Universidad Nacional Autónoma de México

Facultad de Química

Evaluación de la Matriz de Coulomb mediante Unidades de Procesamieto Gráfico y la aproximación de Resolución de la Identidad

T E S I S

QUE PARA OBTENER EL TÍTULO DE

QUÍMICO

P R E S E N T A

Alfonso Esqueda García

TUTOR

Dr. Jorge Martín del Campo Ramírez

SUPERVISOR TÉCNICO

M. en C. Juan Felipe Huan Lew Yee

Ciudad Universitaria, CDMX, 2021

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

JURADO ASIGNADO

PRESIDENTE:	Dr. Amador Bedolla Carlos
VOCAL:	Dr. Orgaz Baqué Luis Emilio
SECRETARIO:	Dr. Martín del Campo Ramírez Jorge
1er. SUPLENTE:	Dr. Álvarez Idaboy Juan Raúl
2° SUPLENTE:	Dr. Barrios Vargas José Eduardo

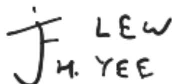
SITIO DONDE SE DESARROLLÓ EL TEMA:

Departamento de Física y Química Teórica, Facultad de Química, UNAM

ASESOR DEL TEMA:

Dr. Jorge Martín del Campo Ramírez

SUPERVISOR TÉCNICO:



M. en C. Juan Felipe Huan Lew Yee

SUSTENTANTE:

Alfonso Esqueda García

Agradecimientos

- Al Dr. Jorge Martín del Campo Ramírez por todo el tiempo invertido en mi desarrollo académico, por su inestimable amistad y sobretodo por alentarme siempre a superarme a mí mismo.
- Al asesor técnico de este trabajo, el M. en C. Juan Felipe Huan Lew Yee. Por su valioso tiempo, su amistad sincera y sobre todo por ser, para mí, un modelo a seguir.
- A los miembros del jurado de mi examen de grado por sus sugerencias para mejorar el presente trabajo y su valioso tiempo para llevar a cabo el examen de grado.
- Al CONACyT Proyecto CB-2016-282791 por los fondos brindados.
- A LANCAD-UNAM-DGTIC-270 por los recursos computacionales proporcionados.
- A mis padres, hermanos, abuelos y el resto de mi familia por su apoyo incondicional y su paciencia. Sin su soporte este logro en mi vida no sería posible.
- A mis colegas, Xiaomin Huang, Rodrigo Cortés, Demetrio Cumplido, Neftali Rodríguez, por su importante amistad y por ser una fuente de inspiración para mí.
- A una gran amiga Sarahí Olalde por su compañía, la motivación que siempre me brindó durante la carrera y su confianza en mí.

未来の私、動き続ける

Resumen

En el presente trabajo se desarrolló un código modular, escrito en *CUDA/C++*, para la evaluación de Integrales de Repulsión Electrónica (ERIs) de tres centros, aprovechando el poder de las Unidades de Procesamiento Gráfico (GPU). Así mismo se programó un generador automático de código en *Python* para el desarrollo de una rutina capaz de evaluar la matriz de Coulomb mediante la aproximación de Resolución de la Identidad (RI), en el método de Hartree-Fock, utilizando los kernels de evaluación de ERIs creados. La rutina para la evaluación de la matriz de Coulomb con RI (J_{RI}) se implementó en el programa de estructura electrónica *Green.128*. La validación del método se realizó mediante cálculos de energía, con la base cc-pVDZ, para un conjunto de moléculas que contiene a los alcanos lineales desde metano hasta octadecano más las moléculas de benceno, fenol, clorobenceno y ribavirina (un medicamento utilizada como tratamiento contra COVID-19).

Para la validación del método se obtuvieron errores mínimos y máximos de $min = -3.80 \times 10^{-05}$ $max = -4.55 \times 10^{-03}$ Hartrees y de $min = 7.93 \times 10^{-07}$ $max = 6.47 \times 10^{-06}$ Hartrees para el error absoluto y el error relativo respectivamente. El tiempo de ejecución de la rutina para la evaluación de J_{RI} se comparó contra dos versiones de Green.128; una con ejecución serial en 1 CPU y otra con ejecución paralela con 6 CPUs. Las aceleraciones máximas registradas contra ambas versiones son de $20.66x$ y $18.34x$ respectivamente.

Abstract

In this work is presented a modular code, written in *CUDA/C++*, for the evaluation of three centers Electronic Repulsion Integrals (ERIs), which takes advantage of the power of Graphical Processing Units (GPU). In addition an automatic code generator was created in *Python* for the development of a routine capable of evaluating the Coulomb matrix via the Resolution of Identity (RI) for the Hartree-Fock method, using the kernels for ERIs evaluation created. The routine for the evaluation of the J_{RI} matrix was implemented in the electronic structure software *Green.128*. The validation of the method was made through the calculation of energy, with the basis cc-pVDZ, for a set of molecules that contain the lineal alkanes from methane to octadecane plus the molecules of benzene, phenol, chlorobenzene and ribavirine (a medicament used as a treatment against COVID-19).

For the validation of the method the minimum and maximum absolute and relative errors of $min = -3.8\times10^{-05}$ $max = -4.55\times10^{-03}$ Hartrees and $min = 7.93\times10^{-07}$ $max = 6.47\times10^{-06}$ Hartrees were respectively obtained. The execution time of the routine for the evaluation of J_{RI} was compared against two versions of Green.128; one with serial execution in 1 CPU and another with parallel execution in 6 CPUs. The registered maximum accelerations against both version are of $20.66x$ and $18.24x$ respectively.

Índice

Agradecimientos II

Resumen IV

Abstract V

1. Introducción 1

2. Motivación 4

3. Marco teórico 5

3.1. Resolución de la Identidad en Hartree-Fock . . . 6
3.2. Algoritmo de implementación de la aproximación RI en HF . . . 7
3.3. Factorización Modificada de Cholesky . . . 8
3.4. Evaluación de las ERIs de tres centros . . . 9
3.5. Unidades de procesamiento gráfico . . . 12

4. Objetivos e hipótesis 16

4.1. Objetivo general . . . 16
4.2. Objetivos particulares . . . 16
4.3. Hipótesis . . . 17

5. Metodología 18

5.1. Hardware y software utilizados . . . 18
5.2. Evaluación de ERIs primitivas de tres centros en GPU . . . 18
5.3. Implementación en Green.128 . . . 20
5.4. Validación del método . . . 22

6. Análisis de resultados 24

6.1. Green.128 GPU vs. Green.128 original . . . 24
6.2. Green.128 GPU vs. Green.128 múltiples CPUs . . . 27

6.3. Perfil de Ejecución Green.128 versión GPU 33
6.4. Generador de código para evaluación de ERIs 34

7. Conclusiones y Perspectivas **42**

Apéndices **44**
A. Teorema del producto de gaussianas 44
B. Implementación del algoritmo K_{RI} 46

Índice de figuras

3.1. Esquema de ejecución de un programa de C++/CUDA para el uso de GPU. En un inicio la CPU tiene el control de la ejecución del programa, esto se realiza de manera serial, hasta que un llamado a un kernel CUDA se realiza y el control de la ejecución se cede momentáneamente a la GPU. 15

5.1. Diagrama de árbol de las ERIs auxiliares involucradas en el kernel para la evaluación de ERIs [ps|p], las ERIs auxiliares son evaluadas desde la parte exterior hacia la parte interior del diagrama. 20

5.2. Diagrama de ejecución de la rutina escrita en C++ para la evaluación de la matriz de Coulomb a partir de los kernels de CUDA, y su posterior contribución a la matriz de Fock. 22

5.3. Molécula de Ribavirina $C_8H_{12}N_40_5$ a) Modelo 2D, b) Modelo 3D. 23

6.1. Tiempo de ejecución para la rutina de evaluación de la matriz de Coulomb con la versión GPU. Sólo fueron incluidos los resultados para los alcanos lineales pues es más sencillo observar la tendencia que si se agregan los resultados para las moléculas aromáticas y la Ribavirina. 30

6.2. Tiempo de ejecución para la rutina de evaluación de la matriz de Coulomb con las tres versiones de Green.128. Sólo fueron incluidos los resultados para los alcanos lineales pues es más sencillo observar la tendencia que si se agregan los resultados para las moléculas aromáticas y la Ribavirina. 33

Índice de tablas

3.1. Procedimiento para la evaluación de la matriz de Coulomb en Hartree-Fock mediante RI. Los pasos en azul se realizan una vez, al iniciar el SCF y los pasos en negro se repiten en cada iteración del ciclo. 7

6.1. Número de funciones base normales y auxiliares, debidas a la base cc-pVDZ, para cada molécula 25

6.2. Comparación entre los ciclos de SCF necesarios para llegar a la convergencia, la energía obtenida y el tiempo de ejecución de la rutina de evaluación de la matriz de Coulomb RI. Los valores de energía para Green.128 GPU indican con color azul los decimales hasta los que el resultado sigue siendo el mismo que para Green.128 original. 26

6.3. Valores de error absoluto y relativo para cada molécula calculada con la versión Green.128 con GPU tomando como referencia los valores obtenidos por el Green.128 en su versión original. En verde se presentan los errores mínimos y en rojo los errores máximos. 28

6.4. Valores de aceleración obtenida por la versión de Green.128 con GPU tomando como referencia la versión original. 29

6.5. Comparación de los valores de energía, el número de ciclos del SCF y el tiempo de ejecución de la rutina de evaluación de la matriz de Coulomb. Para la versión de múltiples CPUs los cálculos siempre fueron realizados utilizando 6 CPUs de manera simultánea. En color azul se muestran los decimales hasta los cuales la energía de Green.128 es igual que para la versión de múltiples CPUs. 31

6.6. Valores de la aceleración en la evaluación de la matriz de Coulomb tomando como referencia la versión Green.128 múltiples CPUs. En verde se presenta la aceleración máxima obtenida. 32

6.7. Perfil de ejecución para la rutina de evaluación de la matriz de Fock en Green.128 versión GPU. En las primeras dos columnas se muestran los tiempos totales para la evaluación de J_{RI} y la factorización de Cholesky de la matriz G. Las últimas tres columnas muestran los porcentajes del tiempo necesario para la evaluación de las tres partes más importantes de la rutina diseñada para la implementación del código modular en Green.128. 34

B.1. Procedimiento para la evaluación de la matriz de Coulomb en Hartree-Fock mediante RI. Los pasos en azul se realizan una vez al iniciar el SCF y los pasos en negro se repiten en cada iteración del ciclo. 46

Índice de algoritmos

1. Descomposición Modificada de Cholesky 9

Índice de listados de código

6.1. Código para generar el kernel para evaluación de las ERIS primitivas del tipo sss, los parámetros que recibe la función son simplemente la angularidad de cada centro. 35

6.2. Kernel de *CUDA/C++* para la evaluación de ERIs sss creado a partir del generador de código en python. 35

6.3. Código para generar los llamados a los kernels necesarios para llevar a cabo el paso 3 del algoritmo de J_{RI}. 37

6.4. Ejemplo de los llamados a los kernels *CUDA* requeridos para la obtención de las ERIs primitivas, contraídas y su contribución al vector x_Q. 38

6.5. Código para generar los llamados a los kernels *CUDA* para realizar la contribución de la matriz de Coulomb a la matriz de Fock. 40

6.6. Ejemplo de los llamados a los kernels *CUDA* requeridos para la obtención de los elementos de la matriz de Coulomb y su contribución a la matriz de Fock. 40

CAPÍTULO 1

Introducción

En las últimas dos décadas la química computacional se volvió una de las áreas de la química con mayor crecimiento y en la actualidad, la capacidad interpretativa y predictiva de los cálculos teóricos permiten realizar aplicaciones tan sofisticadas como la selección de moléculas óptimas para llevar a cabo una reacción química específica o la generación de datos espectroscópicos teóricos de diversas moléculas de interés químico, entre otras. Esto es posible gracias a la capacidad de los dispositivos computacionales modernos que permite aplicar las metodologías teóricas previamente desarrolladas a sistemas cada vez más grandes con gran precisión. Sin embargo, siempre existe la necesidad de alcanzar precisiones mayores y tiempos de cómputo cada vez menores y esta necesidad se debe a que las áreas de interés de la Química Computacional aumentan constantemente, por lo que el obtener mejores resultados con menor costo y tiempo permite a los profesionales del área llevar a cabo investigaciones más sofisticadas en intervalos de tiempo cada vez más accesibles.

Existen dos acercamientos en la búsqueda de un factor costo/rendimiento cada vez más favorable, el primero es a través de nuevos desarrollos metodológicos donde se emplean nuevos modelos o aproximaciones para disminuir el escalamiento de los sistemas estudiados. El segundo se encuentra relacionado con las ciencias computacionales, y es donde se utilizan nuevos algoritmos y arquitecturas de cómputo para acelerar los métodos teóricos existentes. Si bien los desarrollos metodológicos son cruciales y ofrecen mejoras sin precedentes, también es importante mantenerse en contacto con los avances que proveen las ciencias computacionales para obtener el mejor rendimiento posible.

Dentro de las metodologías utilizadas para aumentar la eficiencia computacional de los programas de estructura electrónica se encuentra la aproximación de Resolución de la Identidad (RI, por sus siglas en inglés) que se basa en la expresión de las Integrales de Repulsión Electrónica (ERIs, por sus siglas en inglés) de cuatro centros mediante ERIs de tres centros cuya evaluación es menos costosa computacionalmente. La aproximación de RI puede ser

utilizada en la mayoría de teorías y modelos de la química cuántica debido a que las ERIs de cuatro centros se encuentran presentes en éstos. Un ejemplo de lo anterior es el Ciclo de Campo Autoconsistente (SCF, por sus siglas en inglés), el cual forma parte de modelos como el de Hartree-Fock y también de la Teoría de los Funcionales de la Densidad (DFT, por sus siglas en inglés). En el SCF la aproximación de RI se utiliza para simplificar la evaluación de las ERIs de cuatro centros en los términos de Coulomb e Intercambio.

La aproximación de RI posee 3 ventajas principales sobre la evaluación de ERIs de cuatro centros. En primer lugar, la evaluación de ERIs de tres centros es computacionalmente más sencilla ya que involucra un menor número de operaciones de punto flotante. En segundo lugar, dependiendo de los términos donde se aplique la aproximación de RI es posible diseñar algoritmos de cálculo que disminuyan el escalamiento en comparación con los algoritmos que utilizan ERIs de cuatro centros. Y por último la aproximación de RI requiere un menor uso de la memoria RAM ya que permite expresar todas las ERIs de cuatro centros utilizando un menor número de ERIs de tres centros, lo cual puede reducirse aún más al tomar en cuenta la simetría de las ERIs.

Una de las limitantes en el uso de la aproximación de RI es la alta barrera de conocimiento para llevar a cabo la reproducción del código necesario para su implementación. Este problema se intensifica debido a que la mayoría de las implementaciones computacionales de la aproximación de RI no son modulares y de fácil acceso. En la actualidad es posible utilizar librerías de código libre para evaluación de ERIs de cuatro centros en CPU como LIBINT2[1] y LIBRETA[2] para la implementación de la aproximación de RI. Dichas librerías pueden ser modificadas de manera sencilla para evaluar ERIs de tres centros como ERIs de cuatro centros donde el último centro posee exponente cero y por lo tanto se convierte en la unidad. Sin embargo, estas implementaciones presentan algunos inconvenientes. Primero que nada no se obtiene ventaja alguna en el tiempo de cómputo ya que las ERIs de tres centros aún se calculan como ERIs de cuatro centros debido a que estas librerías no están optimizadas para evaluar ERIs de tres centros. En segundo lugar estas librerías sólo nos permiten realizar la evaluación de las ERIs de tres centros necesarias para implementar la aproximación de RI en vez de presentar una implementación completa de la aproximación de RI. Y por último están diseñadas únicamente para su funcionamiento en Unidades de Procesamiento Central.

La limitación del hardware donde pueden ser utilizadas las librerías antes mencionadas sugiere un problema debido a que día con día aumenta el tamaño de los sistemas químicos que son estudiados mediante la química cuántica computacional. Por lo tanto también aumentan los requerimientos computacionales para su estudio en un periodo de tiempo accesible. Este aumento es solucionado por el uso de nuevas arquitecturas de cómputo paralelo masivo. Uno de los avances más importantes en esta área vino de la mano del mercado de los videojuegos de consumo, que en las últimas tres décadas ha creado una demanda creciente en

los requerimientos computacionales para su óptima ejecución. Dicha demanda se cumple día a día a través del desarrollo de las Unidades de Procesamiento Gráfico (GPUs, por sus siglas en inglés) o tarjetas de gráficos como se les conoce comúnmente.

Las GPUs eran utilizadas tradicionalmente en los videojuegos para la actualización de los colores desplegados por cada pixel en las pantallas, esto explota el gran número de transistores dedicados al procesamiento de datos que contienen las GPUs. Sin embargo la similitud que poseía este problema con las operaciones matriciales desencadenó la investigación del uso de GPUs para el cómputo científico y con el tiempo se estableció la Programación de GPUs de Propósito General (GPGPU, por sus siglas en inglés). A medida que la programación de tarjetas de gráficos se volvió más accesible se crearon varias implementaciones para GPUs en la química computacional y actualmente el uso de GPUs es una opción muy popular en la evaluación de ERIs.

Se han reportado trabajos que implementan la aproximación de RI mediante el uso de GPUs.[3, 4] En la literatura se presenta al lector con las decisiones de diseño de código seguidas para la creación de los códigos que implementan la aproximación de RI en GPU. Sin embargo siguen sin ser programas de código de acceso libre lo cual limita su uso y eleva el tiempo que los investigadores del área necesitan invertir para ser capaces de reproducir y utilizar una implementación de RI que aproveche el poder de cómputo de las GPUs.

CAPÍTULO 2

Motivación

La motivación del presente trabajo es presentar al público interesado una librería de código de acceso libre que permita resolver ERIs de tres centros y utilizar rutinas para implementar la aproximación de RI en diversos modelos y teorías aprovechando el poder de cómputo que proveen las GPUs. La librería planteada permitiría a los usuarios la creación de códigos de estructura electrónica más sofisticados que se beneficien de las ventajas asociadas al uso de la aproximación de RI y de la rapidez que ofrece una implementación de paralelismo mediante GPUs sin la necesidad de invertir tiempo extra en la programación del código necesario para esto.

Por lo tanto, con la creación de este trabajo, se pretende sentar las bases para la creación de la librería GALLETA (GPU Accelerated Library for Long ERIs Transformations with Auxiliary Basis) del grupo de trabajo del Dr. Jorge Martín del Campo Ramírez.

CAPÍTULO 3

Marco teórico

La parte esencial de cualquier cálculo *ab inito* es la evaluación de un gran número de Integrales de Repulsión Electrónica (ERIs) de dos electrones

$$[ab|cd] = \int \int \varphi_a(\mathbf{r_1})\varphi_b(\mathbf{r_1})r_{12}^{-1}\varphi_c(\mathbf{r_2})\varphi_d(\mathbf{r_2})d\mathbf{r_1}d\mathbf{r_2} \tag{3.1}$$

donde φ son funciones base monoelectrónicas y $\mathbf{r}$ las coordenadas electrónicas. Las funciones más utilizadas en la actualidad como funciones base son las de tipo Gaussiano (GTOs, por sus siglas en inglés), definidas por

$$\varphi_a(\mathbf{r}) = (x - A_x)^{l_a}(y - A_y)^{m_a}(z - A_z)^{n_a}e^{\alpha_A(\mathbf{r}-\mathbf{A})^2} \tag{3.2}$$

donde A_i representan la i-ésima coordenada del A-ésimo núcleo en el cual está centrada la función de base. Usualmente los orbitales se representan mediante una combinación lineal de funciones Gaussianas primitivas. La contracción de estas funciones Gaussianas primitivas permite evaluar ERIs contraídas a partir de ERIs primitivas, según

$$(ab|cd) = \sum_{\mu=1}^{K_a}\sum_{\nu=1}^{K_b}\sum_{\sigma=1}^{K_c}\sum_{\lambda=1}^{K_d} D_\mu D_\nu D_\sigma D_\lambda [a_\mu b_\nu|c_\sigma d_\lambda] \tag{3.3}$$

donde los corchetes se utilizan para denotar a las ERIs primitivas mientras que los paréntesis denotan a las ERIs contraídas. La propiedad principal de una función base y una ERI es el momento angular total $L_a = l_a + m_a + n_a$ y $L = L_a + L_b + L_c + L_d$, respectivamente.

Debido a que la evaluación de las ERIs de cuatro centros escala con el tamaño del sistema como $O(N^4)$, donde N es el número de funciones base, el código que evalúa las ERIs es de extrema importancia puesto que este paso se puede volver tardado y dominar el tiempo de cómputo del cálculo a medida que el sistema estudiado aumenta de tamaño.

3.1. Resolución de la Identidad en Hartree-Fock

Debido a la alta demanda computacional para el cálculo de las ERIs de cuatro centros se han desarrollado métodos aproximados para su evaluación. Una de estas aproximaciones es la de la Resolución de la identidad (RI)[5, 6, 7], ésta se lleva a cabo al introducir un conjunto de funciones base auxiliares además del conjunto de funciones de base originales para expresar las ERIs de cuatro centros mediante ERIs de tres centros. La aproximación de RI se puede resumir en la siguiente ecuación:

$$(ab|cd) \approx (ab|cd)_{RI} = \sum_{PQ} (ab|P) G^{-1}_{PQ} (Q|cd) \tag{3.4}$$

donde P y Q simbolizan funciones del conjunto base auxiliar, y G^{-1}_{PQ} denota el inverso de la matriz métrica de Coulomb, la cual está representada por la siguiente ecuación:

$$G_{PQ} = (P|Q) = \int \frac{P(\mathbf{r_1})Q(\mathbf{r_2})}{r_{12}} d\mathbf{r_1} d\mathbf{r_2} \tag{3.5}$$

La ecuación (3.4) descompone las ERIs de cuatro centros en integrales de dos y tres centros lo que a su vez permite la factorización de varias ecuaciones.

En el método de Hartree-Fock es posible utilizar la aproximación de RI en los términos de Coulomb e Intercambio que aparecen en las ecuaciones que definen la matriz de Fock. Al aplicar la aproximación de RI para el término de Coulomb (en sistemas de capa cerrada):

$$J_{ab} = \sum_{cd}^{N} (ab|cd) D_{cd} \tag{3.6}$$

se obtiene la siguiente factorización

$$J_{ab} \approx J^{RI}_{ab} = \sum_{PQ} (ab|P) G^{-1}_{PQ} \sum_{cd} (Q|cd) D_{cd} \tag{3.7}$$

donde, la matriz de Densidad (**D**)se calcula según

$$D_{cd} = 2 \sum_{i} \mathbf{C}_{ai} \mathbf{C}_{id} \tag{3.8}$$

donde i denota los orbitales ocupados y **C** los coeficientes de expansión de orbitales moleculares. La factorización obtenida representa una gran ventaja debido a que reduce el escalamiento formal de $O(N^4)$ a $O(N^3)$.[7] En el apéndice B se muestran también las ecuaciones necesarias para obtener la matriz de Intercambio.

Aunque la aproximación de RI sólo reduce el escalamiento para el cálculo de la matriz de Coulomb, en general simplifica el cálculo sin añadir complicaciones significativas ya que la evaluación de ERIs de tres centros es menos demandante computacionalmente que la evaluación de ERIs de cuatro centros. Además es posible optimizar la implementación para utilizar menos memoria RAM. Las cualidades anteriores hacen de la implementación de RI en GPU una alternativa eficiente a los métodos tradicionales de evaluación de integrales de cuatro centros en GPU.[8]

3.2. Algoritmo de implementación de la aproximación RI en HF

Utilizando la ecuación (3.7) es posible usar un algoritmo de cálculo para el término de Coulomb en el método de Hartree-Fock mediante la matriz G^{-1} o bien la matriz $G^{-1/2}$ tal como se describe en la literatura.[7] Sin embargo, la inversión de la matriz G es una operación computacionalmente demandante que puede volverse un cuello de botella en la implementación. Se ha reportado que la factorización modificada de Cholesky[9, 10] es una alternativa más rápida que no sacrifica precisión. Por lo tanto en esta implementación se utiliza un algoritmo que hace uso de la factorización modificada de Cholesky para la evaluación de la matriz de Coulomb. En la **tabla 3.2** se presenta el algoritmo para la evaluación del término de Coulomb, adicionalmente en el **apéndice B** puede consultarse el algoritmo para la evaluación de la matriz de intercambio.

Algoritmo para calcular J^{RI}	
Procedimiento	Escalamiento
1. Evaluar G	N_{aux}^2
2. Cholesky LDL^T de G	N_{aux}^3
3. $x_Q = \sum_{\sigma\lambda}(Q\|\sigma\lambda)P_{\sigma\lambda}$	$N_{aux}N^2$
4. $G_{QP}x'_p = x_Q$	N_{aux}^2
5. $J_{\mu\nu} = \sum_P(\mu\nu\|P)x'_p$	$N_{aux}N^2$

Tabla 3.1: Procedimiento para la evaluación de la matriz de Coulomb en Hartree-Fock mediante RI. Los pasos en azul se realizan una vez, al iniciar el SCF y los pasos en negro se repiten en cada iteración del ciclo.

3.3. Factorización Modificada de Cholesky

La implementación de RI involucra la inversión de la matriz métrica de Coulomb para la aproximación de las ERIs de cuatro centros.[5, 7, 3] Tradicionalmente se utilizan los métodos de Descomposición en Valores Singulares (SVD, por sus siglas en inglés) o Descomposición en Valores Propios (ED, por sus siglas en inglés) para llevar a cabo la inversión de la matriz métrica de Coulomb mediante las siguientes operaciones $\mathbf{G}^{-1} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^{T}$ o $\mathbf{G}^{-1} = \mathbf{Q}^{T}\mathbf{\Lambda}^{-1}\mathbf{Q}$ respectivamente. Sin embargo, tanto el método de SVD como ED tienden a ser un cuello de botella cuando el tamaño del sistema es suficientemente grande.

Una alternativa, aplicable a matrices métricas de Coulomb positivo definidas, que es al menos un orden de magnitud más rápida que los métodos de SVD y ED es la descomposición de Cholesky, pero ésta técnica es propensa a fallar en casos donde, debido a errores de redondeo, la matriz de métrica de Coulomb posea valores propios cercanos a cero o bien que sean cero. En estos casos la descomposición de Cholesky tradicional no sería adecuada, por lo tanto en este trabajo se decidió utilizar la descomposición modificada de Cholesky (ModChol) la cual permite asegurar la propiedad de que la matriz métrica G sea siempre positivo definida. El uso de ModChol ya ha sido estudiado e implementado con éxito en otros programas de estructura electrónica.[10, 11]

La técnica de ModChol básicamente consiste en calcular una perturbación simétrica $\mathbf{E}$ tal que al sumarla a la matriz métrica $\mathbf{G}$ se cumpla que $\mathbf{G} + \mathbf{E}$ sea suficientemente positivo definida y bien condicionada. El algoritmo utilizado en la implementación de ModChol en el trabajo presente corresponde al reportado por Cheng y Higham.[9] Dado un parámetro de tolerancia $\delta \geq 0$ se lleva a cabo el cálculo de una matriz de permutación $\mathbf{P}$, una matriz unitaria triangular inferior $\mathbf{L}$ y una matriz de bloques diagonales $\hat{\mathbf{D}}$ cuyos bloques diagonales son de dimensión 1 o 2 tal que

$$\mathbf{P}^{T}(\mathbf{G} + \mathbf{E})\mathbf{P} = \mathbf{L}\hat{\mathbf{D}}\mathbf{L}^{T}, \tag{3.9}$$

para asegurar que $\mathbf{G} + \mathbf{E}$ será positivo definida y bien condicionada el algoritmo realiza una inspección de la matriz $\hat{\mathbf{D}}$ para asegurar que sus valores propios sean positivo definidos. El **algoritmo 1** presenta una descripción del proceso llevado a cabo.

Algoritmo 1: Descomposición Modificada de Cholesky

Calcular la factorización simétrica indefinida $P^T GP = LDL^T$ utilizando la estrategia de pivote BBK[12] con la librería MAGMA.[13];
para *cada bloque diagonal D de $\hat{D}$* **hacer**
  si *D es un bloque 1×1* **entonces**
    si $D < \delta$ **entonces**
      $D = \delta$
    fin
  fin
  si *D es un bloque 2×2* **entonces**
    lleva a cabo su descomposición en valores propios $D = Q^T \Delta Q$
    donde $\Delta = diag(\lambda_1, \lambda_2)$.
    si $\lambda_1 < \delta$ **entonces**
      $\lambda_1 = \delta$
    fin
    si $\lambda_2 < \delta$ **entonces**
      $\lambda_2 = \delta$
    fin
    Construye $D = Q^T \Delta Q$, donde $\Delta = diag(\lambda_1, \lambda_2)$.
  fin
fin

3.4. Evaluación de las ERIs de tres centros

En 1986 Obara y Saika publicaron una expresión que permite evaluar ERIs de cuatro centros de cualquier momento angular al reducir los términos a integrales auxiliares de angularidad s.[14, 15] Actualmente se le conoce como Relación de Recurrencia Vertical (VRR) la cual se presenta a continuación:

$$\begin{aligned}
[(a+1_i)b|cd]^{(m)} =& (P_i - A_i)[ab|cd]^{(m)} + (W_i - P_i)[ab|cd]^{(m+1)} \\
&+ \frac{a_i}{2\zeta}\left([(a-1_i)b|cd]^{(m)} - \frac{\eta}{\zeta+\eta}[(a-1_i)b|cd]^{(m+1)}\right) \\
&+ \frac{b_i}{2\zeta}\left([a(b-1_i)|cd]^{(m)} - \frac{\eta}{\zeta+\eta}[a(b-1_i)|cd]^{(m+1)}\right) \\
&+ \frac{c_i}{2(\zeta+\eta)}[ab|(c-1_i)d]^{(m+1)} + \frac{d_i}{2(\zeta+\eta)}[ab|c(d-1_i)]^{(m+1)}
\end{aligned} \tag{3.10}$$

donde i corresponde a la coordenada x, y ó z, y

$$1_i = (\delta_{ix}, \delta_{iy}, \delta_{iz}), \tag{3.11}$$

$$\zeta = \alpha + \beta, \tag{3.12}$$

$$P_i = \frac{\alpha A_i + \beta B_i}{\alpha + \beta}. \tag{3.13}$$

η y Q_i son los análogos de ζ y P_i, construidos de φ_c y φ_d en vez de φ_a y φ_b. A partir de η y Q_i se construye W_i,

$$W_i = \frac{\zeta P_i + \eta Q_i}{\zeta + \eta}. \tag{3.14}$$

El significado del superíndice m es que las integrales con $m = 0$ son ERIs verdaderas de la forma de la ecuación (3.2), mientras que las integrales con $m > 0$ son integrales auxiliares. Éstas son necesarias para obtener ERIs verdaderas a través de la ecuación (3.10).

Es importante mencionar que las constantes que se encuentran en las **ecuaciones 3.11 - 3.14** son independientes del momento angular de las Gaussianas primitivas, por lo tanto el número de dichas constantes no se incrementa con el momento angular total de la ERI:

$$L = \sum_i^3 (a_i + b_i + c_i + d_i) \tag{3.15}$$

Los otros términos necesarios para aplicar la VRR son las integrales de tipo s, que se calculan según

$$[ss|ss]^{(m)} = (\zeta + \eta)^{-\frac{1}{2}} K_{AB} K_{CD} F_m(T), \tag{3.16}$$

donde

$$T = \frac{\zeta\eta}{\zeta + \eta}(P - Q)^2, \tag{3.17}$$

$$F_m(T) = \int_0^1 t^{2m} exp(-Tt^2) dt, \tag{3.18}$$

$$K_{AB} = 2^{-\frac{1}{2}} \frac{\pi^{\frac{5}{4}}}{\alpha + \beta} exp[-\frac{\alpha\beta}{\alpha + \beta}(A - B)^2], \tag{3.19}$$

y K_{CD} se define de manera similar.

En 1988 Head-Gordon y Pople describieron la obtención y utilidad de otra relación de recurrencia a partir de la ecuación (3.10).[16] Para llegar a dicha relación de recurrencia se

sustrae la VRR para $[a(b+1_i)|cd]^{(m)}$:

$$
\begin{aligned}
[a(b+1_i)|cd]^{(m)} =&(P_i - B_i)[ab|cd]^{(m)} + (W_i - P_i)[ab|cd]^{(m+1)} \\
&+ \frac{a_i}{2\zeta}([(a-1_i)b|cd]^{(m)} - \frac{\eta}{\zeta+\eta}[(a-1_i)b|cd]^{(m+1)}) \\
&+ \frac{b_i}{2\zeta}([a(b-1_i)|cd]^{(m)} - \frac{\eta}{\zeta+\eta}[a(b-1_i|cd]^{(m+1)}) \\
&+ \frac{c_i}{2(\zeta+\eta)}[ab|(c-1_i)d]^{(m+1)} + \frac{d_i}{2(\zeta+\eta)}[ab|c(d-1_i)]^{(m+1)}
\end{aligned}
\tag{3.20}
$$

de la VRR para $[(a+1_i)b|cd]^{(m)}$, ecuación (3.10), la cual se encuentra en términos de las mismas integrales. Esto da como resultado

$$
[a(b+1_i)|cd]^{(m)} = [(a+1_i)b|cd]^{(m)} + (A_i - B_i)[ab|cd]^{(m)} \tag{3.21}
$$

que expresa una ERI dada en términos de otra ERI del mismo momento angular total, pero con una unidad de momento angular desplazada del segundo al primer centro más una segunda ERI de menor momento angular. A la ecuación (3.21) se le llama Relación de Recurrencia Horizontal (HRR), con la finalidad de resaltar el uso que se le da para cambiar el momento angular de la posición 2 a 1 (o 4 a 3).

Una propiedad importante de la HRR, ecuación (3.21), es que la constante en el segundo término involucra solamente los centros de las funciones base. Como resultado la HRR puede ser aplicada a integrales primitivas o contraídas y se expresa como

$$
(a(b+1_i)|cd) = ((a+1_i)b|cd) + (A_i - B_i)(ab|cd). \tag{3.22}
$$

En esta ecuación los mismos coeficientes de contracción deben ser utilizados para los pares $(a, a+1_i)$ y $(b, b+1_i)$ y para c y d en ambos lados.

Para obtener la VRR para ERIs de tres centros tan solo es necesario considerar que en la ecuación (3.10) el último centro tiene exponente 0, lo cual da origen a la siguiente expresión,

$$
\begin{aligned}
[(a+1_i)b|c]^{(m)} =&(P_i - A_i)[ab|c]^{(m)} + (U_i - P_i)[ab|c]^{(m+1)} \\
&+ \frac{a_i}{2\zeta}([(a-1_i)b|c]^{(m)} - \frac{\gamma}{\zeta+\gamma}[(a-1_i)b|c]^{(m+1)}) \\
&+ \frac{b_i}{2\zeta}([a(b-1_i)|c]^{(m)} - \frac{\gamma}{\zeta+\gamma}[a(b-1_i)|c]^{(m+1)}) \\
&+ \frac{c_i}{2(\zeta+\gamma)}[ab|(c-1_i)]^{(m+1)}.
\end{aligned}
\tag{3.23}
$$

De manera semejante a W, el centro U se define como

$$U_i = \frac{\zeta P_i + \gamma C_i}{\zeta + \gamma} \tag{3.24}$$

y las ERIs se reducen a integrales de tipo $[ss|s]^m$

$$[ss|s]^m = \frac{2\pi^{5/2} K_{AB} F_m(T)}{\zeta\gamma\sqrt{\zeta + \gamma}}. \tag{3.25}$$

A partir de la HRR, ecuación (3.21), también es posible obtener una expresión para ERIs de tres centros, la única diferencia es que ésta solo puede aplicarse en el bra,

$$[a(b + 1_i)|c] = [(a + 1_i)b|c] + (A_i - B_i)[ab|c]. \tag{3.26}$$

3.5. Unidades de procesamiento gráfico

La superioridad en la capacidad de cómputo de operaciones de punto flotante que poseen las GPUs sobre las CPUs se debe a que las GPUs están especializadas en realizar cálculos con un alto grado de paralelismo (justamente de lo que se trata el procesamiento de gráficos) y por lo tanto están diseñadas de manera que un mayor número de transistores se encuentran destinados al procesamiento de datos en vez de ser utilizados para control de flujo de los programas o el manejo de memoria *caché*.

Las CPUs destinan gran parte de su arquitectura a sectores de reserva de memoria caché para minimizar el efecto de la latencia de los accesos a la memoria RAM y al control de flujo para buscar en memoria las instrucciones que serán ejecutadas por las unidades de proceso. La GPU enmascara la latencia de los accesos a la memoria mediante el procesamiento de datos, es decir la GPU realiza accesos a la memoria caché y de control de instrucciones al mismo tiempo que se encuentra procesando datos y esto le permite destinar menos sectores de hardware a dichos procesos, siempre y cuando tenga una gran cantidad de datos a procesar que le permite realizar un enmascaramiento efectivo.[17]

El modelo de programación de paralelismo de datos liga la información a ser procesada con las unidades de procesamiento, que en el caso de las GPUs se llaman *hilos*. Las GPUs logran esto mediante el Modelo de Instrucción Singular con Múltiples Hilos (SIMT, por

sus siglas en inglés). Lo anterior significa que todos los hilos de la GPU realizan la misma instrucción sobre datos diferentes. El modelo de programación paralela de CUDA posee una jerarquía de hilos, donde cada unidad de procesamiento se encuentra en un bloque de hilos que a su vez se encuentra en una malla de bloques de una, dos o tres dimensiones. Esto permite realizar los cálculos aún cuando los datos a procesar superan el número de unidades de procesamiento y también permite ajustar el mallado para que se amolde adecuadamente al problema a evaluar.[17]

Además de la jerarquía de hilos, el modelo de programación paralela de CUDA cuenta con una jerarquía de memoria.[17] Los hilos de CUDA son capaces de acceder a diferentes espacios de memoria durante su ejecución. Cada hilo posee un espacio de memoria local que es privada y sólo puede ser leída o escrita por dicho hilo, a su vez cada bloque de hilos posee una memoria compartida a la cual todos los hilos del bloque tienen acceso y cuyo tiempo de vida es igual al del bloque de hilos, por último todos los hilos tienen acceso a la misma memoria global. Cabe destacar que cada uno de estos tres tipos de memoria poseen latencias distintas, es decir el tiempo que tarda un hilo en acceder a los diferentes espacios de memoria y regresar al procesamiento de datos es diferente. La latencia de memoria disminuye de la siguiente manera: global >compartida >local. Es por ello que es muy importante optimizar el código escrito en CUDA para hacer buen uso de la jerarquía de memoria y evitar cuellos de botella debidos a la latencia del acceso a memoria.

Debido a que la GPU es un dispositivo físicamente independiente del CPU y la memoria RAM de la GPU es también independiente de la memoria RAM de la CPU, el código serial de C++ se ejecuta solamente en la CPU y el código paralelo escrito en CUDA se ejecuta en la GPU mediante llamadas de C++ a kernels escritos especialmente para la GPU, por la misma razón los datos a ser procesados por los hilos de la GPU deben ser copiados desde la memoria RAM de la CPU a la memoria RAM de la GPU. Debido a esto los programas escritos con C++ y CUDA se ejecutan de la siguiente manera, la cual está esquematizada en la **figura 3.1**.

1. El programa reserva memoria global en la GPU para recibir los datos que serán computados.

2. Se realiza el copiado de datos de la RAM del CPU hacia la memoria global de la GPU.

3. Toda vez que la información a procesar se encuentra en la memoria global de la GPU se realiza el llamado al *kernel* a ejecutar en la GPU mediante la sintaxis especial de CUDA y es necesario especificar las dimensiones del *grid* de bloques de hilos que es necesario desplegar.

4. Por último, ya que el cómputo se realizó con éxito en la GPU y se sincroniza la escritura de los hilos a la memoria global (es decir, se asegura que todos los hilos han terminado

su trabajo y escribieron en la memoria global). Los datos son copiados de regreso a la memoria RAM del CPU para continuar con la ejecución del programa.

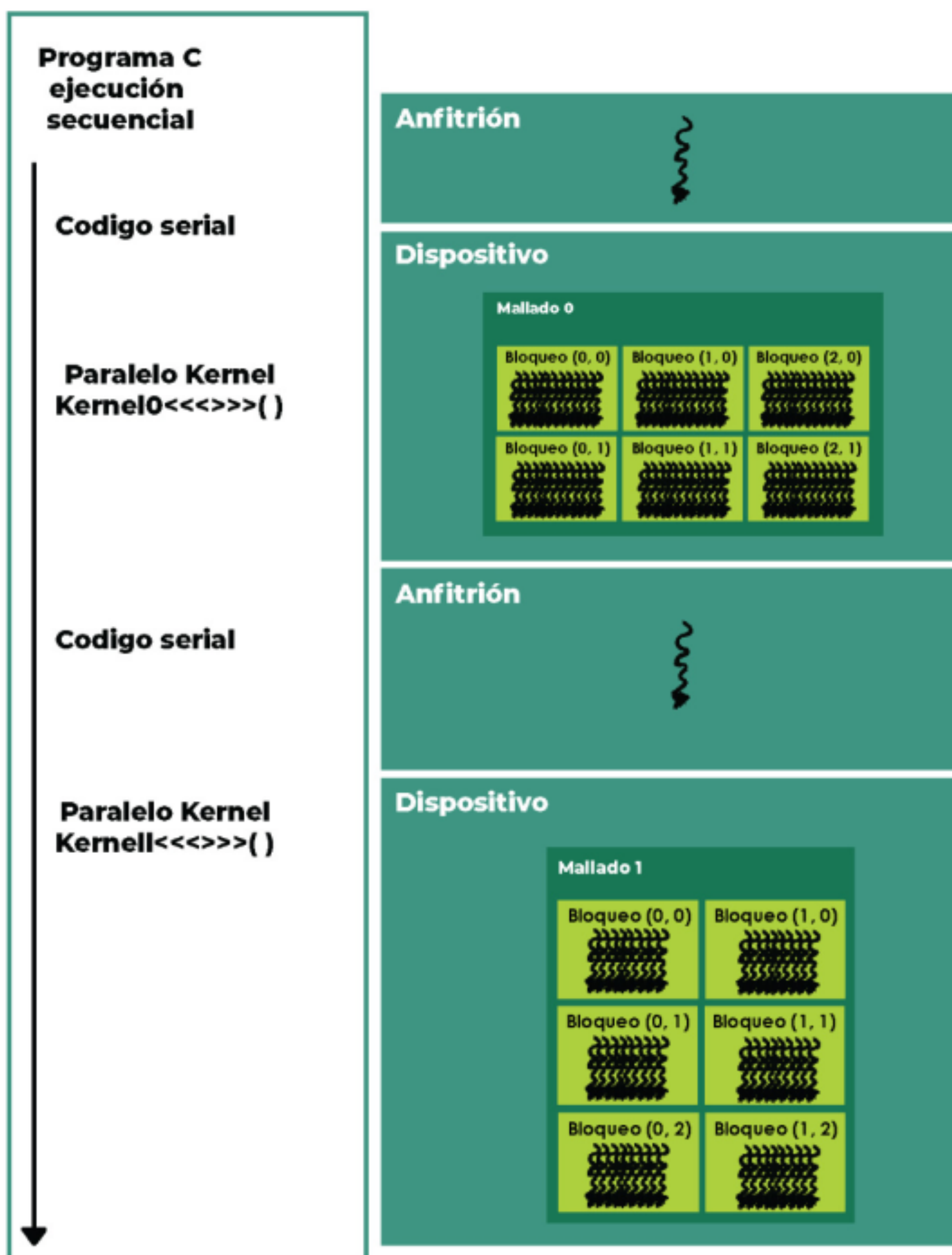


Figura 3.1: Esquema de ejecución de un programa de C++/CUDA para el uso de GPU. En un inicio la CPU tiene el control de la ejecución del programa, esto se realiza de manera serial, hasta que un llamado a un kernel CUDA se realiza y el control de la ejecución se cede momentáneamente a la GPU.

CAPÍTULO 4

Objetivos e hipótesis

4.1. Objetivo general

El objetivo general de este trabajo es la creación de un código modular escrito en CUDA que sea capaz de calcular ERIs de tres centros y llevar a cabo la evaluación de la matriz de Coulomb mediante RI así como su contribución a la matriz de Fock, en el contexto del método de Hartree-Fock, aprovechando el poder de cómputo que provee una GPU. El código mencionado deberá establecer las bases para el desarrollo de la librería GALLETA.

4.2. Objetivos particulares

Los objetivos particulares derivados de lo anterior son:

- Creación de kernels modulares escritos en CUDA para la evaluación de ERIs de tres centros de distintas angularidades en la GPU.
- Creación de una rutina que incorpore los kernels del punto anterior para llevar a cabo la evaluación de la matriz de Coulomb mediante la aproximación de RI en GPU.
- Implementación de la rutina desarrollada en el punto anterior en un programa de estructura electrónica de código libre.
- Validar el método al comparar los resultados de cálculos de energía para una serie de alcanos lineales y moléculas de interés (Benceno, Ribavirina, Clorobenceno y Fenol), con el programa original y el modificado.

4.3. Hipótesis

Si se crean rutinas independientes para la evaluación de ERIs de tres centros de momento angular específico en GPU, será posible desarrollar, con éstas, una rutina modular de GPU para la evaluación de la matriz de Coulomb mediante la aproximación de RI y su contribución a la matriz de Fock. La independencia de las rutinas que evalúan las ERIs de tres centros permitirán obtener un balance de trabajo en la GPU. Lo anterior, aunado al bajo consumo de memoria RAM de la aproximación de RI y el poder de cómputo que poseen las GPUs, permitirá una implementación eficaz, rápida y sencilla del código creado en cualquier programa de estructura electrónica.

CAPÍTULO 5

Metodología

5.1. Hardware y software utilizados

Para el desarrollo de este trabajo se tuvo a disposición el uso de un servidor para cómputo científico, administrado por el grupo de trabajo del Dr. Jorge Martín del Campo Ramírez, en el Departamento de Física y Química Teórica de la Facultad de Química, el servidor posee dos GPUs NVIDIA Tesla K20c con las siguientes especificaciones: arquitectura Kepler, 2496 núcleos de procesamiento, velocidad de reloj de 706 MHz y 5 GB de memoria RAM GDDR5 con un ancho de banda de 208 GB/s. También 16 CPUs Intel Xeon E5-2680 con velocidad de reloj de 2.70 GHz. Y 40 GB de memoria RAM.

Por otro lado, el software utilizado fue el siguiente: compilador del lenguaje CUDA nvcc versión 10.1, compilador del lenguaje C++ g++ versión 6.3.1, Python 3.8, la librería de álgebra lineal para arquitecturas de cómputo heterogéneas MAGMA versión 2.5.1 y por último el programa de estructura electrónica de código abierto Green.128.[18]

5.2. Evaluación de ERIs primitivas de tres centros en GPU

Las rutinas para evaluar las ERIs primitivas de tres centros se crearon utilizando las relaciones de recurrencia obtenidas por Obara-Saika[14, 15] y Head-Gordon-Pople.[16] El requisito más importante que deben cumplir estas rutinas es poseer carácter modular e independiente. Es decir, debe existir una y sólo una rutina para evaluar un tipo específico de integral $[ab|c]$ dada por sus momento angulares l_a, l_b y l_c, y cada rutina debe poder ser utilizada sin la necesidad de invocar llamados a las otras rutinas.

Estas características son de suma importancia ya que sin ellas no sería posible crear una rutina para la evaluación de la matriz de Coulomb que permita hacer uso de los núcleos de procesamiento, o hilos, de la GPU de una manera eficiente. La forma de evaluar las ERIs

primitivas de tres centros, sin disponer de kernels independientes, es crear una función general que evalúe todas las ERIs al mismo tiempo. Sin embargo, las operaciones requeridas para la evaluación de las ERIs varía con la momento angular de los centros que forman parte de la ERI. Por ejemplo, una ERI con centros a, b y c de momento angular d siempre requerirán más operaciones, para su evaluación, que una ERI donde los tres centros tienen momento angular s. La diferencia en la cantidad de operaciones ocasionaría que varios hilos en la GPU terminen su trabajo mientras otros aún se mantienen activos y esto a su vez desencadenaría en una ejecución por parte de la GPU.

Uno de los problemas más grandes de una implementación modular e independiente es que mientras mayor momento angular poseen las ERIs, la dificultad de su programación manual también aumenta considerablemente. Para poder sortear el problema anterior se diseñó un generador automático de código[19] capaz de crear código en un lenguaje de programación arbitrario de manera automática. Se utilizó el lenguaje de programación Python 3.8 para la creación del generador automático de código, de manera que recibe solamente las momentos angulares de los centros del tipo de ERI a evaluar y regresa una rutina escrita en CUDA capaz de evaluar dicho tipo de ERI. Los pasos que sigue el generador automático de código se basan en el esquema de evaluación de Head-Gordon-Pople[16] aplicado a AO-ERIs (ERIs en orbital armónico) de tres centros y son los siguientes:

1. Se aplica la HRR de la ecuación (3.26) al segundo centro hasta volverlo una función s.
2. Se aplica la VRR de la ecuación (3.23) al primer y tercer centro, en ese orden, hasta volverlos funciones s.
3. Se genera un diagrama de árbol que contiene todas las ERIs auxiliares a evaluar en la rutina.
4. Se realiza una búsqueda, desde la parte exterior del diagrama hacia la parte interior, para localizar los términos repetidos con la finalidad de evitar su evaluación múltiple.
5. Una vez que se estableció la jerarquía de evaluación de los términos no repetidos se procede a evaluarlos de manera explícita.
6. Finalmente se obtiene el valor numérico para la ERI a partir de los valores de las ERIs auxiliares que conforman el diagrama de árbol.

En la **figura 5.1** se ejemplifica el diagrama de árbol para el kernel evaluador del bloque de ERIs de momento angular [ps|p].

Los kernels de CUDA generados evalúan las ERIs en orbitales atómicos haciendo uso de un arreglo o mallado tridimensional de hilos en la GPU. Donde cada hilo posee tres índices x, y

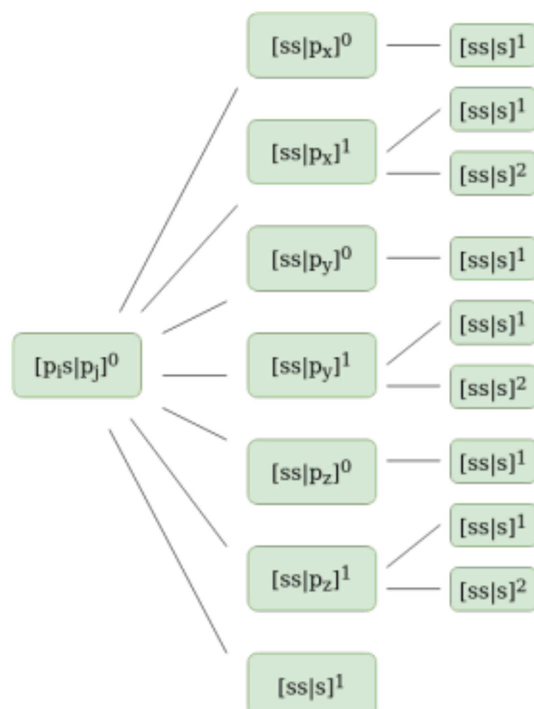


Figura 5.1: Diagrama de árbol de las ERIs auxiliares involucradas en el kernel para la evaluación de ERIs [ps|p], las ERIs auxiliares son evaluadas desde la parte exterior hacia la parte interior del diagrama.

y z que lo caracterizan y se relacionan directamente con los centros a b y c que deberá evaluar. Para que los kernels de CUDA generados funcionen adecuadamente es imprescindible que reciban subconjuntos del conjunto base con los coeficientes y exponentes correspondientes a las momento angulares de los centros del bloque de ERIs a evaluar.

5.3. Implementación en Green.128

La metodología para la obtención de la matriz de Coulomb mediante GPU se llevó a cabo siguiendo el algoritmo de evaluación propuesto por F. Weigend[7, 20], el cual cuenta con cinco pasos:

1. La obtención de la matriz *métrica de Coulomb*.
2. Descomposición LDL^T de Cholesky para evitar el costoso cálculo de la matriz G^{-1}.
3. Evaluación del vector de ajuste x_Q.
4. Obtención de x'_Q.
5. Evaluación de la matriz de Coulomb.

Para la implementación del código creado en este trabajo se dispuso del programa de estructura electrónica Green.128[18], el cual está escrito en C++, las modificaciones realizadas a Green.128 se llevaron a cabo para las partes del código que se encargan de llevar a cabo el SCF dentro del método de Hartree-Fock. Para establecer un puente entre las rutinas creadas en CUDA y Green.128 se hizo una nueva rutina en lenguaje C++ capaz de evaluar la matriz de Coulomb y realizar su contribución a la matriz de Fock.

La rutina descrita recibe directamente de Green.128 los siguientes parámetros: las coordenadas de los centros atómicos de la molécula, los exponentes y coeficientes que forman el conjunto de funciones base utilizado, la matriz métrica de Coulomb descrita por la **ecuación 3.5**, y una serie de arreglos de punteros que sirven para delimitar las coordenadas por centro y las funciones base por momento angular y átomo al que corresponden. Con estos datos la rutina se encarga de llamar a los kernels de CUDA necesarios para llevar a cabo la evaluación de la matriz de Coulomb y posteriormente sumar su contribución a la matriz de Fock.

En la **figura 5.2** se presenta un diagrama de ejecución de la rutina descrita. El primer bloque de instrucciones que se lleva a cabo es la inicialización de los datos. En este paso los coeficientes y exponentes de las funciones base son separadas en subconjuntos de acuerdo a las momento angulares s, p y d, además se crea un arreglo para guardar al vector x_Q y se inicializa en ceros. Una vez que los datos de entrada se han procesado se reserva memoria RAM en la GPU y se copian dichos datos desde la RAM de CPU hacia la GPU.

El segundo bloque de instrucciones se encarga de evaluar las ERIs primitivas y las convierte a ERIs contraídas mediante un kernel de contracción en la GPU que aprovecha un mallado de hilos tridimensional para mapear cada hilo con una ERI contraída. Finalmente se genera el vector x_Q mediante otro kernel CUDA que utiliza un arreglo de hilos de una sola dimensión para mapear cada hilo con un elemento de x_Q. Una vez que se obtiene el vector de ajuste los arreglos que contienen las ERIs primitivas, la matriz de densidad P y los datos de las funciones base dejan de ser necesarios y es pertinente borrarlos para liberar memoria RAM en la GPU.

Posteriormente si la rutina fue invocada en el primer ciclo del SCF es necesario llevar a cabo la descomposición LDL^T de G. Para llevar a cabo la descomposición se utilizó la librería de álgebra lineal para arquitecturas de cómputo heterogéneas *MAGMA*[13] que ya posee las rutinas necesarias para implementarla. Se utilizó una modificación a Cholesky para asegurar la estabilidad numérica de la matriz G, como se reporta en el trabajo de Higham.[9, 21] Y posteriormente se obtiene x'_Q independientemente de si la rutina fue invocada en el primer ciclo del SCF o no.

En el último bloque de instrucciones se realizan llamados, de manera serial, al kernel CUDA que se encarga de evaluar la matriz de Coulomb y realizar su contribución a la matriz de Fock. Dicho kernel es general para cualquier bloque de ERIs contraídas $(ab|c)$ y aprovecha

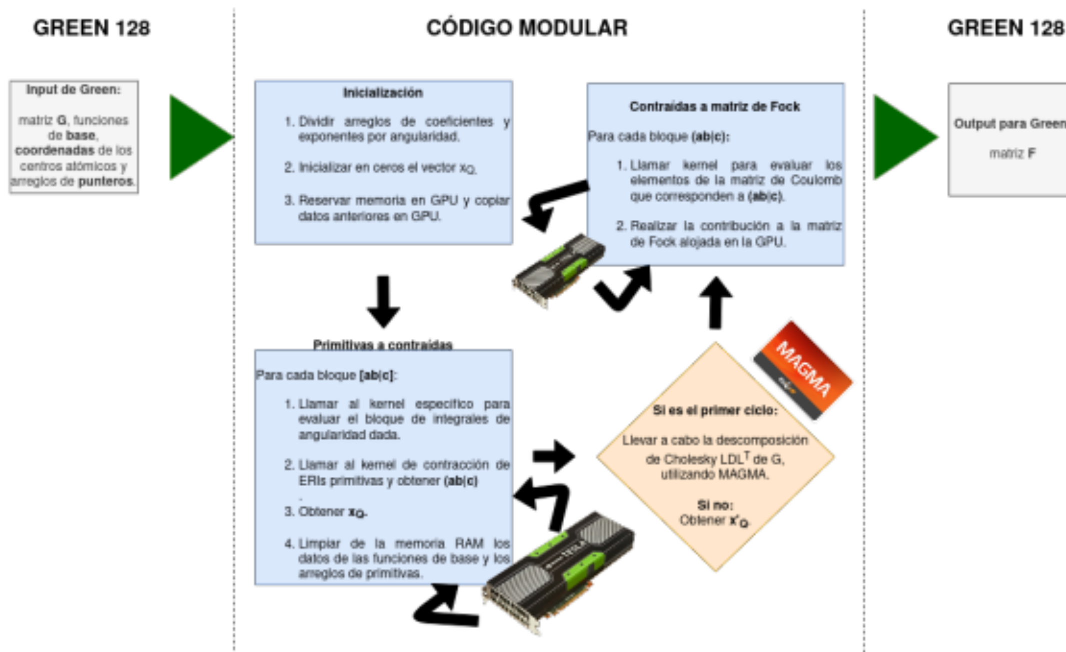


Figura 5.2: Diagrama de ejecución de la rutina escrita en C++ para la evaluación de la matriz de Coulomb a partir de los kernels de CUDA, y su posterior contribución a la matriz de Fock.

un mallado de hilos bidimensional para mapear cada hilo con un elemento de la matriz de Fock. Finalmente la matriz de Fock que reside en la memoria de la GPU es copiada hacía la matriz de Fock de Green.128 y la memoria reservada en GPU se limpia por completo.

Si bien la dificultad de escribir la rutina en C++ para la evaluación de Coulomb y su contribución a la matriz de Fock no era grande, el trabajo que se requería era repetitivo por lo cual se crearon otros programas de generación de código automática para obtener el código requerido para los bloques de instrucciones mencionados con anterioridad.

5.4. Validación del método

Para llevar a cabo la validación del método se realizaron cálculos de energía para un conjunto de moléculas arbitrario. Dicho conjunto conforma los primeros 18 alcanos lineales desde metano hasta octadecano, benceno, fenol, clorobenceno y la molécula de ribavirina que es un medicamento antiviral recetado tradicionalmente para el tratamiento de hepatitis c en pacientes con VIH, sin embargo en tiempos recientes se ha probado como antiviral en el tratamiento de la infección de COVID-19. En la **figura 5.3** se muestran los modelos 2D y 3D de esta molécula.

Todos los cálculos se realizaron utilizando el método de Hartree-Fock, la base cc-pVDZ y la aproximación RI. Para poder realizar la evaluación del método se utilizaron tres compilacio-

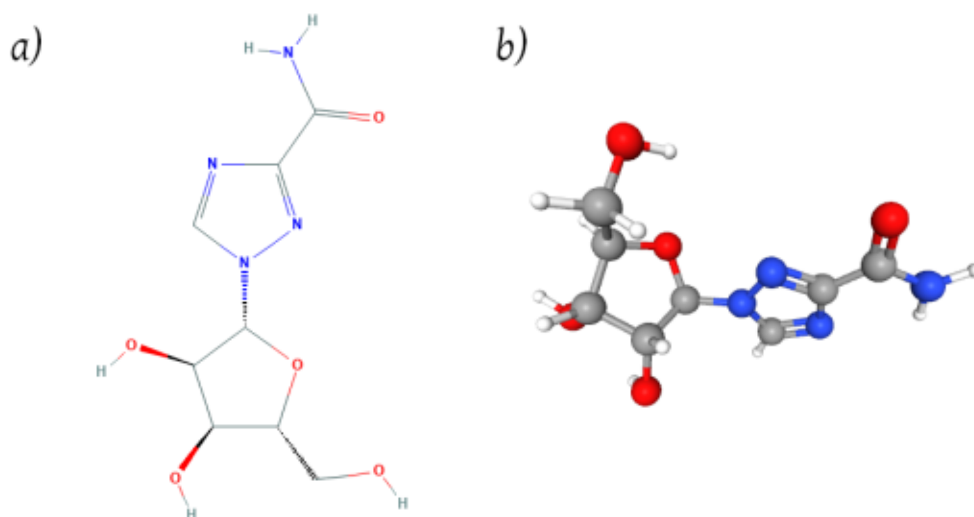


Figura 5.3: Molécula de Ribavirina $C_8H_{12}N_40_5$ a) Modelo 2D, b) Modelo 3D.

nes diferentes de Green.128. La primer compilación es el programa original cuya ejecución es serial en un solo CPU, el programa modificado para utilizar GPU en la evaluación de la matriz de Coulomb es la segunda compilación y por último se utilizó una versión de Green.128, desarrollada por el asesor técnico de esta tesis, cuya ejecución es paralela en varios CPUs.

A partir de los resultados obtenido se realizaron comparaciones, entre las tres diferentes compilaciones de Green.128, para los valores de energía obtenidos para cada molécula así como los pasos del SCF requeridos para la convergencia y el tiempo de ejecución de la rutina de evaluación de la matriz de Coulomb.

CAPÍTULO 6

Análisis de resultados

En la **tabla 6.1** se muestran los números de funciones base normales y funciones de base auxiliares para cada una de las moléculas calculadas. Debido a que el número de funciones base auxiliares es mayor al número de funciones base normales e inspeccionando la **tabla 3.2** se puede notar que el paso más computacionalmente demandante de todo el algoritmo es la descomposición de Cholesky de la matriz métrica de Coulomb. Por ello la rutina es eficiente en el sentido en que sólo realiza dicha descomposición una vez en el primer ciclo del SCF y guarda el resultado en memoria para su uso posterior en los ciclos restantes.

6.1. Green.128 GPU vs. Green.128 original

La implementación del código modular en GPU para la evaluación de la matriz de Coulomb se realizó directamente sobre una versión específica del programa Green.128, en la metodología nos referimos a ésta como la compilación original de Green.128 en lo futuro también haremos referencia a esta versión como Green.128 original. Para referirnos a la compilación modificada con el código modular para la evaluación de la matriz de Coulomb en GPU utilizaremos de ahora en adelante el término Green.128 GPU.

Los resultados obtenidos por Green.128 original son la comparación inmediata para los resultados obtenidos por Green.128 GPU. Los resultados más importantes a comparar son los ciclos del SCF realizados, la energía de las moléculas y el tiempo de ejecución de las rutinas para la evaluación de la matriz de Coulomb.

En la **tabla 6.2** se presentan resultados para los ciclos del SCF necesarios para llegar a la convergencia, la energía en Hartrees de las moléculas y los errores relativos y absolutos de la versión Green.128 GPU con respecto a la versión Green.128 original. Es apreciable que los valores de energía obtenidos por la versión de GPU no son exactamente los mismos a los obtenidos mediante la versión original. Esto es una consecuencia inherente de utilizar GPUs

Número de Funciones Base		
Molécula	**NBF**	**NAUX**
C_1H_4	42	60
C_2H_6	74	102
C_3H_8	106	144
C_4H_{10}	138	186
C_5H_{12}	170	228
C_6H_{14}	202	270
C_7H_{16}	234	312
C_8H_{18}	266	354
C_9H_{20}	298	396
$C_{10}H_{22}$	330	438
$C_{11}H_{24}$	362	480
$C_{12}H_{26}$	394	522
$C_{13}H_{28}$	426	564
$C_{14}H_{30}$	458	606
$C_{15}H_{32}$	490	648
$C_{16}H_{34}$	522	690
$C_{17}H_{36}$	554	732
$C_{18}H_{38}$	586	774
Benceno	162	198
Fenol	184	222
Clorobenceno	207	219
Ribavirina	434	516

Tabla 6.1: Número de funciones base normales y auxiliares, debidas a la base cc-pVDZ, para cada molécula

ya que las unidades de cálculo de éstas no poseen la misma precisión numérica que una CPU. los ciclos de SCF necesarios para llegar a la convergencia son los mismos excepto para el caso de $C_{10}H_{22}$ donde fue necesario un ciclo extra del SCF para alcanzar la convergencia. Lo anterior se debe a la estricta tolerancia que tiene el programa para evaluar la convergencia del SCF, la tolerancia usada en Green.128 es de 1×10^{-6} valor que para el caso del lenguaje de programación *C++* es el valor máximo de precisión.

Para una mejor apreciación de la diferencia en el valor de la energía entre ambas versiones del código se calculó el error relativo y el absoluto tomando a los valores de energía de la versión original como el valor de referencia, en la **tabla 6.3** se presentan estos resultados. En color verde se observan los errores mínimos. Para el caso del error absoluto el error mínimo se obtuvo en el cálculo de metano que tuvo la menor cantidad de ciclos del SCF y también el menor número de funciones base. Lo anterior es relevante debido a que el uso de GPU para operaciones de punto flotante conlleva una incertidumbre asociada debido a que el hardware no está tan especializado en la precisión numérica como un CPU. Por lo tanto al

Comparación entre Green.128 GPU vs. Green.128 Original						
	Green.128 Original			Green.128 GPU		
Molécula	**# Ciclos**	**Energía (Hartrees)**	**$t_{Coulomb}$ (s)**	**# Ciclos**	**Energía (Hartrees)**	**$t_{Coulomb}$ (s)**
C_1H_4	8	-40.198583	0.19	8	-40.198545	2.23
C_2H_6	8	-79.233771	0.90	8	-79.233624	2.68
C_3H_8	9	-118.270436	2.43	9	-118.270136	2.7
C_4H_{10}	11	-157.306811	5.02	11	-157.306340	2.72
C_5H_{12}	11	-196.343087	8.78	11	-196.342417	2.78
C_6H_{14}	12	-235.379364	13.90	12	-235.378468	2.79
C_7H_{16}	12	-274.415612	21.21	12	-274.414478	3.23
C_8H_{18}	17	-313.451872	28.70	17	-313.450484	3.39
C_9H_{20}	18	-352.488131	38.30	18	-352.486473	3.74
$C_{10}H_{22}$	14	-391.524365	49.80	15	-391.522426	4.25
$C_{11}H_{24}$	13	-430.560641	62.98	13	-430.558409	4.58
$C_{12}H_{26}$	13	-469.596873	78.15	13	-469.594337	5.54
$C_{13}H_{28}$	12	-508.633138	95.46	12	-508.630288	6.01
$C_{14}H_{30}$	13	-547.669368	129.76	13	-547.666193	6.48
$C_{15}H_{32}$	13	-586.705627	136.02	13	-586.702120	7.70
$C_{16}H_{34}$	14	-625.741854	168.87	14	-625.738006	8.42
$C_{17}H_{36}$	14	-664.778137	195.90	14	-664.773941	9.48
$C_{18}H_{38}$	15	-703.814347	224.06	15	-703.809795	10.87
Benceno	9	-230.712792	7.40	9	-230.712430	2.48
Fenol	14	-305.569330	10.16	14	-305.568868	2.55
Clorobenceno	13	-689.618307	10.97	13	-689.617760	2.79
Ribavirina	17	-902.012610	87.28	17	-902.010821	5.8

Tabla 6.2: Comparación entre los ciclos de SCF necesarios para llegar a la convergencia, la energía obtenida y el tiempo de ejecución de la rutina de evaluación de la matriz de Coulomb RI. Los valores de energía para Green.128 GPU indican con color azul los decimales hasta los que el resultado sigue siendo el mismo que para Green.128 original.

utilizar un menor número de veces la GPU para operaciones de punto flotante hace que dicha incertidumbre se propague en menor medida. Los errores máximos en octadecano pueden explicarse de la misma manera. Por último es importante notar que dado los resultados del error absoluto se puede apreciar que la versión de Green.128 con GPU desestima, en general, el valor de la energía.

Para una mejor perspectiva se realizó la conversión de los errores máximos y mínimos (absoluto y relativo) de Hartrees a kcal/mol. El error absoluto máximo es de -2.85 kcal/mol y el error relativo máximo es de 4.05×10^{-3} kcal/mol. Por otro lado el error absoluto mínimo es de 0.02 kcal/mol y el error relativo mínimo es de 4.97×10^{-4} kcal/mol. El valor de la precisión química deseada, encontrado en la literatura[22, 23], es de 1 kcal/mol. De las comparaciones previas podemos concluir que mientras menor es el tamaño del sistema el error asociado al uso de la versión GPU es menor al de la precisión química. Sin embargo conforme el tamaño del sistema el error absoluto supera al valor de la precisión química. Aun cuando el error máximo cometido se encuentra en el mismo orden de magnitud para sistemas más grandes este podría cobrar mayor importancia. Por otro lado que el error absoluto sea siempre negativo quiere

decir que el valor de la energía es subestimado de manera sistemática. Lo cual sugiere que es reparable para obtener valores más precisos y mejorar la calidad del algoritmo diseñado. Si por otra parte el error está asociado a la GPU utilizada, entonces una GPU más reciente con mejor tecnología mejoraría los resultados obtenidos.

Para justificar el uso de GPU, aún cuando brinda una incertidumbre numérica mayor, es necesario comparar los tiempos de ejecución para la rutina de evaluación de la matriz de Coulomb en ambas versiones del código, en la **tabla 6.4** se presentan los resultados para la aceleración de la versión de Green.128 en GPU contra la versión original. El cálculo de la aceleración obtenida se realizó con la siguiente fórmula:

$$Aceleración = \frac{t_{Original}}{t_{GPU}} \tag{6.1}$$

donde $t_{Original}$ y t_{GPU} son el tiempo de evaluación de la matriz de Coulomb en el la versión del código original y la versión con GPU respectivamente.

El valor máximo obtenido para la aceleración de la versión de Green.128 GPU con respecto a la versión original para la rutina de evaluación de la matriz de Coulomb es de $20.66x$. Es decir la rutina de la versión de GPU es casi 21 veces más rápida que la de la versión original. Además de la **tabla 6.4** podemos apoyarnos de la **figura 6.1** para concluir que dicha aceleración no es la máxima obtenible.Como es posible apreciar en la gráfica la curva que representa el tiempo de ejecución de la versión del código con GPU aún no presenta un comportamiento asintótico lo que sugiera que con una GPU con mayor capacidad de almacenamiento sería posible calcular sistemas de mayor tamaño y la aceleración, en teoría podría ser mayor a $\approx 21x$.

6.2. Green.128 GPU vs. Green.128 múltiples CPUs

Como se mencionó en la metodología, los resultados del código desarrollado en el presente trabajo (Green.128 versión GPU) se compararon también con una versión de Green.128 cuya implementación permite su ejecución en varias CPUs de manera simultánea, en lo posterior nos referiremos a esta versión como *Green.128 múltiples CPUs*. Dicha versión fue desarrollada por el asesor técnico de esta Tesis; el Maestro en Ciencias Juan Felipe Huan Lew Yee.

Para habilitar la ejecución paralela, en la versión Green.128 múltiples CPUs, se utilizó la librería *OpenMP* la cual es una Interfaz de Programación Aplicación (API) que permite la manipulación simultánea de diferentes procesos o *hilos*, a nivel del sistema operativo, que hacen posible la ejecución de una sección de código en varias CPUs al mismo tiempo. Además de esto la versión hace uso de la librería *LIBRETA*[2] para la evaluación de las ERIs. La cual se puede decir que es la mejor opción de software de acceso libre disponible actualmente para

Error absoluto y relativo vs. Green.128 original		
Molécula	**Error absoluto (Hartrees)**	**Error relativo (Hartrees)**
C_1H_4	-3.80E-05	9.45E-07
C_2H_6	-1.47E-04	1.86E-06
C_3H_8	-3.00E-04	2.54E-06
C_4H_{10}	-4.71E-04	2.99E-06
C_5H_{12}	-6.70E-04	3.41E-06
C_6H_{14}	-8.96E-04	3.81E-06
C_7H_{16}	-1.13E-03	4.13E-06
C_8H_{18}	-1.39E-03	4.43E-06
C_9H_{20}	-1.66E-03	4.70E-06
$C_{10}H_{22}$	-1.94E-03	4.95E-06
$C_{11}H_{24}$	-2.23E-03	5.18E-06
$C_{12}H_{26}$	-2.54E-03	5.40E-06
$C_{13}H_{28}$	-2.85E-03	5.60E-06
$C_{14}H_{30}$	-3.17E-03	5.80E-06
$C_{15}H_{32}$	-3.51E-03	5.98E-06
$C_{16}H_{34}$	-3.85E-03	6.15E-06
$C_{17}H_{36}$	-4.20E-03	6.31E-06
$C_{18}H_{38}$	-4.55E-03	6.47E-06
Benceno	-3.62E-04	1.57E-06
Fenol	-4.62E-04	1.51E-06
Clorobenceno	-5.47E-04	7.93E-07
Ribavirina	-1.79E-03	1.98E-06

Tabla 6.3: Valores de error absoluto y relativo para cada molécula calculada con la versión Green.128 con GPU tomando como referencia los valores obtenidos por el Green.128 en su versión original. En verde se presentan los errores mínimos y en rojo los errores máximos.

la evaluación de ERIs para códigos de estructura electrónica. Lo anterior hace de la versión Green.128 múltiples CPUs una muy referencia para evaluar no sólo el desempeño numérico de la versión Green.128 GPU sino especialmente su eficacia en cuanto a la velocidad de ejecución respecta.

Tal y como se describe en el capítulo de la metodología los cálculos de la versión múltiples CPUs fueron realizados utilizando la misma base (cc-pVDZ) y con 6 procesadores de manera paralela. El número de funciones base y funciones base auxiliares son exactamente los mismos que para las versiones original y GPU y se muestran en la **tabla 6.1**. En la **tabla 6.5** se presenta una comparación entre la versión Green.128 GPU desarrollada en este trabajo y la versión múltiples CPUs.

Realizando una inspección entra las **tablas 6.2** y **6.5** se puede observar que los resultados para la energía en la versión original y la de múltiples CPUs son prácticamente los mismos (salvo para $C_{18}H_{38}$ donde hay una diferencia de $1x10^{-6}$ Hartrees). La observación anterior se

Aceleración en la evaluación de la matriz de Coulomb vs. Green.128 original	
Molécula	**Aceleración**
C_1H_4	0.09x
C_2H_6	0.34x
C_3H_8	0.90x
C_4H_{10}	1.85x
C_5H_{12}	3.16x
C_6H_{14}	4.98x
C_7H_{16}	6.57x
C_8H_{18}	8.47x
C_9H_{20}	10.24x
$C_{10}H_{22}$	11.72x
$C_{11}H_{24}$	13.75x
$C_{12}H_{26}$	14.11x
$C_{13}H_{28}$	15.88x
$C_{14}H_{30}$	20.02x
$C_{15}H_{32}$	17.66x
$C_{16}H_{34}$	20.06x
$C_{17}H_{36}$	20.66x
$C_{18}H_{38}$	20.61x
Benceno	2.98x
Fenol	3.98x
Clorobenceno	3.93x
Ribavirina	15.05x

Tabla 6.4: Valores de aceleración obtenida por la versión de Green.128 con GPU tomando como referencia la versión original.

explica ya que la integración de las ERIs en la versión original de Green.128 se lleva a cabo mediante las relaciones de recurrencia de Obara y Saika[15] y Head-Gordon y Pople.[16] Y en la versión de múltiples CPUs la integración de las ERIs se realiza mediante la librería *LIBRETA*, la cual cuenta con un algoritmo que a tiempo de vuelo evalúa el método de evaluación de ERIs más eficaz de entre tres opciones[2]: la relaciones de recurrencia horizontal de Obara y Saika en conjunto con la relación de recurrencia vertical de Head-Gordon y Pople , el método de McMurchie-Davidson y la cuadratura de Dupuis-Rys-King. Sin embargo, tanto el método de McMurchie-Davidson como la cuadratura de Dupuis-Rys-King son eficientes para ERIs de angularidad y número de contracción elevados. Para los casos de estudio de este trabajo (funciones base con angularidad máxima d y números de contracción relativamente bajos) el método predominante para evaluación de ERIs en *LIBRETA* serán las relaciones de recurrencia de Obara y Saika y Head-Gordon y Pople, mismo método que es utilizado por la versión original de Green.128 lo que da como consecuencia los mismos resultados para la energía en ambas versiones de referencia.

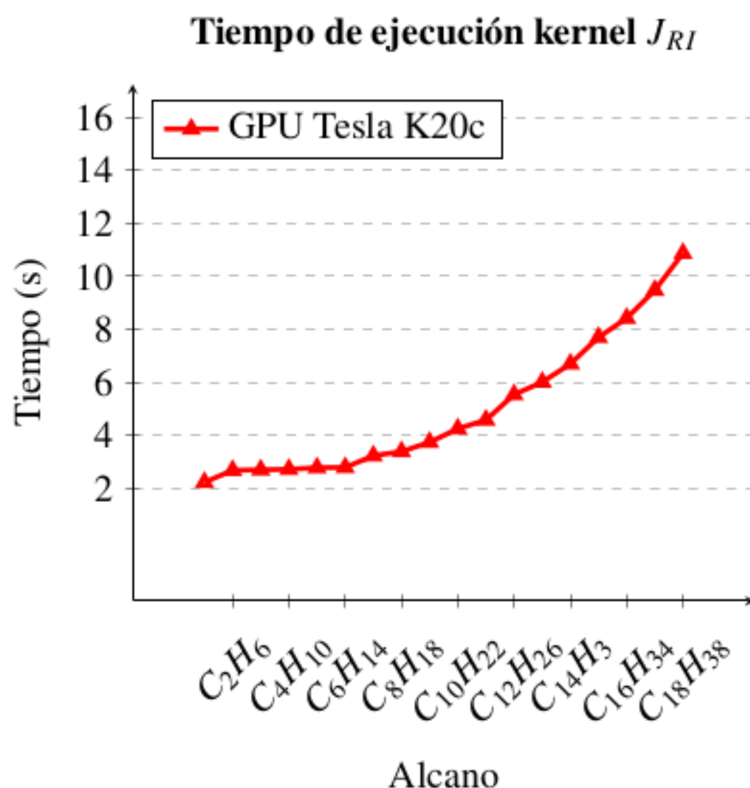


Figura 6.1: Tiempo de ejecución para la rutina de evaluación de la matriz de Coulomb con la versión GPU. Sólo fueron incluidos los resultados para los alcanos lineales pues es más sencillo observar la tendencia que si se agregan los resultados para las moléculas aromáticas y la Ribavirina.

Los errores absolutos y relativos entre la versión de GPU tomando como referencia la versión de múltiples CPUs son exactamente los mismos que los presentados en la **tabla 6.3** y por lo tanto el mismo análisis realizado en la sección anterior aplica para este caso. Sin embargo, debido al uso simultáneo de varios procesadores en la versión de múltiples CPUs los resultados para los tiempos de evaluación de la matriz de Coulomb y la aceleración de la versión GPU contra la versión múltiples CPUs serán diferentes. Estos resultados se presentan en la **tabla 6.6**. En la **figura 6.2** se muestra también las curvas del tiempo de ejecución del kernel J_{RI} para los diferentes alcanos con las tres versiones de Green.128.

Nuevamente la molécula para la cual se presenta una mayor aceleración es el octadecano ya que es el cálculo para el cual se involucra un mayor número de funciones base y funciones base auxiliares, y por lo tanto se requieren más operaciones de punto flotante para la evaluación de la matriz de Coulomb. Gracias a estos resultados es posible apreciar la gran ventaja que presenta el uso del código modular para evaluación de la matriz de Coulomb mediante GPU. Aún cuando la API *OpenMP* no es la mejor opción para paralelización de grano fino, lo cual

Comparación Green.128 GPU vs. Green.128 múltiples CPUs						
	Green.128 múltiples CPUs			Green.128 GPU		
Molécula	**# Ciclos**	**Energía (Hartrees)**	**t Coulomb (s)**	**# Ciclos**	**Energía (Hartrees)**	**t Coulomb (s)**
C_1H_4	8	-40.198583	0.10	8	-40.198545	2.23
C_2H_6	8	-79.233771	0.45	8	-79.233624	2.68
C_3H_8	9	-118.270436	1.25	9	-118.270136	2.70
C_4H_{10}	11	-157.306811	2.69	11	-157.306340	2.72
C_5H_{12}	11	-196.343087	4.99	11	-196.342417	2.78
C_6H_{14}	12	-235.379364	8.28	12	-235.378468	2.79
C_7H_{16}	12	-274.415612	12.80	12	-274.414478	3.23
C_8H_{18}	17	-313.451872	18.69	17	-313.450484	3.39
C_9H_{20}	18	-352.488131	26.24	18	-352.486473	3.74
$C_{10}H_{22}$	14	-391.524365	35.61	15	-391.522426	4.25
$C_{11}H_{24}$	13	-430.560641	47.06	13	-430.558409	4.58
$C_{12}H_{26}$	13	-469.596873	60.40	13	-469.594337	5.54
$C_{13}H_{28}$	12	-508.633138	76.13	12	-508.630288	6.01
$C_{14}H_{30}$	13	-547.669368	95.12	13	-547.666193	6.48
$C_{15}H_{32}$	13	-586.705627	115.72	13	-586.702120	7.70
$C_{16}H_{34}$	14	-625.741854	139.78	14	-625.738006	8.42
$C_{17}H_{36}$	14	-664.778137	167.54	14	-664.773941	9.48
$C_{18}H_{38}$	15	-703.814348	199.31	15	-703.809795	10.87
Benceno	9	-230.712792	3.45	9	-230.712430	2.48
Fenol	14	-305.569330	4.89	14	-305.568868	2.55
Clorobenceno	13	-689.618307	5.50	13	-689.617760	2.79
Ribavirina	17	-902.012610	62.4	17	-902.010821	5.80

Tabla 6.5: Comparación de los valores de energía, el número de ciclos del SCF y el tiempo de ejecución de la rutina de evaluación de la matriz de Coulomb. Para la versión de múltiples CPUs los cálculos siempre fueron realizados utilizando 6 CPUs de manera simultánea. En color azul se muestran los decimales hasta los cuales la energía de Green.128 es igual que para la versión de múltiples CPUs.

se puede apreciar en la baja diferencia en el tiempo de ejecución entre la versión original y la de múltiples CPUs, sí es una de las opciones más utilizadas para una paralelización rápida y con bajos requerimientos para su implementación. Contra 6 CPUs trabajando de manera simultánea el código desarrollado en el presente trabajo obtuvo una aceleración máxima de $18.34x$ utilizando únicamente una CPU.

En el trabajo de *Kalinowski et al*[3] se reportaron aceleraciones de hasta 30× contra implementaciones de software serial y paralelo en CPUs. Lo cual se encuentra dentro del orden de magnitud de los resultados obtenidos. Los valores para la aceleración obtenidos en el presente trabajo son mejorables tanto por la mejora de la memoria RAM de la GPU para cálculo de sistemas más grandes donde el paralelismo sea mejor aprovechado, como por más poder de cálculo haciendo uso de GPUs más recientes. Así mismo pueden utilizarse nuevas metodologías para obtención de ERIs de tres centros.[4] Por ejemplo, otro método como el de McMurchie-Davidson. En las cuales el poder de cómputo necesario para evaluar las ERIs contraídas sea menor al requerido en el presente trabajo.

Es posible realizar una comparación superficial sobre los beneficios de utilizar el código desarrollado en el presente trabajo mediante GPU. Para empezar podemos comparar los

Aceleración en la evaluación de la matriz de Coulomb vs. Green.128 múltiples CPUs	
Molécula	**Aceleración**
C_1H_4	0.04x
C_2H_6	0.17x
C_3H_8	0.46x
C_4H_{10}	0.99x
C_5H_{12}	1.79x
C_6H_{14}	2.97x
C_7H_{16}	3.96x
C_8H_{18}	5.51x
C_9H_{20}	7.02x
$C_{10}H_{22}$	8.38x
$C_{11}H_{24}$	10.28x
$C_{12}H_{26}$	10.90x
$C_{13}H_{28}$	12.67x
$C_{14}H_{30}$	14.68x
$C_{15}H_{32}$	15.03x
$C_{16}H_{34}$	16.60x
$C_{17}H_{36}$	17.67x
$C_{18}H_{38}$	18.34x
Benceno	1.39x
Fenol	1.92x
Clorobenceno	1.97x
Ribavirina	10.76x

Tabla 6.6: Valores de la aceleración en la evaluación de la matriz de Coulomb tomando como referencia la versión Green.128 múltiples CPUs. En verde se presenta la aceleración máxima obtenida.

precios actuales de los procesadores y la GPU utilizados en el desarrollo del presente trabajo: CPU Intel Xeon E5-2680 con un precio de $4,694.10 pesos mexicanos y la GPU Tesla K20c es de $18,349.56 pesos mexicanos (ambos precios son aproximados y fueron consultados en www.ebay.com). Ya que el máximo número de CPUs utilizadas en la comparación fue 6, el precio de 6 CPUs sería de $28,164.60, este precio es mayor al de la GPU Tesla K20c utilizada en este trabajo por lo que podemos decir que además de que el valor de comprar 6 CPUs es mayor al de comprar una GPU, en este trabajo, una GPU demostró tener un rendimiento 18 veces más rápido que 6 CPUs trabajando de manera paralela. Si además toma en cuenta el costo de las tarjetas madres necesarias para la correcta implementación de ambas arquitecturas y se le agrega el costo de una CPU al de la GPU debido a que la GPU no puede trabajar de manera independiente (Es la CPU quien le brinda instrucciones de control al GPU), el beneficio en tiempo es 18 veces mayor y tanto los costos de ventilación como el tamaño del servidor disminuyen para el caso de la arquitectura con GPU.

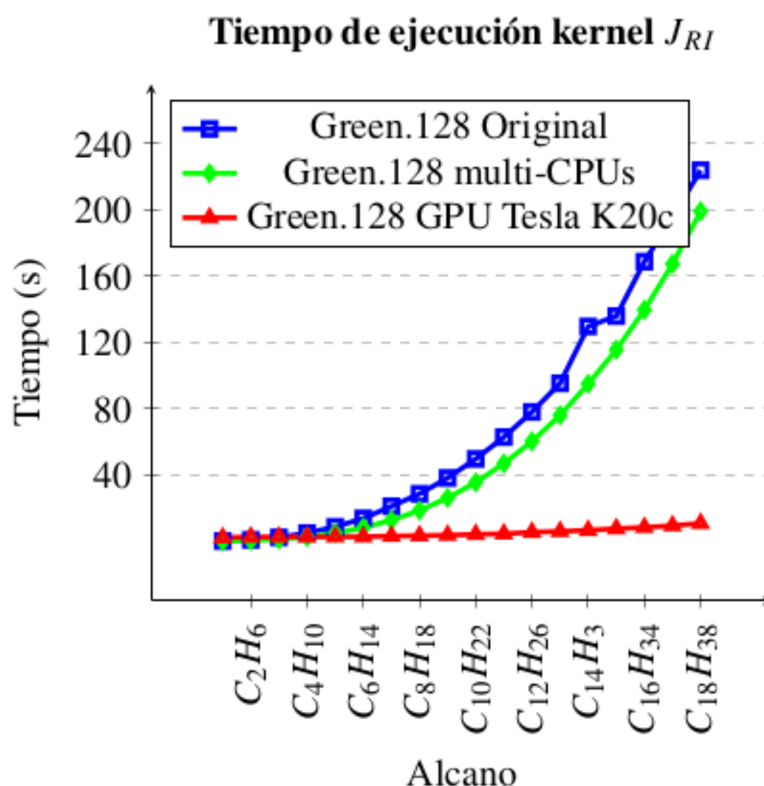


Figura 6.2: Tiempo de ejecución para la rutina de evaluación de la matriz de Coulomb con las tres versiones de Green.128. Sólo fueron incluidos los resultados para los alcanos lineales pues es más sencillo observar la tendencia que si se agregan los resultados para las moléculas aromáticas y la Ribavirina.

6.3. Perfil de Ejecución Green.128 versión GPU

Para la versión Green.128 GPU además de obtener el tiempo de ejecución total de la rutina de evaluación de J_{RI} también se realizó el perfil de ejecución de las partes de la rutina que se describen en la **figura 5.2**, los resultados se presentan en la **tabla 6.7**. Primero que nada es importante recalcar la estabilidad de la evaluación de la descomposición de Cholesky, aún cuando en la **tabla 3.2** se puede apreciar que éste es el paso con el mayor escalamiento en todo el algoritmo de evaluación de la matriz de Coulomb con RI. Los tiempos de ejecución se mantienen más o menos constantes a pesar de que el número de funciones base auxiliares aumentan para cada alcano sucesivo e incluso en el caso de las moléculas aromáticas y la Ribavirina. La eficacia de la descomposición de Cholesky se debe al uso de la librería *MAGMA* para su evaluación. La descomposición no se muestra como un porcentaje debido a que sólo se lleva a cabo una vez en el primer ciclo del SCF lo cual reduce el tiempo de ejecución de los ciclos restantes.

Perfil de ejecución de la rutina de evaluación de J_{RI} en Green.128 versión GPU					
Molécula	**Tiempo Total J_{RI}**	**Tiempo Cholesky**	**% Vector de ajuste x_Q**	**% x'_Q**	**% J a matriz de Fock**
C_1H_4	2.23	0.87	83.17	6.97	9.86
C_2H_6	2.68	1.20	75.46	14.60	9.94
C_3H_8	2.70	1.06	81.76	8.64	9.60
C_4H_{10}	2.72	0.81	80.80	10.73	8.47
C_5H_{12}	2.78	0.84	80.00	10.95	9.05
C_6H_{14}	2.79	0.86	86.00	4.84	9.16
C_7H_{16}	3.23	0.86	83.50	7.27	9.23
C_8H_{18}	3.39	0.90	86.26	4.71	9.03
C_9H_{20}	3.74	1.09	85.21	5.37	9.42
$C_{10}H_{22}$	4.25	0.93	87.20	3.48	9.32
$C_{11}H_{24}$	4.58	1.06	86.31	3.39	10.30
$C_{12}H_{26}$	5.54	0.97	86.36	4.51	9.13
$C_{13}H_{28}$	6.01	0.99	88.47	2.24	9.29
$C_{14}H_{30}$	3.14	1.27	81.14	5.42	13.44
$C_{15}H_{32}$	7.70	1.05	88.87	2.48	8.65
$C_{16}H_{34}$	8.42	1.24	89.79	1.68	8.53
$C_{17}H_{36}$	9.48	1.24	89.90	1.66	8.44
$C_{18}H_{38}$	10.87	1.12	90.60	1.33	8.07
Benceno	2.48	0.91	84.15	5.77	10.08
Fenol	2.55	0.93	84.26	6.01	9.73
Clorobenceno	2.79	0.90	84.70	6.07	9.23
Ribavirina	5.80	1.07	87.49	4.03	8.48

Tabla 6.7: Perfil de ejecución para la rutina de evaluación de la matriz de Fock en Green.128 versión GPU. En las primeras dos columnas se muestran los tiempos totales para la evaluación de J_{RI} y la factorización de Cholesky de la matriz G. Las últimas tres columnas muestran los porcentajes del tiempo necesario para la evaluación de las tres partes más importantes de la rutina diseñada para la implementación del código modular en Green.128.

Las últimas tres columnas de la **tabla 6.7** representan el porcentaje llevar a cabo los pasos 3, 4 y 5, respectivamente, del algoritmo para el cálculo de J_{RI} que se encuentra en la **tabla 3.2**. Como es de esperarse los pasos que mayor porcentaje contribuyen son la obtención del vector de ajuste y la contribución de las ERIs a la matriz de Fock debido a que ambos escalan como $N_{aux}N^2$. La razón por la cual el porcentaje de la generación del vector de ajuste es mayor es que este porcentaje involucra también la evaluación de las ERIs primitivas y su posterior contracción a ERIs contraídas. Los resultados para el porcentaje de la evaluación del vector de ajuste también nos permiten concluir que la evaluación de las integrales ya que a pesar de ser los pasos más demandantes los tiempos de ejecución para la rutina completa no van más allá de un par de decenas de segundos.

6.4. Generador de código para evaluación de ERIs

Gracias a los resultados anteriores es posible establecer la eficacia de las rutinas de evaluación de las ERIs primitivas mediante las relaciones de recurrencia de Obara Saika y

Head-Gordon Pople así como el generador de código de CUDA escrito en *python*. Gracias al generador de código es posible aumentar la angularidad de las integrales que sean evaluadas con la GPU. Lo anterior se logra al definir las angularidades de los centros de las ERIs en la rutina encargada de generar el código en CUDA para el kernel de evaluación de ERIs. Como un ejemplo en el **listado de código 6.1** se presenta el código en *python* requerido para generar el kernel de evaluación de las ERIs primitivas de tipo sss.

```
generate_kernel(0,0,0)
```

Listado de código 6.1: Código para generar el kernel para evaluación de las ERIS primitivas del tipo sss, los parámetros que recibe la función son simplemente la angularidad de cada centro.

En el **listado de código 6.2** se presenta el código de *CUDA/C++* creado mediante el generador de código de *Python* para evaluar las ERIs primitivas de tipo **sss**. En las líneas 2 a 7 se crea la declaración de la función junto con los parámetros que debe recibir en su llamado, tales como las coordenadas de los centros, los exponentes y coeficientes de las funciones gaussianas y la angularidad de cada centro. Posteriormente se realiza la declaración de las variables a utilizar para llevar a cabo el producto de gaussianas utilizado en las relaciones de recurrencia. De la línea 13 a la 19 se crea un arreglo que contiene una serie de prefactores utilizados para evaluar las funciones de Boys que aparezcan en las relaciones de recurrencia, este arreglo se escribe únicamente una vez en la memoria compartida de la GPU y es accesible para todos los hilos. En las líneas 22 a 27 se utilizan los índices de cada hilo para seleccionar la ERI del lote a resolver, así mismo se introduce código para que una vez que un hilo termine de trabajar busque una nueva ERI a resolver (en el caso de que el número de ERIs sea mayor al número de hilos desplegados). La parte que resta del código se encarga de realizar la contracción gaussiana del bra mediante el producto de gaussianas y por último se evalúan las funciones de Boys necesarias para obtener los intermediarios de las relaciones de recurrencia y así obtener las ERIs primitivas.

```

__global__ void sss(double *a, double *b, double *caux, double *exp_a,
                    double *exp_b, double *exp_caux, double *coef_a,
                    double *coef_b, double *coef_caux, int la, int lb,
                    int lcaux, int nprimis_a, int nprimis_b,
                    int nprimis_aux, double *primeri,
                    int *atomidx_a, int *atomidx_b, int *atomidxaux) {

  double Rab2, Rpcaux2, p, mulexpab, Kab, coef, t;
  double Rp[3], Rw[3];
```

```
11  int atidxa, atidxb, atidxaux;
12  double fis[1];
13  __shared__ double ftab[2*GALLETA_MAX_L_I+6+1][GALLETA_MAX_TAB_GAM+1];
14
15  if (threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0) {
16    // Initialize ftab
17    init_boysfunc(ftab);
18  }
19  __syncthreads();
20
21
22  for (int x=blockIdx.x * blockDim.x + threadIdx.x; x<nprimis_a; x+=
     blockDim.x * gridDim.x) {
23    for (int y=blockIdx.y * blockDim.y + threadIdx.y; y<nprimis_b; y+=
     blockDim.y * gridDim.y) {
24      for (int z=blockIdx.z * blockDim.z + threadIdx.z; z<nprimis_aux; z
     +=blockDim.z * gridDim.z) {
25        atidxa = atomidx_a[x];
26        atidxb = atomidx_b[y];
27        atidxaux = atomidxaux[z];
28
29
30        // Bra contraction by the Gaussian Product Theorem
31        Rab2 = pow(a[atidxa*3]-b[atidxb*3], 2) + pow(a[atidxa*3+1]-b[
     atidxb*3+1], 2) + pow(a[atidxa*3+2]-b[atidxb*3+2], 2);
32        p = exp_a[x] + exp_b[y];
33        mulexpab = exp_a[x] * exp_b[y];
34        Kab = exp(-(mulexpab/p)*Rab2);
35        Rp[0] = (exp_a[x]*a[atidxa*3] + exp_b[y]*b[atidxb*3]) / p;
36        Rp[1] = (exp_a[x]*a[atidxa*3+1] + exp_b[y]*b[atidxb*3+1]) / p;
37        Rp[2] = (exp_a[x]*a[atidxa*3+2] + exp_b[y]*b[atidxb*3+2]) / p;
38
39        // Get Rw
40        Rw[0] = ((p*Rp[0]) + (exp_caux[z]*caux[atidxaux*3])) / (p +
     exp_caux[z]);
41        Rw[1] = ((p*Rp[1]) + (exp_caux[z]*caux[atidxaux*3+1])) / (p +
     exp_caux[z]);
42        Rw[2] = ((p*Rp[2]) + (exp_caux[z]*caux[atidxaux*3+2])) / (p +
     exp_caux[z]);
43
44        coef = 2*pow(GALLETA_PI, 5.0/2.0) / (p*exp_caux[z]*pow(p+exp_caux
     [z],1.0/2.0));
45        Rpcaux2 = pow(Rp[0]-caux[atidxaux*3], 2) + pow(Rp[1]-caux[
     atidxaux*3+1], 2) + pow(Rp[2]-caux[atidxaux*3+2], 2);
46        t = ((p*exp_caux[z]) / (p+exp_caux[z])) * Rpcaux2;
```

```
47
48          // Get Boys function
49          boysfunc(0, t, fis, ftab);
50          primeri[(x*1+0)*1*nprimis_b*1*nprimis_aux+(y*1+0)*1*nprimis_aux+(
        z*1+0)] = coef * Kab * fis[0];
51        }
52      }
53    }
54  }
```

Listado de código 6.2: Kernel de *CUDA/C++* para la evaluación de ERIs sss creado a partir del generador de código en python.

Debido a la naturaleza modular del código creado, en el sentido de que existen varias rutinas para evaluar las integrales en bloques de angularidades específicas, crear las rutinas para implementar la aproximación de RI se vuelve una tarea de gran tamaño y repetitiva. Por esta razón se crearon más rutinas para generación de código en *Python*. La primera de estas rutinas tiene la finalidad de generar de manera automática los llamados a los kernels en *CUDA* necesarios para evaluar las ERIs primitivas, realizar su contracción a ERIs contraídas y por último obtener el vector de ajuste x_Q. En el **listado de código 6.3** se presenta el código generador en *Python* para crear los llamados a las rutinas hasta una angularidad de ERIs **ddd**. Es importante recalcar cómo después de generar las ERIs contraídas, LAS ERIs primitivas son borradas de la memoria RAM de la GPU con la finalidad de liberar espacio para la posterior reserva de memoria necesaria para llevar a cabo el paso número 5 del algoritmo de J_{RI}. En el **listado de código 6.4** se muestra una parte del código en *CUDA/C++* generado.

```
1
2  otxt = "" # this variable will hold the first output of the generator
3  ldict = {0: "s", 1: "p", 2: "d"} # angularity dictionary
4  adict = {0: "0", 1: "1", 2: "2"} # same as angularity dict but with
       integers
5  odict = {0: "1", 1: "3", 2: "6"} # orientations dict
6
7  for a in range(0, 3):
8      for b in range(0, 3):
9          for c in range(0, 3):
10             txtbuffer = ""
11             txtbuffer += "  // (" + ldict[a] + ldict[b] + "|" + ldict[c]
       + ") ---\n"
12
13             txtbuffer += "  // allocation of primitive eris\n"
14             txtbuffer += alloc_peri(ldict[a], ldict[b], ldict[c],
```

```
15                                      odict[a], odict[b], odict[c])
16
17              txtbuffer += "  // evaluation of primitive eris\n"
18              txtbuffer += eval_peri(ldict[a], ldict[b], ldict[c],
19                                     adict[a], adict[b], adict[c])
20
21              txtbuffer += "  // allocation of contracted eris\n"
22              txtbuffer += alloc_ceri(ldict[a], ldict[b], ldict[c],
23                                      odict[a], odict[b], odict[c])
24
25              txtbuffer += "  // evaluation of contracted eris\n"
26              txtbuffer += eval_ceri(ldict[a], ldict[b], ldict[c],
27                                     odict[a], odict[b], odict[c])
28
29              txtbuffer += "  // free primitive ERIs array from GPU\n"
30              txtbuffer += free_peri(ldict[a], ldict[b], ldict[c])
31
32              txtbuffer += "  // contribution of contracted ERIs to Fitting
    Tensor\n"
33              txtbuffer += get2fitK(ldict[a], ldict[b], ldict[c],
34                                    odict[a], odict[b], odict[c])
35
36              txtbuffer += "  // ---\n\n"
37              otxt += txtbuffer
38
39  print(otxt)
```

Listado de código 6.3: Código para generar los llamados a los kernels necesarios para llevar a cabo el paso 3 del algoritmo de J_{RI}.

```
1
2  // (sd|s) ---
3    // allocation of primitive eris
4    cudaMalloc((double**)&d_primeri_sds, n_s*n_d*n_s_aux*1*6*1*sizeof(
      double));
5    // evaluation of primitive eris
6    sds <<<grid,block>>> (d_r, d_r, d_rcaux, d_exps_s, d_exps_d,
      d_exps_aux_s,
7                          d_coefs_s, d_coefs_d, d_coefs_aux_s, 0, 2, 0,
8                          n_s, n_d, n_s_aux, d_primeri_sds, d_atomidx_s,
9                          d_atomidx_d, d_atomidxaux_s);
10   // allocation of contracted eris
11   cudaMalloc((double**)&d_contreri_sds, n_s_shells*n_d_shells*n_s_aux
      *1*6*1*sizeof(double));
12   // evaluation of contracted eris
```

```
13  primi2contr <<<grid,block>>> (d_ll_shellidxs_s, d_ul_shellidxs_s,
14                                d_ll_shellidxs_d, d_ul_shellidxs_d,
     d_primeri_sds,
15                                d_contreri_sds, d_coefs_s, d_coefs_d,
     n_s_shells, n_d_shells, n_d,
16                                n_s_aux, 1, 6, 1, d_ncsto_s, d_ncsto_d,
     d_ncsto_s_aux);
17  // free primitive ERIs array from GPU
18  cudaFree(d_primeri_sds);
19  // contribution of contracted ERIs to Fitting Tensor
20  contr2fittingK <<<grid4,block4>>> (n_s_shells, n_d_shells, n_s_aux,
21                                     HOMO, d_ll_ao_shell_s,
     d_ll_ao_shell_d, d_ll_ao_aux_shell_s,
22                                     d_contreri_sds, 1, 6, 1, nbas,
     d_Cvalues, d_FitTensor,
23                                     auxnco);
24  // ---
25
26  // (sd|p) ---
27  // allocation of primitive eris
28  cudaMalloc((double**)&d_primeri_sdp, n_s*n_d*n_p_aux*1*6*3*sizeof(
     double));
29  // evaluation of primitive eris
30  sdp <<<grid,block>>> (d_r, d_r, d_rcaux, d_exps_s, d_exps_d,
     d_exps_aux_p,
31                        d_coefs_s, d_coefs_d, d_coefs_aux_p, 0, 2, 1,
32                        n_s, n_d, n_p_aux, d_primeri_sdp, d_atomidx_s,
33                        d_atomidx_d, d_atomidxaux_p);
34  // allocation of contracted eris
35  cudaMalloc((double**)&d_contreri_sdp, n_s_shells*n_d_shells*n_p_aux
     *1*6*3*sizeof(double));
36  // evaluation of contracted eris
37  primi2contr <<<grid,block>>> (d_ll_shellidxs_s, d_ul_shellidxs_s,
38                                d_ll_shellidxs_d, d_ul_shellidxs_d,
     d_primeri_sdp,
39                                d_contreri_sdp, d_coefs_s, d_coefs_d,
     n_s_shells, n_d_shells, n_d,
40                                n_p_aux, 1, 6, 3, d_ncsto_s, d_ncsto_d,
     d_ncsto_p_aux);
41  // free primitive ERIs array from GPU
42  cudaFree(d_primeri_sdp);
43  // contribution of contracted ERIs to Fitting Tensor
44  contr2fittingK <<<grid4,block4>>> (n_s_shells, n_d_shells, n_p_aux,
45                                     HOMO, d_ll_ao_shell_s,
     d_ll_ao_shell_d, d_ll_ao_aux_shell_p,
```

```
46                                         d_contreri_sdp, 1, 6, 3, nbas,
    d_Cvalues, d_FitTensor,
47                                         auxnco);
48  // ---
```

Listado de código 6.4: Ejemplo de los llamados a los kernels *CUDA* requeridos para la obtención de las ERIs primitivas, contraídas y su contribución al vector x_Q.

La última rutina generadora de código se presenta en el **listado de código 6.5** y gracias a esta se construyen los llamados a los kernels en *CUDA* para realizar el paso 5 del algoritmo de J_{RI} que se encarga de sumar las contribuciones del vector x'_Q y las ERIs contraídas a la matriz de Coulomb. Dicha contribución se realiza directamente sobre la matriz de Fock. Nuevamente se presenta un ejemplo del código de *CUDA/C++* generado en el **listado de código 6.6**.

```
1
2  """ Portion of code to contribute to the Fock Matrix. """
3
4  otxt2 = "" # This variable will hold the last part of the outputed text
5
6  for a in range(0, 3):
7      for b in range(0, 3):
8          for c in range(0, 3):
9              txtbuffer = ""
10             txtbuffer += "  // (" + ldict[a] + ldict[b] + "|" + ldict[c]
    + ") ---\n"
11
12             txtbuffer += "  // contribution to Fock Matrix\n"
13             txtbuffer += get2fock(ldict[a], ldict[b], ldict[c],
14                                   odict[a], odict[b], odict[c])
15
16             txtbuffer += "  // free contracted ERIs array from GPUs
    memory\n"
17             txtbuffer += free_ceri(ldict[a], ldict[b], ldict[c])
18
19             txtbuffer += "  // ---\n\n"
20             otxt2 += txtbuffer
21
22 print(otxt2)
```

Listado de código 6.5: Código para generar los llamados a los kernels *CUDA* para realizar la contribución de la matriz de Coulomb a la matriz de Fock.

```
1
```

```
2 // (ss|d) ---
3   // contribution to Fock Matrix
4   contr2FockM <<<grid2,block2>>> (d_contreri_ssd, d_FockM, nshells,
                                    n_s_shells,
5                                    n_s_shells, n_d_aux, d_w_d, sf,
6                                    d_ll_ao_shell_s, d_ll_ao_shell_s, nbas,
      1, 1, 6);
7   // free contracted ERIs array from GPUs memory
8   cudaFree(d_contreri_ssd);
9   // ---
10
11  // (sp|s) ---
12  // contribution to Fock Matrix
13  contr2FockM <<<grid2,block2>>> (d_contreri_sps, d_FockM, nshells,
                                    n_s_shells,
14                                   n_p_shells, n_s_aux, d_w_s, sf,
15                                   d_ll_ao_shell_s, d_ll_ao_shell_p, nbas,
      1, 3, 1);
16  // free contracted ERIs array from GPUs memory
17  cudaFree(d_contreri_sps);
18  // ---
```

Listado de código 6.6: Ejemplo de los llamados a los kernels *CUDA* requeridos para la obtención de los elementos de la matriz de Coulomb y su contribución a la matriz de Fock.

A partir de los **listados de código 6.4 y 6.6** se puede ver que gracias a que las ERIs primitivas se eliminan de la memoria tan pronto como se generan el requerimiento de la memoria RAM, de la rutina en GPU para evaluar la matriz de Coulomb mediante la aproximación de RI, se reduce de manera drástica. La cantidad de memoria RAM máxima utilizada será únicamente la necesaria para almacenar las ERIs contraídas y la matriz de Fock. Para el caso de los resultados del trabajo presente fue posible manejar sin problema alguno todos los ciclos del SCF para la molécula $C_{18}H_{38}$ con tan sólo 5GB de memoria RAM disponible.

CAPÍTULO 7

Conclusiones y Perspectivas

En el trabajo se desarrollaron con éxito rutinas en CUDA para la evaluación de ERIs de tres centros mediante GPU, las rutinas desarrolladas presentan un carácter modular ya que se crearon diferentes rutinas para evaluar ERIs de angularidad específica. Esta propiedad de las rutinas permitió realizar una implementación eficaz de la aproximación de RI para la evaluación de la matriz de Coulomb mediante GPU en el código de estructura electrónica Green.128. Los resultados para los valores de energía obtenidos para el conjunto de moléculas estudiadas permiten concluir que la implementación es numéricamente robusta, por otro lado las aceleraciones obtenidas hacen evidente la superioridad en eficiencia de utilizar el código desarrollado. No sólo para evaluar ERIs de tres centros, sino también para paralelizar completamente la implementación de la aproximación de RI en el método de Hartree-Fock.

Las rutinas generadoras de código escritas en *Python* permiten la creación instantánea de nuevos kernels de *CUDA* para evaluar ERIs de tres centros de cualquier angularidad específica. Y las rutinas generadoras de los llamados a los kernels *CUDA* permiten crear rutinas para implementar la evaluación de la matriz de Coulomb mediante la aproximación de RI de manera rápida, además de que las rutinas creadas utilizan el mínimo de memoria RAM requerida gracias al cuidadoso diseño de éstas y la naturaleza modular de los kernels evaluadores de ERIs de tres centros.

Por todo lo anterior podemos concluir que se logró con éxito el desarrollo de un código modular para la evaluación de ERIs de tres centros y éste servirá de manera exitosa como la base de código para la librería *GALLETA*. La cual facilitará el uso de GPUs para la implementación de la aproximación de RI y cualquier otra metodología que requiera la evaluación de ERIs de tres centros. Como perspectivas del trabajo se pueden mencionar: ampliar la metodología desarrollada para llevar a cabo la evaluación de la matriz de Intercambio mediante la aproximación de RI, la introducción de diferentes técnicas para la evaluación de las ERIs tales como el método de McMurchie-Davidson o la cuadratura de Rys y por último es posible

explorar la posibilidad de aprovechar la naturaleza modular de los kernels *CUDA* para realizar rutinas que sean capaces de manipular más de una GPU al mismo tiempo y así disminuir aún más los tiempos de ejecución de la implementación de RI.

Apéndices

A. Teorema del producto de gaussianas

El teorema del producto de gaussianas estipula que para cualesquiera dos funciones gaussianas con momento angular arbitrario y con los centros A y B su producto será una nueva función gaussiana con centro en P. Lo anterior se puede demostrar al considerar dos funciones gaussianas

$$\begin{aligned}\varphi_k(\mathbf{r}) &= (x - A_x)^{a_x}(y - A_y)^{a_y}(z - A_z)^{a_z} e^{-\alpha_k(\mathbf{r}-\mathbf{A})^2},\\ \varphi_l(\mathbf{r}) &= (x - B_x)^{b_x}(y - B_y)^{b_y}(z - B_z)^{b_z} e^{-\alpha_l(\mathbf{r}-\mathbf{B})^2}.\end{aligned} \tag{A.1}$$

Al multiplicar las exponenciales de ambas funciones, se tiene que:

$$e^{-\alpha_k(\mathbf{r}-\mathbf{A})^2} e^{-\alpha_l(\mathbf{r}-\mathbf{B})^2} = e^{[-(\alpha_k+\alpha_l)\mathbf{r}\cdot\mathbf{r}+2(\alpha_k\mathbf{A}+\alpha_l\mathbf{B})\cdot\mathbf{r}-\alpha_k\mathbf{A}\cdot\mathbf{A}-\alpha_l\mathbf{B}\cdot\mathbf{B}]} \tag{A.2}$$

Y se puede definir el exponente de la nueva gaussiana como $\gamma = \alpha_k + \alpha_l$ y el centro de ésta se encuentra en el punto $\mathbf{P} = \frac{\alpha_k\mathbf{A}+\alpha_l\mathbf{B}}{\gamma}$. De tal manera que se cumple la siguiente igualdad:

$$e^{-\alpha_k\mathbf{r_A}^2-\alpha_l\mathbf{r_B}^2} = Ke^{[-\gamma(\mathbf{r}\cdot\mathbf{r}-\mathbf{r}\cdot\mathbf{P}+\mathbf{P}\cdot\mathbf{P})]} \tag{A.3}$$

Si tomamos el caso donde r sea un vector nulo y usando la sustitución de la ecuación (A.2) en la ecuación (A.3) obtenemos la siguiente expresión:

$$Ke^{-\gamma\mathbf{P}\cdot\mathbf{P}} = e^{\alpha_k\mathbf{A}\cdot\mathbf{A}-\alpha_l\mathbf{B}\cdot\mathbf{B}} \tag{A.4}$$

y despejando para K, obtenemos:

$$K = e^{\alpha_k\mathbf{A}\cdot\mathbf{A}-\alpha_l\mathbf{B}\cdot\mathbf{B}+\gamma\mathbf{P}\cdot\mathbf{P}} \tag{A.5}$$

Al expandir $\mathbf{P}\cdot\mathbf{P}$ en la ecuación (A.5) tenemos,

$$
\begin{aligned}
K &= exp[-\alpha_k \mathbf{A}\cdot\mathbf{A} - \alpha_l \mathbf{B}\cdot\mathbf{B} + \gamma^{-1}(\alpha_k^2 \mathbf{A}\cdot\mathbf{A} + 2\alpha_k\alpha_l \mathbf{A}\cdot\mathbf{B} + \alpha_l^2 \mathbf{B}\cdot\mathbf{B})] \\
&= exp[\gamma^{-1}(-\alpha_k^2 \mathbf{A}\cdot\mathbf{A} - \alpha_k\alpha_l \mathbf{A}\cdot\mathbf{A} - \alpha_k\alpha_l \mathbf{B}\cdot\mathbf{B} + \alpha_k^2 \mathbf{A}\cdot\mathbf{A} + 2\alpha_k\alpha_l \mathbf{A}\cdot\mathbf{B} + \alpha_l^2 \mathbf{B}\cdot\mathbf{B})] \\
&= exp[-\gamma^{-1}\alpha_k\alpha_l(\overline{\mathbf{AB}}^2)].
\end{aligned}
\tag{A.6}
$$

donde

$$
\overline{\mathbf{AB}}^2 = \mathbf{A} - \mathbf{B}. \tag{A.7}
$$

Por lo tanto, cualquier producto de funciones gaussianas tendrá el siguiente resultado:

$$
\begin{aligned}
\varphi_k(r)\varphi_l(r) =& (x - A_x)^{a_x}(y - A_y)^{a_y}(z - A_z)^{a_z} \\
& \times (x - B_x)^{b_x}(y - B_y)^{b_y}(z - B_z)^{b_z} \\
& \times e^{-\gamma^1 \alpha_k\alpha_l(\overline{\mathbf{AB}}^2)} e^{\gamma(\mathbf{r}-\mathbf{P})^2}.
\end{aligned}
\tag{A.8}
$$

B. Implementación del algoritmo K_{RI}

Se presenta la ecuación para obtener la matriz de intercambio mediante la aproximacion de RI.

$$K_{ab} = \sum_{cd}^{N} (ac|bd) D_{cd} \tag{B.9}$$

$$K_{ab} \approx K_{ab}^{RI} = \sum_{P}^{N_{aux}} \sum_{c}^{N} (ac|P)) \sum_{Q}^{N_{aux}} (G^{-1})_{PQ} \sum_{d}^{N} (Q|bd) D_{cd} \tag{B.10}$$

Esta expresión para el intercambio permite realizar optimizaciones sobre la implementación, pero el escalamiento se mantiene como $O(N^4)$.

En la **tabla B** se presenta el algoritmo para la evaluación de la matriz de intercambio del método de Hartree-Fock mediante la aproximación de RI.

Algoritmo para calcular K^{RI}	
Procedimiento	Escalamiento
1. Evaluar G	N_{aux}^2
2. Cholesky LDL^T de G	N_{aux}^3
3. $(x^i)_{Q\nu} = \sum_\lambda C_\lambda (P\|\nu\lambda)$	$N_{occ} N_{aux} N^2$
4. $G_{PQ}(x'^i)_{Q\nu} = (x^i)_{Q\nu}$	$N_{occ} N_{aux}^2 N$
5. $K_{\mu\nu} = \sum_i \sum Q (x^{iT})_{\mu Q} (x^i)_{Q\nu}$	$N_{occ} N_{aux} N^2$

Tabla B.1: Procedimiento para la evaluación de la matriz de Coulomb en Hartree-Fock mediante RI. Los pasos en azul se realizan una vez al iniciar el SCF y los pasos en negro se repiten en cada iteración del ciclo.

Bibliografía

[1] E. F. Valeev, "Libint: A library for the evaluation of molecular integrals of many-body operators over gaussian functions." http://libint.valeyev.net/, 2020. version 2.7.0-beta.4.

[2] J. Zhang, "LIBRETA: Computerized Optimization and Code Synthesis for Electron Repulsion Integral Evaluation," *Journal of Chemical Theory and Computation*, vol. 14, pp. 572–587, 2018.

[3] J. Kalinowski, F. Wennmohs, and F. Neese, "Arbitrary Angular Momentum Electron Repulsion Integrals with Graphical Processing Units: Application to the Resolution of Identity Hartree-Fock Method," *Journal of Chemical Theory and Computation*, vol. 13, pp. 3160–3170, 2017.

[4] J. Kussmann, H. Laqua, and C. Ochsenfeld, "Highly efficient resolution-of-identity density functional theory calculations on central and graphics processing units," *Journal of Chemical Theory and Computation*, vol. 17, no. 3, pp. 1512–1521, 2021. PMID: 33615784.

[5] O. Vahtras, J. Almlöf, and M. W. Feyerisen, "Integral approximations for LCAO-SCF calculations," *Chemical Physics Letters*, vol. 213, no. 5, 1993.

[6] M. Feyereisen, G. Fitzgerald, and A. Komomicki, "Use of approximate integrals in ab initio theory. an application in mp2 energy calculations," *Chemical Physics Letters*, vol. 208, no. 5, 6, pp. 359–363, 1993.

[7] F. Weigend, "A fully direct RI-HF algorithm: Implementation, optimised auxiliary basis sets, demonstration of accuracy and efficiency," *Physical Chemistry Chemical Physics*, vol. 4, pp. 4285–4291, 2002.

[8] K. Yasuda and H. Maruoka, "Efficient calculation of two-electron integrals for high angular basis functions," *International Journal of Quantum Chemistry*, vol. 114, no. 9, pp. 543–552, 2014.

[9] S. H. Cheng and N. J. Higham, "A modified Cholesky algorithm based on a symmetric indefinite factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 19, no. 4, pp. 1097–110, 2016.

[10] J. F. H. Lew-Yee, R. Flores-Moreno, J. L. Morales, and J. M. del Campo, "Asymmetric density fitting with modified Cholesky decomposition applied to second-order electron propagator," *Journal of Chemical Theory and Computation*, vol. 16, no. 3, pp. 1597–1605, 2020. PMID: 31967819.

[11] J. N. Pedroza-Montero, F. A. Delesma, J. L. Morales, P. Calaminici, and A. M. Köster, "Variational fitting of the fock exchange potential with modified Cholesky decomposition," *The Journal of Chemical Physics*, vol. 153, no. 13, p. 134112, 2020.

[12] G. H. Golub and V. L. C. F., *Matrix computations*. Johns Hopkins Univ Press, 2013.

[13] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, pp. 232–240, June 2010.

[14] S. Obara and A. Saika, "Efficient recursive computation of molecular integrals over Cartesian Gaussian functions," *Journal of Chemical Physics*, vol. 84, no. 3963, 1986.

[15] S. Obara and A. Saika, "General recurrence formulas for molecular integrals over Cartesian Gaussian functions," *Journal of Chemical Physics*, vol. 89, no. 1540, 1988.

[16] M. Head-Gordon and J. A. Pople, "A method for two-electron Gaussian integral and integral derivative evaluation using recurrence relations," *Journal of Chemical Physics*, vol. 89, no. 5777, 1988.

[17] "NVIDIA Developer Zone cuda toolkit documentation." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`. Consultado el 20 de noviembre de 2020.

[18] R. Flores-Moreno, A. Venegas-Reynoso, J. A. Guerrero, J. J. Villalobos, B. Zuñiga-Gutierrez, L. F. Morales, H. N. Gonzalez, J. Mojica, J. Valdez, A. Flores-Ramos, D. Gomez, and V. M. Medel, "Green.128," 2017.

[19] G. Mazur, M. Makowski, R. Łazarski, R. Włodarczyk, E. Czajkowska, and M. Glanowski, "Automatic code generation for quantum chemistry applications," *International Journal of Quantum Chemistry*, vol. 116, no. 18, pp. 1370–1381, 2016.

[20] F. Weigend, M. Kattanek, and R. Ahlrichs, "Approximated electron repulsion integrals: Cholesky decomposition versus resolution of the identity methods," *Journal of Chemical Physics*, vol. 130, no. 164106, 2009.

[21] N. J. Higham, "Cholesky factorization," *WIREs Computational Statistics*, vol. 1, no. 2, pp. 251–254, 2009.

[22] J. A. Pople, "Quantum chemical models (Nobel lecture)," *Angewandte Chemie International Edition*, vol. 38, no. 13-14, pp. 1894–1902, 1999.

[23] P.-F. Loos, N. Galland, and D. Jacquemin, "Theoretical 0–0 energies with chemical accuracy," *The Journal of Physical Chemistry Letters*, vol. 9, no. 16, pp. 4646–4651, 2018. PMID: 30063359.