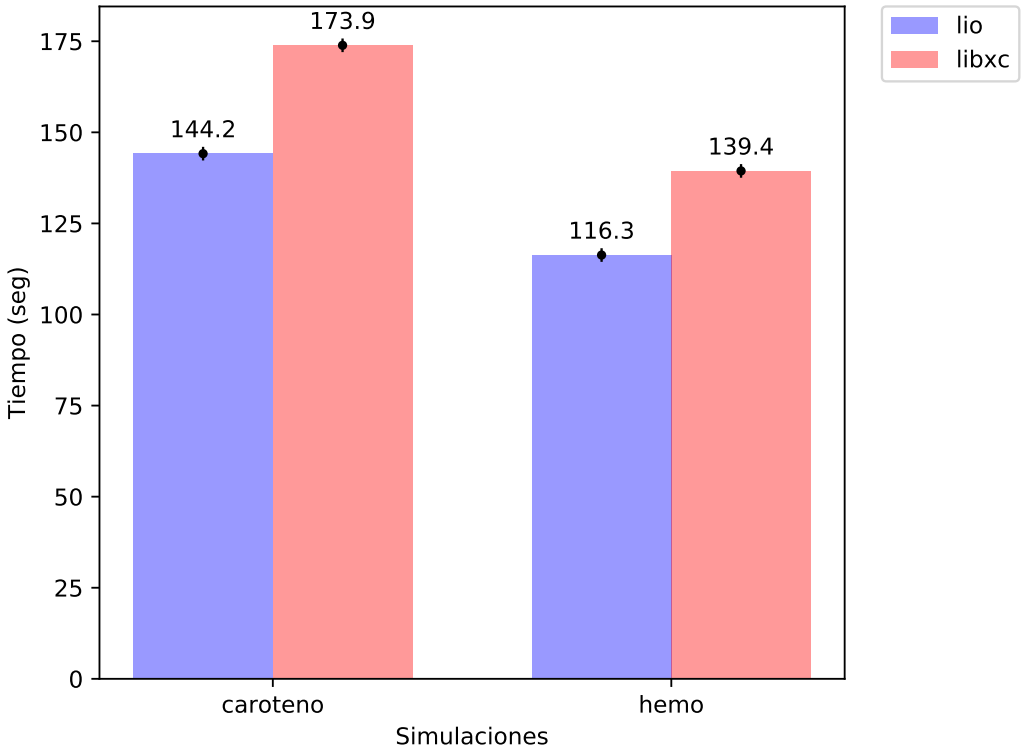

Fig. 5.1: Comparación de los tiempos totales de las simulaciones (obtenido de 100 corridas) entre `LIO` y la versión original de `Libxc` que funciona en CPU.

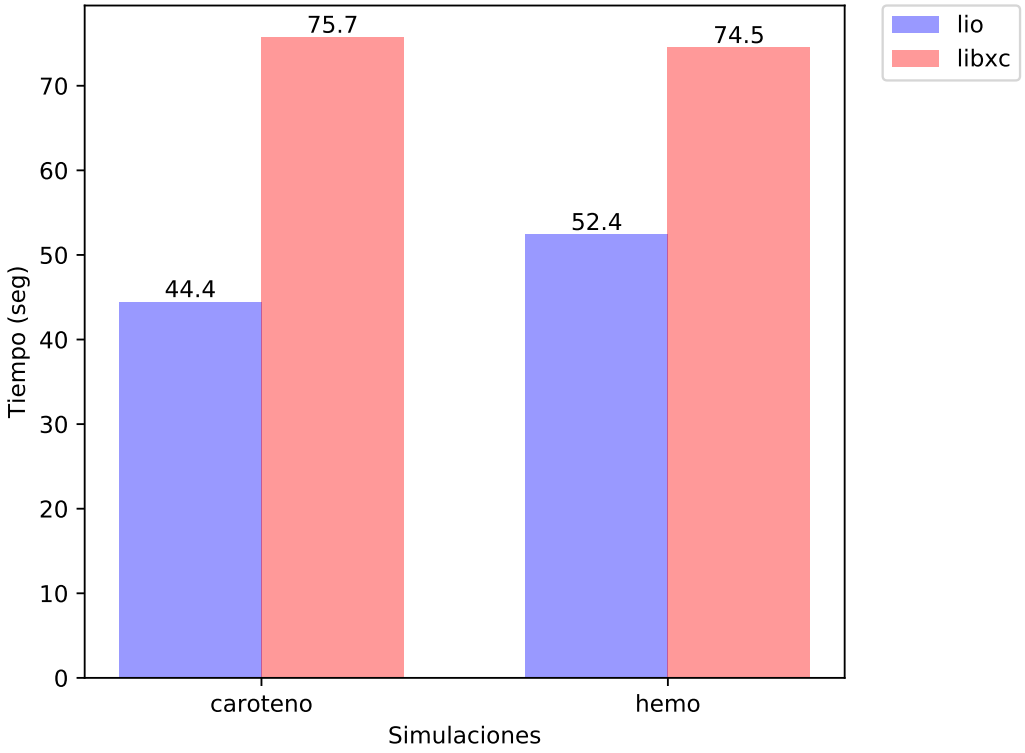

Fig. 5.2: Tiempo que insume la simulación en calcular los funcionales de intercambio y correlación entre `LIO` y la versión original de `Libxc` que funciona en CPU, obtenidos luego de 100 corridas. Se omiten las barra de errores ya que los mismos son menores a $1x10^{-7}$.

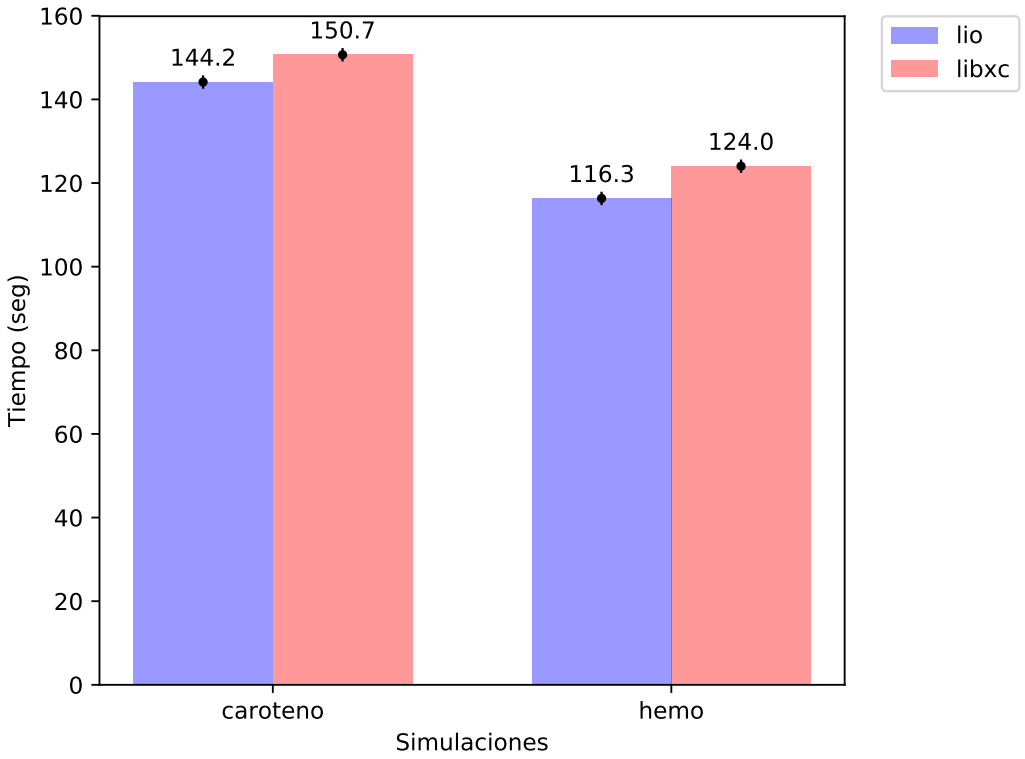

Fig. 5.3: Comparación de los tiempos totales de las simulaciones entre `LIO` y la primera implementación de `Libxc` en GPU, obtenidos luego de 100 corridas.

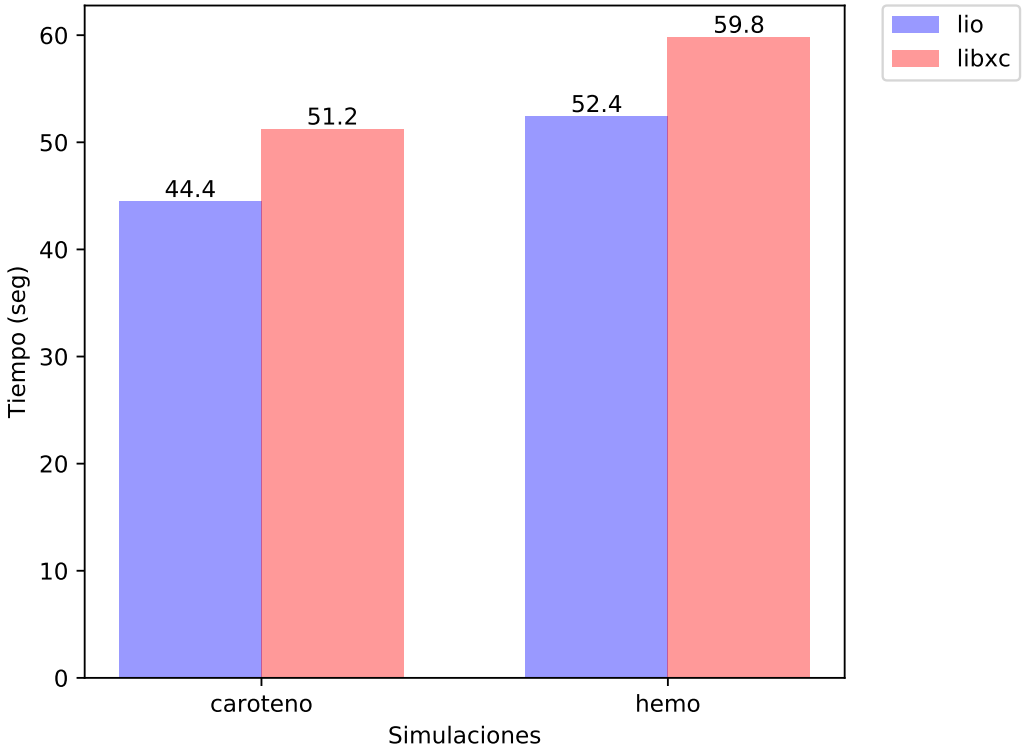

Fig. 5.4: Tiempo que insume la simulación en calcular los funcionales de intercambio y correlación entre `LIO` y la primera implementación de `Libxc` en GPU, obtenidos luego de 100 corridas. Se omiten las barra de errores ya que los mismos son menores a $1x10^{-6}$.

funcionales en GPU. Como es de esperar, los resultados no superan los obtenidos con la implementación original de `LIO`.

Threads y registros

Una de las tareas más *artesanales* del trabajo consistió en realizar pruebas de performance variando la cantidad de *threads* que se utilizan para paralelizar el cálculo de los funcionales. Como complemento, también se realizaron pruebas variando la cantidad máxima de registros de memoria que utilizan los kernels para encontrar la mejor configuración de estas dos variables con las cuales se obtienen los mejores resultados. Los datos que muestra la tabla 5.1 corresponden a los resultados que se obtuvieron al variar la cantidad de *threads* y registros para la primera versión en GPU del funcional `gga_c_pbe`. Como puede verse, a mayor cantidad de *threads* se observa un decremento de la performance. Esto significa que la traducción directa del funcional de CPU a GPU no escala de la manera esperada al incrementar la cantidad de *threads* que se utilizan para paralelizar el cálculo. Por otro lado, podemos notar que independientemente de la cantidad de *threads* que se utilicen para paralelizar el cálculo, siempre se obtienen mejores resultados cuando el kernel se configura para que utilice 63 registros de memoria (cantidad máxima permitida por hardware).

Threads	#Registros	Tiempo(us)
512	16	684,46
512	32	545,91
512	63	444,31
256	16	484,56
256	32	357,2
256	63	275,29
128	16	425,43
128	32	319,26
128	63	243,34
64	16	383,64
64	32	305,66
64	63	228,5

Tab. 5.1: Tiempo de ejecución del funcional `gga_c_pbe` en función de la cantidad de registros y *threads* asignados al *kernel* para la primera versión de `Libxc` en GPU, obtenidos luego de 100 corridas.

5.1.3. Resultados obtenidos utilizando la versión optimizada de Libxc en GPU

En el gráfico de la figura 5.5 se ve la comparación entre los tiempos de las simulaciones cuando utilizamos la versión original de `LIO` contra la versión que utiliza la versión optimizada de la biblioteca de `Libxc`.

El gráfico de la figura 5.6 muestra los tiempos que insume la simulación en realizar sólo el cálculo de los funcionales.

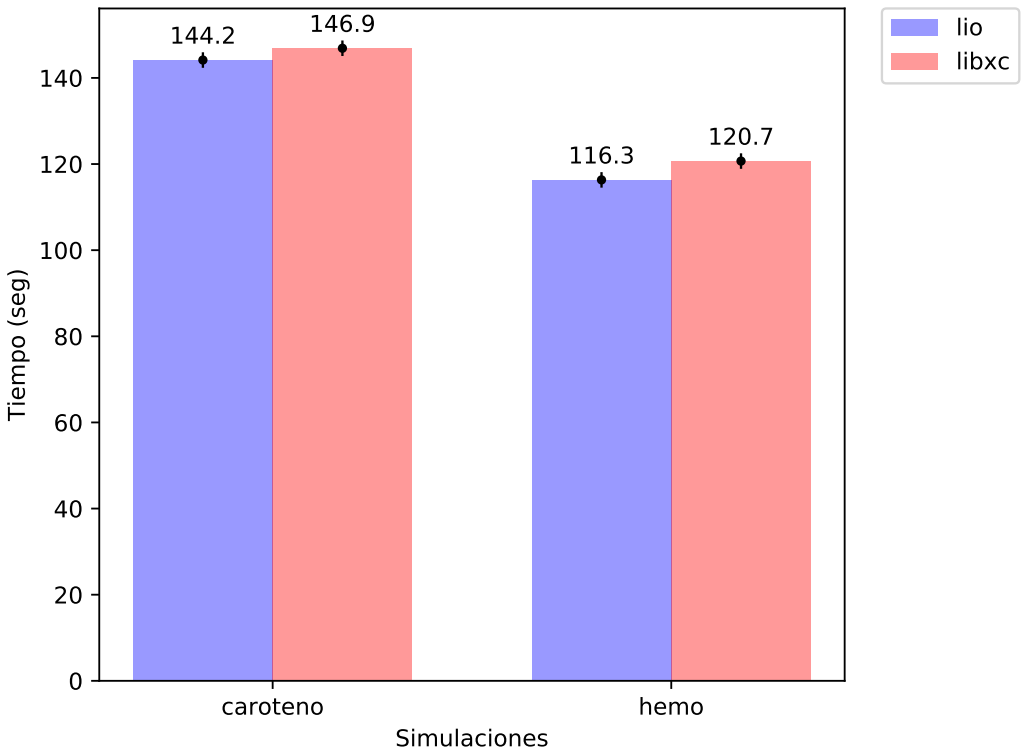

Fig. 5.5: Comparación de los tiempos totales de las simulaciones entre `LIO` y la versión optimizada de la biblioteca Libxc, obtenidos luego de 100 corridas.

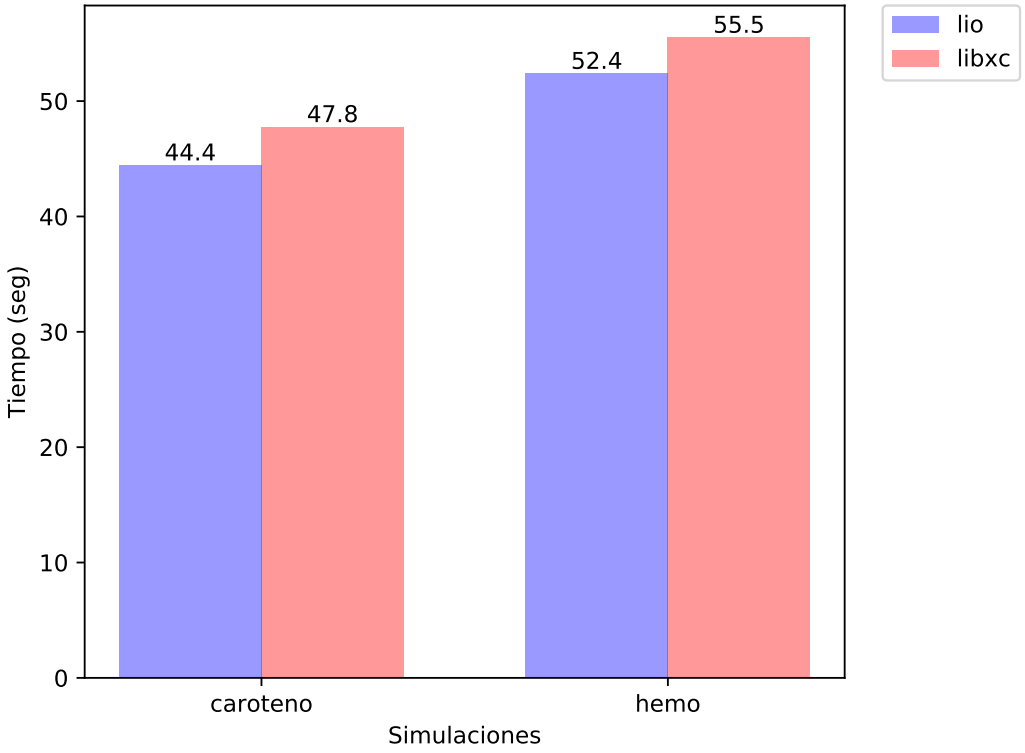

Fig. 5.6: Tiempo que insume la simulación en calcular los funcionales de intercambio y correlación de `LIO` y la versión optimizada de la biblioteca de Libxc, obtenidos luego de 100 corridas. Se omiten las barra de errores ya que los mismos son menores a $1x10^{-6}$.

Con la versión optimizada de `Libxc` se obtiene una mejora en los tiempos de las simulaciones con respecto a sus versiones anteriores. Si tenemos en cuenta que utilizamos una biblioteca que no fue diseñada para ejecutarse en GPU, el impacto en las simulaciones es marginal. Además, esta ampliación de la versión de `LIO` cuenta con más de 30 funcionales de `Libxc` preparados para ser ejecutados en GPU ampliando drásticamente las posibilidades a sus usuarios.

El gráfico de la figura 5.7 muestra el porcentaje de la diferencia con la versión original de `LIO` de los tiempos totales de las ejecuciones de las simulaciones para las distintas implementaciones de la biblioteca de `Libxc`. En este gráfico se puede ver cómo mejoran los tiempos de ejecución en las distintas versiones de la biblioteca de Libxc. Luego de aplicar varias técnicas de optimización a la implementación de los funcionales en GPU, logramos que el impacto en simulaciones sea marginal ($< 4\,\%$) con respecto a la implementación original de `LIO`. Sin embargo debemos volver a mencionar que, lo poco que se pierde en tiempo se gana en diversidad ya que ahora la nueva implementación de `LIO` cuenta con una amplía gama de nuevos funcionales.

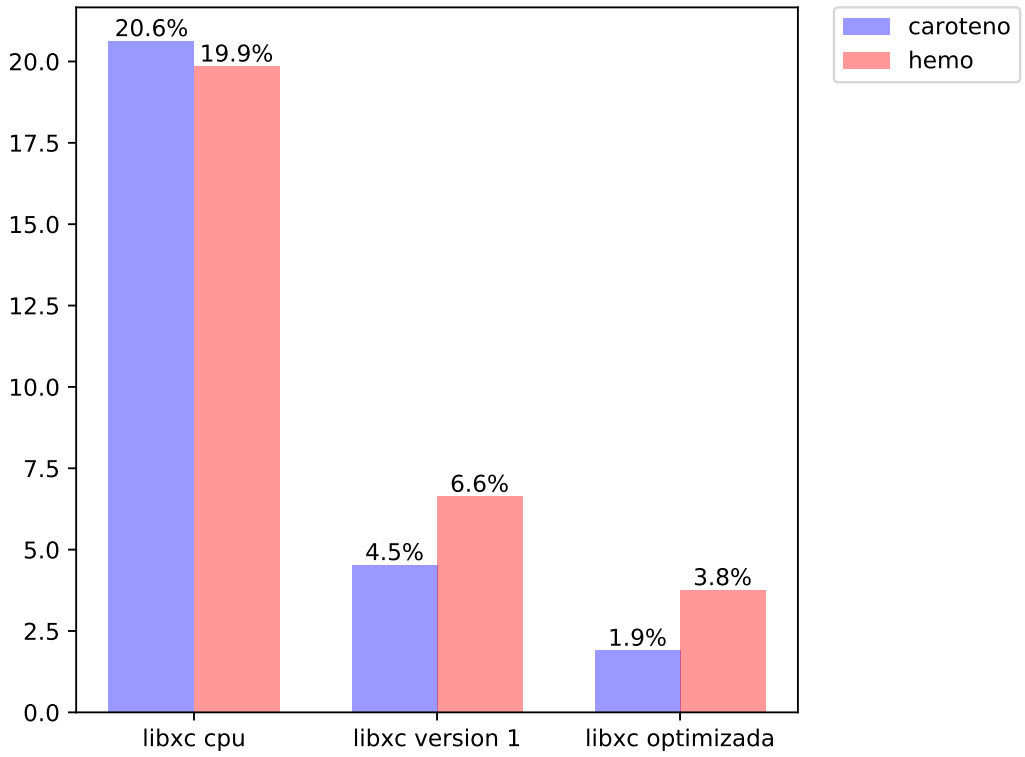

Fig. 5.7: Diferencia expresada en porcentajes de los tiempos totales de las simulaciones para las distintas versiones de la biblioteca de `Libxc` con respecto a la versión original de `LIO`, luego de haber realizado 100 simulaciones. Las barras de errore se omiten ya que estos son menores a $1x10^{-3}$

5.2. Ocupación de bloques de memoría

El gráfico de la figura 5.8 muestra los resultados finales de la ocupación de bloques en memoria de la placa de video que usa la versión final de la biblioteca de `Libxc` contra los bloques que utilizaba la primera versión de la biblioteca para los funcionales `gga_c_pbe` y `gga_x_pbe`. Como se puede ver en los gráficos, luego de aplicar varias técnicas de optimización se logró duplicar el uso de bloques de memoria disponibles en la placa de video para realizar el cálculo de los funcionales.

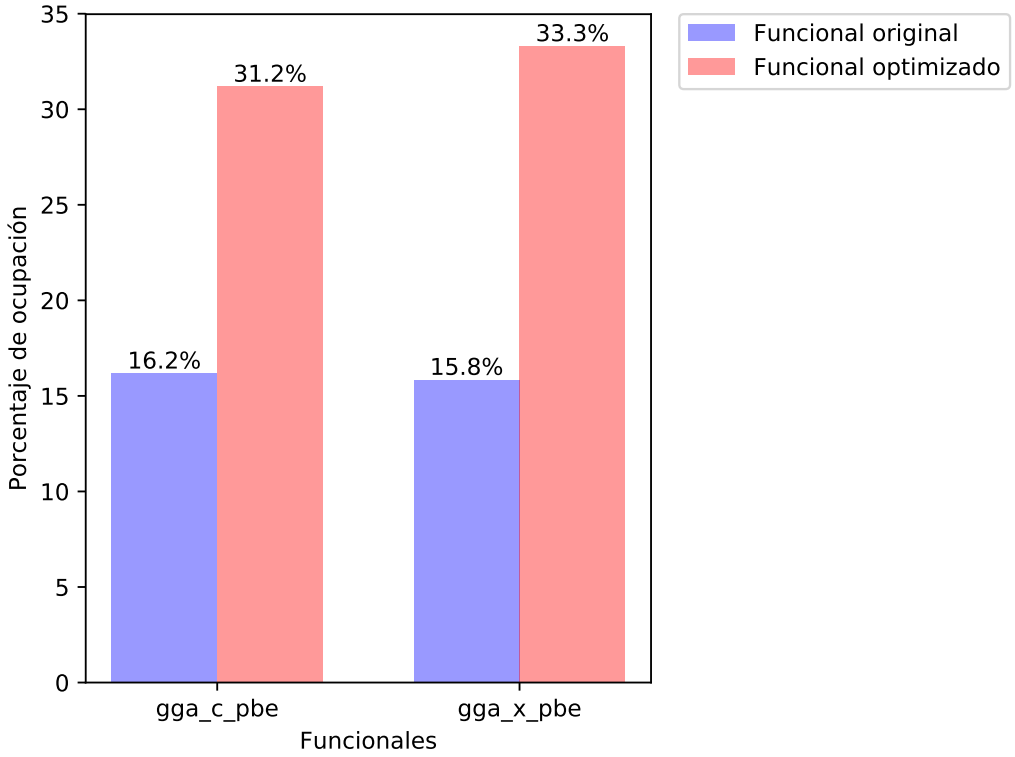

Fig. 5.8: Porcentaje de ocupación de bloques del código de los funcionales de la biblioteca de `Libxc`.

5.3. Atascamientos (stalls)

En los gráficos de las figuras 5.9 y 5.10 se ven los problemas que presenta el funcional `gga_c_pbe` en su versión original y en la versión optimizada.

Los gráficos muestran que se logró obtener una mejora del 9 % con respecto a los problemas de acceso a memoria. Por otro lado, también podemos notar que la dependencia de instrucciones se incrementó en un 9 %. Esto está directamente relacionado a la gran cantidad de variables que utilizan los algoritmos de los funcionales. Es por esta razón que no se pueden observar mejoras significativas en los tiempos de ejecución. Finalemente, no se pudo avanzar más sobre estos problemas ya que para hacerlo se requiere realizar modificaciones a la forma en que se realizan los cálculos de los funcionales y eso queda fuera del alcance en el presente trabajo.

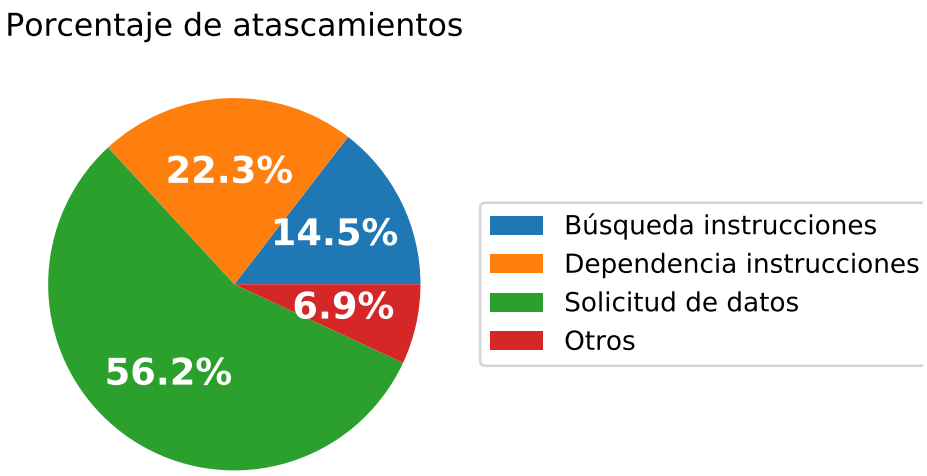

Fig. 5.9: Porcentaje de causas de cuellos de botella en el cálculo del funcional `gga_c_pbe` de la versión original de la biblioteca de `Libxc`, obtenidos luego de 100 corridas.

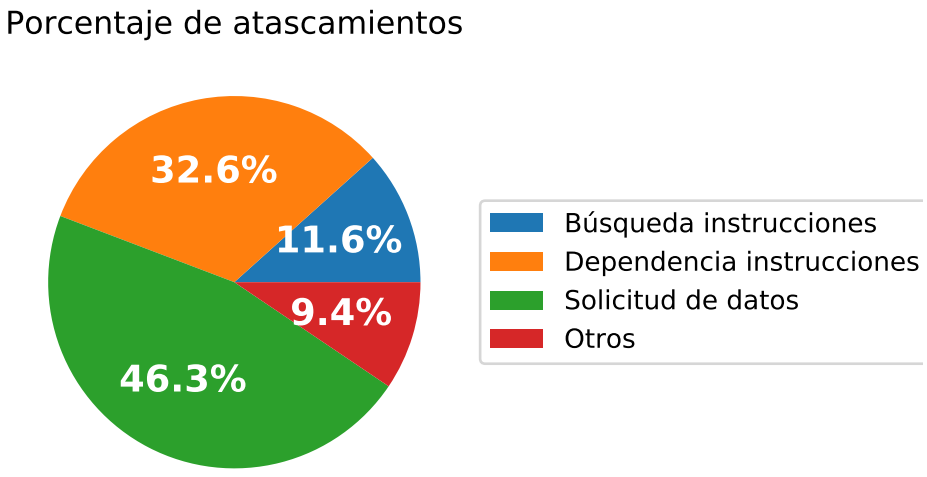

Fig. 5.10: Porcentajes de causas de cuellos de botella en la versión optimizada de `Libxc`, obtenidos luego de 100 corridas.

5.4. Configuración de memoria cache

En la sección 4.6.5, la tabla 4.2 mostraba cómo variaba el porcentaje de causas de atascamientos (stalls) de la versión original del funcional `gga_c_pbe` para las distintas configuraciones del uso de memoria cache de la placa de video.

Configuración de Cache	Búsqueda de instrucciones	Dependencia de instrucciones	Solicitud de datos	Otros
Default	14,54 %	22,33 %	56,19 %	6,94 %
cudaFuncCachePreferShared	13,69 %	23,56 %	55,12 %	7,13 %
cudaFuncCachePreferL1	10,8 %	24 %	57,19 %	8,01 %
cudaFuncCachePreferEqual	13,06 %	25,53 %	53 %	8,41 %

Tab. 5.2: Porcentaje de causas de cuellos de botella del funcional `gga_c_pbe` original, para las distintas configuraciones de memoria cache, obtenidos luego de 100 repeticiones.

En la tabla 5.2 se muestra cómo varía el porcentaje de cuellos de botella según la configuracíon de la memoria cache que ha sido configurado en el funcional. Como se introdujo en las secciones anteriores, la mayor cantidad de problemas se encuentra en los accesos a memoria principal de la placa de video y ninguna de las configuraciones probadas permite obtener una mejora en performance.

En la tabla 5.3 se ve cómo varían los porcentajes de las causas de los cuellos de botella según la configuración de memoria cache para el funcional `gga_c_pbe` luego de aplicarle las optimizaciones.

Configuración de Cache	Búsqueda de instrucciones	Dependencia de instrucciones	Solicitud de datos	Otros
Default	10,57 %	30,39 %	43,26 %	15,78 %
cudaFuncCachePreferShared	10,69 %	30,06 %	43,12 %	16,13 %
cudaFuncCachePreferL1	10,8 %	24 %	52,19 %	13 %
cudaFuncCachePreferEqual	11,06 %	29,53 %	43 %	16,4 %

Tab. 5.3: Porcentaje de causas de cuellos de botalla del funcional `gga_c_pbe` optimizado, para las distintas configuraciones de memoria cache, obtenidos luego de 100 repeticiones.

Si bien en la tabla 5.3 se puede notar un decremento en los porcentajes a los accesos de datos en la memoria principal, también se ve cómo se incrementan los problemas por la dependencia de instrucciones. Cabe resaltar que utilizar la configuración de memoria `cudaFuncCachePreferL1` provoca un gran incremento en la solicitud de datos con respecto a las otras configuraciones disponibles.

5.5. Estructuras de datos alineadas en memoria

Los gráficos de la figura 4.18 y de la figura 4.19 muestran cómo varía el tiempo de copia a memoria de la placa de video de las estructuras de datos que utiliza `Libxc` en los casos en los que las estructuras se encuentran alineadas en memoria y cuando no lo están.

Los tiempos de copia de las estructuras de datos a la placa de video se incrementan significativamente cuando las estructuras de datos superan los 200 bytes. Por otro lado, no se notan mejoras en los rendimientos si las estructuras de datos son menores a 64 bytes (ver tabla 5.4) .

Estructura	Tamaño (bytes)
`xc_gga_work_x_t`	64
`xc_func_info_type`	96
`xc_func_type`	192
`xc_gga_work_c_t`	552

Tab. 5.4: Estructura de datos utilizada por los funcionales de `Libxc`.

El mismo patrón del gráfico 4.18 se puede ver en la figura 4.19 que muestra el *throughput* que se obtiene cuando los datos se encuentran alineados en memoria y cuando no lo están.

5.6. Otros compiladores

En esta sección se presentan los resultados obtenidos cuando se utilizan distintos compiladores. La tabla 4.6 mostramos cómo fue variando la performance del funcional `gga_c_pbe` en función de los compiladores que fueron utilizados. En la misma podemos ver que al compilar el funcional `gga_c_pbe` con el compilador de Nvidia (*nvcc*) se obtuvieron los mejores resultados. Tambíen se ve que realizar la compilación con `clang` es mucho mejor que generar el código fuente del funcional y luego generar binario con el compilador de Nvidia (*nvcc - ptx*).

5.7. Calidad numérica de los resultados

Uno de los aspectos más preocupantes de la traducción de los funcionales de la biblioteca de `Libxc` y de su integración a `LIO` fue la diferencia entre los tipos de datos numéricos respecto a la precisión utilizada. Todos los funcionales de la biblioteca de `Libxc` están implementados utilizando números de doble precisión mientras que `LIO` realiza las simulaciones utilizando precisión mixta (doble o simple precisión dependiendo del caso). Este escenario realza la necesidad de comparar los resultados finales de las simulaciones para comprobar el error numérico de los resultados obtenidos de las sucesivas implementaciones de `Libxc` en GPU.

En la tabla 5.5 se ve el error numérico que se produce en el valor final de la energía calculado con las distintas implementaciones de la biblioteca de `Libxc`. La unidad correspondiente a los resultados que se obtienen de las simulaciones de `LIO` es la unidad atómica de energía *Hartree*, en donde 1 *Hartree* es igual a 627,5 kcal/mol y los errores menores a 1 kcal/mol se consideran aceptables.

Versión de Libxc	Hemo		Caroteno	
	error	error relativo	error	error relativo
Versión original en cpu	28×10^{-2}	12×10^{-6}	52×10^{-2}	13×10^{-6}
Primera implementación en gpu	37×10^{-2}	15×10^{-6}	12×10^{-2}	13×10^{-6}
Versión final en gpu	33×10^{-2}	31×10^{-6}	1×10^{-2}	13×10^{-6}

Tab. 5.5: Tabla comparativa del error numérico expresado en kcal/mol entre los resultados finales de las simulaciones de `LIO` para las distintas implementaciones de la biblioteca de `Libxc`.

6. CONCLUSIONES

La simulación numérica de problemas QM/MM es una herramienta moderna que ayuda a estudiar fenómenos químicos a partir de principios básicos de la física. Sin embargo, el elevado costo computacional asociado a estas técnicas limita tanto los sistemas y procesos que pueden ser estudiados así como la calidad de los modelos implementados. Por ello, el desarrollo y mejora de herramientas computacionales resulta de gran interés, permitiendo extender la frontera del tiempo simulado y estudiar fenómenos o sistemas no abarcables previamente.

En este trabajo, se estudió la ampliación de las capacidades del programa `LIO` mediante la integración con la biblioteca `Libxc`. Se pudo observar que es posible extender la funcionalidad de `LIO` rediseñando los algoritmos de cálculo de DFT para aprovechar todas las prestaciones disponibles de `Libxc`.

En la ampliación con la versión CPU de `Libxc`, se extendió la funcionalidad de `LIO` en al menos 180 nuevos funcionales para realizar experimentos preliminares y determinar si es factible realizar la traducción de los funcionales a GPU. Como es de esperar, esto produjo un deterioro en el rendimiento del código. Un análisis más de detallado reveló que la mayoría del tiempo de las simulaciones se insume en el cálculo de los funcionales por lo que se procedió a realizar su traducciones para que puedan ser ejecutadas en GPU.

En la ampliación de `LIO` con la versión GPU de la biblioteca de `Libxc`, la velocidad final alcanzada de las simulaciones es, en el peor caso, sólo un 4 % mayor a la obtenida con respecto a la versión original de `LIO`. Como contrapartida, esta nueva versión de `LIO` cuenta con más de 30 nuevos funcionales para realizar experimentos. Es de destacar que los beneficios de disponer de estos funcionales adicionales supera largamente el marginal costo en performance.

Realizar las optimizaciones en la traducción de los funcionales de `Libxc` no es una tarea sencilla, pero 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. Las mejoras pasaron por encontrar los limitantes de las distintas funciones del cómputo, reestructurando las paralelizaciones, almacenando más resultados temporales y aprovechando los cambios en la arquitectura de las GPU con respecto a las memorias *on-chip*. También se tuvo en consideración el comportamiento de las memorias caches, y de cómo aprovecharlas efectivamente para lograr una mejorar en el rendimiento, cosa que no es viable en otro tipo de arquitecturas. Corresponde remarcar que realizar optimizaciones a algoritmos que se ejecutan en la placa de video resultó una tarea que roza lo *artesanal* debido a la gran variedad de variables involucradas en el proceso (threads, distintos tipos de memoria, precisión numérica, arquitecturas, por nombrar algunas) y a la gran cantidad de experimentos que se deben realizar hasta encontrar los resultados esperados.

El proceso de adaptar y modificar programas diseñados originalmente para funcionar en CPU para que se ejecuten en GPU no es una tarea trivial. Requiere realizar un estudio profundo del problema a resolver, los tipos de datos y los procesos involucrados. Se necesitan muchas horas de capacitación previa así como también un importante cambio de paradigma en la forma de programación y de la forma de pensar los algoritmos. Durante la realización de este trabajo notamos que los resultados obtenidos al realizar una traduc-

ción directa de los funcionales de la biblioteca de `Libxc` sin realizar modificaciones a los algoritmos producen como resultado código de baja performance en GPU. Al comenzar a trabajar directamente sobre el código fuente de los funcionales y aplicando técnicas de optimizaciones específicas de programación en *gpgpu* pudimos obtener mejoras notables en el rendimiento, que si bien no son las esperadas, deja la puerta abierta para seguir trabajando en futuras mejoras.

Como corolario de aplicar sucesivas optimizaciones a la traducción de los funcionales se pudo establecer un procedimiento para realizar la traducción del resto de los funcionales de la biblioteca de `Libxc`. Lamentablemente no se pudo lograr automatizar dicho procedimiento ya que cada funcional define estructuras de datos auxiliares diferentes para realizar los cálculos. Esto dificulta la tarea de estandarizar el proceso de traducción mediante scripts o alguna técnica de traducción directa de código fuente.

Los caminos que se abren en base al trabajo realizado se puede resumir en:

Explotar más paralelismo

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

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

Traducir nuevos funcionales

1. Realizar la traducción a GPU de los 150 funcionales restantes de `Libxc`.

2. Explorar técnicas que permitan una automatización en la traducción.

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] 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

[3] 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

[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] M. A. Marques, M. J. Oliveira, T. Burnus, Libxc: A library of exchange and correlation functionals for density functional theory, Computer Physics Communications 183 (10) (2012) 2272 – 2281.
URL http://www.sciencedirect.com/science/article/pii/S0010465512001750

[7] 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.

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

[9] Nvidia, Cuda programming guide, Tech. rep., Nvidia (2017).

[10] 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

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

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

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

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

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

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

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

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

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

[20] 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

[21] 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.

[22] Nvidia Corporation, NVIDIA's Next Generation CUDA Compute Architecture: Maxwell, Tech. rep., Nvidia Corporation (2014).
URL http://www.nvidia.com/content/PDF/maxwell_white_papers/NVIDIAMaxwellComputeArchitectureWhitepaper.pdf

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

[24] 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: 3 de octubre de 2019.

[25] 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

[26] M. F. y Juan Pablo Darago, Optimización de cómputo qm/mm empleando arquitecturas masivamente paralelas., Master's thesis, Universidad de Buenos Aires (2015).

[27] C. Lattner, V. Adve, Llvm: a compilation framework for lifelong program analysis amp; transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86.

[28] 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

[29] 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

[30] J. P. Perdew, K. Burke, M. Ernzerhof, Generalized gradient approximation made simple, Phys. Rev. Lett. 77 (1996) 3865–3868.
URL https://link.aps.org/doi/10.1103/PhysRevLett.77.3865

[31] NVIDIA, Cuda c best practices guide (2017).
URL https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

Apéndice

A. EQUIPAMIENTO USADO PARA CORRER LAS PRUEBAS

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

Las pruebas de GPU se realizaron en las siguientes máquinas.

- 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
- 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
- FFYB - LAB6: Intel Core i5-3330 @ 3.0 GHz - 16GB DDR3 - Quadro P6000

Estas máquinas contaron con el siguiente software:

- C/C++ Compiler 2016
- Fortran Compiler 2016
- Nvidia CUDA 8.0

Las pruebas de CPU se realizaron en las siguientes máquinas:

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

Estas maquinas contaron con el siguiente software:

- C/C++ Compiler 2016
- Fortran Compiler 2016

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.

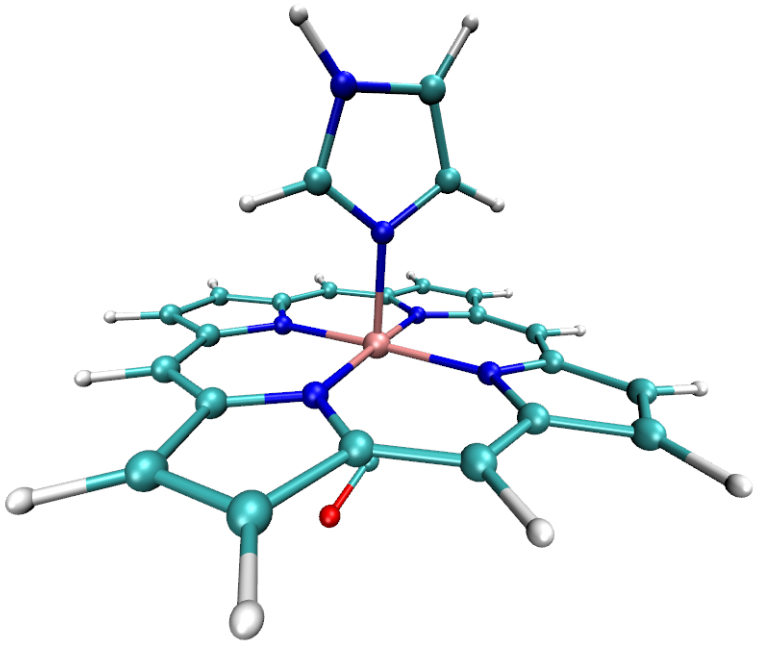

Fig. B.1: Render de hemo

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 gran número 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.

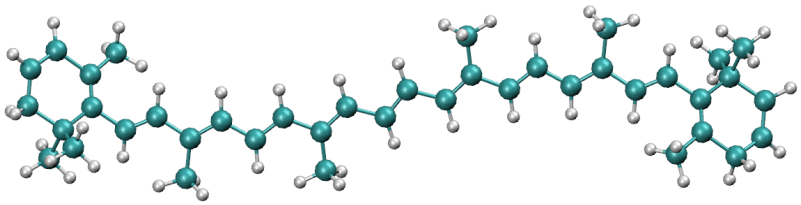

Fig. B.2: Render de caroteno

C. CANTIDAD DE VARIABLES DE LOS FUNCIONALES DE LIBXC

La tabla C.1 y la tabla C.1 muestran la cantidad de variables que utilizan los algoritmos que realizan los cálculos de los funcionales de Libxc para las familias de funcionales *gga* y *lda* respectivamante.

Funcional	#Variables	Funcional	#Variables
gga_x_2d_pbe	6	*gga_c_wi*	83
gga_x_pbea	6	*gga_x_beefvdw*	102
gga_k_tflw	7	*gga_x_am05*	103
gga_x_herman	7	*gga_x_am05*	103
gga_k_ol1	8	*gga_x_sogga11*	112
gga_x_g96	8	*gga_c_lm*	146
gga_k_ol2	9	*gga_c_op_xalpha*	167
gga_x_2d_b86	11	*gga_c_am05*	174
gga_x_rpbe	14	*gga_c_lyp*	194
gga_x_optx	15	*gga_c_rev_a*	198
gga_x_pbe	17	*gga_c_s1*	221
gga_x_09x	18	*gga_c_p86*	229
gga_x_2d_b86_m	18	*gga_x_htbs*	233
gga_x_mpbe	18	*gga_x_kt*	255
gga_x_rge2	18	*gga_x_n12*	404
gga_x_2d_b88	19	*gga_c_th1*	440
gga_x_b86	20	*gga_c_th3*	476
gga_k_pearson	21	*gga_c_gp*	507
gga_x_pw86	25	*gga_c_pbe*	544
gga_x_lag	26	*gga_c_pbecu*	533
gga_k_thakkar	27	*gga_c_th2*	569
gga_k_dk	30	*gga_c_an_e0*	572
gga_x_th_a	30	*gga_c_sogga11*	640
gga_x_ak13	30	*gga_x_pbepow*	642
gga_x_b88	30	*gga_x_ft97*	668
gga_x_bayesian	30	*gga_c_zpbeint*	683
gga_x	33	*gga_c_bmk*	706
gga_x_eg93	33	*gga_c_op_b88*	716
gga_x_ssb_sw	35	*gga_c_op_g96*	718
gga_x_ap	36	*gga_c_regtpss*	733
gga_x_sg4	36	*gga_c_op_pw91*	748
gga_x_vmt	38	*gga_c_zvpbeint*	748
gga_x_dk87	39	*gga_c_pw91*	752
gga_x_pbeint	40	*gga_c_op_pbe*	804
gga_x_lg93	43	*gga_c_b97*	813
gga_x_pbetrans	44	*gga_c_pbel*	814
gga_x_vmt84	48	*gga_c_cth_a*	916
gga_k_meyer	55	*gga_c_sg4*	934
gga_x_lv_rpw86	55	*gga_c_q2d*	1252
gga_c_w94	57	*gga_c_gapl*	1652
gga_x_airy	58	*gga_c_op*	1845
gga_x_pw91	58	*gga_x_hjs*	2133
gga_x_q2d	58	*gga_x_hjs_b88_v2*	2317
gga_x_bc	75	*gga_c_ga*	2445
gga_c_a	82	*gga_c_ft97*	7321

Tab. C.1: Funcionales de la familia *gga* y la cantidad de variables que utilizan para realizar los cálculos.

Funcional	#Variables
lda_c_2d_amgb	8
lda_rpa	8
lda_wigner	12
lda_k_tf	16
lda_x_2d	19
lda_x_1d_ehwlrg	26
lda_x	34
lda_x_zlp	48
lda_04	56
lda_lp96	60
lda h hiyo	63
lda_gombas	64
lda_k_zlp	64
lda_x_teter93	77
lda_hl	79
lda_gk72	102
lda_x_rel	108
lda_2d_prm	112
lda_c_1d_loos	128
lda_2d_amgb	145
lda_vwn_1	175
lda_vwn_rpa	175
lda_pz	177
lda_c_1d	187
lda_pw	231
lda_vwn	281
lda_vwn_4	281
lda_vwn_2	379
lda_vwn_3	415
lda_x_erf	458
lda_ml1	475
lda_x_ksdt	1344
lda_pk09	6912

Tab. C.2: Funcionales de la familia *lda* y la cantidad de variables que utilizan para realizar los cálculos.

D. TIEMPOS DE EJECUCIÓN DE LOS FUNCIONALES

A continuación detallamos los tiempos de ejecución de los funcionales *gga_c_pbe* y *gga_x_pbe* que fueron usados como caso de estudio de la implementación realizada. Las tablas D.1 y D.2 muestran los resultados para las primeras versiones de los funcionales. Las tablas D.3 y D.4 muestran los resultados de las versiones optimizadas de los mismos.

Threads	#Registros	Tiempo(us)
512	16	684,46
512	32	545,91
512	63	444,31
256	16	484,56
256	32	357,2
256	63	275,29
128	16	425,43
128	32	319,26
128	63	243,34
64	16	383,64
64	32	305,66
64	63	228,5

Tab. D.1: Tiempo de ejecución del funcional *gga_c_pbe* en función de la cantidad de registros y threads asignados al kernel para pa primera versión de Libxc en *gpu*.

Threads	#Registros	Tiempo(us)
512	16	19,96
512	32	11,42
512	63	10,33
256	16	15,26
256	32	10,54
256	63	10,45
128	16	11,55
128	32	9,13
128	63	8,9
64	16	11,26
64	32	9,89
64	63	9,53

Tab. D.2: Tiempo de ejecución del funcional *gga_x_pbe* en función de la cantidad de registros y threads asignados al kernel para pa primera versión de Libxc en *gpu*.

Threads	#Registros	Tiempo(us)
512	16	95,26
512	32	78,42
512	63	69,79
256	16	72,913
256	32	58,6
256	63	52,03
128	16	64,61
128	32	52,29
128	63	47,43
64	16	59,15
64	32	48,57
64	63	46,61

Tab. D.3: Tiempo de ejecución del funcional *gga_c_pbe* en función de la cantidad de registros y threads asignados al kernel para pa la versión final de Libxc en *gpu*.

Threads	#Registros	Tiempo(us)
512	16	13,02
512	32	10,19
512	63	5,47
256	16	9,66
256	32	8
256	63	7,97
128	16	8,2
128	32	7,13
128	63	7,07
64	16	8,12
64	32	7,7
64	63	7,42

Tab. D.4: Tiempo de ejecución del funcional *gga_x_pbe* en función de la cantidad de registros y threads asignados al kernel para pa la versión final de Libxc en *gpu*.

E. OPCIONES DE COMPILACIÓN

En esta sección vamos a detallar las opciones de compilación más utilizadas en las pruebas de compilación de los funcionales de *Libxc* para los distintos compiladores que se utilizaron en este trabajo.

clang - opt

A continuación se detallan las opciones que se utilizaron para compilar los funcionales de *Libxc* con *clang* para generar código intermedio. Luego a ese código intermedio se le aplicaron optimizaciones utilizando la herramienta *opt* de LLVM. Las opciones de compilador con las cuales se hicieron las pruebas son las siguientes:

```
-adce - Aggressive Dead Code Elimination
-add-discriminators  - Add DWARF path discriminators
-alignment-from-assumptions - Alignment from assumptions
-amode-opt - Optimize addressing mode
-argpromotion - Promote 'by reference' arguments to scalars
-bdce - Bit-Tracking Dead Code Elimination
-codegenprepare - Optimize for code generation
-constmerge - Merge Duplicate Global Constants
-dce - Dead Code Elimination
-deadargelim - Dead Argument Elimination
-die - Dead Instruction Elimination
-elim-avail-extern - Eliminate Available Externally Globals
-generic-to-nvvm - Ensure that the global variables are in the global address space
-global-merge - Merge global variables
-globaldce - Dead Global Elimination
-globalopt - Global Variable Optimizer
-indvars - Induction Variable Simplification
-infer-address-spaces - Infer address spaces
-inferattrs - Infer set function attributes
-instcombine - Combine redundant instructions
-instsimplify - Remove redundant instructions
-interleaved-access - Lower interleaved memory accesses to target specific intrinsics
-internalize - Internalize Global Symbols
-intervals - Interval Partition Construction
-ipconstprop - Interprocedural constant propagation
-ipsccp - Interprocedural Sparse Conditional Constant Propagation
-load-store-vectorizer - Vectorize load and store instructions
-localizer - Move/duplicate certain instructions close to their use
-loop-deletion - Delete dead loops
-loop-instsimplify - Simplify instructions in loops
-loop-unroll - Unroll loops
-mem2reg  - Promote Memory to Register
```

```
-memcpyopt - MemCpy Optimization
-memdep - Memory Dependence Analysis
-memoryssa - Memory SSA
-mergefunc  - Merge Functions
-mergeicmps - Merge contiguous icmps into a memcmp
-msan - MemorySanitizer: detects uninitialized reads.
-nvptx-assign-valid-global-names - Assign valid PTX names to globals
-nvptx-lower-aggr-copies - Lower aggregate copies, and llvm.mem* intrinsics into loops
-nvptx-lower-alloca - Lower Alloca
-nvptx-lower-args - Lower arguments (NVPTX)
-reg2mem - Demote all values to stack slots
-regbankselect - Assign register bank of generic virtual registers
-enable-name-compression - Enable name string compression
-enable-no-infs-fp-math - Enable FP math optimizations that assume no +-Infs
-enable-no-nans-fp-math - Enable FP math optimizations that assume no NaNs
-enable-no-signed-zeros-fp-math - Enable FP math optimizations that assume the sign of 0
```