Aceleración de cálculos de estructura electrónica mediante el uso de Procesadores Gráficos Programables

Tesista

Matías Alejandro Nitsche

Director

Dr. Mariano C. González Lebrero (DQIAQF)

Co-director

Dr. Esteban Mocskos (DC)

DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Tesis para optar por el Título de Licenciado en Ciencias de la Computación

Resumen

En este trabajo se presenta una implementación de cálculo de estructura electrónica, basado en la Teoría de los Funcionales de la Densidad (DFT)[1], especialmente diseñada para ser ejecutada sobre un procesador gráfico (GPU). Como base de los desarrollos se utiliza un software existente, Molecole[2]. Siguiendo la literatura sobre el tema se implementan los algoritmos eficientes conocidos. Sin embargo, se introducen adaptaciones novedosas que tienen en cuenta tanto las características del hardware sobre el cual se trabaja, así como los tipos particulares de sistemas moleculares en los que se enfoca.

El atractivo de este tipo de hardware es su gran poder de cómputo, producto de la arquitectura altamente paralelizada que implementa. En los cálculos en los cuales se puede aprovechar este paralelismo es posible obtener mejoras importantes en los tiempos de ejecución.

Además de las mejoras en performance, se tiene en cuenta otro aspecto importante: la precisión numérica con la que se opera. Únicamente los procesadores gráficos más actuales y costosos soportan operaciones de precisión doble, y su rendimiento en comparación con las operaciones de precisión simple es aproximadamente diez veces menor[3]. Por estas razones y dado que la implementación original (Molecole) utiliza operaciones de punto flotante de precisión doble, también se analiza el impacto que tiene el uso de operaciones de precisión simple en el resultado final del cálculo.

Con el software desarrollado se logra una implementación que está a la par del reconocido Gaussian '03[4] en términos de calidad numérica, pero con una reducción en los tiempos de cómputo de hasta aproximadamente diez veces ($10x$) al comparar con una ejecución serial, y de hasta cuatro veces (4x), al comparar con una ejecución paralelizada en cuatro cores de CPU. Se realizan también comparaciones con el software original Molecole, SIESTA[5] y finalmente con una implementación para procesador convencional (CPU), análoga a la desarrollada en GPU.

Por último, se espera que la experiencia lograda a través de la resolución de los diversos problemas encontrados al trabajar sobre este tipo de hardware sea generalizable a futuros proyectos. Es más, los resultados obtenidos no solo motivan a avanzar con el trabajo presentado, sino que también demuestran la utilidad de estos procesadores para acelerar tanto cálculos como los presentados así como posiblemente otros similares.

Abstract

This work presents an implementation of electronic structure calculations based on the Density Functionals Theory (DFT)[1], specifically designed to be executed on a graphics processor (GPU). As the basis of the developments an existing software is used: Molecole[2]. Following the literature on this subject, the known efficient algorithms are implemented. However, novel adaptations that take into account both the characteristics of the used hardware, and the particular types of molecular systems in which this work focuses, are introduced.

The appeal of such hardware is its great computational power, which is a product of the highly parallelized architecture it implements. In calculations where this paralellism can be exploited, it is possible to obtain significant improvements in execution times.

In addition to execution performance improvements, another important aspect is taken into account: the numerical accuracy. Only the latest and costly graphics processors support double-precision operations, and its performance compared to single-precision operations is approximately ten times lower[3]. For these reasons and because the original implementation (Molecole) uses double-precision floating-point operations, the impact over the final result of single-precision operations is analyzed.

With the software developed in this work an implementation which is at par with the known Gaussian '03 [4] in terms of numerical quality, but with a reduction in computing time to about ten times ($10x$) is achieved, by comparing to a serial execution, and to four times ($4x$) by comparing to a parallelized execution over four CPU cores. Other comparisons are also made between the original software Molecole, SIESTA[5] and an implementation that targets conventional processors (CPU), but analogous to the one developed for GPU.

Finally, it is to be expected that the experience gained through solving the various problems encountered when working on this type of hardware could be generalized to future projects. Moreover, the results not only motivate to advance with the present work, but also demonstrate the usefulness of these processors to speed up these type of calculations and possibly other similar.

Índice general

Capítulo 1

Introducción

1.1. Objetivos

El objetivo principal de esta tesis es obtener una implementación eficiente (a partir de un software existente, Molecole[2]) de cálculo de estructura electrónica, basado en la Teoría de los Funcionales de la Densidad (DFT)[1], especialmente diseñada para ser ejecutada sobre un procesador gráfico. Por estar basados en una arquitectura altamente paralelizada, estos procesadores pueden mejorar considerablemente el rendimiento de cierto tipo de cálculos, siendo su principal atractivo la baja relación costo/poder computacional. Además, su crecimiento en términos de capacidad de procesamiento parece estar dejando atrás al de los procesadores convencionales (figura 2.1).

1.2. Trabajos relacionados

El uso de procesadores gráficos en aplicaciones científicas está en gran auge desde los últimos años. Algunos ejemplos cercanos a la química computacional corresponden a trabajos de simulación de *n*-cuerpos (dinámica gravitacional)[6], de dinámica molecular[7], química cuántica[8, 9, 10, 11] y quantum Monte-Carlo[12]. En todos los casos se obtuvieron incrementos de performance considerables, que motivan el uso de este tipo de hardware en cálculos de química computacional basados en el método de DFT.

En un trabajo previo hemos intentado[13] alcanzar objetivos similares a los de la presente tesis. Sin embargo, dicho trabajo se basó en algoritmos de alta complejidad temporal y espacial que, si bien facilitaron la implementación en la arquitectura paralelizada en cuestión, son ineficientes en términos generales. De todos modos se pudo obtener un *speedup* en performance de aproximadamente cinco veces (5x) en comparación con un procesador convencional, lo que motivó la implementación de un algoritmo más eficiente en este mismo hardware.

En Stratmann *et al*[14] se presentan algoritmos de complejidad lineal para las porciones computacionalmente más costosas de un cálculo basado en el método de DFT. En el trabajo más reciente de Yasuda[15], se adaptaron estos mismos algoritmos al procesador gráfico, obteniendo así aceleraciones de entre cinco a diez veces (10x) en comparación con un procesador relativamente moderno (Intel Core2 Duo, 2.8GHz).

Sin embargo, mientras que la implementación de Yasuda se enfoca principalmente en la resolución de sistemas de gran tamaño, en este trabajo resulta de interés la simulación de sistemas de tamaño moderado (en vistas de la inserción de este cálculo en una simulación de dinámica molecular). Por ello resulta conveniente introducir algunas variaciones en los algoritmos para así mejorar el rendimiento de la implementación en estos casos.

1.3. Estructura de la tesis

En las siguientes secciones de este capítulo se exponen los fundamentos teóricos necesarios para comprender el marco en el que se insertan los distintos cálculos que se buscan resolver. Se detalla el método que se implementa y se exhibe la porción del mismo que resulta de interés en este trabajo.

En el capítulo 2, se analiza en detalle la arquitectura sobre la cual se trabaja. Se tienen en cuenta tanto las características físicas del hardware como las del modelo de programación asociado. Finalmente, se presentan de manera concisa los distintos aspectos de performance necesarios a tener presentes al trabajar con este tipo de procesadores.

Más adelante, en el capítulo 3, se exponen los algoritmos que se desarrollaron a partir de los trabajos existentes en la literatura, y de sus diferencias con respecto a estos últimos.

Luego, en el capítulo 4, se explica el funcionamiento del código desarrollado para el procesador gráfico, justificando las decisiones que se tomaron en cada caso y detallando cómo se tuvieron en cuenta los aspectos de performance previamente mencionados.

Todos los resultados obtenidos, que incluyen la búsqueda de parámetros óptimos, análisis de performance de la implementación y de la calidad numérica de los cálculos realizados, se presentan en el capítulo 5.

Por último, en el capítulo número 6, se discuten todos los resultados obtenidos y el trabajo a futuro que deja esta tesis.

1.4. Fundamentos teóricos

1.4.1. La mecánica cuántica

La mecánica que rige el comportamiento de las partículas muy livianas (del orden del electrón) se denomina *mecánica cuántica*. Esta postula que la dinámica de los sistemas físicos se puede describir mediante una *función de onda* Ψ. Bajo esta teoría, dicha función de onda debe satisfacer la llamada *Ecuación de Schrödinger*. La versión general se conoce como *Ecuación de Schrödinger dependiente del tiempo*, y se expresa como sigue:

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

donde $\vec{R} = (\vec{r}_1, \ldots, \vec{r}_n)$ es el vector de todas las posiciones de las partículas de masa m, que se mueven en un espacio afectado por un campo externo V (que puede ser el potencial electrostático generado por el núcleo de una molécula), $\hbar$ es la constante de Planck dividida por 2π e $i = \sqrt{-1}$.

Se dice que es *dependiente del tiempo*, ya que V puede depender de t. Si esto no ocurre, se puede separar (1.1) en sus componentes espaciales y temporales. De este modo, se define como *Ecuación de Schrödinger independiente del tiempo* a la siguiente:

$$-\frac{\hbar^2}{m}\nabla^2\Psi(\vec{R}) + V(\vec{R})\Psi(\vec{R}) = E\Psi(\vec{R}) \tag{1.2}$$

donde E corresponde a la energía asociada a la función de onda Ψ. Es usual abreviar esta ecuación mediante el operador Hamiltoniano:

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

con lo que la ecuación (1.2) ahora queda:

$$\hat{H}\Psi(\vec{R}) = E\Psi(\vec{R})$$

que es la forma más comúnmente utilizada.

La mecánica cuántica, además, postula que cualquier propiedad del sistema se puede obtener a partir de la función de onda (también conocida como la *estructura electrónica*, cuando las partículas son electrones), y por ello su obtención es de gran interés en química. El problema, entonces, se reduce a encontrar una solución a esta ecuación diferencial. Lamentablemente solo se conoce la solución exacta para sistemas con un único electrón. Para los sistemas polielectrónicos se pueden obtener soluciones aproximadas. Sin embargo, el costo computacional de las mismas es, en principio, elevado. Este costo crece notablemente con el tamaño del sistema, con orden cúbico en DFT, orden cuatro en Hartree-Fock y órdenes mayores para métodos más precisos. Por ello, las técnicas más refinadas sólo se pueden aplicar a sistemas pequeños (moléculas en fase gaseosa).

Para sistemas más complejos (solutos en solución, proteínas, entre otros) es necesario simplificar aún más el problema. Una técnica muy utilizada consiste en resolver la ecuación de Schrödinger únicamente para una porción del sistema, modelando el resto mediante una parametrización sin detalle electrónico (es decir, aplicando la mecánica clásica). De esta manera se logra obtener un cálculo con detalle electrónico en la porción reactiva del sistema (por ejemplo, un soluto) incluyendo el resto (el solvente) como modulador de su comportamiento, con un costo computacional moderado. Este tipo de técnicas se denominan *híbridas* o QM-MM (*Quantum Mechanic - Molecular Mechanic*)[16, 17, 18].

Además, las propiedades de estos sistemas complejos dependen de un gran número de configuraciones posibles, llamado *ensamble*, por lo que es necesario emplear algún método de muestreo que permita obtenerlo. Una técnica muy utilizada es la llamada *simulación de dinámica molecular,* que consiste en resolver las ecuaciones del movimiento de Newton para el sistema y dejarlo evolucionar en el tiempo. Para obtener un ensamble representativo, en general es necesario obtener tanto las fuerzas como la energía (es decir, resolver la ecuación de Schrödinger del sub-sistema cuántico) para un grupo de configuraciones muy grande (del orden del millón o más).

Las capacidades de cálculo actuales solo permiten realizar simulaciones de dinámica molecular híbrida en sistemas relativamente pequeños (de hasta veinte átomos, aproximadamente) sobre un intervalo de tiempo reducido (hasta cientos de picosegundos). El presente trabajo busca extender el campo de aplicación tanto en tamaño del sistema como en tiempo de simulación, disminuyendo el costo computacional asociado. Sin embargo, debido a la gran cantidad de configuraciones distintas que se deben tener en cuenta, y que en una dinámica molecular se debe resolver el problema electrónico por cada fracción de tiempo simulada, el poder computacional requerido para resolver sistemas de gran tamaño (de varios cientos de átomos) resulta prohibitivo. Por estas razones, en este trabajo nos restringiremos a la resolución de sistemas de tamaño moderado (de hasta aproximadamente ochenta átomos) en vistas de utilizar esta implementación, en un futuro, en simulaciones de dinámica molecular.

1.4.2. Teoría del funcional de densidad (DFT)

Dentro de las aproximaciones posibles para (1.2) existen dos grandes ramas: la basada en los postulados de Hartree y Fock, y la basada en la Teoría de los Funcionales de la Densidad, desarrollada por Hohenberg y Kohn. Este trabajo sigue esta última.

El modelo DFT tiene sus bases en los trabajos realizados por Thomas y Fermi en 1920, en los que se postula que las propiedades de un sistema de electrones están dadas por (es decir, son función de) la *densidad electrónica* ρ. Esta determina, en un sistema electrónico que se encuentra en un estado dado, la probabilidad de encontrar un electrón en un lugar determinado del espacio.

Por otro lado, Hohenberg y Kohn demostraron dos teoremas que permiten describir el problema electrónico por medio de la densidad electrónica. El primer teorema establece que Ψ (y por ende, E) se encuentra unívocamente determinada por ρ, según:

$$\rho(\vec{r}_i) = \int \Psi^*(\vec{r}_1, \ldots, \vec{r}_n)\Psi(\vec{r}_1, \ldots, \vec{r}_n)d\vec{r}_1 \ldots, d\vec{r}_{i-1}, d\vec{r}_{i+1}, \ldots, d\vec{r}_n, \forall i \in [1, n]$$

donde Ψ^* corresponde al conjungado de Ψ. De este modo la energía se puede considerar un *funcional* de la densidad, notada entonces como $E[\rho]$. Las diferentes contribuciones a la energía total del sistema

se pueden separar de la siguiente forma:

$$E[\rho] = T[\rho] + V_{ee}[\rho] + V_{ne}[\rho]$$

donde T corresponde a la energía cinética, V_{ee} y V_{ne} a la energía potencial que surge de la interacción electrón-electrón y núcleo-electrón, respectivamente.

La segunda contribución a DFT se conoce como el *Teorema de Variación de Hohenberg y Kohn*, que establece que para cualquier densidad ρ', tal que $\int \rho'(r)dr = N$ y $\rho'(r) \geq 0, \forall r \in \mathbb{R}^3$, entonces:

$$E[\rho'] \geq E_0 = E[\rho]$$

Este resultado da un criterio para construir un algoritmo iterativo con el fin de obtener una aproximación ρ' de ρ minimizando $E[\rho']$. En adelante se notará con ρ a la densidad aproximada, para simplificar la notación.

1.4.2.1. Método de Kohn-Sham

Hasta el momento solo se vinculó la densidad con la energía y no se dio un método para calcularlas. Kohn y Sham[19] propusieron un método que se basa en calcular la densidad ρ de un sistema ficticio con ciertas propiedades, según:

$$\rho(\vec{r}) = \sum_{i=1}^{N_{oc}} |\chi_i(\vec{r})|^2 \tag{1.3}$$

donde los χ_i son los llamados *orbitales atómicos* y N_{oc} la cantidad de éstos que se encuentran ocupados por algún electrón (y por ende, es proporcional a la cantidad de átomos del sistema). A su vez, los χ_i son expresados como una combinación lineal de funciones ϕ_k, denominadas contracciones:

$$\chi_i(\vec{r}) = \sum_{\mu=1}^{M} C_{i\mu}\phi_k(\vec{r}) \tag{1.4}$$

$$\phi_k(\vec{r}) = \sum_{\nu}^{N_k} \gamma_\nu f_\nu(r) \tag{1.5}$$

donde C es una matriz de coeficientes. La cantidad de contracciones (M) es proporcional a la cantidad de átomos del sistema. La cantidad N_k de funciones f_ν es fija y relativamente pequeña. Estas funciones f_ν son, en nuestra implementación, Gaussianas centradas en el núcleo de algún átomo $\vec{a}_\nu = (x_0, y_0, z_0)$:

$$f_\nu(x, y, z) = N(x - x_0)^i (y - y_0)^j (z - z_0)^k e^{-\alpha_\nu |(x,y,z)-(x_0,y_0,z_0)|^2} \tag{1.6}$$

donde N es un factor de normalización. Según los exponentes i, j, k se diferencian tres casos posibles: si estos son todos iguales a cero, se trata de una función de tipo s, si algún exponente es 1 se trata de una de tipo p, y si la suma de los tres es 2, tipo d. Cada contracción estará formada únicamente por un solo tipo de función f_ν (con las distintas combinaciones posibles para los exponentes i, j, k, de acuerdo al tipo). En adelante, siempre que se hable de *funciones base,* o incluso *funciones* a secas, se estará queriendo referir a las M contracciones ϕ_k (teniendo en cuenta su expansión en combinación lineal de Gaussianas).

Ahora, a partir de (1.3) y (1.4), se puede reescribir ρ como:

$$\rho(\vec{r}) = \sum_{i=1}^{N_{oc}} \left| \sum_{k=1}^{M} C_{ik}\phi_k(\vec{r}) \right|^2 \tag{1.7}$$

Con esta aproximación, y planteando la diferencia entre el sistema ficticio y el real, se obtiene un nuevo funcional para la energía:

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

donde las contribuciones a $E[\rho]$ son todas conocidas excepto por $E_{xc}[\rho]$, la *energía de intercambio y correlación*. La calidad de la aproximación de la energía $E[p]$ depende principalmente de este término, cuya magnitud es relativamente pequeña. Existen distintos métodos para estimarlo. El más simple de todos es el conocido como *Aproximación por Densidad Local* o LDA (que es el utilizado en este trabajo), y supone estimar $E_{xc}[\rho]$ de la siguiente forma:

$$E_{xc}[\rho] = \int \rho(\vec{r})\varepsilon_{xc}(\rho(\vec{r}))d\vec{r} \tag{1.8}$$

donde ε_{xc}, denominado *funcional de intercambio y correlación*, depende exclusivamente de la densidad local y se aproxima al valor de la energía de intercambio y correlación de una gas de electrones de la misma densidad, cuya solución exacta es conocida. Este funcional se puede escribir como la suma del funcional de intercambio ε_x, dado por:

$$\varepsilon_x(\rho(\vec{r})) \quad = \quad -\frac{3}{4}\left(\frac{3\rho(\vec{r})}{\pi}\right)^{1/3}$$

y el de correlación ε_c, el cual tiene varias definiciones posibles. En este trabajo se calculará ε_c según el método expuesto por Vosko[20].

Existen otros métodos de estimación para $E_{xc}[\rho]$, tal como GGA (que considera, además de la densidad, su gradiente), pero cuyo costo computacional es mayor. En ciertos casos, la calidad de los resultados arrojados por el método LDA es más que suficiente.

1.4.2.2. Aproximación numérica de la energía de intercambio y correlación

El método de Kohn-Sham permite obtener una aproximación de ρ, a partir de la cual es ahora posible determinar la energía de intercambio y correlación $E_{xc}[\rho]$. Sin embargo, dado que la integral de la ecuación (1.8) no tiene solución analítica, es necesario integrar numéricamente mediante un método de cuadratura numérica. Este tipo de métodos permiten aproximar una integral de la forma:

$$I = \int F(\vec{r})d\vec{r} \tag{1.9}$$

mediante la siguiente suma discreta:

$$I \simeq \sum_i \omega_i F(\vec{r}_i) \tag{1.10}$$

donde los $\vec{r}_i$ se denominan puntos de integración (o grilla), y los $\omega_i \in \mathbb{R}$, pesos de integración.

Según cómo se haga la asignación de pesos, de la densidad de la grilla y de los puntos que la conforman, se tendrá una mejor o peor aproximación de (1.9). Por ello, teniendo en cuenta la geometría del problema, se realizaron varios trabajos para encontrar distribuciones óptimas de estos pesos y puntos, para el caso de la integral en cuestión.

El esquema más utilizado para obtener las puntos $\vec{r}_i$ consiste en la generación de estos a partir de alguna grilla de tamaño fijo, definida sobre la superficie de una esfera de radio 1. Este tipo de grillas base fueron estudiadas por Lebedev en 1975[21] y 1976[22]. Mediante la superposición de varias grillas de este tipo, en forma de capas concéntricas, se obtiene una serie de puntos apropiados para la aproximación de la energía asociada a un átomo (figura 1.1). En general, para sistemas poliatómicos la totalidad de la grilla se obtiene como resultado de la unión de las obtenidas para cada átomo. Lo que se obtiene es un conjunto de puntos que se pueden notar como $\vec{r}_{k_g}$, refiriéndose de esta forma a la posición del g-ésimo punto de la grilla, asociado al k-ésimo átomo del sistema.

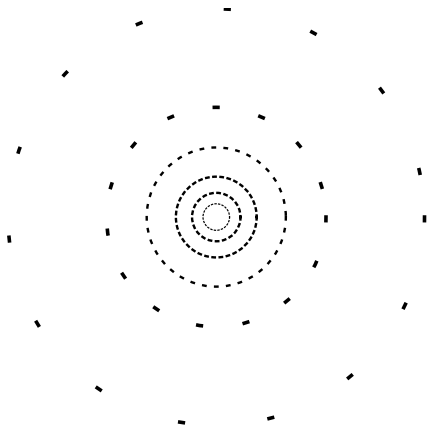


Figura 1.1: Esquema en dos dimensiones de la distribución de puntos en capas concéntricas, sobre una esfera de radio 1, para una grilla asociada a un átomo dado. La mayor concentración de puntos se encontrará rodeando el núcleo.

Concretamente, los puntos correspondientes a un átomo de posición $\vec{a}$, se definen como:

$$\vec{r} \in \mathbb{R}^3 / \vec{r} = \vec{s_j}.\lambda_i + \vec{a}$$

donde $\vec{s_j}$ corresponde a la posición del j-ésimo punto de la grilla base y λ_i al factor de escalamiento asociado a la i-ésima capa. Este último factor se obtiene como:

$$\begin{aligned} \lambda_i &= r_m \left(\frac{1+x_i}{1-x_i} \right), i \in [1, \kappa] \\ x_i &= cos(\frac{\pi i}{\kappa + 1}) \end{aligned}$$

donde κ corresponde a la cantidad de capas y r_m a la mitad del *Radio de Slater* (ambos dependen del elemento químico propio del átomo en cuestión). Con estos factores se obtiene una grilla que es mucha más densa cerca del núcleo, donde la densidad ρ cambia más abruptamente (figura 1.2). De esta forma, esta transformación de los puntos $\vec{s_j}$ de la grilla resulta en un conjunto de puntos $\vec{r}$ apropiados para las características particulares del tipo de átomo en cuestión.

Por otro lado, Becke[23] estudió la asignación de pesos apropiada para el caso de los puntos distribuidos sobre la superficie de una esfera. Becke descompone los pesos originalmente definidos como ω_i en (1.10), en una serie de pesos ω'_g (fijos), y otra serie p_k, definida según:

$$\begin{aligned} p_k(\vec{r}_{k,g}) &= w_k(\vec{r}_{k,g}) / \sum_j^K w_j(\vec{r}_{j,g}) \\ w_k(\vec{r}_{k,g}) &= \prod_{i \neq k}^K s(\mu_{ki}) \\ \mu_{ki} &= \frac{|\vec{a}_k - \vec{r}_{k,g}| - |\vec{a}_i - \vec{r}_{i,g}|}{|\vec{a}_k - \vec{a}_i|} \end{aligned} \tag{1.11}$$

donde $w_k(\vec{r}_{k,g})$ corresponde al peso parcial del punto g perteneciente al átomo k, cuya posición es $\vec{a}_k$ y K, a la cantidad de átomos del sistema. La función s, llamada *cell function*, es un polinomio que se define convenientemente dentro del rango $\mathbb{R}_{[0,1]}$[14]. Dado que los pesos ω'_g son fijos para una grilla base

dada, estos pueden ser precomputados dado que no dependen de los átomos presentes en el sistema (al contrario de p_k).

Así, aplicando (1.10) a la ecuación (1.8), se llega finalmente a la siguiente expresión:

$$E_{xc}[\rho] \simeq \sum_{k}^{K}\sum_{g}^{G}\omega'_g p_k(\vec{r}_{k,g})F(\vec{r}_{k,g}), K, G \in \mathbb{N} \tag{1.12}$$

que permite obtener una aproximación de la energía de intercambio y correlación, como funcional de la densidad ρ.

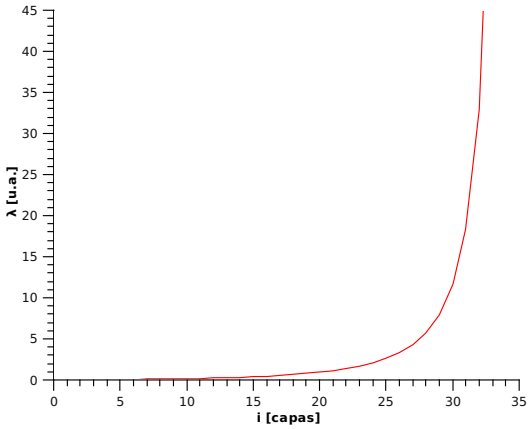


Figura 1.2: Factores de escalamiento λ_i, para el caso de un átomo de oxígeno, con $r_m = 0.57$ y $\kappa = 35$. Se puede ver que hasta la capa veinte (es decir, casi el 60 % de las primeras capas), los puntos no se encuentran a mas de 1 u.a. del núcleo, mientras que el resto se aleja considerablemente.

1.4.2.3. Matriz de Kohn-Sham

El paso restante para la resolución del problema consiste en determinar una matriz de coeficientes C para (1.7) de forma de poder calcular (1.12).

El método de Hartree-Fock conlleva resolver un conjunto de ecuaciones llamadas *Ecuaciones de Roothaan*. Para el caso de DFT, se puede llegar a un sistema análogo de ecuaciones, denominadas *Ecuaciones de Kohn-Sham*. A su vez, la matriz asociada a este sistema se denomina *Matriz de Kohn-Sham*. Así como la energía de intercambio y correlación contribuye al valor de la energía total del sistema, también existe una matriz de Kohn-Sham $\Gamma \in \mathbb{R}^{M\times M}$ asociada a $E_{xc}[\rho]$, donde M corresponde a la cantidad de funciones base del sistema. Por simplicidad, cuando se hable de la Matriz de Kohn-Sham se estará refiriendo a la matriz Γ asociada a $E_{xc}[\rho]$. Esta matriz se define entonces como:

$$\begin{aligned} \Gamma_{ij/i<j} &= \sum_{k,g}\Gamma'_{ij}(k,g) \\ \Gamma'_{ij}(k,g) &= \phi_i(\vec{r}_{k,g})\phi_j(\vec{r}_{k,g})p_k(\vec{r}_{k,g})\omega'_g y \\ y &= f(\varepsilon_{xc}(\rho(\vec{r}_{k,g}))) \end{aligned} \tag{1.13}$$

Dado que Γ resulta simétrica, solo es necesario obtener una de las matrices triangulares contenidas en ella.

Se puede ver a partir de (1.13) que el sistema de ecuaciones depende indirectamente de C, a través de ρ. De este modo, lo que queda determinado es una método iterativo de aproximación de ρ, que consiste en la resolución de un sistema de ecuaciones en cada paso. Partiendo de una matriz inicial aproximada

de coeficientes, cada iteración determinará una nueva matriz C. El objetivo final es la minimización de la energía $E[\rho]$, asociada a la densidad ρ obtenida en cada paso. Así, basándose en el *Teorema de Variación de Hohenberg y Kohn*, este método converge a la mejor aproximación posible para la densidad ρ y, por ende, para la energía $E[p]$.

Capítulo 2

El procesador gráfico

Originalmente, las operaciones matemáticas necesarias para graficar objetos tridimensionales eran resueltas por el procesador principal (CPU). A medida que las aplicaciones gráficas (principalmente juegos) fueron evolucionando, fue necesario delegar estas tareas a un procesador especializado para poder alcanzar rendimientos aceptables. De este modo surgieron los procesadores gráficos o GPU, presentes en cualquier placa de video actual. Estos procesadores implementan una arquitectura específicamente orientada al tipo de operaciones que deben ejecutar. Dado que la mayoría consiste en la aplicación de una simple transformación sobre una gran cantidad de píxeles o vértices tridimensionales sin interdependencias de los resultados, la arquitectura tiene como objetivo permitir una ejecución altamente paralelizada.

Más adelante, con la aparición de la programabilidad de estos procesadores, se dio un gran paso evolutivo. Se hizo posible definir porciones de código que fueran ejecutadas por el GPU, para transformar de manera programática vértices y píxeles. Estos pequeños programas fueron denominados *shaders*, dado que se los utilizaba principalmente para generar efectos complejos de iluminación. Originalmente, el lenguaje de programación utilizado era primordialmente Cg[24], desarrollado por NVIDIA. Este lenguaje ad-hoc incluía tipos de datos especializados (como *color*, *vértice*, etc.) y conceptos propios del renderizado (como las *texturas*). Con el tiempo fueron apareciendo nuevos lenguajes similares (GLSL, HLSL)[25], a medida que se desarrollaban nuevos y más poderosos procesadores gráficos (figura 2.1).

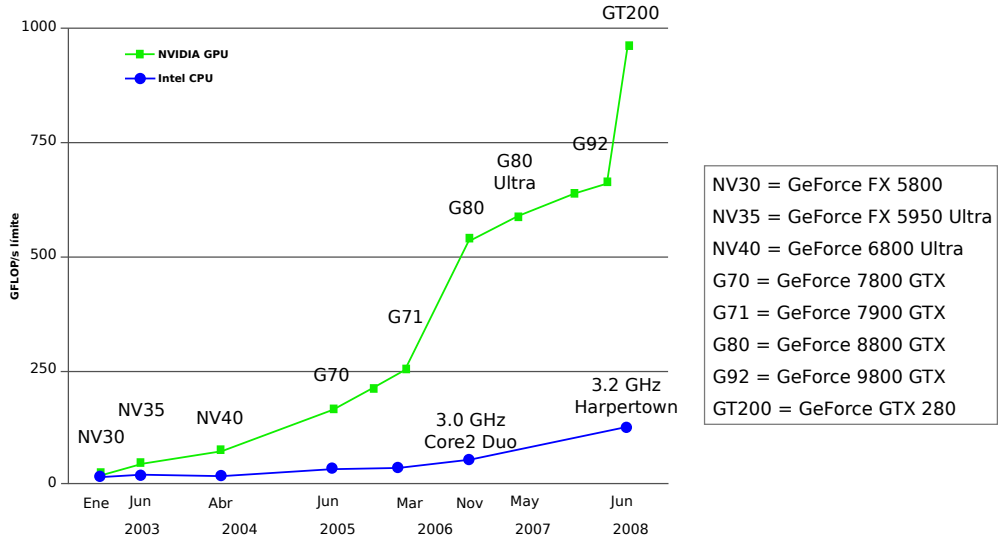


Figura 2.1: Poder de procesamiento (en GFLOP/s) en función del tiempo, para la línea de procesadores de NVIDIA y de Intel. Gráfico tomado de [26].

Sin embargo, con el tiempo se hizo evidente que esta programabilidad podía ser explotada de modo de utilizar el GPU como un co-procesador matemático. Esta práctica se dio a conocer como GPGPU o *General Processing on the GPU*. Dado que los elementos del lenguaje estaban fuertemente orientados a aplicaciones gráficas, era necesario darles otras interpretaciones más abstractas (un *color* podía ser interpretado como tres variables de punto flotante, una textura como una matriz de datos, etc.). Esto era propenso a confusiones y a la vez limitaba considerablemente el desarrollo de algoritmos complejos.

A la par del creciente uso de GPUs en este tipo de aplicaciones, los principales fabricantes (ATI, recientemente adquirido por AMD, y NVIDIA) comenzaron a desarrollar un entorno de programación general (es decir, no orientado a gráficos) que pudiera explotar las características del hardware. NVIDIA, por su parte, desarrolló CUDA[26] (Compute Unified Device Architecture), que comprende un driver especializado y un compilador para un lenguaje ad-hoc derivado de C. Por otro lado, ATI se basó en un framework pre-existente de programación paralela llamado Brook[27], adaptándolo de modo de poder utilizarlo con su línea de procesadores gráficos programables. Recientemente surgió en ambos fabricantes el interés en desarrollar soporte para un lenguaje orientado al cómputo intensivo, llamado OpenCL[28].

De aquí en adelante los detalles de arquitectura y del modelo de programación se refieren al caso del fabricante NVIDIA, dado que ese fue el hardware disponible para el desarrollo de este trabajo.

2.1. Modelo de programación

Dado que el modelo de programación de estos procesadores sirve para entender y justificar la arquitectura que implementan, este aspecto será descrito primero.

2.1.1. Componentes del *framework*

CUDA comprende principalmente tres componentes:

1. Un *driver* especializado, capaz de exponer los aspectos necesarios del hardware
2. Una biblioteca o *runtime* que se comunica con este driver para presentar una interfaz de alto nivel al usuario
3. Un compilador especializado (nvcc) que genera tanto el código máquina para el GPU, como el código para CPU necesario para hacer de nexo entre ambos procesadores.

El código a ejecutar en GPU se define siempre dentro de un método o *kernel* que puede ser invocado desde código C convencional. Estos métodos se escriben en un lenguaje C con extensiones sintácticas propias de CUDA. El compilador nvcc, al encontrar una invocación a un método de GPU (con la sintaxis apropiada), traduce este al código máquina correspondiente.

2.1.2. Descripción del modelo

El modelo de programación asociado a este tipo de procesadores es de tipo SIMD: Single Instruction, Multiple Data. Es decir, el procesamiento consiste en aplicar una misma operación a múltiples datos. De hecho, NVIDIA define este modelo como SIMT: Single Instruction, Multiple Threads, dado que en realidad una misma instrucción será ejecutada en paralelo por varios threads distintos (generalmente sobre datos también distintos).

Desde el punto de vista del usuario, los *kernels* para GPU son ejecutados por múltiples threads, donde el paralelismo a nivel instrucción es resuelto por el procesador mismo. Para aprovechar las características del hardware (que se detallarán más adelante), es necesario definir los threads en forma de grilla (uni o bi-dimensional), y agrupados en conjuntos de igual tamaño, llamados *bloques*. De este modo, cada thread tendrá asociado un identificador $(blockIdx, threadIdx)$, donde $blockIdx$ corresponde al

índice del bloque al cual pertenece el thread, y *threadIdx* al número de thread dentro del bloque mismo (figura 2.2).

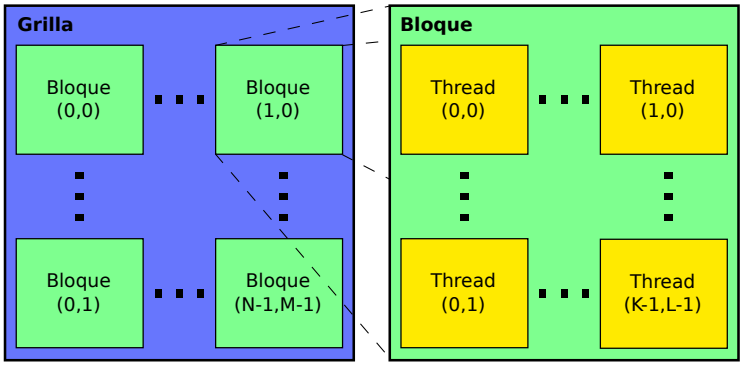


Figura 2.2: Agrupamiento de *threads* en una grilla de *bloques*

En el listado 2.1 se puede ver un ejemplo de un *kernel* de GPU que calcula de forma paralelizada el producto de dos series de números enteros $\{b_i\}, \{c_i\}$, donde cada producto $a_i = b_i \times c_i$ es resuelto por el i-ésimo thread (en este caso se presume que hay tantos datos como threads en ejecución).

Algoritmo 2.1 Ejemplo de un *kernel* de GPU

```
void metodo(int *a, int* b, int* c) {
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  a[i] = b[i] * c[i];
}
```

Las variables `threadIdx`, `blockIdx` y `blockDim` contienen el índice al thread actual, el bloque al que pertenece el mismo y el tamaño de cada bloque, respectivamente. En el ejemplo se utilizó una grilla unidimensional, y por ello solo se accede a la componente x de cada variable.

Una invocación al *kernel* del ejemplo se muestra en el listado 2.2, donde las variables g y b contienen la definición del tamaño de la grilla (medida en cantidad de bloques) y de los bloque de la misma (medida en cantidad de threads). La sintaxis utilizada para indicar g y b corresponde a la *configuración de la llamada*, y es una de las extensiones al lenguaje introducidas con CUDA.

Algoritmo 2.2 Ejemplo de invocación de un *kernel* de GPU

```
int* a;
int* b;
int* c;
...
metodo<<<g,b>>>(a, b, c);
...
```

Los parámetros de entrada a un *kernel* pueden ser valores primitivos (como un entero o un real) o punteros a memoria de video (de la cual se puede leer o escribir), que debe ser inicializada previamente. La salida generada por el *kernel* se produce indirectamente a través de la escritura en memoria de video, con punteros recibidos por parámetro.

En el entorno de CUDA también se definen tipos de datos de estilo vectorial (float4, float2, uint3, etc.), implementados internamente mediante una estructura de datos típica de C. Los operadores asociados a estos tipos (como la suma y el producto, entre otros) están definidos mediante funciones también de C, que son compiladas a lenguaje de ensamblador de GPU al ser invocadas por un *kernel*. En otras palabras, estos no son tipos primitivos del procesador. Cada operación sobre un tipo de datos vectorial es descompuesta en varias operaciones sobre cada una de sus componentes.

2.2. Arquitectura

El GPU comprende una serie de multiprocesadores (o *stream processors*), con capacidad de direccionar a un espacio de memoria global (la memoria de video o *device memory*). Cada multiprocesador tiene su propia memoria compartida (*shared memory*), que puede ser utilizada para intercomunicar los procesadores que lo conforman.

La velocidad de la memoria compartida es comparable a la de un registro de procesador, mientras que los accesos a memoria global involucran cientos de ciclos. Dado que el mismo programa se aplica numerosas veces sobre distintos datos, no hay necesidad de un control de flujo optimizado (predicción de saltos, etc). No existe tampoco una caché asociada a la memoria global, por lo que se favorece el código con gran intensidad aritmética (en contraste con la cantidad de operaciones de memoria).

En realidad, la memoria de video se encuentra segmentada. Además de la memoria global, existen otros segmentos tales como la memoria *constante* y la de *textura* (espacios solo de lectura, con cachés asociadas). Sin embargo, estos segmentos tienen restricciones de tamaño y direccionamiento con lo que no es posible prescindir de la memoria global para la mayoría de las transferencias.

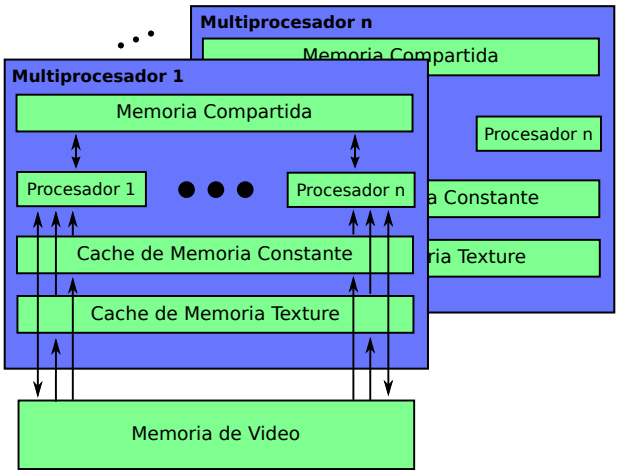


Figura 2.3: Esquema de la arquitectura de un GPU

Cada thread será ejecutado por un procesador, con su propio *instruction pointer* y estado de registros. La asignación entre threads y procesadores está dada por la configuración de la llamada al método. Los bloques ejecutan siempre sobre un mismo multiprocesador, de modo de poder permitir la intercomunicación de los threads contenidos en él, a través del uso de la memoria compartida. Cada bloque es dividido por el *scheduler* en subconjuntos de dieciséis threads llamados *warps* (que, a su vez, se divide en dos *half-warps*). Cuando un *warp* comienza a ejecutar, el puntero a instrucción coincide para todos los threads del mismo. Mientras los threads no diverjan en su comportamiento (por ejemplo, condicionando la ejecución de una operación según el número de thread), el puntero avanza para todos los threads por igual. Los threads divergentes deben ser serializados, reduciendo así su performance. Al momento de volver a converger en su ejecución, el puntero vuelve a avanzar para todos los threads simultáneamente.

Para arbitrar el acceso a la memoria compartida, se implementa un sistema de sincronización por *barriers*. Cuando un procesador ejecuta una determinada instrucción de sincronización (`__syncthreads()`) para un thread dado, este último no continuará ejecutando hasta que todos los otros threads del bloque hayan pasado por esta misma instrucción. La sincronización solo es posible a nivel bloque.

Para lanzar una ejecución de código GPU, los datos sobre los cuales se operarán deben estar disponibles en la memoria de video (el procesador no puede operar sobre la memoria principal, o memoria del *host*). La transferencia entre memoria del *host* y la memoria de video puede hacerse mediante DMA (el usuario solo debe ejecutar un método similar a `memcpy`).

2.3. Hardware utilizado

Principalmente se utilizó una placa de video con un procesador GeForce 8800 GTX, que fue adquirida en el momento que pertenecía al tope de la gama de procesadores compatibles con CUDA. Recientemente se pudo tener acceso a un hardware más avanzado: Tesla C1060. Los detalles técnicos de ambas se exhiben en el apéndice A.1.

El hardware con el que se comenzó el trabajo (GeForce 8800 GTX) no cuenta con soporte para operaciones de precisión doble. Por ello, una de las preguntas iniciales en este trabajo es si la precisión simple alcanza para obtener resultados dentro de un margen de error aceptable en un cálculo de DFT.

Si bien el hardware más actual, como la placa de video Tesla C1060, cuenta con soporte para operaciones precisión doble, la performance puede ser hasta diez veces menor que utilizando precisión simple. Por estas razones, dentro de los análisis que se hacen sobre la implementación, se tiene en cuenta la calidad numérica de los resultados.

2.4. Aspectos de performance

Existen varios aspectos que deben ser tenidos en cuenta para lograr un código eficiente. No es trivial tener en cuenta todos los aspectos simultáneamente debido a que estos se afectan mutuamente. En general, solo es posible intentar por prueba y error distintas variaciones en la implementación que prioricen un aspecto u otro por vez. Además, puede ocurrir que no sea posible extraer tanto paralelismo como podría ser aprovechado por la arquitectura, por un límite intrínseco del problema a resolver.

2.4.1. Accesos a memoria

Dado que los accesos a memoria son considerablemente más costosos que las operaciones aritméticas, es necesario minimizar su utilización y, a la vez, hacerlo de forma eficiente. Para lograr esta eficiencia se pueden agrupar un conjunto de accesos a memoria en una sola transacción, lo que ocurre automáticamente siempre que se cumpla que:

1. cada thread accede a una palabra de 32 bits (resulta en una transacción de 64 bytes), de 64 bits (una transacción de 128 bytes) o de 128 bits (dos transacciones de 128 bytes),

2. el *k*-ésimo thread del *warp* accede al *k*-ésimo word dentro de la transacción (sin embargo, no es necesario que todos los threads hagan un acceso),

3. y los accesos comienzan en una dirección múltiplo del tamaño de la transacción.

Cuando hay una correspondencia uno-a-uno de los datos a cargar con los threads que están ejecutando (ver ejemplo 2.3), la segunda condición se cumple trivialmente al acceder a los mismos indexando con el número de thread.

Algoritmo 2.3 Ejemplo trivial de acceso *eficiente* a memoria global

```
void metodo(float* entrada, uint n, float* salida) {
  ...
  float a = entrada[threadIdx.x];
  ...
}
```

Sin embargo, cuando esta correspondencia no existe, la condición puede ser cumplida de otro modo. Un ejemplo de este tipo de casos es el del algoritmo 2.4, donde todos los threads acceden a los mismos n datos. En una iteración cualquiera del ciclo, todos los threads estarán requiriendo el mismo dato en paralelo. Los n accesos serán serializados, dado que no existe un mecanismo automático de broadcast del dato en estos casos.

Algoritmo 2.4 Ejemplo de acceso *ineficiente* a memoria global

```
void metodo(float* entrada, uint n, float* salida) {
  float resultado = 0;
  for (uint i = 0; i < n; i++) {
    resultado += entrada[i] * threadIdx.x;
  }
  salida[threadIdx.x] = resultado;
}
```

Para evitar este problema, es posible seguir un patrón de acceso como el presentado en el algoritmo 2.5. Las principales diferencias con el algoritmo original son: el particionado de las n iteraciones en grupos de `BLOCK_SIZE` (la cantidad de threads de un bloque, para una configuración de llamada con una grilla unidimensional), y el orden de acceso a `entrada`, de a tandas de este mismo tamaño (ver línea 5). Para cada tanda, los `BLOCK_SIZE` threads de un bloque requerirán `BLOCK_SIZE` elementos simultáneamente, formándose así una transacción de memoria que agrupa varios accesos adyacentes. Los datos son, entonces, cargados primero a memoria compartida, para luego ser utilizados durante el cálculo de la línea 8, dado que esta memoria sí puede ser accedida en paralelo sin generar serialización de instrucciones (al menos, bajo ciertas condiciones que se detallan en la sección 2.4.4).

Algoritmo 2.5 Ejemplo de acceso *eficiente* a memoria global

```
1  void metodo(float* entrada, uint n, float* salida) {
2    float resultado = 0;
3    __shared__ entrada_sh[BLOCK_SIZE];
4    for (uint i = 0; i < n; i += BLOCK_SIZE) {
5      entrada_sh[threadIdx.x] = entrada[i + threadIdx.x];
6      __syncthreads();
7      for (uint ii = 0; ii < BLOCK_SIZE && (i + ii < n); ii++)
8        resultado += entrada_sh[i] * threadIdx.x;
9      __syncthreads();
10   }
11   salida[threadIdx.x] = resultado;
12 }
```

El acceso a memoria global en este caso cumple con los tres requerimientos antes mencionados, siempre que `BLOCK_SIZE` sea múltiplo de 16 (la cantidad de palabras de 32, 64 o 128 bits que contendrá el acceso agrupado). Para conjuntos de datos $n-$dimensionales (como matrices), es necesario

que las primeras $n-1$ dimensiones sean múltiplo de 16. De este modo se puede seguir cumpliendo el requerimiento de alineamiento[1].

Otra diferencia que se puede apreciar entre el algoritmo 2.4 y el 2.5 es el uso de la instrucción de sincronización `__syncthreads()`. Dado que en el ciclo más interno cada thread accederá a la totalidad de los datos del arreglo `entrada_sh`, y que es posible que durante la transferencia paralelizada alguno de los threads complete la misma antes que el resto (por cuestiones de scheduling), es necesario arbitrar el acceso al medio compartido. Por un lado se debe asegurar que ningún thread llegue a leer un elemento de `entrada_sh` antes de que el encargado de escribirlo haya terminado, por lo que se incluye una primera llamada a `__syncthreads()` entre la escritura y la lectura de dicho arreglo (línea 6). Por otro lado, tampoco debe darse que un thread que terminó de ejecutar el ciclo interno comience a cargar un nuevo dato que todavía podría estar siendo leído por otros threads, por lo que se hace una segunda sincronización entre la lectura y la escritura de la variable[2] (línea 9).

Más allá del patrón de acceso a memoria, es posible esconder la latencia asociada a la memoria global si existen suficientes operaciones aritméticas independientes que pueden ser ejecutadas mientras se espera que se complete el acceso mismo.

2.4.2. Transferencia de datos

Para grandes transferencias de datos entre memoria *host* y *device*, es posible tener un ancho de banda mucho mayor si se hace uso de memoria bloqueada a paginación (*pinned memory*). Con el *runtime* de CUDA, es posible marcar un espacio de memoria *host* como no-paginable, permitiendo al driver realizar transferencias por DMA. Sin embargo, de acuerdo a la documentación de referencia[26], este beneficio solo se obtiene si se reutiliza un mismo espacio de memoria para varias transferencias de gran cantidad de datos. En otros casos, puede incluso resultar contraproducente[3].

En los casos en los que la memoria *host* y *device* sean físicamente las mismas (en placas de video integradas al motherboard), la transferencia es innecesaria. De hecho, en estos caso debe usarse específicamente *pinned memory*.

2.4.3. Control de flujo

Además de las operaciones aritméticas y de memoria, hay que tener en cuenta a las instrucciones de control de flujo (es decir, los ifs, while, etc.). En esta arquitectura, un salto o *branch* puede ser particularmente costoso en el caso en donde distintos threads de un mismo *warp* tomen caminos distintos (*divergent branches*, según la documentación). En estos casos el compilador debe serializar las instrucciones, lo cual puede tener un impacto importante en la performance.

Para el caso de los ciclos for, el compilador es capaz de evaluar la posibilidad (o incluso, se lo puede forzar mediante una directiva especial) de hacer *loop unrolling* o *desenrollamiento de ciclos*. Al eliminar o reducir el impacto de la condición de fin de ciclo, es posible mejorar el rendimiento considerablemente.

En cuanto a los *if*s, puede ser beneficioso reemplazar la ejecución condicional de ciertas operaciones aritméticas con una ejecución incondicional, de modo que la contribución de los casos indeseados al resultado final sea neutra.

2.4.4. Conflicto de bancos de memoria compartida

Así como existe un patrón eficiente de acceso a memoria global, lo mismo sucede con la memoria compartida. Este espacio de memoria se encuentra repartido en un cierto número de bancos físicos (die-

[1] Para los procesadores más actuales (como es el caso de la Tesla C1060), la segunda condición para el acceso eficiente en realidad no es necesaria. Es decir, se permiten los accesos desordenados dentro de un warp. Sin embargo, siguiendo el patrón propuesto se puede asegurar que el tamaño y cantidad de transacciones serán los óptimos.

[2] Notar que sería análogo ubicar la segunda llamada a `__syncthreads()` previo a la carga de datos en la línea 5.

[3] Dado que no se conocen las operaciones involucradas en el acceso a una zona de memoria no-paginable (ni las requeridas para marcar a esta como tal), no se pueden precisar las razones por las cuales solo en estas condiciones se pueden lograr mejoras en el rendimiendo.

ciséis, en la arquitectura utilizada), de modo de permitir un acceso paralelizado. El direccionamiento a esta memoria está definido de tal forma que un acceso a dos palabras adyacentes de 32 bits, caerá en dos bancos también adyacentes. Esto implica que si hay k bancos, cada k accesos secuenciales se vuelve a acceder al mismo banco, generando un *bank conflict*. Estos conflictos solo pueden ser resueltos mediante la serialización de los accesos.

Dado que en la especificación de hardware actual, la cantidad de threads de un *half-warp* coincide con la cantidad de bancos existentes, si el i-ésimo thread accede a la i-ésima dirección asociada a una palabra de 32 bits, no se producirán conflictos. Además, dado que los accesos a memoria compartida hechos por los threads de un *warp*, se hacen en dos pasos (uno para el primer *half-warp* y otro para el segundo), solo puede haber conflictos entre threads de un mismo *half-warp*.

Un patrón de acceso muy común, corresponde a un indexamiento sobre una variable, mediante dos índices i, j y un ancho fijo n, de la forma $i \times n + j$, donde i se corresponde con el índice absoluto del thread dentro del bloque. Esto ocurre generalmente al acceder a un arreglo bidimensional, donde cada thread accede a n elementos consecutivos del mismo. En estos casos, los threads i e $i + k$ accederán al mismo banco siempre que $s \times n$ sea múltiplo de la cantidad de bancos b. O lo que es equivalente, siempre y cuando k sea múltiplo de $\frac{b}{d}$, donde $d = \text{mcd}\{b, n\}$. Consecuentemente, no habrá conflicto únicamente cuando el tamaño de un *half-warp* sea menor que $\frac{b}{d}$. Para los dispositivos actuales, esto sucede únicamente si $d = 1$. Por último, dado que en la especificación actual b es una potencia de dos, la condición se reduce a pedir que n sea impar.

2.4.5. Tamaño óptimo de bloques

Existen varios factores que determinan cuántos threads por bloque conviene utilizar. Cómo mínimo, es necesario tener tantos bloques como multiprocesadores. En caso contrario, habría multiprocesadores ociosos. Sin embargo, es deseable que haya al menos dos bloques *activos* por multiprocesador. Esto permite que las latencias asociadas a la sincronización por *barriers* y a los accesos a memoria de video puedan ser escondidas mediante la ejecución de otro bloque. Para lograr esto es necesario no solo que haya el doble de bloques que de multiprocesadores, sino que la cantidad de memoria compartida y de registros necesarios por bloque sea lo suficientemente baja.

Además, el tamaño de los bloques debería ser elegido de modo que sea múltiplo del tamaño de un *warp*, para evitar *warps* incompletos. Para bloques de al menos 64 threads, el compilador es capaz de manejar eficientemente los bancos de memoria de registros de modo de esconder las latencias asociadas a dependencias lectura-escritura. En general, más threads por bloque permite un manejo más eficiente de los tiempos asignados a cada *warp*. El problema es que, a mayor tamaño de bloque habrá mas recursos utilizados por thread. El balance de estos factores en general puede ser obtenido por experimentación con distintos tamaños.

Sabiendo la cantidad de registros y de memoria compartida por thread que un determinado *kernel* requiere, es posible determinar el tamaño de bloque que maximice la cantidad de *warps* activos por multiprocesador en relación a la cantidad máxima de *warps* activos en general (que estará dada por la cantidad de multiprocesadores, registros y memoria disponible, entre otros factores). Esta medida se denomina *occupancy*. Sin embargo, no necesariamente una mayor *occupancy* implica mejor rendimiento, dado que siempre se estará sujeto a las particularidades del código en cuestión.

2.4.6. Memoria local y registros

La memoria local corresponde al espacio asignado para las variables estáticas declaradas directamente en el código de un *kernel*. Dicho espacio es en realidad un segmento de la memoria global, con lo que la latencia asociada a su acceso es la misma. Sin embargo, dado que una variable declarada dentro de un kernel (y que no será asignada a un registro, sino a memoria local) resulta en una dirección de memoria distinta para cada thread, los requerimientos de acceso eficiente a memoria se cumplen automáticamente.

Si bien en general, el acceso a registros no implica un costo extra en ciclos de procesador, en casos de dependencias lectura-escritura que el compilador no pueda resolver, o que no puedan ser escondidas mediante la ejecución de otros threads, esto puede no ser así. Sin embargo, si existen al menos 192 threads activos por multiprocesador, estas latencias pueden ser ignoradas.

La cantidad total de registros disponibles por multiprocesador es un recurso que se divide por cada thread, y se asigna durante la compilación del código correspondiente. Como fue mencionado, reduciendo el tamaño de los bloques es posible disminuir la cantidad de registros necesarios para la ejecución de cierto kernel, en caso de que se esté excediendo el límite físico disponible por el hardware utilizado. Sin embargo, la eficiencia en el uso de registros estará sujeta a las decisiones realizadas por el compilador. Por esta razón, puede resultar beneficioso reescribir el kernel de alguna forma equivalente, con el objetivo de disminuir la cantidad de registros requeridos por thread y posiblemente incrementar el índice de *occupancy*. En general, mediante una directiva al compilador, es posible limitar explícitamente la cantidad de registros que un *kernel* puede usar por thread. Sin embargo, el efecto de este límite será el de alojar variables locales a la memoria local por lo que, en la mayoría de los casos, esto tendrá un impacto negativo en el rendimiento.

2.4.7. Memoria Constante

Para los conjuntos de datos que serán leídos y no modificados en un *kernel*, puede ser útil el segmento de memoria constante. Dado que este espacio de memoria tiene una caché asociado, en caso de producirse un *miss*, el costo es el mismo que en el caso de un acceso a memoria global. Si se produce un *hit* de caché, el tiempo de acceso es comparable al de un registro.

Sin embargo, para obtener máxima performance se requiere que todos los threads de un *warp* accedan al mismo dato simultáneamente. En este caso se produce un *miss*, seguido de un *broadcast* del dato a todos estos threads. Si no se cumple este patrón de acceso, el costo de una lectura es lineal con la cantidad de direcciones distintas accedidas por los threads. Por ello, en los casos en donde cada thread requiera un dato distinto, puede ser más conveniente precargar una serie de datos de memoria global a memoria compartida, usando así a esta última como una caché.

2.4.8. Explotación de paralelismo

En general, lo más importante es exponer correctamente el paralelismo del problema a resolver. Es posible que existan distintos algoritmos paralelizados, con distintas ventajas y desventajas. Compartir datos comunes a través de la memoria compartida suele ser deseable. Sin embargo, hay que tener en cuenta que en ciertas situaciones, debido al inmenso poder de cómputo de estos procesadores, recomputar datos en cada thread en vez de cargarlos de memoria puede dar mejores resultados de rendimiento.

El paralelismo también puede ser explotado a nivel host, haciendo uso de *streams*. Un *stream* corresponde a un pipeline de kernels que se aplican en serie a un flujo de datos de entrada. Si se tienen varios de estos pipelines, es posible hacer un uso más eficiente de los tiempos muertos intercalando la ejecución de los distintos kernels. Este esquema corresponde justamente a una arquitectura clásica tipo *pipe & filter*.

Por último, si se hacen uso de threads a nivel sistema, podría ser posible utilizar el CPU mientras el GPU se encuentra computando. Sin embargo, muchos kernels suelen correr muy rápidamente, y hacer uso del CPU en esos casos podría introducir latencias adicionales.

Capítulo 3

Algoritmos

A partir de las ecuaciones presentadas en el capítulo 1, es posible llegar a una implementación en forma casi directa mediante la simple traducción de sumatorias y productorias en ciclos de tipo `for`. Sin embargo, dada la gran cantidad de términos involucrados, la complejidad temporal resultaría muy alta. Justamente Molecole, utiliza este tipo de algoritmos simples pero computacionalmente exigentes (de orden tres o incluso orden cuatro). Sin embargo, existen otros algoritmos más eficientes para este tipo de cálculos, que permiten reducir la complejidad a un orden lineal[14], que serán descritos en las siguientes secciones de este capítulo.

Conociendo las capacidades de los procesadores gráficos, una pregunta que surge es qué pasaría si directamente se explotara el paralelismo asociado a una implementación basada en ciclos `for` que calculan valores independientes. Efectivamente, en un trabajo previo[13] se siguió este camino, obteniendo aceleraciones de hasta casi cinco veces (5x). Sin embargo, dado que los requerimientos espaciales resultaron muy altos (este tipo de paralelización requiere un almacenamiento de valores proporcional a la cantidad de iteraciones de estos ciclos) y que la complejidad temporal era demasiado elevada, se hizo necesario considerar alguna implementación alternativa.

Stratmann[14] describe un método por el cual es posible obtener algoritmos de complejidad lineal para los cálculos en cuestión relegando, al menos en principio, cierta precisión numérica en los resultados (pero siempre manteniéndose dentro de un margen de error aceptable). Por otra parte, Yasuda[15] implementó estos mismos algoritmos en el procesador gráfico obteniendo aceleraciones en el rendimiento de hasta diez veces (10x). Sin embargo, el enfoque estuvo puesto, como en la mayoría de las implementaciones de DFT, en el tratamiento de sistemas relativamente grandes como el Taxol ($C_{47}H_{51}NO_{14}$) o la Valinomicina ($C_{54}H_{90}N_6O_{18}$).

Restringiéndose a la resolución de sistemas más pequeños (como es el caso de este trabajo) es posible hacer otras presunciones y llegar a una implementación distinta que se comporte mejor para estos. Así, en este trabajo se describen los algoritmos de complejidad lineal para un cálculo de DFT, a la vez que se los analiza desde el punto de vista de su posterior implementación sobre un procesador gráfico y su aplicación a sistemas de tamaño relativamente pequeño.

3.1. Esquema general del cálculo

En la mayoría de las implementaciones de cálculos de DFT (tal como es el caso de Molecole e incluso de otras implementaciones más eficientes), el mayor tiempo computacional está asociado a la obtención de los pesos de integración, de la densidad, de la energía de intercambio y correlación, y de la matriz de Kohn-Sham. Por esta razón, los esfuerzos se centran en optimizar estas porciones del cálculo.

Más concretamente, y restringiéndose a las porciones más costosas de un cálculo de energía bajo el método de DFT, se requieren determinar los siguientes términos:

1. Las posiciones de los puntos de integración: $\vec{r}_{k,g}$

2. Los pesos de integración: $p_k(\vec{r}_{k,g})$
3. El valor de las funciones base en cada punto: $\phi_i(\vec{r}_{k,g})$
4. La densidad en cada punto: $\rho(\vec{r}_{k,g})$
5. La matriz de Kohn-Sham: Γ_{ij}

Partiendo de una matriz de coeficientes inicial C se puede obtener una primera aproximación para ρ (pasos 3 y 4), a partir de la cual se puede plantear y resolver el sistema de ecuaciones asociado a Γ (paso 5). Estos últimos dos pasos se repiten en forma iterativa hasta minimizar la energía (como funcional de la densidad), de acuerdo a algún criterio de convergencia. En la última iteración, entonces, se calcula la energía final $E_{xc}[\rho]$, lo que generalmente se hace con una grilla más densa que la utilizada durante la convergencia, con el objetivo de mejorar la calidad de los resultados.

En una implementación simple basada en ciclos `for`, como la descrita previamente, la complejidad temporal de los algoritmos asociados a cada paso será proporcional a la cantidad de iteraciones de los mismos. Así, y recordando que K corresponde a la cantidad de átomos del sistema, M a la cantidad de funciones base utilizadas y G al tamaño (constante) de la grilla base, la complejidad temporal para cada uno de los pasos mostrados previamente resulta:

1. $O(G \times K) = O(K)$
   El cálculo de las posiciones implica recorrer los G puntos de la grilla base para cada uno de los K átomos del sistema
2. $O(G \times K \times K^2) = O(K^3)$
   Para obtener los pesos de integración es necesario determinar el peso de cada átomo relativo a los demás ($O(K^2)$) para cada uno de los $G \times K$ puntos de la grilla total
3. $O(G \times K \times M) = O(K \times M)$
   Este paso implica obtener el valor de las M funciones base sobre cada uno los $G \times K$ puntos de la grilla total
4. $O(G \times K \times N_{oc} \times M) = O(K^2 \times M)$
   Según la ecuación (1.7), el costo asociado al cálculo de la densidad sobre un punto de la grilla es $O(N_{oc} \times M)$. Y sabiendo que $N_{oc} = \frac{1}{2}\sum_{i=1}^{K} n_e(i)$, donde $n_e(i)$ corresponde al número de electrones del átomo i, se puede ver que N_{oc} resulta proporcional a K.
5. $O(G \times K \times M^2) = O(K \times M^2)$
   Cada uno de los $G \times K$ puntos de la grilla tiene asociado una matriz Γ propia y esta tiene dimensionalidad $M \times M$.

Excluyendo los sistemas más pequeños, estos costos computacionales son demasiado altos. En la práctica, este tipo de cálculos pueden requerir demasiado tiempo de cómputo, por lo que es necesario partir de algoritmos más eficientes.

3.2. Complejidad lineal

Dado que la resolución del cálculo de los orbitales moleculares está dada por la calidad de la base Gaussiana utilizada, la cantidad de funciones base a computar en general será muy alta. Sin embargo, la definición de una base en sí no tiene (ni puede tener) en cuenta las particularidades de la distribución espacial de los átomos del sistema. Por otro lado, la grilla base con la que se construye la nube de puntos de integración final generalmente se extiende considerablemente en el espacio con el objetivo de poder aproximar incluso las funciones base más difusas (es decir, las de exponentes menores).

Por estas razones, de acuerdo a la distribución espacial del sistema, el valor de muchas funciones base sobre una gran parte de los puntos de la grilla será despreciable (es decir, su valor cae debajo de

cierto umbral). Descartar estos pares punto-función de antemano permitiría restringirse únicamente al cálculo de magnitudes significativas. Lo que se obtiene después de este proceso es un conjunto puntos, cada uno con un conjunto de funciones que se pueden denominar *significativas* para el mismo. En definitiva, con este método se deja de considerar la totalidad de las M funciones para cada punto de integración, para solo tener en cuenta una cantidad acotada.

Dado que el valor de las funciones Gaussianas utilizadas decaen rápidamente en función de la distancia respecto del núcleo sobre el cual están centradas, para sistemas suficientemente grandes la cantidad de funciones significativas por punto se puede acotar con una constante, la cual estará dada por el umbral que se utilice para clasificar una función como significativa o no-significativa.

Sin embargo, en términos de implementación no es práctico tener definido un conjunto de funciones significativas por cada uno de los puntos de la grilla. El costo de almacenamiento, de construcción y recorrido de estos conjuntos podría aplacar los beneficios que se obtienen al descartar las funciones *no-significativas*. Para evitar este problema, se pueden agrupar puntos espacialmente cercanos y obtener un solo conjunto para los mismos. Si una función es significativa para algún punto del grupo, lo será para el grupo en sí. Si bien con este agrupamiento potencialmente se incluirían algunas funciones no-significativas para algunos puntos del grupo, la cantidad sería mínimo).

Determinar el tamaño apropiado para estos grupos (es decir, la cantidad de puntos que contendrán) es un factor importante, ya que para tamaños muy grandes no se estarán descartando tantas funciones como se podrían. Por otro lado, para conjuntos muy pequeños se estaría acercando al esquema de determinar funciones significativas por punto. De todos modos está claro que el tamaño esta directamente relacionado con la velocidad de decaimiento de las funciones de dicho grupo (la cual es muy alta, por tratarse de una función Gaussiana).

3.2.1. Criterio de selección de funciones

Una función se podría considerar significativa cuando su valor sobre un punto dado fuera mayor que cierto umbral ε. Sin embargo, un criterio basado en el valor de la función es costoso, dado que requiere de la evaluación de las M funciones sobre los $G \times K$ puntos. Es posible tener un criterio distinto, pero de resultados similares. Dado que la velocidad de decaimiento de una función Gaussiana está dada mayormente por el exponente, una función f_v (1.6) puede considerarse como no-significativa si dicho exponentes es mayor que un δ dado. En nuestro caso, este fue el criterio que utilizamos.

En otras palabras, una función f_v se considerará significativa para un punto dado (x_1, y_1, z_1) si y solo si:

$$-\alpha_\nu d^2 > \delta \tag{3.1}$$

donde se define:

$$d = |(x_1, y_1, z_1) - (x_0, y_0, z_0)|$$

Teniendo en cuenta que una contracción se escribe como combinación lineal de funciones Gaussianas, que la cantidad de funciones por contracción es relativamente baja y que los α_ν difieren sustancialmente entre uno y otro (para así representar una contracción con el menor número posible de Gaussianas), el criterio para clasificar será aplicado directamente a su función f_v más difusa. Es decir, se reescribe el criterio anterior tomando $\alpha_{\text{mín}}$ en vez de α_ν, según:

$$\alpha_{\text{mín}} = \min_\nu \{\alpha_\nu\}$$

Teniendo una función fija, en principio, este criterio se debería aplicar a cada uno de los puntos de cada grupo. Para evitar este recorrido costoso se puede relajar la condición (3.1), simplemente redefiniendo d como la distancia entre el núcleo asociado a la función y el grupo mismo. Esta distancia núcleo-grupo se define como la mínima distancia que puede haber entre un punto cualquiera del grupo

y el núcleo (contemplando así incluso al punto más lejano). Dependiendo de la geometría tridimensional que define un grupo de puntos (un cubo, por ejemplo), la distancia se calcula de distinta forma, por lo que este aspecto se describe en la siguiente sección.

Por último, con el criterio así definido, y sabiendo que la distancia grupo-punto se obtiene con costo $O(1)$, el algoritmo de selección pasa de tener un costo $O(K \times M)$ a tener costo $O(C \times M)$, donde C es la cantidad de grupos que conforman la partición.

3.2.2. Construcción de grupos

La forma comúnmente utilizada en este tipo de cálculos para agrupar puntos de integración espacialmente cercanos consiste en definir primero un prisma que contenga a todo el sistema y luego particionar este en cubos regulares.

Lo que se busca, entonces, es un prisma tal que encierre únicamente la porción del espacio que contenga a todos los puntos *significativos*. Es decir, puntos tales que exista el menos una función considerada significativa para los mismos. Para construir este prisma, se puede buscar, para cada átomo, la distancia máxima a la que puede encontrarse un punto significativo, la cual estará dada por la función más difusa que aún resulte significativa sobre el mismo. Concretamente, esta distancia o "radio de acción" se puede encontrar despejando d de la inecuación (3.1), tomando el mínimo $\alpha_{\text{mín}}$ entre todas las contracciones (teniendo en cuenta así al exponente correspondiente a la contracción más difusa). Con este radio d, para cada átomo queda determinada entonces una esfera que lo rodea, y así puede construirse el prisma de forma tal de abarcar la totalidad de las mismas. En consecuencia, el prisma contendrá a la totalidad de los puntos significativos del sistema.

Si se define el prisma resultante como $P = (\vec{v_0}, \vec{v_1})$, mediante dos de sus vértices opuestos $\vec{v_0}, \vec{v_1}$, donde:

$$\vec{v_1} - \vec{v_0} = (w_x, w_y, w_z)$$

se puede aplicar ahora un particionamiento basado en cubos. Como resultado, se obtiene un segundo prisma $P' = (\vec{v_0'}, \vec{v_1'})$, tal que:

$$\vec{v_1'} - \vec{v_0'} = \left(\left\lceil \frac{w_x}{\ell} \right\rceil, \left\lceil \frac{w_y}{\ell} \right\rceil, \left\lceil \frac{w_z}{\ell} \right\rceil \right) . \ell$$

donde ℓ corresponde a la longitud de las aristas de los cubos. Este esquema de particionado basado en cubos fue definido originalmente por Stratmann[14] y también implementado por Yasuda[15].

Sin embargo, dado que el particionado es regular y que la distribución de puntos es poco homogénea (la densidad de puntos es mucho mayor alrededor de los núcleos), la elección de cubos como único método de particionamiento genera, en la mayoría de los casos, un gran número de grupos con pocos puntos (dependiendo de ℓ). Como consecuencia, habrá varios grupos vecinos poco poblados, que bien podrían ser unidos para así lograr una menor redundancia de cálculo de funciones compartidas.

Por estas razones, parece más apropiado un particionamiento adaptativo, donde el tamaño de los cubos varíe teniendo en cuenta la densidad irregular de puntos. Sin embargo, este modo de particionamiento puede resultar costoso computacionalmente. Por ello, en este trabajo se introdujo una variación por sobre el esquema comúnmente usado, que no requiere un particionado adaptativo.

La variación consiste en introducir un segundo tipo de grupos, que convive con el basado en cubos, de esferas centradas en los átomos. El propósito de estas esferas es el de agrupar en un solo conjunto la mayor parte de los puntos asociados a un átomo (y que se encuentran rodeándolo). Los puntos que no caigan dentro de estas esferas serán asignados a los cubos correspondientes a esa porción del espacio.

En cuanto a la distancia grupo-átomo, en el caso de las esferas, se puede tomar simplemente como la distancia entre el punto más cercano de la esfera al átomo en cuestión, es decir:

$$d = |\vec{a_k} - \vec{c}| - r$$

donde $\vec{c}$ corresponde a la posición de la esfera, y r a su radio. Para el caso de los cubos, dado que estos se encuentran alineados a los ejes cartesianos, la distancia se puede obtener descomponiendo el vector en cuestión en sus tres componentes (figura 3.1).

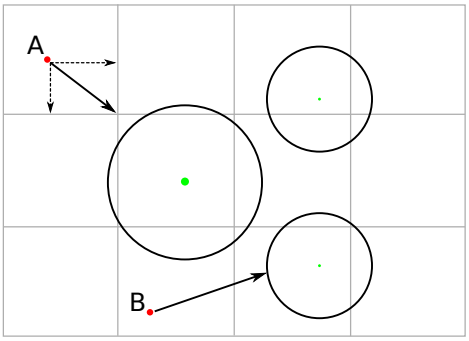


Figura 3.1: Esquema en dos dimensiones del particionamiento basado en cubos y esferas. Los puntos rojos ilustran un punto cualquiera de la grilla, mientras que las flechas representan los vectores distancia entre estos y un cubo (punto A) y una esfera (punto B). Los puntos verdes corresponden a la posición de un átomo.

Este particionamiento híbrido tiene las siguientes consecuencias:

- La cantidad de funciones por punto resulta ser menor que utilizando un particionamiento únicamente basado en cubos. Posiblemente esto se deba a que dentro de las esferas correspondientes a cada átomo queden encerrados los puntos que más funciones significativas en común tienen.
- Se puede tomar un ℓ mayor que si no se utilizaran esferas, y aún así obtener una cantidad de funciones por punto menor. Esto tiene como ventaja asociada una cantidad de puntos por grupo mayor, que será beneficioso en una implementación que paralelice el cómputo por punto (como se verá más adelante).

3.2.3. Nuevo análisis de costos

Como fue mencionado, el método de clasificación efectivamente permite acotar con una constante la cantidad de funciones que se deben tratar en el cálculo de la energía, en función del tamaño del sistema. En otras palabras, lo que se logra es una reducción en la complejidad temporal de casi todos los pasos asociados al cálculo de la energía de intercambio y correlación.

En primer lugar, el cálculo de las posiciones de los puntos de integración (paso 1) mantiene su costo, dado que no involucra a las funciones base utilizadas. Sin embargo, el cálculo de los pesos de integración (paso 2) está indirectamente relacionado con dichas funciones, por lo que su complejidad sí se ve reducida. En principio, según (1.11), para obtener estos pesos es necesario visitar todos los pares (i, j) de los K átomos del sistema (con costo $O(K^2)$). Sin embargo, se puede ver[14] que basta con visitar únicamente a los pares de átomos que tengan alguna función significativa (con respecto a un grupo de puntos dado) centrada en sí mismos. Estos átomos se pueden considerar entonces como "átomos significativos" para el grupo. Si bien el peso correcto para estos átomos sería $\frac{1}{K}$, en realidad se puede tomar un peso igual a cero (excluyéndolos así de este cálculo) sin perder precisión en el valor de la energía. De este modo, la cantidad de átomos a recorrer por cada punto ahora también puede acotarse con una constante. El costo final del paso 2 resulta ser finalmente $O(K)$.

Nuevamente, debido a que la cantidad de funciones significativas (M') ahora puede acotarse con una constante, el costo asociado al cálculo de los valores de las funciones sobre los puntos de la grilla también resulta lineal en la cantidad de átomos K. Análogamente, el costo computacional del cálculo de la matriz de Kohn-Sham (paso 5) queda también reducido a $O(K)$.

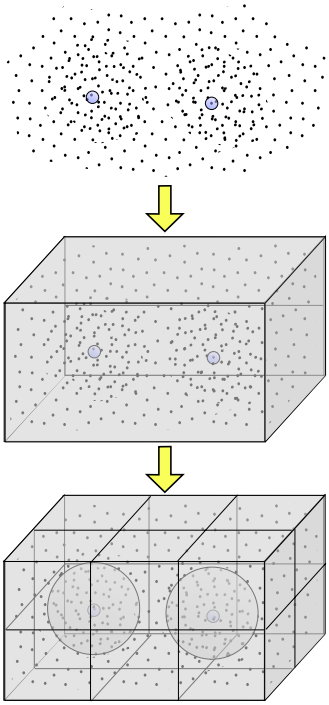


Figura 3.2: Esquema del algoritmo de particionamiento. El primer paso consiste en la construcción del prisma, mientras que el segundo, en la partición del mismo en cubos y esferas.

A diferencia de todos estos cálculos, el de la densidad (y energía) no resulta en un costo lineal, sino cuadrático (es decir, $O(K^2)$). Esta particularidad se debe al uso de la expresión de la densidad como fue dada en (1.7). Sin embargo, existe otra forma de escribir este término, como sigue:

$$\rho(\vec{r}) = \sum_{i=1}^{M}\sum_{j=1}^{M} P_{ij}\phi_i(\vec{r})\phi_j(\vec{r}) \tag{3.2}$$

donde P se denomina Matriz Densidad, y está dada por:

$$P_{ij} = 2\sum_{k=1}^{\frac{1}{2}N_{oc}} C_{ik}C_{jk}$$

Con esta nueva expresión, dado que la matriz P_{ij} puede ser precomputada, sí es posible llegar a una implementación lineal en K. Sin embargo, la expresión (1.7) es más apropiada para una implementación en el procesador gráfico. De hecho, para los sistemas analizados (ver sección 5.2), la expresión utilizada parecería ser más eficiente en la mayoría de los casos, y solo marginalmente más costosa en los sistemas más grandes. Esto se puede ver si se plantea una expresión del costo aproximado de ambas implementaciones, y se analiza la relación entre estos en función del tamaño del sistema (figura 3.3).

De todos modos, queda pendiente como trabajo a futuro investigar los resultados de una implementación totalmente lineal en GPU para el cómputo de la densidad.

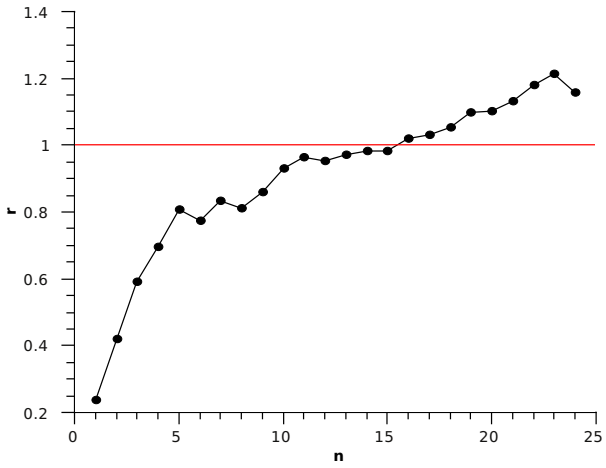


Figura 3.3: Comparación de costo teórico entre la implementación basada en (1.7) y la basada en (3.2), para el cálculo de la densidad. Si se define a M_i' como la cantidad de funciones significativas asociadas al i-ésimo grupo, y C a la cantidad de éstos, el costo asociado a (1.7) se aproxima como $\sum_i N_{oc} \times M_i'$, mientras que el asociado a (3.2), como $\sum_i M_i' \times M_i'$. El gráfico corresponde a la relación $r = \frac{\sum_i N_{oc} \times M_i'}{\sum_i M_i' \times M_i'}$ entre ambos, en función de la cantidad n de moléculas de agua presentes en el sistema.
A partir de un $n = 15$, r resulta ser mayor a 1, indicando que recién para sistemas de más de 15 moléculas de agua la expresión (3.2) resultaría más conveniente.

Capítulo 4

Implementación

En este capítulo se describen los *kernels* implementados, incluyendo las variantes evaluadas. Además, se detalla cómo se tuvieron en cuenta los aspectos de performance expuestos en la sección 2.4.

4.1. Consideraciones previas

La primera decisión que se tomó fue la de determinar la unidad de paralelismo. Es decir, del cómputo de cuáles datos será responsable, y por ende cuántos recursos necesitará cada thread. El particionamiento del espacio de integración en grupos de puntos, además de acotar las interacciones a determinar, permite aplicar una estrategia *divide & conquer* sobre la totalidad del cálculo. Es decir, se pueden definir los *kernels* de manera tal que operen en forma paralelizada sobre un grupo de puntos por vez. Si bien en principio se podría hacer que los *kernels* resolvieran la totalidad del cálculo en una sola llamada, los recursos requeridos por cada thread y la complejidad del código resultante se convertirían en factores limitantes para el rendimiento.

Otro factor que se tuvo en cuenta es que una gran cantidad de datos necesarios en cada iteración son constantes, y por ello podrían ser precalculados y almacenados. A diferencia de lo que se pensaría al desarrollar un algoritmo para ser ejecutado en un procesador convencional, en los GPUs el recálculo de datos puede resultar más eficiente que un repetido acceso a memoria. Esto es resultado de su enorme capacidad de procesamiento, la alta latencia asociada a los accesos a memoria y la inexistencia de varios niveles de caché (en el caso de la memoria global). Efectivamente, los dos *kernels* definidos por Yasuda[15] (uno para el cálculo de la densidad y otro para el de la matriz de Kohn-Sham) recalculan de forma altamente redundante las funciones necesarias en cada iteración. Además, si bien con las técnicas mencionadas es posible reducir la cantidad de funciones por punto a determinar, en general (y en particular, en la implementación de Yasuda), para sistemas relativamente grandes no es razonable precalcular y almacenar estos valores en la memoria de video debido al tamaño limitado de esta última. Sin embargo, en nuestro caso estamos interesados en sistemas relativamente pequeños, con lo que sí resulta posible considerar una implementación de este tipo.

Así, se decidió explorar los pros y contras de dos implementaciones posibles: una versión que recalcule las funciones base y otra que las precalcule en un primer paso, para así poder acceder a estos valores almacenados en cada iteración.

4.2. Esquema general de implementación

A grandes rasgos, la porción del cómputo de interés en este trabajo puede resumirse mediante el siguiente pseudo-código:

Algoritmo 4.1 Pseudo-código de la totalidad del cálculo

```
inicializacion()
P = particionamiento()
W = pesos(P)
F = funciones(P)

hasta determinar convergencia
  d = densidad(P, F, R, W)
  R = matriz_kohn_sham(P, d, F, R)
fin

e = energia(P, R)
```

El primer paso, el de la inicialización, consiste en la carga de datos (parámetros constantes) desde memoria principal a memoria de video. El segundo, corresponde al particionamiento del sistema en grupos de puntos de integración, lo que incluye la selección de funciones significativas para cada grupo determinado. Habiendo construido la partición `P`, se pueden determinar los pesos de integración y, en el caso de la versión con precálculo, los valores de las funciones base `F` asociadas a cada punto. En cada iteración se calcula la densidad `d` y, a partir de ella, la matriz de Kohn-Sham. En el último paso, se calcula la energía asociada a la matriz `R` obtenida.

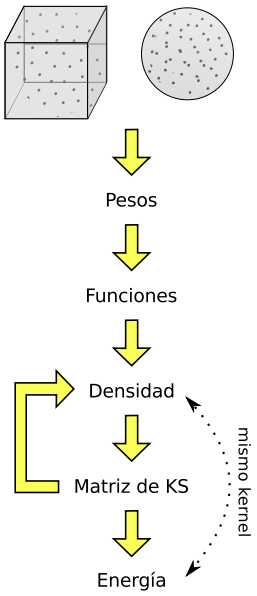


Figura 4.1: Esquema de ejecución de los *kernels*

Los métodos `pesos()`, `funciones()`, `densidad()/energía()` y `matriz_kohn_sham()` involucran la resolución de los respectivos cálculos sobre cada grupo de puntos individualmente, mediante sucesivos llamados a distintos *kernels* de GPU (figura 4.1). Debido a esta subdivisión del cálculo, los valores de las funciones, la densidad y matriz de Kohn-Sham se almacenan de forma separada por cada grupo. Dado que se resuelve de a un grupo por vez, esto implica que el costo de almacenamiento en la memoria de video se ve reducido al utilizar este esquema de particionamiento. El valor de las funciones, en realidad, es un caso especial. Para la variante de la implementación en la cual estos valores son precalculados, sí es necesario almacenar la totalidad de los mismos en un paso previo. De esta forma, el costo de almacenamiento es proporcional a la cantidad de grupos y de funciones significativas por cada uno de estos. Sin embargo, en el caso de la variante con recálculo, este costo es prácticamente nulo

debido a que el cálculo correspondiente al método `funciones()` no es ejecutado y, por ende, no es necesario dicho almacenamiento.

Los kernels implementados fueron cuatro en total, y éstos tienen asociados el cálculo de: (a) los pesos de integración, (b) los valores de las funciones base en cada punto, (c) la matriz de Kohn-Sham y (d) la densidad/energía. En (a), (b) y (d), cada thread es responsable del cálculo correspondiente a un punto, mientras que en (c) cada thread calculará un elemento de dicha matriz. Esto implica que en el primer caso (a, b y d), la grilla de threads es unidimensional, mientras que en el segundo (c), bidimensional. En las siguientes secciones se detallan cada uno de estos *kernels*.

4.3. Pesos de integración

En una implementación serial, el cálculo de los pesos típicamente se resolvería iterando sobre cada uno de los puntos contenidos en los grupos. Sin embargo, dado que el cálculo asociado a un punto es independiente al del resto, estas iteraciones podrían ser resueltas completamente en paralelo. Por esta razón, este *kernel* se configura con una grilla unidimensional de $\lceil \frac{p}{B} \rceil$ bloques de B threads, donde p corresponde a la cantidad de puntos del grupo. Dado que la cantidad de threads en ejecución será entonces múltiplo de B, pero que p no necesariamente lo sea, algunos threads del último bloque podrían quedar ociosos. Sin embargo, esto no representa un problema dado que dichos threads pueden ser descartados por el *scheduler* al ejecutar la instrucción `return`.

Experimentando con distintos valores para B, se determinó que con bloques de 128 threads se obtienen los menores tiempos de ejecución.

Algoritmo 4.2 Pseudo-código (resumido) para el kernel asociado al cálculo de los pesos de integración

```
// corresponde el thread a un punto del grupo o no?
thread_valido = (threadAbs < p);

// carga de parámetros de memoria global a compartida
for (i = 0; i < K; i += B) {
  if (i + thread < K) {
    a'_thread = a_{i+thread};
    r~_m^thread = r_m^{i+thread};
  }
}

if (!thread_valido) return;

// cálculo de pesos
P_total = 0;
for (i = 0; i < nucleos; i++) {
  P = 1;
  for (j = 0; j < nucleos; j++) {
    if (i == j) continue;
    μ_{i,j} = calcular_mu(a'_i, a'_j, r_thread, r~_m^thread);
    P *= μ_{i,j};
  }
  P_total += P;
}
```

El código de este kernel (algoritmo 4.2)[1] está compuesto principalmente por dos ciclos `for` anidados, en donde cada uno de estos recorre los núcleos asociados a las funciones significativas del grupo. Para calcular el valor de μ_{ki} de (1.11), en cada una de estas iteraciones se requieren tanto la posición del átomo (un valor de tipo `float3`) como un factor constante r_m (un `float`) asociado al mismo. Debido a

[1] La variable `thread` corresponde a un índice que identifica al thread dentro de un bloque. Asimismo, la variable `threadAbs` identifica a un thread dentro del total de la grilla.

que ambos parámetros serán accedidos frecuentemente por cada thread, estos pueden ser almacenados en dos arreglos residentes en memoria compartida. Sin embargo, si se utiliza un solo arreglo de tipo `float4` para almacenar ambos parámetros en memoria global, se puede requerir un registro menos durante la compilación y también realizar una sola transacción de 16 bytes durante la transferencia.

Por otro lado, dado que estos dos datos no cambian durante la simulación, parecería razonable hacer uso del segmento de memoria solo-lectura que ofrece CUDA (denominado *memoria constante*). El principal beneficio que se obtiene con su uso es la existencia de una caché que permite pagar el alto costo asociado a un acceso a memoria únicamente en caso de fallo de caché o *miss*. Sin embargo, en total, este costo resultaría mayor que si se alojaran estos datos en memoria global previo a la ejecución, y luego se hiciera una transferencia a la memoria compartida (siguiendo el patrón de acceso expuesto en la sección 2.4.1). Esto se debe a que ante cada *miss* se transferirá únicamente un valor de tipo `float4`, mientras que en el caso del uso de memoria global y compartida, se pueden lograr transferencias de varios valores en una misma transacción. Además, el uso de memoria compartida en este caso se asemeja al de una caché pero con la diferencia de que se puede tener un control más fino sobre los *miss* que se produzcan[2]. Por último, dado que todos los threads accederán al mismo elemento del arreglo alojado en memoria compartida simultáneamente, no se producirán conflictos de bancos.

En cuanto al uso de memoria global, el espacio requerido para el cálculo de los pesos es proporcional a la cantidad de puntos contenidos en un grupo dado.

4.4. Funciones base

Este segundo *kernel* permite calcular el valor de cada una de las funciones significativas de un grupo, sobre cada uno de los puntos contenidos en este. Como en el caso anterior, el cálculo asociado a cada punto es independiente al resto y por ello la unidad de paralelismo elegida es la misma. El tamaño de bloque más adecuado también resultó ser de 128 threads.

Algoritmo 4.3 Pseudo-código para el kernel asociado al cálculo de las funciones significativas

```
thread_valido = (threadAbs < p);

for (i = 0; i < M'; i += B) {
  // copia de parámetros de memoria global a memoria compartida
  if (i + thread < M') {
    N'_thread = N_{i+thread}; // cantidad de Gaussianas asociadas a φ_{i+thread}
    for (j = 0; j < N'_thread; j++) {
      // parámetros γ y α asociados a cada Gaussiana
      γ'_{thread,j} = γ_{i+thread,j};
      α'_{thread,j} = α_{i+thread,j};
    }
  }
  __syncthreads();

  if (thread_valido) {
    for (ii = 0; ii < B && (i + ii < M'); ii++) {
      φ = 0;
      for (j = 0; j < N'_ii; j++) φ += γ'_{ii,j} · f_j(r⃗_thread, α'_{ii,j});
      φ_{i+ii,thread} = φ;
    }
  }

  __syncthreads();
}
```

[2]Tanto el tamaño de las líneas de caché de la memoria constante como el algoritmo de reemplazo asociado no son descritos por el fabricante del hardware

A grandes rasgos, el código (algoritmo 4.3) comprende un ciclo de tipo `for` que recorre las M' funciones significativas asociadas al grupo. En este caso, a diferencia del kernel anterior, la cantidad de parámetros necesarios durante el cálculo (correspondientes a las características de cada función) es tal que no es posible almacenar la totalidad de los mismos en memoria compartida, debido a la capacidad limitada de esta última (ver apéndice A.1). Por esta razón, el ciclo principal se descompone en tandas de tamaño fijo (igual al de los arreglos de memoria compartida), en la forma descrita en la sección 2.4.1. Así, en cada tanda se transfiere una porción de los parámetros de memoria global a memoria compartida, y luego se los accede desde esta última durante el cálculo de las funciones. Con el objetivo de satisfacer el requerimientos de acceso eficiente a memoria global, el tamaño de los arreglos (y por ende, de las tandas) es igual al B, el tamaño de bloque utilizado. Nuevamente, por las mismas razones que las presentadas en el inciso anterior, no se producen conflictos de bancos de memoria compartida.

Debido a este esquema de acceso a memoria compartida, es necesario encerrar el cómputo realizado en una tanda entre instrucciones de sincronización. Esto asegura que ningún thread que haya finalizado su ejecución en la tanda actual, intente cargar un nuevo valor en una posición que todavía este siendo accedida por otro thread. De forma análoga, con este mecanismo también se evita el problema inverso, de la lectura anticipada.

Dado que la cantidad de threads en ejecución puede ser mayor que la cantidad de puntos del grupo a resolver (como en el kernel anterior), algunos threads deberían ser terminados con la instrucción `return`. Sin embargo, dado que se está utilizando la instrucción de sincronización, si un thread no llegara a ejecutarla (por haber terminado con `return`), se produciría un espera infinita. Por esta razón, los threads inválidos no son terminados con un `return`, sino que continúan su ejecución junto al resto pero evitando realizar accesos inválidos a memoria.

Por último, en el caso en que este kernel sea utilizado (es decir, cuando se utiliza la variante de implementación con precálculo), el costo de almacenamiento resulta proporcional a la cantidad de funciones significativas del grupo, por la cantidad de puntos presentes en el mismo.

4.5. Densidad / energía

Para el cálculo de la densidad (y por ende, de la energía, debido a que se puede obtener esta a partir de la primera) se evaluaron dos implementaciones posibles. El primero de los *kernels* desarrollados requiere que el valor de las funciones sea precalculado y almacenado en memoria global en un paso previo. El segundo *kernel*, por el contrario, calcula estos valores por sí mismo y los almacena en memoria compartida. Debido a las limitaciones espaciales de esta memoria, no es posible almacenar la totalidad de las funciones, por lo que inevitablemente este cálculo deberá repetirse varias veces durante la ejecución del kernel.

Independientemente de estas dos variantes, si se observa la definición de la densidad en (1.7), se pueden proponer dos implementaciones posibles: (a) un ciclo externo de N_{oc} iteraciones con uno interno de M' iteraciones o (b), inversamente, un ciclo externo de M' iteraciones con otro interno de N_{oc} iteraciones. En el primer caso, cada iteración del ciclo externo determinará el valor de un orbital χ_i, de modo que resulta posible acumular el valor de χ_i^2 directamente, obteniendo así un resultado parcial de la densidad en el punto actual. Sin embargo, esto implica obtener por cada χ_i, las totalidad de las M' funciones significativas. Por el otro lado, si bien este problema no se da con (b) (dado que cada función es requerida una sola vez), para poder calcular χ_i^2, en este caso se requieren almacenar los N_{oc} valores correspondientes a cada orbital χ_i. En las siguientes secciones se verá que el camino más apropiado es (b), en las dos variantes consideradas.

4.5.1. Variante con precálculo

En esta variante de la implementación del cálculo de la densidad, los valores de las M' funciones significativas son leídos desde la memoria global, los cuales fueron almacenadas como resultado de la ejecución del kernel descrito en 4.4.

El orden de iteración de los dos ciclos principales que componen este kernel corresponde al caso (b) mencionado previamente. Es decir, el valor de cada función se utiliza, en principio, una sola vez durante el cálculo. Dado que estos valores se encuentran almacenados en memoria, un acceso repetido a los mismos podría generar una disminución importante el rendimiento.

Sin embargo, este orden de las iteraciones implica que cada thread requiere un espacio de almacenamiento proporcional a N_{oc}, de forma de poder tener el resultado parcial para cada orbital. Como estos N_{oc} valores serán accedidos múltiples veces durante el cálculo (concretamente, M' veces cada uno), no es concebible almacenar los mismos en la memoria global, sino únicamente en la memoria compartida. Si bien de esta forma se evita pagar el alto costo de acceso a memoria global, ahora se debe trabajar dentro de las limitaciones espaciales de la memoria compartida.

Como en el caso del kernel previamente descrito, la forma de trabajar dentro de estas limitaciones espaciales consiste en partir el ciclo interno en tandas de tamaño N_t (algoritmo 4.4). Es decir, en vez de tratar para cada función la totalidad de los N_{oc} valores asociados a cada orbital, se calculan únicamente N_t valores por vez. De esta forma, se puede reducir el requerimiento espacial para los resultados parciales de los orbitales a un tamaño constante y relativamente pequeño, N_t. Esto significa que dicho valor deberá ser elegido teniendo en cuenta los recursos disponibles por thread, según el tamaño de bloque elegido en la configuración (128 threads, en este caso), los registros requeridos por el código y los recursos que ofrece el procesador en cuestión. Probando con distintos valores posibles, los mejores tiempos de ejecución se lograron tomando $N_t = 8$.

Algoritmo 4.4 Pseudo-código para el kernel de la densidad (con precálculo)

```
ρ = 0; // densidad correspondiente al punto asociado al thread actual
thread_valido = (threadAbs < p);

for (i = 0; i < N_oc; i += N_t) {
  for (l = 0; l < N_t; l++) χ_l = 0; // memoria compartida

  for (j = 0; j < M'; j++) {
    __syncthreads();
    if (threadIdx < N_t) {
      // carga de memoria global a memoria compartida
      if (i + thread < N_oc) C'_thread = C_{i+thread,j};
      else C'_thread = 0;
    }
    __syncthreads();

    if (thread_valido) {
      // carga de funciones desde memoria global
      φ' = φ_{j,thread};
      for (k = 0; k < N_t; k++) χ_j = C'_k * φ';
    }
  }

  for (l = 0; l < N_t; l++) ρ += χ_j^2;
}

ρ = 2ρ;
```

Partir el ciclo interno en tandas de tamaño N_t tiene también el beneficio de permitir almacenar los valores de la matriz C en memoria compartida, dado que solo se necesitan aquellos que serán requeridos para calcular la contribución a la densidad de la tanda actual. Por otro lado, también puede verse que en realidad se accederá al valor de cada función no una vez sola, sino una vez por tanda (es decir, $\left\lceil \frac{N_{oc}}{N_t} \right\rceil$ veces). Sin embargo, dado que los sistemas a tratar son relativamente pequeños, N_{oc} será lo suficientemente chico de forma que esta implementación resulta en una cantidad mucho menor de accesos que si se hubiera seguido el orden de iteración (a).

4.5.2. Variante con recálculo

En este segundo caso, la primera diferencia de este kernel respecto del anterior es el reemplazo de la carga del valor de $\phi_{j,p}$ desde memoria global, por una invocación a un método que lo calcula. Debido a que en los casos en donde se invoca un método interno de GPU el compilador en realidad inserta el cuerpo de dicho método en el punto de invocación, este kernel tiene un requerimiento de registros mayor que en el caso anterior. Como consecuencia, es necesario reducir considerablemente el tamaño de los bloques a tan solo 16 threads (tanto para no exceder los límites físicos como para obtener un rendimiento aceptable). Por otro lado, dado que la memoria compartida utilizada para el cálculo de la densidad es proporcional al tamaño de bloque (debido a que cada thread requiere su propio espacio de almacenamiento), esta reducción trae aparejado un consumo de memoria compartida mucho menor, en comparación con el primer kernel. De esta forma, el tamaño N_t de las tandas de orbitales se pudo incrementar a 16. En estas condiciones se genera un índice de *occupancy* muy bajo (ver definición en 2.4.5), dando a notar que los recursos utilizados limitan severamente la cantidad de *warps* que pueden ser ejecutados simultáneamente. Sin embargo, como se verá luego, con este código se logra un rendimiento muy bueno.

Por otro lado, si se vuelve al problema de la expresión de la densidad que se decidió tomar en esta implementación, cabe la duda de si se podría lograr una reducción importante de los recálculos si se pasara al caso lineal (ver (3.2)). Sin embargo, el problema actual que se presentaría sería nuevamente el del límite de los recursos disponibles. En principio este cambio implicaría duplicar el espacio asociado a los parámetros que permiten calcular las funciones base, dado que se requerirían dos series de valores $\phi_{i,p}, \phi_{j,p}$ simultáneamente durante el cálculo. Por otro lado, se podría prescindir de la variable χ en memoria compartida (de tamaño proporcional $N_t \times B$) ya que sería posible acumular directamente sobre ρ, con lo que no está claro si esto sería realmente un limitante. De todas formas, la cantidad de registros necesarios sigue siendo un problema que no es fácil de tratar, dado que este factor no depende más que del código mismo.

4.6. Matriz de Kohn-Sham

El último *kernel* implementado es el que permite calcular los valores de los elementos de la matriz de Kohn-Sham.

En este caso, la unidad de paralelismo se toma de modo tal que cada thread determine el valor de un elemento distinto de la matriz. Es decir, la grilla de threads se define como bidimensional. Sin embargo, dado que solo se requiere determinar una submatriz triangular de la matriz de Kohn-Sham (por ser simétrica), esta configuración implica que del total de los threads lanzados, la mitad de ellos serán innecesarios. Por ello, para mitigar el desaprovechamiento de recursos se puede proceder de la siguiente forma. En primer lugar, dado que la grilla de threads se encuentra subdividida en bloques, es posible directamente descartar (con `return`) aquellos en los cuales no exista ningún thread válido (es decir, bloques fuera del triángulo en cuestión). Para los threads inválidos restantes, si se utiliza la instrucción `return` se podría generar una serialización de instrucciones por no estar descartando *warps* o bloques enteros. Así, se decidió mantener estos threads en ejecución pero evitar que escriban su resultado en memoria global. Este esquema pareció tener un mejor rendimiento que si se utilizara el `return` condicional de estos threads inválidos.

El código de este kernel (algoritmo 4.5) consiste de un ciclo principal, que itera la totalidad de los puntos del grupo actual, de forma de calcular la contribución de cada uno al valor del elemento de la matriz que estará calculando un thread dado. Observando nuevamente (1.13), se ve que en cada iteración se requiere calcular el producto entre el peso de integración y los valores del par de funciones ϕ_i, ϕ_j, asociados a un punto dado, y un coeficiente y que es función de la densidad. Debido a la alta cantidad de puntos contenidos en la mayoría de los grupos, y que todos estos valores se encuentran almacenados en memoria global, es necesario transferir los mismos a memoria compartida de forma de obtener tiempos de acceso aceptables durante el cómputo de la matriz.

Nuevamente, el ciclo principal se separa en tandas de tamaño B (es decir $B_{xy} \times B_{xy}$). Para el caso del factor y, se reserva un arreglo en memoria compartida de tamaño B. En cada iteración del ciclo principal (es decir, al comienzo de cada tanda) cada uno de los threads del bloque cargará un valor en dicho arreglo, logrando así una transferencia eficiente. Para los valores de las funciones, se utilizan dos arreglos bidimensionales (ϕ' y ϕ'' en el pseudo-código) que contendrán los valores de las funciones que serán accedidas por los threads de un bloque dado. Si se reescribe (1.13) de forma matricial:

$$\Gamma = \sum_r wy\Phi(\vec{r})\Phi(\vec{r})^T$$

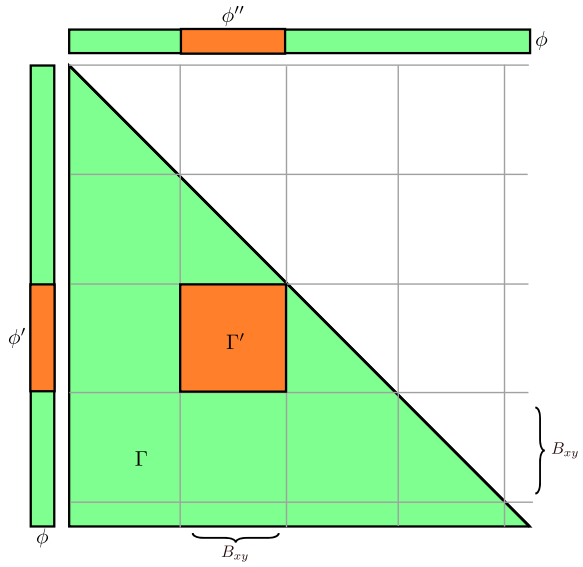
con $\Phi(r) \in \mathbb{R}^{M'}$, donde:

$$\Phi^i(r) = \phi_i(r)$$

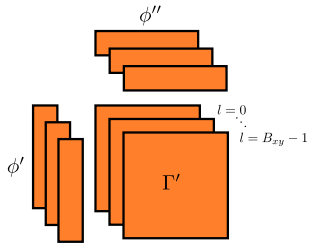
se ve que para el cálculo de una submatriz Γ' de Γ (asociada a un bloque de threads dado) de dimensión $B_{xy} \times B_{xy}$, se requieren únicamente dos sub-vectores de $\Phi(\vec{r})$ de tamaño B_{xy} cada uno (figura 4.2a). Por esta razón, el kernel carga estos dos sub-vectores en los arreglos ϕ' y ϕ''. De hecho, no solo se cargan los B_{xy} valores correspondientes a Γ' para el punto actual, sino para B_{xy} puntos (figura 4.2b). De esta forma, durante el cálculo de una tanda de $B_{xy} \times B_{xy}$ puntos, se cargan cada B_{xy} iteraciones del ciclo interno, B_{xy} funciones (en cada arreglo) para B_{xy} puntos distintos. Si bien parecería más apropiado cargar directamente B funciones distintas para el punto actual, dado que no se estarían satisfaciendo los requerimientos para una transferencia eficiente (por cuestiones de direccionamiento a memoria global), el rendimiento sería menor que con el esquema presentado.

Algoritmo 4.5 Pseudo-código para el kernel del cálculo de la Matriz de Kohn-Sham

```
if (bloque_invalido) return;

int i = threadAbs.x; // columna
int j = threadAbs.y; // fila

bool thread_valido = (i < M' && j < M' && i <= j); // triangulo inferior

γ = 0;
for (k = 0; k < p; k += B_xy * B_xy) {
  __syncthreads();
  // y' en memoria compartida, de tamaño B_xy^2
  if (k + thread.xy < p) y'[thread.xy] = y[k + thread.xy];
  __syncthreads();

  for (l = 0; l < B_xy * B_xy && k + l < p; l++) {
    if (l % B_xy == 0) {
      __syncthreads();
      if (k + l + thread.x < p) {
        // ... carga del arreglo φ' y φ'' (memoria compartida) con los valores
        // de φ correspondientes a la submatriz Γ' asociada al bloque actual
      }
      else {
        // para puntos invalidos
        φ' = 0;
        φ'' = 0;
      }
      __syncthreads();
    }
  }
  γ += φ'_thread.x * φ''_thread.y;
}
Γ_i,j = γ;
```



(a) Subdivisión de Γ en submatrices a partir de la grilla bidimensional de bloques de threads de tamaño B_{xy}



(b) Asignación de arreglos ϕ' y ϕ'', para varios puntos $k = 0, \ldots, B_{xy} - 1$ distintos, a las submatrices Γ' respectivas.

Figura 4.2: Esquema de asignación de bloques de threads a submatrices Γ'

Así como para el kernel de la densidad, se evaluó una implementación con recálculo y se pudo obtener un rendimiento comparable al de la otra variante, para el caso del cálculo de la matriz de Kohn-Sham no se pudo obtener el mismo resultado. Al evaluar una implementación con recálculo para este kernel, el rendimiento resultó ser extremadamente bajo debido a varias razones. En primer lugar, el código era demasiado complejo por lo que se requerían demasiado registros de procesador. En segundo lugar, debido a la configuración bidimensional elegida para la ejecución de este kernel (donde un par de threads i, j están asociados a un elemento $\Gamma_{i,j}$), la invocación simultánea en threads separados del método responsable del cálculo de cada contracción generaba una importante serialización de instrucciones. Esto se debe a que dicho método requiere de una ejecución condicional para diferenciar el tipo de función en particular que debe calcular (tipo s, p o d), de forma tal que threads dentro de un mismo warp siguen inevitablemente caminos distintos. Por último, el espacio requerido en memoria compartida para el almacenamiento de los parámetros necesarios para el cálculo de las funciones también representa un problema importante. Por estas razones, solo se consideró en la implementación actual la variante con precálculo para este kernel. Como consecuencia, incluso en los casos en donde se utiliza la variante con recálculo del kernel asociado a la densidad, es necesario precalcular las funciones de forma de poder contar con ellas en memoria global al momento de ejecutar el kernel asociado a la matriz de Kohn-Sham.

En resumen, con la implementación presentada en esta sección todas las transferencias de memoria se hacen de forma eficiente, no existen conflictos de bancos ni serialización de instrucciones y el índice de *occupancy* resulta ser 0.667, indicando un buen aprovechamiento de recursos. Como se verá en el siguiente capítulo, este kernel presenta un rendimiento más que aceptable, teniendo en cuenta los tiempos de ejecución sobre los sistemas evaluados.

Capítulo 5

Resultados

En este capítulo se analizan principalmente dos aspectos de la implementación desarrollada:

- la ganancia en rendimiento obtenida y
- la calidad numérica de los resultados,

mediante la comparación de las siguientes implementaciones:

1. la desarrollada para ejecutar en GPU
2. la implementación equivalente, para ser ejecutada en CPU
3. la implementación original, Molecole[2]
4. dos aplicaciones ampliamente utilizadas: Gaussian '03[4] y SIESTA[5].

La implementación en GPU corresponde a una versión modificada del software Molecole (escrito en el lenguaje de programación Fortran77), en la cual las porciones del cálculo mencionadas en la sección 3.1 son reemplazadas por *kernels* de GPU. El código (C/C++) necesario para preparar los parámetros necesarios antes y después del cálculo en GPU (permitiendo así su interacción con el código original de Molecole) se encapsula en una biblioteca dinámica.

Por otro lado, la implementación en CPU consiste en una versión modificada de dicha biblioteca, en la cual el conjunto de *kernels* desarrollados para GPU (en lenguaje CUDA) son traducidos a código C++ convencional. El cálculo del cual sería responsable cada thread en la versión paralelizada es realizado en forma secuencial, en sucesivas iteraciones de ciclos for. Por cuestiones de simplicidad, el cálculo del valor de las funciones en ambos casos se hace mediante el kernel correspondiente para GPU.

Dado que se busca que la implementación en CPU sirva para comparar con el desempeño del mismo algoritmo utilizado pero sobre GPU, la traducción a C++ de los kernels incluyó el uso de operaciones de punto flotante de precisión simple, tal como fueron implementados estos en GPU. De esta forma, se puede saber que no se está pagando injustamente el costo computacional extra asociado al uso de precisión doble, ni que la calidad numérica estará dada por este factor.

Hay que remarcar que, si bien para GPU se analizaron dos variantes de la implementación (una con precálculo y otra que con recálculo), para el caso de CPU no parece razonable recalcular datos dado que la existencia de varios niveles de caché permite mitigar considerablemente la alta latencia asociada a los accesos a memoria principal. Aún más, para un nodo de cómputo convencional, es posible tener una memoria principal de gran capacidad a muy bajo costo. Por estas razones, la implementación en CPU almacena la totalidad de las funciones base en memoria principal.

Gaussian '03 corresponde a una implementación también de cálculos de DFT con utilización de bases Gaussianas, al igual que Molecole. Es un software comercial, producido por un gran equipo de trabajo, y utilizado extensivamente en química computacional. Esta implementación sirve como referencia en

términos de calidad numérica y performance alcanzable para este tipo de cálculos. Gaussian '03 también utiliza operaciones de precisión doble, por lo que deberá tenerse en cuenta al comparar posteriormente los resultados obtenidos entre las implementaciones. Además, dado que Gaussian '03 puede ser ejecutado no solo en forma serial sino también en forma paralelizada sobre varios cores de un mismo procesador, se tendrán en cuenta estos dos tipos de ejecuciones al realizar las comparaciones de performance.

SIESTA implementa un método DFT distinto. La principal diferencia consiste en el uso de funciones base numéricas (es decir, tabuladas) en vez de funciones Gaussianas. La matriz de Kohn-Sham también se define de forma distinta, mediante un operador también numérico. Este método permite llegar a una implementación de complejidad lineal en el cálculo de la energía, pero no con las técnicas descritas en esta tesis. Dado que SIESTA es ampliamente utilizado en simulaciones QM-MM o híbridas[29, 30, 31, 32], es valiosa la comparación con la aplicación de GPU.

5.1. Hardware utilizado

Dado que se contó con dos procesadores gráficos distintos, la mayoría de las mediciones de performance y calidad numérica se realizaron sobre ambos. Sin embargo, en algunos casos se muestran únicamente los resultados asociados al hardware de mayor poder de cómputo (el procesador Tesla C1060), mientras que en los otros casos en los que resulta valiosa la comparación entre ambos procesadores, se exhiben las diferencias encontradas entre estos.

La implementación para CPU fue ejecutada en la unidad de cómputo en donde se encuentra instalada la placa de video Tesla C1060 (ver especificaciones en apéndice A.2).

5.2. Evaluación de la implementación

Las distintas implementaciones se evaluaron mediante la simulación de una serie de sistemas de entre 1 a 24 moléculas moléculas de agua (H_2O) en una configuración fija. Las funciones base utilizadas son denominadas *de calidad doble Z,* más funciones de polarización (DZP)[33]. Los coeficientes y exponentes utilizados se exhiben en el apéndice B. Para la medición de tiempo de cómputo se utilizó una grilla de integración de 116 puntos por capa, mientras que para la comparación de los valores finales de la energía, se utilizó una grilla más densa (de 194 puntos por capa) en el último paso del algoritmo iterativo[21, 22, 23]. En el caso de Gaussian se utilizó la misma base y una grilla similar, de 110 puntos por capa (utilizando el parámetro CoarseGrid).

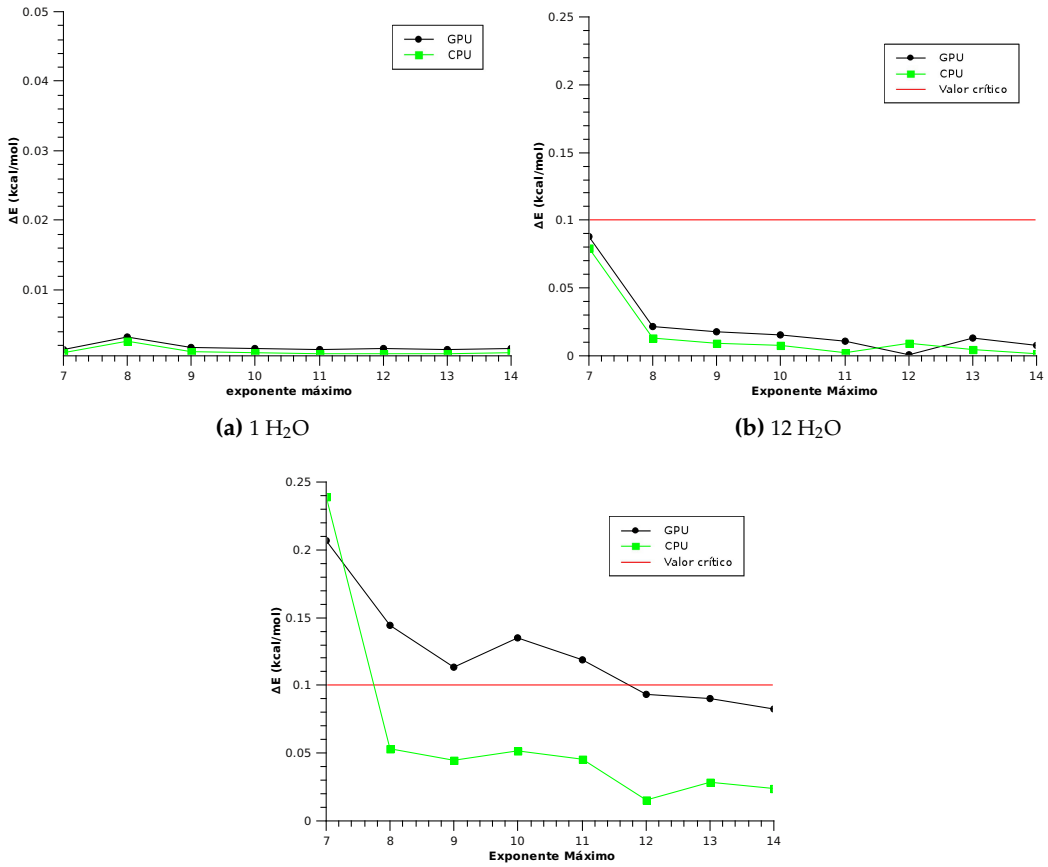
5.2.1. Parámetros óptimos

Las distintas variables que rigen el particionamiento del sistema (el tamaño de los cubos y de las esferas) y la selección de funciones significativas (el exponente máximo permitido) determinan tanto la performance como la calidad numérica de los resultados de una simulación. Por ello, antes de poder comparar las distintas implementaciones es necesario primero determinar los valores óptimos para los parámetros pertinentes.

En cuanto a la calidad numérica, dado que el *software* Molecole considera la totalidad de las funciones Gaussianas (es decir, no las clasifica en significativas y no-significativas), se pueden buscar los parámetros en cuestión minimizando el error absoluto existente entre la energía calculada con esta implementación y las calculadas mediante las implementaciones en GPU y CPU. De hecho, estas tres implementaciones utilizan exactamente la misma grilla, base Gaussiana y criterio de convergencia (que no sucede exactamente con Gaussian '03 ni SIESTA), con lo que el valor de la energía no depende de ningún parámetro más allá de los que se tratan en esta sección.

Así, para determinar el exponente máximo utilizado para la clasificación de funciones, se tuvo como objetivo minimizar dicho error absoluto (figura 5.1). Para determinar qué exponente tomar se decidió

fijar en 0.1 $^{kcal}/_{mol}$ una cota superior para el error, valor que en general permite asegurar que no se está relegando demasiada precisión en el cálculo.



(a) 1 H_2O **(b)** 12 H_2O

(c) 24 H_2O

Figura 5.1: Relación entre la precisión y el exponente máximo (Tesla C1060), para sistemas compuestos por 1, 12 y 24 moléculas de agua.

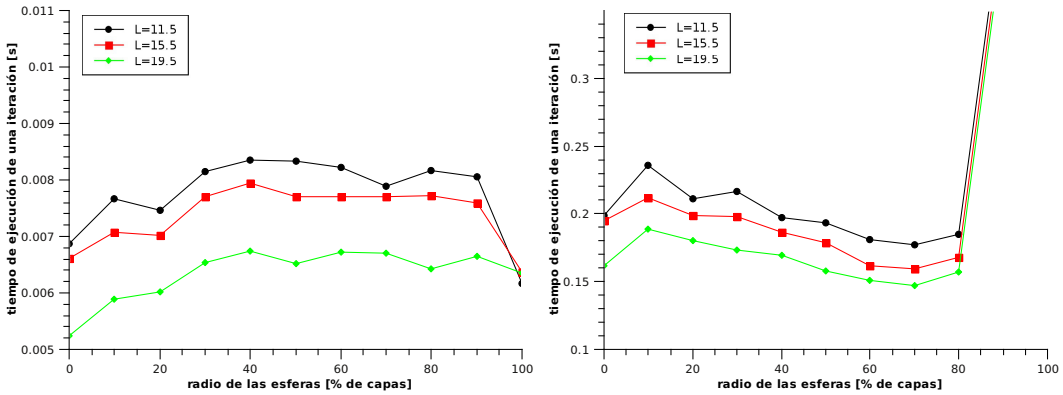
Se puede ver que la utilización del GPU implica una pérdida de calidad numérica, en comparación con los resultados que se pueden obtener con la implementación en CPU, por lo que es necesario un exponente más alto para llegar a resultados aceptables. Teniendo en cuenta que en ambos casos los cálculos fueron realizados con operaciones de precisión simple, estos resultados posiblemente puedan deberse a las diferencias que presenta el procesador gráfico respecto del estándar de punto flotante (conocido como IEEE 754) [26], tales como la introducción de instrucciones del tipo multiplicación-suma (que truncan el resultado intermedio de la multiplicación) o la falta de soporte para números denormalizados.

Teniendo en cuenta estos resultados, se decidió utilizar para los posteriores análisis un exponente máximo de 8 para el caso de CPU y uno de 12, para GPU.

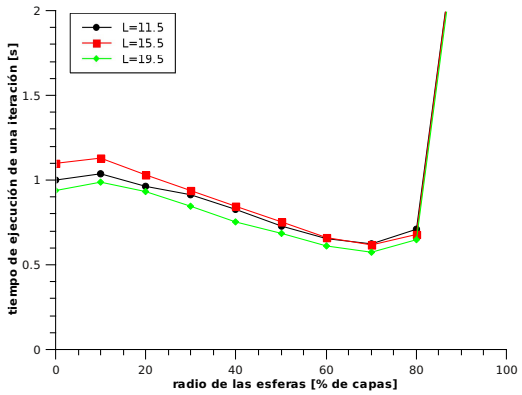
En las simulaciones realizadas se utilizó siempre el mayor radio posible para la esferas de la partición, pero evitando generar su intersección o la inclusión de varios átomos en una sola esfera. Si bien en principio la superposición de esferas no debería representar un problema (siempre y cuando se asegure que todo punto de la grilla esté asignado a un solo grupo, ya sea esfera o cubo), se observaron variaciones considerables en el valor de la energía al incluir más del 60 % de las capas, punto a partir del cual se produce dicha superposición. Por esta razón, y debido a que la diferencia entre los tiempos de ejecución (de una iteración del algoritmo) obtenidos con un radio correspondiente al 60 % de las capas

y uno mayor es despreciable, se decidió tomar este primer valor para el radio de las esferas en todas las simulaciones realizadas.

Si se observan los tiempos de ejecución en función del radio de las esferas (figura 5.2) se puede ver que con esta forma de agrupamiento se puede lograr una disminución en los tiempos de ejecución, como era esperado. De hecho, para el sistema de 24 moléculas de agua, se ve que con el radio elegido (60 %) se reduce el tiempo de ejecución en un 25 %. El menor tiempo se logra con un radio correspondiente al 70 % de las capas, lo que reduce en un 35 % dicho tiempo. Sin embargo, como fue explicado, este valor no fue elegido para el resto de las simulaciones debido a la aparición de variaciones considerables para el valor de la energía final a partir de radios mayores al 60 % de las capas.



(a) 1 H_2O **(b)** 12 H_2O



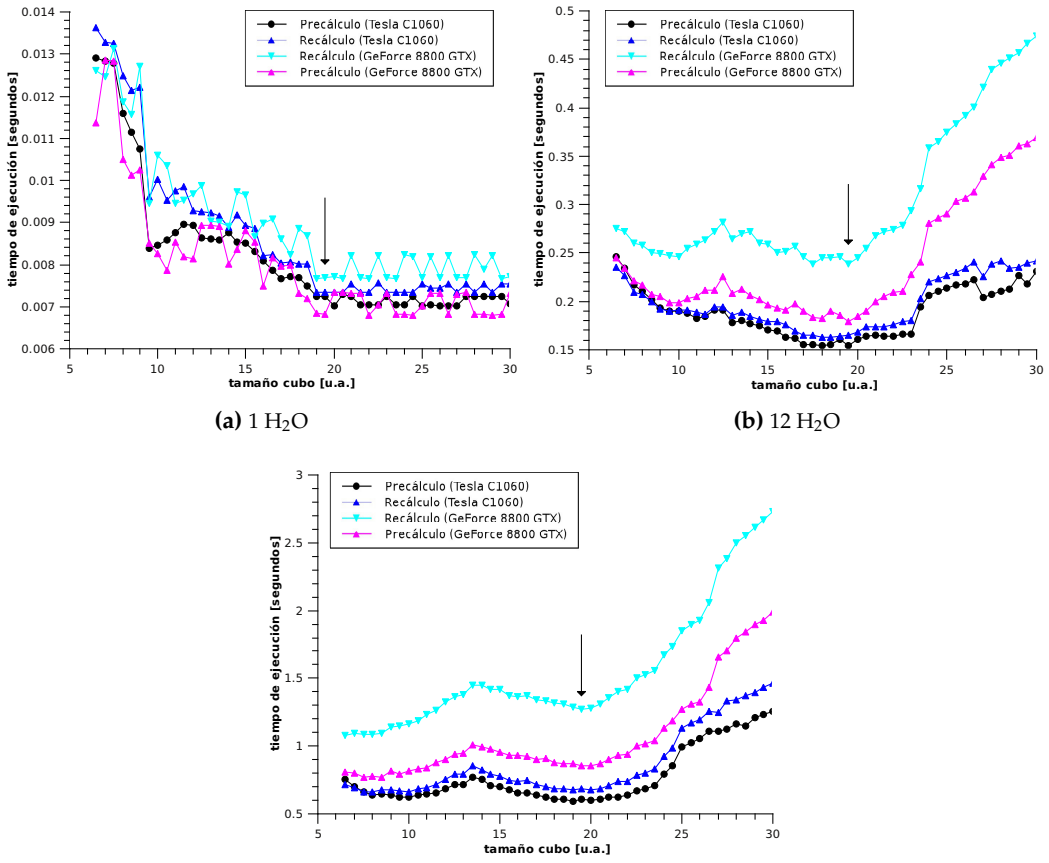
(c) 24 H_2O

Figura 5.2: Tiempos de ejecución de una iteración en función del radio de las esferas, para distintos tamaños de cubo (L) (Tesla C1060), implementación con precálculo (la versión con recálculo se comporta de forma equivalente, pero con tiempos de ejecución ligeramente superiores).

Sabiendo que, por un lado, por cada grupo de puntos de la partición se hace una invocación a los dos kernels computacionales, y por otro, que la unidad de paralelización en sendos casos corresponde a un punto de integración y un par de funciones, resulta conveniente que estos grupos contengan suficientes puntos y funciones significativas de modo de no desperdiciar los recursos del procesador gráfico. Es decir, al generar pocos grupos con muchos puntos se logra que cada cálculo se lance con un gran número de threads, disminuyendo así los tiempos de ejecución dado que se logra un mejor aprovechamiento de los recursos del procesador. Por otro lado, con este agrupamiento se logra también mejorar la eficiencia en la clasificación de funciones. Dado que los puntos cercanos a los núcleos de un átomo tendrán en común una gran cantidad de funciones base con valores no-despreciables, es conveniente que estos

compartan la misma lista de funciones significativas, lo que se logra agrupándolos en una esfera. De todas formas, el beneficio asociado al uso de este tipo de grupos eventualmente se revierte para radios mayores debido a que se comienzan a determinar como significativas demasiadas funciones base (lo que también ocurre al tomar cubos de gran tamaño, como se verá más adelante).

Finalmente, teniendo fijo el exponente máximo y el radio de las esferas a utilizar, solo resta determinar el tamaño óptimo de los cubos de la partición. Para encontrar este valor, se buscó minimizar el tiempo asociado a la ejecución de una iteración del algoritmo de convergencia. Observando los resultados obtenidos (figura 5.3) se puede determinar un tamaño para los cubos de 19.5 ua. (marcado con una flecha en todos los casos). Este valor parece ser adecuado para todo el rango de tamaños de sistemas evaluados.



(a) 1 H_2O **(b)** 12 H_2O

(c) 24 H_2O

Figura 5.3: Relación entre el tiempo de ejecución de una iteración y el tamaño de los cubos (Tesla C1060)

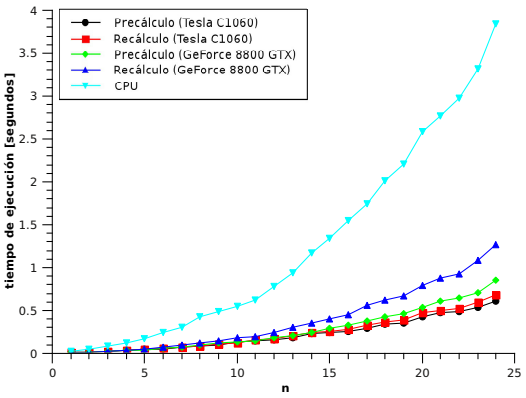
Haciendo este mismo análisis para la implementación análoga en CPU, se determina que el tamaño óptimo para los cubos es mucho menor, de 3.5 ua. aproximadamente. Esto probablemente se deba al hecho de que, en la práctica, el costo de acceso a la memoria principal resulta menos costoso en relación a GPU (debido a la presencia de caché), y que además no se trabaja dentro de las limitaciones de capacidad de la memoria de video (por lo que se pueden almacenar muchas más funciones precalculadas).

5.2.2. Performance

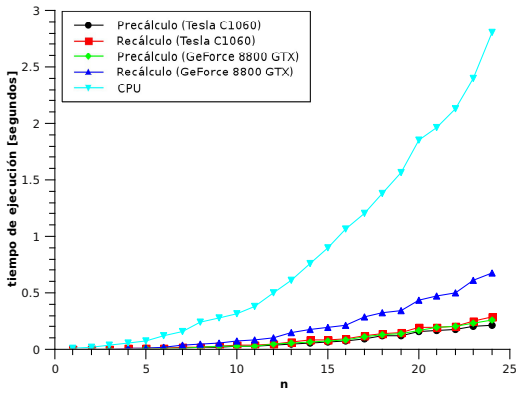
Para el análisis de performance se comparan los tiempos de ejecución entre las distintas implementaciones, tanto de la totalidad de las invocaciones hechas a cada kernel durante una iteración, como el

tiempo total de la misma (esto implica que en este último caso se incluye el tiempo de preparación de parámetros y transferencia de los mismos entre el CPU y GPU). El objetivo de esta medición es poder tener una base de comparación entre la performance de la aplicación desarrollada y otras pre-existentes.

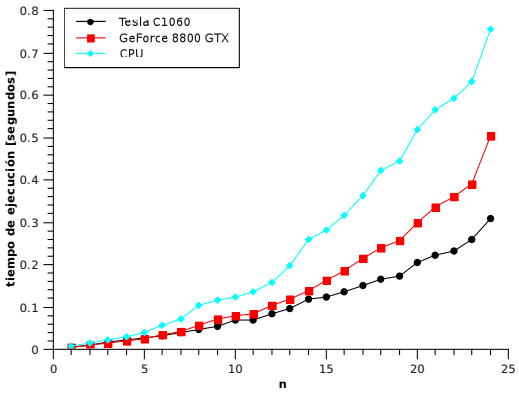
Haciendo uso de los parámetros óptimos obtenidos, se hizo una primera medición (figura 5.4a) del tiempo de ejecución por iteración, comparando la implementación en GPU (en ambos procesadores gráficos, para las variantes con recálculo y precálculo) con la versión implementada en CPU (análoga a la variante con precálculo). Las mediciones fueron realizadas utilizando la función `gettimeofday` del sistema operativo, que tiene una resolución apropiada (del orden del microsegundo).



(a) Una iteración



(b) Kernel asociado a la densidad



(c) Kernel asociado a la matriz de Kohn-Sham

Figura 5.4: Tiempos de ejecución en función del tamaño del sistema (*n* corresponde a la cantidad de moléculas de agua presentes)

Además, se midieron los tiempos de ejecución asociados al conjunto de todas las invocaciones realizadas en una iteración, de los dos kernels involucrados durante la convergencia: el del cálculo de la densidad (figura 5.4b) y el del cálculo de la Matriz de Kohn-Sham (figura 5.4c).

Por último, para analizar el incremento en performance obtenido respecto de las otras implementaciones con las que se cuenta, se presentan los tiempos reales de ejecución de una iteración para 1, 12 y 24 moléculas de agua (figura 5.5), así como los *speedup* relativos obtenidos en cada caso. Tanto para el caso de Molecole, como el de las implementaciones para GPU y CPU, la medición se obtuvo utilizando la función `gettimeofday`. Por otro lado, en el caso de SIESTA, los tiempos correspondientes se extrajeron directamente de los resultados generados por la simulación, que incluyen un análisis de tiempos de

ejecución desglosado por método (profiling). Por último, debido a que no se cuenta con acceso al código fuente de Gaussian '03 y que este tampoco permite extraer un análisis de tiempos de ejecución a partir de los resultados de la simulación, se decidió estimar el tiempo de ejecución de una iteración mediante la medición de distintas ejecuciones de una misma simulación, limitando la cantidad de iteraciones a realizar en cada una.

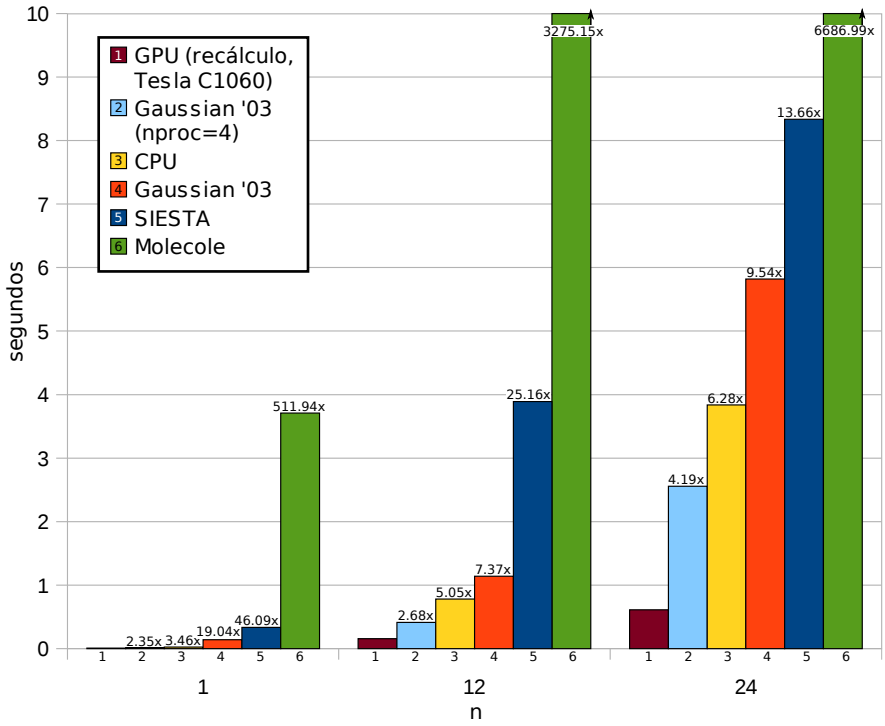


Figura 5.5: Análisis comparativo de performance, entre las distintas implementaciones, para sistemas de 1, 12 y 24 moléculas de agua respectivamente. Las medidas corresponden al tiempo de ejecución (real) de una iteración del algoritmo. Los valores sobre las barras corresponden al *speedup* (o relación de tiempo) obtenido comparando el tiempo de ejecución de la implementación en GPU con respecto a cada una de las otras evaluadas.
Para el caso de Gaussian '03 se incluye tanto una medición de una ejecución serial, como una paralelizada sobre cuatro cores de CPU (`nproc=4`).

Con estos resultados se hacen evidentes dos hechos. En primer lugar, con respecto a Molecole, se pueden obtener mejoras importantes en los tiempos de ejecución de una simulación mediante la implementación de los algoritmos descritos en este trabajo. Esto puede verse comparando los resultados obtenidos con la implementación en CPU con los obtenidos con Molecole. Es más, para los sistemas evaluados, los tiempos de ejecución son menores que los obtenidos con SIESTA y Gaussian '03. En segundo lugar, se puede ver que mediante la ejecución en GPU de estos mismos algoritmos es posible alcanzar mejoras significativas. Tomando como referencia el software Gaussian '03, con la implementación para GPU se obtiene una relación de tiempos de hasta cuatro veces ($4x$) para el caso de la ejecución paralelizada, y de hasta diez veces ($10x$) en la ejecución serial, para el sistema de mayor tamaño. De hecho, con esta misma referencia, con el procesador GTX 8800 (implementación con precálculo) se puede lograr una relación de hasta tres ($3x$) y siete ($7x$) veces, respectivamente.

Hay que aclarar que si bien se obtuvieron mejoras considerables directamente con la implementación en CPU, el uso del procesador gráfico abre nuevos horizontes en la explotación del paralelismo expuesto por el código desarrollado, mediante el uso de futuros procesadores más potentes.

5.2.3. Calidad numérica

Para analizar la precisión de los resultados arrojados por la implementación para GPU, se compara la energía total (es decir, la suma de la energía de intercambio y correlación con las demás energías calculadas con el método) obtenida para los distintos sistemas evaluados.

Sin embargo, dado que existen ciertas diferencias entre las implementaciones (principalmente, existe una diferencia menor en la descripción de las grillas) no basta con comparar el error absoluto entre las energías calculadas entre las distintas implementaciones. Lo que sí se puede hacer es analizar alguna propiedad que se pueda extraer a partir de distintos cálculos y comparar esta entre las distintas implementaciones.

La propiedad que se decidió analizar fue la denominada *energía de formación* de un grupo (o *cluster*) de moléculas de agua. Tomando como E_1 a la energía asociada a un sistema de una molécula, y a E_n de n moléculas, se define como energía de formación:

$$E_F = E_n - nE_1$$

De esta forma, se puede obtener la diferencia entre las E_F asociadas a cada implementación. La diferencia entre la E_F asociada a la implementación en GPU y cada una de las otras evaluadas (ΔE_F) sirve entonces como base de análisis de la calidad de los resultados obtenidos. Además de este *error absoluto* ΔE_F, se calcula el *error relativo* $\Delta E_F / E_F$ en cada caso, de forma de independizarse de la unidad utilizada ($^{\text{kcal}}/_{\text{mol}}$). En la tabla5.1 se presentan estas diferencias, medidas entre 1 y 24 moléculas de agua (H_2O) (es decir, $n = 24$), entre la implementación para GPU y las otras consideradas.

Implementación	$\Delta E_F \left[^{\text{kcal}}/_{\text{mol}}\right]$	$\Delta E_F / E_F$
Molecole	0.06	0.0004
Traducción a CPU	0.07	0.0005
Gaussian '03	0.36	0.003
SIESTA	8.98	0.063

Cuadro 5.1: Diferencias entre las energías de formación entre las distintas implementaciones, calculadas a partir de 1 y 24 moléculas de agua (H_2O)

En primer lugar, se observa que la diferencia con respecto a Molecole es mínima. Dado que esta última considera la totalidad de las funciones base, este resultado habla bien del método de eliminación de funciones no-significativas. De todas formas, el resultado es esperable dado que justamente se ajustaron los parámetros de la implementación en GPU de modo de controlar el error absoluto de la energía. Aún más, ambas implementaciones utilizan exactamente la misma grilla y funciones base. Por otro lado, la diferencia en energía de formación entre la implementación en GPU y en CPU también es muy baja. Nuevamente, se están utilizando las mismas grillas y, especialmente, los mismos valores de las funciones significativas, dado que (como fue previamente mencionado) ambas implementaciones calculan estas con un *kernel* de GPU. Comparando con el reconocido Gaussian, se puede ver que también se logran muy buenos resultados. Teniendo en cuenta que Gaussian es un software altamente utilizado en química computacional y que cuenta con un gran equipo de desarrollo, da mucha confianza sobre la correctitud tanto del algoritmo utilizado como sobre su implementación en GPU. Por último, se puede ver que al comparar con SIESTA se obtienen resultados muy distintos, lo que es esperable debido al método utilizado por esta aplicación. Este resultado podría ser mejorado pero con un costo computacional muy alto.

5.3. Requerimientos espaciales

A diferencia de la implementación con precálculo desarrollada en este trabajo, la mayoría de las implementaciones existentes para resolver este tipo de cálculos no suelen precomputar y almacenar los valores de las funciones significativas debido a que esto podría limitar seriamente la escalabilidad del algoritmo. Por ello, resulta interesante tener una medida concreta del uso de memoria de video en función del tamaño del sistema, observado durante una iteración de las distintas simulaciones evaluadas.

En la figura 5.6 se puede observar que el máximo uso de memoria observado es bastante bajo incluso para los sistemas de mayor tamaño. En relación a la capacidad de la memoria de video que posee el procesador gráfico Tesla C1060 (ver apéndice A.1), el máximo consumo registrado durante una iteración (es decir, dado por el grupo con más puntos y más funciones) representa un mero 3.6 %, aproximadamente.

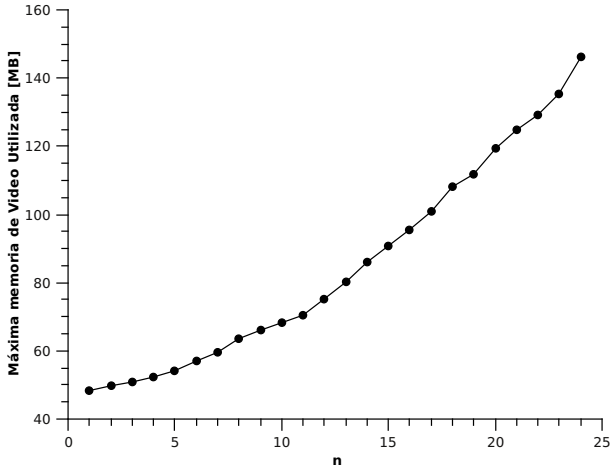


Figura 5.6: Uso máximo de memoria de video durante una iteración (medido en MB)

Si bien en este trabajo se busca principalmente mejorar el rendimiento de los algoritmos presentados en el caso de sistemas de tamaño moderado, estos resultados dan una idea de la escalabilidad de la implementación con precálculo, que resultó ser la de menor tiempo de ejecución en todos los casos.

Capítulo 6

Conclusiones

En este trabajo se evaluaron principalmente dos puntos:

- una implementación distinta a la tradicional para acelerar cálculos DFT
- su desempeño al ser ejecutada en un procesador gráfico

Se habla de una implementación con diferencias respecto de las descritas en la literatura, principalmente por la utilización de un agrupamiento de puntos de integración híbrido, ya que está basada en cubos y esferas. Se pudo ver mediante los resultados obtenidos, que este nuevo esquema de particionamiento mejora considerablemente el desempeño en una implementación orientada a procesadores gráficos.

Por otro lado, con respecto a la implementación de Yasuda[15] existen otras diferencias, todas basadas en la premisa de que se simularán preferentemente sistemas relativamente pequeños. La expresión utilizada para aproximar la densidad electrónica parece ser más adecuada para estos casos. Además, en nuestra implementación (a diferencia del caso de Yasuda) se pudo contemplar una versión de los *kernels* que precalculara parámetros necesarios, resultando ser esta más eficiente que la que los recalcula en cada paso. De todas formas, el procesador gráfico Tesla C1060 mostró ser capaz de equiparar el desempeño de ambas versiones de dichos kernels, debido a su inmenso poder de cómputo. Aún mas, con el procesador 8800 GTX, también se obtuvieron buenos resultados, lo cual no es un dato menor debido a que éste tiene un precio mucho menor al del otro.

Es necesario remarcar que si bien Yasuda logró obtener aceleraciones de hasta diez veces, en comparación con un procesador moderno, la comparación fue hecha entre la implementación en GPU y esta misma traducida para ser ejecutada en CPU. Es decir, en ambos casos el algoritmo recalcula el valor de las funciones significativas. Por el contrario, en la presente tesis, en cuanto a la traducción de los kernels a código de CPU, se apuntó a obtener una implementación eficiente para este tipo de procesador. En otras palabras, los kernels en CPU precalculan los parámetros necesarios y los almacenan en memoria principal, dado que este esquema es el más apropiado para este tipo de arquitectura (en la cual el poder de cómputo no opaca el costo de acceso a memoria principal). Además, tanto el tamaño de los cubos y el exponente elegidos fueron los más apropiados de acuerdo a los tiempos de ejecución resultantes en el CPU. En resumen, se podría decir que la comparación hecha en esta tesis se realizó tomando la mejor implementación para GPU y la mejor para CPU, en vez de la misma en ambas. De esta forma se obtienen resultados menos artificiales y más útiles para un potencial usuario final, interesado en la implementación desarrollada.

La importancia de las mejoras en performance obtenidas se hacen evidentes cuando se tiene en cuenta el tiempo de cómputo que implica una dinámica molecular. Para poner un ejemplo, se puede considerar una simulación de un cluster de 24 moléculas de agua, de 200 picosegundos de duración, con un paso (o *timestep*) de 0.2 femtosegundos de largo. En este caso se estaría hablando de 1 millón de ejecuciones de un cálculo de energía mediante DFT. Teniendo en cuenta que uno de estos cálculos en general implica aproximadamente 6 iteraciones del algoritmo de convergencia, se puede estimar que el tiempo de cómputo de esta simulación utilizando el software Gaussian '03 (teniendo en cuenta

los datos presentados en la figura 5.5, para su ejecución paralelizada sobre cuatro núcleos) tomaría aproximadamente 177 días. Por otro lado, con el software desarrollado para GPU, el tiempo de cálculo se reduciría a aproximadamente 42 días.

Dado que los mejores resultados se obtuvieron con la implementación basada en el precálculo de funciones, también es importante tener en cuenta que, en relación a la capacidad de la memoria de video disponible con los procesadores gráficos más actuales, el costo de almacenamiento en este caso no es tan alto como podría haberse esperado. Esto implica que, si bien los esfuerzos en este trabajo estuvieron centrados en sistemas de tamaño relativamente pequeño, los algoritmos desarrollados todavía permiten tratar sistemas más grandes antes de verse limitados por este aspecto. Aún más, estos resultados dan mayor confianza para eventualmente obtener una implementación basada en el método GGA, y que incluya el cálculo de las fuerzas presentes en el sistema (necesarias en una dinámica molecular), dado que esto implicaría un uso de memoria mayor aún.

Por último, en cuanto a la calidad de los resultados numéricos es necesario recalcar que, contrario a lo que posiblemente se hubiera pensado en un principio, este tipo de cálculos pueden ser resueltos con precisión simple sin introducir errores numéricos significativos. De hecho, los resultados de la implementación desarrollada para GPU estuvieron a la par de los obtenidos con Gaussian '03, que utiliza operaciones de precisión doble.

En general, los resultados obtenidos fueron muy satisfactorios, motivando a continuar con la implementación de cálculos relacionados (por ejemplo, la utilización de métodos más complejos para la aproximación de la densidad, bajo el método DFT), y a tener presente las posibilidades que ofrece esta arquitectura para acelerar cálculos que puedan ser paralelizados.

6.1. Qué se aprendió

Luego del desarrollo realizado sobre procesadores gráficos, nos enfrentamos a muchas dificultades intrínsecas a este tipo de arquitectura altamente paralelizada. Principalmente, fue necesario comprender sus límites y virtudes, para así intentar explotarlos al máximo.

No todo cálculo que exhiba cierto grado de paralelismo podrá aprovechar las particularidades de este hardware, debido a que existen restricciones como la capacidad de la memoria principal o de la memoria compartida, lo cual a veces obliga a rediseñar el código de formas que no son muy eficientes. Sin embargo, siguiendo la estrategia de *divide & conquer*, es posible reducir el cálculo en su totalidad a porciones más simples que puedan ser resueltas por el GPU de forma más eficiente que un procesador convencional.

También pudimos encontrar formas relativamente generalizables a cualquier implementación en GPU que permiten satisfacer los aspectos de performance expuestos. Principalmente, el uso eficiente de la memoria global es el principal factor que se debe tener en cuenta para acelerar los cálculos.

Con respecto a este último punto cabe mencionar que, a la fecha, NVIDIA ya anunció su próxima generación de procesadores denominada Fermi que, además de tener un poder de cómputo mucho mayor, contará con una jerarquía de caché real. En otras palabras, el tiempo de acceso a memoria podría dejar de ser un factor crítico para el rendimiento, por lo que en aplicaciones como las presentadas se podrían lograr mejoras en performance aún mayores.

6.2. Trabajo a futuro

En primer lugar, sería deseable que en un trabajo posterior se pudiera explotar un segundo nivel de paralelismo que surge a partir de la resolución simultánea de varios grupos de puntos a la vez. Actualmente, la implementación recorre en forma secuencial estos grupos, resolviendo uno por vez. Para introducir esta mejora se podría aprovechar la posibilidad que provee CUDA de utilizar varias placas de video simultáneamente. Incluso podría ser valioso hacer uso de librerías de procesamiento paralelo para múltiples nodos de cómputo, como MPI, cada uno con varias placas de video.

Por otro lado, si bien en las pruebas realizadas la búsqueda de parámetros óptimos se trata básicamente de un método de prueba y error (midiendo el tiempo de ejecución de una iteración), en una implementación madura se podría querer recurrir a otra tipo de medición para decidir qué parámetros tomar. Una opción podría consistir en definir alguna función de costo, parametrizada con dichos parámetros, para cada porción del cálculo. Esta función posiblemente podría ser minimizada de alguna forma menos costosa que con el método actual. Sin embargo, esto probablemente implique que aún así se requiera obtener efectivamente el particionado, para así saber la cantidad de funciones significativas y puntos por grupo resultantes. Aún más, obtener una función de costo puede no ser fácil, dado que no se conocen todos los detalles de la arquitectura implementada por el GPU de forma de poder predecir para todos los casos el desempeño teórico de un cálculo dado.

También queda pendiente evaluar la posibilidad de implementar una versión de costo lineal para el cálculo de la densidad, dado que esto probablemente permitiría alcanzar mejoras importantes en el rendimiento a medida que se busque tratar sistemas más grandes que los evaluados.

Por último, y como ya fue mencionado, también es necesario considerar para los próximos desarrollos, esquemas de aproximación más refinados, como lo es el método GGA (en contraste con LDA, utilizado en este trabajo). Esta variante del método implica una cantidad mucho mayor de términos a determinar, por lo que resulta atractivo aplicar el poder de cómputo de los procesadores gráficos también en este caso. De esta forma, se podrían obtener simulaciones confiables de dinámica molecular y a un costo reducido, mediante el uso de procesadores gráficos.

Agradecimientos

Después de tanto tiempo y esfuerzo invertido en este trabajo quiero (y debo) agradecer:

A Andrés Colubri, quien fue el que originalmente vino con la novedosa idea de utilizar procesadores gráficos para cómputo general. Sin esta presentación probablemente habríamos quedado detrás de la ola, teniendo en cuenta que muchos investigadores en el mundo se interesaron por aplicar esta idea en temas similares de química cuántica.

Yendo un poco más atrás, a Darío, quien me presentó la oportunidad de trabajar en el grupo al poco tiempo de estar cursando la carrera. Empezando primero como administrador del cluster, para luego incursionar en tareas de investigación.

A Nano, con quien trabajé bastante a lo largo de todo el tiempo que tengo en el grupo, primero dándome una mano y enseñandome a destripar las máquinas del cluster, y luego orientándome tanto en témas de química como en el laberinto de código `Fortran77` de Molecole. Gracias también por las largas y divertidas sesiones de brainstorming para resolver los problemas que se nos iban presentando (alguna que otra vez con sus epifanías producto del insomnio), y más que nada por tener siempre buena onda y hacer de temas a veces tediosos un trabajo copado.

Estoy agradecido también por haber podido presentar nuestro trabajo en un congreso, mucho antes de lo que hubiera esperado hacer algo así.

También agradezco al grupo de general, por las pequeñas (y a veces no tanto) ayudas que me dieron varios mientras trabajaba en Amber y Molecole (al mismo tiempo que solucionaba problemas de permisos, una impresora que dejaba de andar o encontraba otra fuente quemada).

A Esteban Mocskos por guiarme más en los aspectos computacionales de esta tesis y de encontrar algún que otro anglicismo del cual no me puedo despegar.

A Adrian Roitberg y Turjanski, por traer la GeForce 8800 en su momento, y más adelante proveerme acceso el cluster de la Universidad de Florida (EE.UU.), que me permitió evaluar el programa sobre la placa Tesla.

Y por el lado personal, en primer lugar agradezco a mis viejos por haberme inculcado (voluntaria e involutariantemente) el gusto por la ciencia y la investigación.

A Dagma por siempre estar ahí cuando lo necesite, en cualquier aspecto (porque sabe que es mutuo).

A Oma por tener una actitud admirable en casi todo aspecto, y más que nada por ser una abuela copada.

Y por último, agradezco a Ale por ser una parte tan importante de mi vida, por darme ganas de aprovechar cada momento y de compartir no solo el gusto por la ciencia y la curiosidad, sino también muchas otras cosas que disfrutamos y disfrutaremos juntos. En fin, por demasiadas razones para poner en palabras.

Gracias =)

Bibliografía

[1] R.G. Parr and W. Yang. *Density functional theory of atoms and molecules*. Oxford University Press, New York, 1989.

[2] Darío A. Estrin, G. Corongiu, and E. Clementi. *Methods and Techniques in Computational Chemistry*. METECC, 1993.

[3] NVIDIA. Tesla C1060 computing processor. `http://www.nvidia.com/object/product_tesla_c1060_us.html`.

[4] MJ Frisch, GW Trucks, HB Schlegel, GE Scuseria, MA Robb, JR Cheeseman, JA Montgomery Jr, T. Vreven, KN Kudin, JC Burant, et al. Gaussian 03, Revision C. 02, Gaussian. *Inc., Wallingford, CT*, 2004.

[5] J.M. Soler, E. Artacho, J.D. Gale, AI Garcia, J. Junquera, P. Ordejon, and D. Sánchez-Portal. The SIESTA method for ab initio order-N materials simulation. *Journal of Physics Condensed Matter*, 14(11):2745–2780, 2002.

[6] Tsuyoshi Hamada and Toshiaki Iitaka. The chamomile scheme: An optimized algorithm for n-body simulations on programmable graphics processing units, 2007.

[7] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[8] Koji Yasuda. Two-electron integral evaluation on the graphics processor unit. *Journal of Computational Chemistry*, 29(3):334–342, 2008.

[9] Leslie Vogt, Roberto Olivares-Amaya, Sean Kermes, Yihan Shao, Carlos Amador-Bedolla, and Alan Aspuru-Guzik. Accelerating resolution-of-the-identity second-order moller-plesset quantum chemistry calculations with graphical processing units. *The Journal of Physical Chemistry*, 112:2049–2057, 2008.

[10] Ivan S. Ufimtsev and Todd J. Martinez. Quantum chemistry on graphical processing units. 1. strategies for two-electron integral evaluation. *Journal of Chemical Theory and Computation*, 4:222–231, 2008.

[11] L. Genovese, M. Ospici, T. Deutsch, J.F. Méhaut, A. Neelov, and S. Goedecker. Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *The Journal of Chemical Physics*, 131:034103, 2009.

[12] Amos G. Anderson, William A. Goddard III, and Peter Schröder. Quantum monte carlo on graphical processing units. *Computer Physics Communications*, 177(3):298 – 306, 2007.

[13] Matías Nitsche, Mariano González Lebrero, and Darío Estrín. Accelerating DFT calculations using the Graphical Processor Unit. In *Jornadas Argentinas de Informática - JAIIO HPC*, number 37, UTN-FRSF, Ciudad de Santa Fe, Provincia de Santa Fe, Argentina, 2008.

[14] R. Eric Stratmann, Gustavo E. Scuseria, and Michael J. Frisch. Achieving linear scaling in exchange-correlation density functional quadratures. *Chemical Physics Letters*, 257(3-4):213 – 223, 1996.

[15] Koji Yasuda. Accelerating density functional calculations with graphics processing unit. *Journal of Chemical Theory and Computation*, 4(8):1230–1236, August 2008.

[16] M.C.G. Lebrero, D.E. Bikiel, M.D. Elola, D.A. Estrin, and A.E. Roitberg. Solvent-induced symmetry breaking of nitrate ion in aqueous clusters: A quantum-classical simulation study. *The Journal of Chemical Physics*, 117:2718, 2002.

[17] D.E. Bikiel, F. Di Salvo, M.C.G. Lebrero, F. Doctorovich, and D.A. Estrin. Solvation and Structure of LiAlH4 in Ethereal Solvents. *Inorg. Chem*, 44(15):5286–5292, 2005.

[18] C.M.A. Guardia, M.C. González Lebrero, S.E. Bari, and D.A. Estrin. QM–MM investigation of the reaction products between nitroxyl and O2 in aqueous solution. *Chemical Physics Letters*, 463(1-3):112–116, 2008.

[19] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140(4A):A1133–A1138, Nov 1965.

[20] S. H. Vosko, L. Wilk, and M. Nusair. Accurate spin-dependent electron liquid correlation energies for local spin density calculations: a critical analysis. *Canadian Journal of Physics*, 58:1200–+, 1980.

[21] V. I. Lebedev. Values of the nodes and weights of ninth to seventeenth order gauss-markov quadrature formulae invariant under the octahedron group with inversion. *USSR Computational Mathematics and Mathematical Physics*, 15(1):44 – 51, 1975.

[22] V. I. Lebedev. Quadratures on a sphere. *USSR Computational Mathematics and Mathematical Physics*, 16(2):10 – 24, 1976.

[23] A. D. Becke. A multicenter numerical integration scheme for polyatomic molecules. *The Journal of Chemical Physics*, 88(4):2547–2553, 1988.

[24] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. In *International Conference on Computer Graphics and Interactive Techniques*, pages 896–907. ACM New York, NY, USA, 2003.

[25] R.J. Rost. *OpenGL (R) Shading Language*. Addison-Wesley Professional, 2005.

[26] NVIDIA. *NVIDIA CUDA Programming Guide 2.2*, 4 2009.

[27] Brook language. `http://graphics.stanford.edu/projects/brookgpu/lang.html`.

[28] A. Munshi. The OpenCL specification version 1.0. Technical report, Technical report, Khronos OpenCL Working Group, 2009.04. 02.

[29] A. Crespo, D.A. Scherlis, M.A. Martí, P. Ordejon, A.E. Roitberg, and D.A. Estrin. A DFT-based QM-MM approach designed for the treatment of large molecular systems: application to chorismate mutase. *Journal of Physical Chemistry B-Condensed Phase*, 107(49):13728–13736, 2003.

[30] M.A. Marti, A. Crespo, S.E. Bari, F.A. Doctorovich, and D.A. Estrin. QM- MM Study of Nitrite Reduction by Nitrite Reductase of Pseudomonas aeruginosa. *J. Phys. Chem. B*, 108(46):18073–18080, 2004.

[31] A. Crespo, M.A. Marti, S.G. Kalko, A. Morreale, M. Orozco, J.L. Gelpi, F.J. Luque, and D.A. Estrin. Theoretical study of the truncated hemoglobin HbN: exploring the molecular basis of the NO detoxification mechanism. *J. Am. Chem. Soc*, 127(12):4433–4444, 2005.

[32] L. Capece, M.A. Marti, A. Crespo, F. Doctorovich, and D.A. Estrin. Heme protein oxygen affinity regulation exerted by proximal effects. *Journal of the American Chemical Society*, 128(38):12455–12461, 2006.

[33] N. Godbout, D.R. Salahub, J. Andzelm, and E. Wimmer. Optimization of Gaussian-type basis sets for local spin density functional calculations. Part I. Boron through neon, optimization technique and validation. *Canadian Journal of Chemistry*, 70(2):560–571, 1992.

Apéndice A

Especificaciones técnicas del hardware utilizado

A.1. Procesadores gráficos

Modelo	GeForce 8800 GTX	Tesla C1060
Arquitectura	G80	G200
Core clock [Mhz]	575	602
Procesadores Totales	128 (16MP×8P)	240 (30MP×8P)
Clock procesadores [Mhz]	1350	1300
Tipo de bus de memoria	GDD3	GDD3
Ancho de bus de memoria [bits]	384	512
Capacidad de memoria [MB]	768	4096
Clock de memoria [Mhz]	1800	1600
Ancho de banda [GB/s]	86.4	102
Poder de procesamiento [GFLOPS]	518	936
Capacidad de memoria compartida [KB]	16	16
Cantidad de Registros por Multiprocesador	8192	16384

A.2. Nodo de Cómputo de Referencia

Procesador	Intel(R) Xeon(R) CPU E5462
Clock [Ghz]	2.80
Cores	4
Capacidad memoria [GB]	16
Modelo memoria	PC2 6400

Apéndice B

Funciones base utilizadas

B.1. Base correspondiente al Oxígeno

	f_ν por tipo					
N_k	6	2	1	4	1	1
tipo	s	s	s	p	p	d

Figura B.1: Funciones Gaussianas que componen las distintas contracciones de la base asociada a un átomo de Oxígeno. La cantidad de contracciones es $M = 6$.

α_ν	γ_ν
5222.9022000	-0.001936
782.5399400	-0.014851
177.2674300	-0.073319
49.5166880	-0.245116
15.6664400	-0.480285
5.1793599	-0.335943
10.6014410	0.078806
0.9423170	-0.567695
0.2774746	1.000000
33.4241260	0.017560
7.6221714	0.107630
2.2382093	0.323526
0.6867300	0.483223
0.1938135	1.000000
0.8000000	1.000000

Figura B.2: Exponentes y coeficientes de las distintas funciones Gaussianas. Las funciones están ordenadas por tipo: s, p y d.

B.2. Base correspondiente al Hidrógeno

	f_ν por tipo		
N_k	4	1	1
tipo	s	s	s

Figura B.3: Funciones Gaussianas que componen las distintas contracciones de la base asociada a un átomo de Hidrógeno. La cantidad de contracciones es $M = 3$.

α_ν	γ_ν
5222.9022000	-0.001936
782.5399400	-0.014851
177.2674300	-0.073319
49.5166880	-0.245116
15.6664400	-0.480285
5.1793599	-0.335943

Figura B.4: Exponentes y coeficientes de las distintas funciones Gaussianas. Las funciones están ordenadas por tipo: s, p y d.