UNIVERSIDAD NACIONAL DE CÓRDOBA

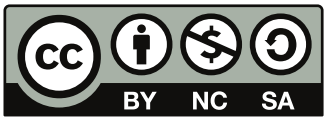
FACULTAD DE MATEMÁTICA, ASTRONOMIA, FÍSICA Y COMPUTACIÓN.

Optimización del cómputo para la resolución del problema de una y dos partículas en un pozo de potencial usando B-splines

Tesis realizada por Emanuel Emilio Lupi para la Licenciatura en Ciencias de la Computación en la Universidad Nacional de Córdoba

---

Dirigida por:
Doctor Nicolás Wolovick
Licenciado Mariano Garagiola



Esta obra está bajo una Licencia Creative Commons
Atribución-NoComercial-CompartirIgual 4.0 Internacional

Agradecimientos

- A mi familia, quienes siempre serán los primeros agradecidos en cualquier logro de mi vida por formarme como persona, por el amor y el apoyo incondicional que siempre me brindan.
- A mis directores Nicolás y Mariano por la calidez humana y la gran ayuda que siempre me prestaron a lo largo del proyecto.
- A mis amigos y compañeros.

Resumen

La ecuación de Schrödinger independiente del tiempo predice que las funciones de onda pueden tener la forma de ondas estacionarias, denominados estados estacionarios (también llamados "orbitales", como en los orbitales atómicos o los orbitales moleculares). A partir de los estados estacionarios es posible encontrar los estados dependientes del tiempo de un sistema físico. La ecuación de Schrödinger independiente del tiempo es la ecuación que describe los estados estacionarios.

En una gran variedad de situaciones físicas, por ejemplo cuando se quiere conocer la función de onda espacial, la ecuación de Schrödinger es una ecuación diferencial en derivadas parciales de segundo orden

$$E\psi(r) = \left[\frac{-\hbar^2}{2\mu}\nabla^2 + V(r)\right]\psi(r)$$

la cual, a su vez es una ecuación de autofunciones y autovalores, donde las autofunciones son los estados estacionarios de la ecuación y los autovalores son las energías asociadas a cada autofunción.

Las ecuaciones diferenciales de segundo orden son complejas de resolver ya que no existe una forma analítica para la solución de una ecuación arbitraria, es por eso que en la mayoría de los casos se buscan soluciones aproximadas. Uno de los métodos usados para resolver la ecuación de Schrödinger independiente del tiempo es el método variacional de Rayleigh-Ritz. Este método consiste en escribir una representación matricial de la ecuación diferencial, a partir de un conjunto de funciones, y luego calcular los autovalores y autovectores de dicha matriz.

En este trabajo se recibió una implementación del método variacional de Rayleigh-Ritz, en el que se usaba el conjunto de funciones conocido como B-splines, el cual se optimizó utilizando estructuras de datos y modificaciones algorítmicas más eficientes.

Índice general

Capítulo 1

Introducción

Para el desarrollo del trabajo que presentamos en este informe, es necesario estudiar y comprender varias herramientas y técnicas que usaremos a lo largo del proyecto. A continuación presentamos un resumen de los temas que abordaremos.

1.1. Motivación

En la gran mayoría de los problemas de mecánica cuántica no es posible encontrar una solución analítica a la ecuación de Schrödinger por lo que se buscan soluciones aproximadas en forma numérica. En un gran número de problemas, para obtener una buena aproximación de la solución real es necesario recurrir a tamaños de matrices grandes. Por lo que el cómputo y memoria de las mismas se hace cada vez más lento y pesado, siendo de suma importancia tener un software lo más eficiente en tiempo y espacio para dicho cálculo.

1.2. Objetivos del trabajo

El objetivo del trabajo es mejorar una implementación existente del método variacional de Rayleigh-Ritz. La mejora es en cuanto a eficiencia de tiempo y memoria del cálculo del sistema. Estas mejoras se puede lograr a través de estructuras de datos específicas y utilizando arquitecturas de GPU en algunas rutinas.

1.3. Estructura de la Tesis

La tésis esta estructurada de la siguiente manera:

- Nociones Preliminares: Se presentan las estucturas de datos a utilizar, ventajas y desventajas de las mismas
- Implementación Numérica del Método de Rayleigh-Ritz: Se desarrolla la implementación del método la cual se optimizaá.
- Modelos Computacional CPU y GPU: Se explican los conceptos de las arquitecturas GPGPU y CPU y sus técnicas de programación para su mejor desempeño.

- Optimización: Se desarrollan variaciones estructurales y algorítmicas.
- Resultados: Se presentan los resultados obtenidos por las modificaciones.
- Conclusiones: Se presentan conclusiones del trabajo y posibles mejoras.

Capítulo 2

Nociones preliminares

2.1. Método variacional de Rayleigh-Ritz.

Entre las formas de dependencia de los parámetros variacionales, una ampliamente utilizada es la de los parámetros lineales:

Se considera una función variacional lineal, que es una combinación de n funciones linealmente independientes:

$$\Psi = c_1 f_1 + c_2 f_2 + \ldots + c_n f_n = \sum_i^n c_i f_i$$

donde c_i son los coeficientes a obtener por métodos variacionales, y donde f_i son funciones que cumple las condiciones límite del problema, es decir están bien condicionadas.

Tenemos que

$$\langle \Psi \mid \Psi \rangle = \langle \sum_j^n c_j f_j \mid \sum_k^n c_k f_k \rangle = \sum_j^n \sum_k^n c_j c_k \langle f_j \mid f_k \rangle = \sum_j^n \sum_k^n c_j c_k S_{jk}$$

(S es la Matriz de solapamiento)

$$\langle \Psi \mid H \mid \Psi \rangle = \langle \sum_j^n c_j f_j \mid H \mid \sum_k^n c_k f_k \rangle = \sum_j^n \sum_k^n c_j c_k H_{jk}$$

El valor esperado de la energía, o *integral variacional*, será:

$$W = \frac{\langle \Psi \mid H \mid \Psi \rangle}{\langle \Psi \mid \Psi \rangle}$$

Tenemos que minimizar W, que dependerá de n variables $\{c_i\}$:

$$W \sum_j^n \sum_k^n c_j c_k S_{jk} = \sum_j^n \sum_k^n c_j c_k H_{jk}$$

Una condición necesaria para que sea mínimo es:

$$\frac{\partial W}{\partial c_i} = 0 \qquad\qquad i = 1, 2, \ldots, n$$

y haciendo la derivada respecto a cada uno de los c_i, tendremos n ecuaciones:

$$\frac{\partial W}{\partial c_i}\sum_j^n\sum_k^n c_jc_kS_{jk} + W\frac{\partial}{\partial c_i}\sum_j^n\sum_k^n c_jc_kS_{jk} = \frac{\partial}{\partial c_i}\sum_j^n\sum_k^n c_jc_kH_{jk}$$

pero

$$\frac{\partial}{\partial c_i}\sum_j^n\sum_k^n c_jc_kS_{jk} = \sum_j^n\sum_k^n \frac{\partial}{\partial c_i}(c_jc_k)S_{jk} =$$

$$\sum_j^n\sum_k^n (c_k\frac{\partial c_j}{\partial c_i} + c_j\frac{\partial c_k}{\partial c_i})S_{jk}$$

y como los c_i son variables independientes, tan sólo para $c_j = c_i$ se verifica que $\frac{\partial c_i}{\partial c_i} = 1$, mientras que el resto será igual a cero, o sea: $\frac{\partial c_j}{\partial c_i} = \delta_{ij}$, luego

$$\frac{\partial}{\partial c_i}\sum_j^n\sum_k^n c_jc_kS_{jk} = \sum_k^n c_kS_{ik} + \sum_j^n c_jS_{ji}$$

Pero $S_{ij} = S^*_{ji}$, y si las funciones son reales, entonces $S_{ij} = S_{ji}$, por lo que

$$\frac{\partial}{\partial c_i}\sum_j^n\sum_k^n c_jc_kS_{jk} = 2\sum_k^n c_kS_{ik}$$

Igualmente, para el término de la derecha, ya que H es hermítico:

$$\frac{\partial}{\partial c_i}\sum_j^n\sum_k^n c_jc_kH_{jk} = 2\sum_k^n c_kH_{ik}$$

así pues, si $\frac{\partial W}{\partial c_i} = 0 \Rightarrow$

$$2W\sum_k^n c_kS_{ik} = 2\sum_k^n c_kH_{ik} \qquad \text{para } i = 1, 2, \ldots, n$$

$$\sum_k^n [(H_{ik} - S_{ik}W)c_k] = 0 \qquad \text{para } i = 1, 2, \ldots, n$$

tenemos un conjunto de n ecuaciones con n incógnitas, que forman un sistema de ecuaciones lineales homogéneo, las cuales para tener una solución distinta de la trivial, debe tener el determinante de los coeficientes igual a cero, (el determinante de los coeficientes de las n variables debe ser nulo):

$$|H_{ik} - S_{ik}W| = 0$$

que se conoce con el nombre de *determinante secular*. El desarrollo del determinante nos proporciona una ecuación algebraica de grado n en la incógnita W, que lógicamente tendrá n raíces (que serán reales), que se pueden agrupar en orden creciente:

$$W_0 \leq W_1 \leq W_2 \ldots\ldots \leq W_{n-1}$$

y si enumeramos los estados del sistema en orden de energías crecientes:

$$E_0 \leq E_1 \leq E_2 \ldots\ldots \leq E_{n-1}$$

Por el teorema variacional (o de Eckart), $W_0 \geq E_0$, pero además, J.K.L. MacDonald, [17], demostró que $E_1 \leq W_1, E_2 \leq W_2, \ldots, E_{n-1} \leq W_{n-1}$.

Si ahora queremos la función de onda de cada estado, debemos sustituir en las ecuaciones originales:

$$\sum_{k}^{n} [(H_{ik} - S_{ik}W)c_k] = 0 \text{ para } i = 1, 2, \ldots, n$$

el valor W del estado en que estemos interesados y obtener los coeficientes, que como ya vimos, nos quedarán en función de uno de ellos, y para determinarlo recurriremos a normalizar la función.

2.2. B-splines

Los B-splines son funciones diseñadas para generalizar polinomios con propositos de aproximar una función dada. De esta manera se pueden obtener valores de funciones, sus dereivadas y sus integrales. A continuación introduciremos algunas definiciones:

1. El polinomio de orden k (maximo grado $k-1$) es: $p_{(x)} = a_0 + a_1 + \ldots + a_{k-1}^{k-1}$.

2. Una función f y sus derivadas que son continuas (dado un intervalo) hasta orden n se le dice que pertenecen a la clase C^n. Por lo tanto C^0 significa que f es continua, C^{-1} significa que f es no es continua.

3. Considere el intervalo $I = [a, b]$ dividido en l intervalos $I_j = [\xi_j, \xi_{j+1}]$ es una secuencia de $l + 1$ puntos en orden creciente estricto. Los $\{\xi_i\}$ sera llamado breakpoints (bps).

Cada B-spline es una función compuesta por diferentes piezas de polinomios en subintervalos adjacentes. Unidas con cierto grado de continuidad en los *bps*. Rara vez estas condiciones son necesarias:

1. Asociemos a bps $\xi_j, j = 2, \ldots, l$, una segunda secuencia de enteros no negativos $v_j, j = 2, \ldots, l, v_j \geq 0$, que definen la condición de continuidad $C^{v_j - 1}$ en el bsp ξ_j asociado. Con el fin bsp ξ_1 y ξ_{l+1} asociamos $v_1 = v_{l+1} = 0$, es decir, no necesitamos ninguna continuidad. Esto es natural ya que sólo estamos interesados en el intervalo [a, b]. Otras restricciones pueden ser dictadas por condiciones de frontera en los extremos, y son fácilmente implementadas como veremos. Por lo tanto, en el ejemplo tenemos $v_j = \{0,2,2,2,2,0\ \}$

2. Finalmente llamemos a nodos otra secuencia de puntos t_i en orden ascendente, no necesariamente distinta, asociada con ξ_j y v_j como sigue:

$$
\begin{aligned}
&t_1 = t_2 = \ldots = t_{\mu_1} = \xi_1; \mu_1 = k \\
&t\mu_1 + 1 = \ldots = t_{\mu_1+\mu_2} = \xi_2 \\
&\ldots \\
&t_{p+1} = \ldots = t_{p=\mu_i} = \xi_i; p = \mu_1 + \mu_2 + \ldots + \mu_{i-1} \\
&\ldots \\
&t_{n+1} = \ldots = t_{k+n} = \xi_{l+1}; \mu_{l+1} = k; n = \mu_1 + \mu_2 + \ldots + \mu_l
\end{aligned}
$$

Donde μ_j es la multiplicidad de los knots t_i a ξ_j y está dada por $\mu_j = k - v_j$ .Actualmente sólo la multiplicidad en el bps interno es importante, y siempre elegiremos la multiplicidad máxima $\mu_1 = \mu_{l+1} = k$ en los extremos.

La opción más común para la multiplicidad de knots en bps internos es la unidad, correspondiente a máxima continuidad, que es C^{k-2}. Esta elección se empleará, a menos que se indique lo contrario. Con esta elección el número de funciones B-spline n está dado por:

$$n = l + k - 1$$

2.2.1. La base de B-splines

Un B-spline $B(x)$ está definida por el orden $k > 0$, y un conjunto de $k + 1$ knots, t_i,. . . ,T_{i+k}, Tal que $t_i < t_{i+k}$. Las propiedades importantes, son las siguientes.

1. $B(x)$ es una pp-función de orden k sobre $[t_i, t_{i+k}]$
2. $B(x) > 0$ para $x \in (t_i, t_{i+k})$.
3. $B(x) = 0$ para $x \notin [t_i, t_{i+k}]$.

Los B-splines están diseñados para tener un soporte mínimo. Algunas propiedades generales de las B-splines son las siguiendo.

1. En cada intervalo (t_i, t_{i+1}), $t_i < t_{i+1}$, exactamente k B-splines no son cero, Esto sería $B_j(x) \neq 0 \; para \; j = i - k + 1, \ldots, i$
2. En la expansión de una función arbitraria.

$$f(s) = \sum_{j=1}^{n} c_j B_j(x) \; = \sum_{j=i-k+1}^{i} c_j B_j(x) \; para \; x \; \in \; [t_i, t_{i+1}]$$

Donde siempre contribuyen k términos, por lo que un número mínimo de operaciones son necesarias.

3. Como los B-splines son no negativos con soporte mínimo, los coeficientes de expansión de una función arbitraria f están próximos a los valores de función en los *knots*. Esto significa que las oscilaciones en los coeficientes se evitan, los errores de cancelación son mínimos y numéricamente estables.

4. Cada intervalo $I_j = [\xi_j, \xi_{j+1}] = [t_i, t_{i+1}]$ se caracteriza por un par de knots consecutivos $t_i < t_{i+1}$. t_i se llama el knot izquierdo del intervalo I_j, y determina los índices de B_i contribuyendo sobre I_j, estos son $B_{i-k+1}, \ldots, B_i$.

5. Están normalizados como $\sum_i B_i(x) = 1$ sobre $[t_k, t_n]$.

6. Para los knots equidistantes cada B_i es sólo una traducción por un intervalo del anterior. Si los knots no son equidistantes hay un cambio suave en la forma.

7. B-splines satisfacen la recursion

$$B_i^k(x) = \frac{x - t_i}{t_{i+k-1} - t_i} B_i^{k-1}(x) + \frac{t_{i+k} - x}{t_{i+k} - t_{i+1}} B_{i+1}^{k-1}(x)$$

   Esto da el algoritmo empleado para la evaluación práctica de las B-splines: dado un punto x, se generan por recursión los valores de todas los k B-splines que no son cero en x.

2.3. Problema de autovalores y autovectores

Como vimos anteriormente este método consiste en: escribir una representación matricial de la ecuación diferencial a partir de un conjunto de funciones, luego calcular los autovalores y autovectores más pequeños de ese sistema. Para ello se utilizó paquete ARPACK [16].

Este paquete está diseñado para calcular algunos autovalores y autovectores correspondientes de una matriz general cuadrada $\mathcal{A}$. Es más apropiado para matrices grandes ralas o estructuradas, donde estructurado significa que un producto vectorial matricial $\mathrm{w} = \mathcal{A}\mathrm{v}$ requiere $\mathcal{O}(m)$ en lugar de $\mathcal{O}(n^2)$ operaciones de punto flotante, donde m es menor que n^2. Este software se basa en una variante algorítmica del proceso Arnoldi llamado Implicitly Restarted Arnoldi Method (IRAM) [3]. Cuando la matriz $\mathcal{A}$ es simétrica, se reduce a una variante del proceso de Lanczos denominado Método Lanczos Implícitamente Reiniciado (IRLM) [18]. Estas variantes pueden ser vistas como una síntesis del proceso de Arnoldi Lanczos con la técnica QR [19] implícitamente desplazada que es adecuada para problemas a gran escala. Para muchos problemas estándar, no se requiere una factorización de matriz.

ARPACK es capaz de resolver grandes problemas simbólicos simétricos, no simétricos y generalizados de áreas de aplicación significativas. El software está diseñado para calcular algunos autovalores con características especificadas por el usuario tales como los de mayor parte real o mayor magnitud. Los requisitos de almacenamiento están en el orden $\mathcal{O}(n \times k)$ de memoria. No se requiere almacenamiento auxiliar. Se calcula un conjunto de vectores de base de Schur para el autoespacio k-dimensional deseado que es numéricamente ortogonal a la precisión de trabajo. Para mas detalles ver [15]

A continuación daremos una breve explicación de metodos numéricos utilizados para resolver el problema de autovalores y autovectores.

2.3.1. Método de la Potencia

El método mas simple para calcular el autovalor dominante junto con su vector propio es el método de la potencia presentado en el Algoritmo 1. Bajo suposiciones suaves encuentra el autovalor de A que tiene el valor absoluto más grande el autovector correspondiente.

```
1 y = z vector inicial
2 do
3 |   v = y/‖y‖_2
4 |   y = A * v
5 |   θ = v · y
6 while ‖y − θv‖_2 ≤ ϵ|θ|;
7 λ = θ
8 x = v
```

Algoritmo 1: Método de las Potencia

Sea x_1 el autovector correspondiente a $\lambda_1 = \lambda_{max}(A)$. El ángulo $\angle(z, x_1)$ entre x_1 y z se define por la relación

$$\cos \angle(z, x_1) = \frac{z \cdot x_1}{\|z\|_2 \ \|x_1\|_2}$$

Si el vector inicial z y el autovector x_1 son perpendiculares entre sí, entonces $\cos \angle(z, x_1) = 0$. En este caso, el método de potencia no converge. Por otro lado, si $\cos \angle(z, x_1) \ \neq 0$, el método de las potencias genera una secuencia de vectores que se vuelven cada vez más paralelos a x_1. Esta condición en $\angle(z, x_1)$ es verdadera con probabilidad muy alta si z se elige al azar.

La convergencia del método de la potencias depende de $|\lambda_2/\lambda_1|$, donde λ_2 es el segundo autovalor más grande de A en magnitud. Esta proporción es generalmente menor que 1, lo que permite una convergencia adecuada. Pero hay casos en los que esta relación puede ser muy cercana a 1, causando convergencia muy lenta. Para discusiones detalladas sobre el método de las potencias, véase Demmel [21], Golub y Van Loan [13], y Parlett [12].

2.3.2. Algoritmo de Lanczos

El algoritmo de Lanczos está estrechamente relacionado con los algoritmos iterativos en el sentido de que sólo necesita acceder a la matriz en forma de operaciones de matriz por vector. Es diferente en el sentido de que hace mucho mejor uso de la información obtenida recordando todas las direcciones calculadas y siempre permite que la matriz opere sobre un vector ortogonal a todos aquellos previamente probados.
En esta sección se describe el algoritmo de Lanczos para el caso de matriz hermitiana.

$$Ax = \lambda x \ ,$$

Donde A es una matriz hermitiana, o en el caso real simétrica.

El algoritmo comienza con un vector de inicio elegido v y construye una base ortogonal V_j del subespacio de Krylov.

$$\mathcal{K}^j(A, v) = \text{span}\{v, VA, A^2v, \ldots, A^{j-1}v\} \ , \tag{2.1}$$

En cada paso sólo una multiplicación matriz-vector. En la nueva base ortogonal V_j el operador A está representado por una matriz real tridiagonal simétrica.

$$T_j = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{j-1} \\ & & \beta_{j-1} & \alpha_j \end{bmatrix} \tag{2.2}$$

T se construye de a una fila y una columna a la vez (por se simétrica).

$$AV_j = V_jT_j + re_j^* \quad \text{with} \quad V_j^*r = 0. \tag{2.3}$$

En cualquier paso j, podemos calcular una *auto solución* de T_j

$$T_j s_i^{(j)} = s_i^{(j)} \theta_i^{(j)} \ , \tag{2.4}$$

Donde el superíndice (j) se utiliza para indicar que estas cantidades cambian para cada iteración j. El valor de Ritz $\theta_i^{(j)}$ y su vector.

$$x_i^{(j)} = V_j s_i^{(j)} \ , \tag{2.5}$$

Será una buena aproximación a un autovalor y autovector de A si el residuo tiene una norma tolerante.

Para calcular el residuo de este par de Ritz se utiliza:

$$r_i^{(j)} = Ax_i^{(j)} - x_i^{(j)}\theta_i^{(j)} = AV_j s_i^{(j)} - V_j s_i^{(j)} \theta_i^{(j)} = (AV_j - V_jT_j)s_i^{(j)} = v_{j+1}\beta_j s_{j,i}^{(j)} \ .$$

Donde su norma satisface:

$$\|r_i^{(j)}\|_2 = |\beta_j s_{i,j}^{(j)}| = \beta_{j,i} \ , \tag{2.6}$$

Por lo que solamente necesitamos monitorear los elementos subdiagonales β_j de T y los últimos elementos $s_{i,j}^{(j)}$ de sus autovectores para obtener una estimación del valor absoluto del residuo. Tan pronto como esta estimación es pequeña, podemos marcar el valor de Ritz $\theta_i^{(j)}$ como convergente con el autovalor λ_i. Obsérvese que el cálculo de los valores de Ritz no necesita la multiplicación matriz-vector. Podemos ahorrar esta operación que lleva mucho tiempo hasta el paso j, cuando la estimación indique la convergencia.

Si se sabe de una buena suposición para el autovector deseado es conveniente usarlo. Por ejemplo, si para una ecuación diferencial parcial discretizada se sabe que el vector propio deseado es suave con respecto a la cuadrícula, se podría comenzar con un vector con todos unos. En otros casos, se elije una dirección aleatoria, como por ejemplo: números aleatorios normalmente distribuidos.

La recursión de Lanczos se construye para que la base V sea ortogonal, pero esto es cierto sólo para el cálculo de precisión infinita. En el algoritmo sólo nos aseguramos de

$r = v$ vector inicial
$\beta_0 = \|r\|_2$ $j = 1$ **do**
  $v_j = r/\beta_{j-1}$
  $r = Av_j$
  $r = r - v_{j-1}\beta_{j-1}$
  $\alpha_j = v_j \cdot r$
  $r = r - v_j\alpha_j$
  reortogonalizar si es necesario
  $\beta_j = \|r\|_2$
  computar el autovalor aproximado $T_j = S\Theta^j S*$
  $verificar\ convergencia$
while *hasta converger*;
convergenciaomputar los autovectores aproximados $X = V_j S$

Algoritmo 2: Algoritmo de Lanczos

que el nuevo vector v_{j+1} sea ortogonal a la precisión de los dos vectores más recientes v_{j-1} y v_j, y la ortogonalidad a los vectores anteriores se sigue de la simetría de A y la recursión. Tan pronto como converge un valor propio, es decir, el par de Ritz tiene un residuo pequeño, todos los vectores de base v_j obtienen perturbaciones en la dirección del autoespacio del valor propio convergente. Como resultado de esto, una copia duplicada de ese valor propio pronto aparecerá en la matriz tridiagonal T [12].

La gran ventaja de este tipo de algoritmo es que la matriz A se accede sólo en una operación matriz por vector en el Algoritmo 2. Cualquier de tipo de esquema de almacenamiento puede ser aprovechado.

Es necesario mantener sólo tres vectores, r, v_j, y v_{j-1}, fácilmente accesibles. Incluso hay una variante en la que sólo se necesitan dos vectores (véase [12] capítulo 13.1]), pero requiere cierta destreza para codificarla.

2.3.3. Algoritmo de Arnoldi

El metódo de Arnoldi se presentó por primera vez como un algoritmo directo para reducir una matriz general en la forma superior de Hessenberg [10]. Más tarde se descubrió que este algoritmo conduce a una buena técnica iterativa para aproximar valores propios de grandes matrices ralas.

El algoritmo funciona para matrices no hermitaneas. Es muy útil para casos en los que la matriz A es grande, pero los productos de matriz por vector son relativamente baratos de realizar. Esta es la situación, por ejemplo, cuando A es grande y ralo. Comenzamos con una presentación del algoritmo básico y luego describimos una serie de variaciones.

2.3.3.1. Algoritmo Básico

El método de Arnoldi es un método de proyección ortogonal sobre un subspacio de Krylov. Comienza con el procedimiento de Arnoldi como se describe en el algoritmo 3. El procedimiento puede ser visto esencialmente como un proceso de Gram-Schmidt midificado para construir una base ortogonal del subespacio de Krylov. $K^m(A, v)$

```
1  Procedimiento Arnoldi;
2  v_1 = v/‖v‖_2
3  for j = 1 to m do
4  |   w = Av_j
5  |   for j = 1 to j do
6  |   |   h_ij = w * v_i
7  |   |   w = w − h_ij vi
8  |   end
9  |   h_{j+1,k} = ‖w‖_2
10 |   if h_{j+1,j} == 0 then
11 |   |   stop
12 |   end
13 |   v_{j+1} = w/h_{j+1,j}
14 end
```

Algoritmo 3: Procedimiento de Arnoldi

El algoritmo 3 se detendrá si el vector w tiene norma 0 (desaparece). Los vectores $v_1, v_2, \ldots, v_m$ forman un sistema ortonormal por construcción y se llaman vectores de Arnoldi. Un argumento de inducción fácil demuestra que este sistema es una base del subespacio de Krylov $K^m(A, v)$.

A continuación se considera una relación fundamental entre las cantidades generadas por el algoritmo. La siguiente igualdad se deriva fácilmente:

$$Av_j = \sum_{i=1}^{j+1} h_{ij} v_i, \quad j = 1, 2, \ldots, m \ . \tag{2.7}$$

Si denotamos por V_m la matriz $n \times m$ con vectores columna $v_1, \ldots, v_m$ y H_m la matriz $m \times m$ Hessenberg cuyas entradas no nulas h_{ij} son definidas por el algoritmo, entonces las siguientes relaciones se mantienen:

$$AV_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^* \tag{2.8}$$

$$V_m^* A V_m = H_m. \tag{2.9}$$

La ecuación 2.9 viene de 2.8 multiplicando las dos partes de la ecuación 2.8 por V_m^* y usando la ortonormalidad de $\{v_1, \ldots, v_m\}$.

Como se observó anteriormente, el algoritmo se descompone cuando la norma de w desaparece en un cierto paso j. Esto ocurre si y sólo si el vector de partida v es una combinación de j vectores propios (es decir, el polinomio mínimo de v_1 es de grado j). Además, el subespacio K_j es entonces invariante y los autovalores y autovectores aproximados son exactos. [9]

Los autovalores aproximados λ_i^m dados por el proceso de proyección sobre K_m son los valores propios de la matriz de Hessenberg H_m. Éstos son conocidos como valores de Ritz. Un autovector aproximado de Ritz asociado con un valor de Ritz λ_i^m es definido por $u_i^m = V_m y_i^m$, donde Y_i^m es un vector propio asociado con el autovalor λ_i^m. Un número de los valores propios del Ritz, típicamente una pequeña fracción de m, generalmente

constituyen buenas aproximaciones para los valores propios correspondientes λ_i de A, y la calidad de la aproximación usualmente mejorará a medida que m aumenta.

El algoritmo original consiste en aumentar m hasta que todos los autovalores deseados de A se encuentren. Para matrices grandes, ésto resulta costoso tanto en términos de cálculo como de almacenamiento. En términos de almacenamiento, necesitamos mantener m vectores de longitud n más una matriz de Hessenberg de m^2 elementos, un total aproximado de $nm + m^2/2$ elementos. Para los costos aritméticos, necesitamos multiplicar v_j por A, al costo de $2 \times N_z$, donde N_z es el número de elementos no nulos en A, y luego ortogonalizar el resultado contra j vectores al costo de $4(j + 1)n$, que aumenta con el paso número j. Por lo tanto, un procedimiento de Arnoldi de m-dimensionales cuesta $\approx nm + m^2/2$ en almacenamiento y $\approx N_z + 2nm^2$ en operaciones aritméticas.

Obtener la norma residual, para un par Ritz, a medida que el algoritmo progresa es bastante eficiente. Sea Y_y un autovector de H_m asociado al autovalor λ_i^m, y sea u_i^m el autovector aproximado de Ritz $u_i^m = V_m y_i^m$. Tenemos la relación.

$$(A - \lambda_i^m I)u_i^m = h_{m+1,m}(e_m^* y_i^m)v_{m+1},$$

y por lo tanto

$$\|(A - \lambda_i^m I)u_i^m\|_2 = h_{m+1,m}|e_m^* y_i^m| \ .$$

Así, la norma residual es igual al valor absoluto del último componente del vector propio Y_i^m multiplicado por $h_{m+1,m}$. Las normas residuales no son siempre indicativas de errores reales en λ_i^m, pero pueden ser muy útiles para decidir la detención.

2.3.3.2. Variantes

La descripción del procedimiento de Arnoldi dado anteriormente se basó en el proceso modificado de Gram-Schmidt. Otros algoritmos de ortogonalización podrían ser utilizados. Una mejora es recuperar la ortogonalidad cuando sea necesario. Cada vez que se calcula el vector final obtenido al final del segundo bucle en el algoritmo anterior, se realiza una prueba para comparar su norma con la norma del w inicial (que es $\|Av_j\|_2$). Si la reducción cae por debajo de un determinado umbral (una indicación de que puede haber ocurrido una cancelación severa), se realiza una segunda ortogonalización. Se sabe por un resultado de Kahan que más de dos ortogonalizaciones son superfluas (véase, por ejemplo, Parlett [12])

Una de las técnicas de ortogonalización más confiables, desde el punto de vista numérico, es el algoritmo Householder [13]. Ésto ha sido implementado para el procedimiento de Arnoldi por Walker [14]. El algoritmo Householder es numéricamente más estable que las versiones de Gram-Schmidt o Gram-Schmidt modificado, pero también es más caro, requiriendo aproximadamente el mismo almacenamiento que el Gram-Schmidt modificado, pero aproximadamente el doble de operaciones. La ortogonalización de Householder es una opción razonable cuando se desarrollan paquetes de software confiables y de propósito general donde la robustez es un criterio crítico.

2.3.3.3. Reinicios explícitos

Como se mencionó anteriormente, las implementaciones estándar del método Arnoldi están limitadas por sus altos requisitos de almacenamiento y cómputo a medida que m crece. Supongamos que estamos interesados en un sólo autovalor / autovector de A, llamado

el autovalor de la parte real más grande de A. Entonces una manera de eludir la dificultad es reiniciar el algoritmo. Después de una corrida con vectores de m Arnoldi, calculamos el vector propio aproximado y lo usamos como vector inicial para la siguiente ejecución con el método de Arnoldi. Este proceso, el más simple de este tipo, es iterado a la convergencia para calcular un par propio. Para el cálculo de otros pares autovector autovalor, y para mejorar la eficiencia del proceso, se han desarrollado una serie de estrategias, que están algo relacionadas. Estos incluyen *procedimientos de deflación* brevemente discutidos en la siguiente Sección, y la estrategia de reinicio explícito descrita en Algoritmo 4

```
1 while True do
2 |   Iterate: Hacer m iteraciones del algoritmo 3
3 |   Reiniciar: Calcular el autovalor aproximado u_1^(m) asociado con el autovalor mas
  |    a la derecha λ_1^(m)
4 |   if satisface then
5 |   |  Stop
6 |   end
7 |   v_1 = u_1^(m)
8 end
```

Algoritmo 4: Método Explícito de Arnoldi Reiniciado para NHEP

2.3.3.4. Deflación

Ahora consideramos la siguiente implementación que incorpora un proceso de deflación. Hasta ahora hemos descrito algoritmos que calculan sólo un autopar. En caso de que se busquen varios autopares, hay dos opciones posibles.

La primera es tomar v_1 como una combinación lineal de los vectores propios aproximados cuando reiniciamos. Por ejemplo, si necesitamos calcular los vectores propios p a la derecha, podemos tomar

$$\hat{v}_1 = \sum_{i=1}^{p} \rho_i \tilde{u}_i,$$

Donde los autovalores están numerados en orden decreciente de sus partes reales. El vector v_1 se obtiene luego de normalizar $\hat{v}_1$. La opción más simple para los coeficientes ρ_i es tomar $\rho_i = 1, i = 1, \ldots, p$. Hay varios inconvenientes a este enfoque, el más importante es que no hay manera fácil de elegir los coeficientes ρ_i de una forma sistemática. El resultado es que para los problemas complejos, la convergencia es difícil de lograr. Una alternativa más confiable es computar un autopar a la vez y usar la deflación. La matriz A puede deflactarse explícitamente construyendo progresivamente los primeros k vectores de Schur. Si ya se ha calculado una base ortogonal previa $U_{k-1} = [u_1, \ldots, u_{k-1}]$ del subespacio invariante, entonces para computar el valor propio λ_k, Puede trabajar con la matriz

$$\tilde{A} = A - U_{k-1} \Sigma U_{k-1}^*,$$

En la que $\Sigma = diag(\sigma_i)$ es una matriz diagonal de desplazamientos. Los autovalores de $\tilde{A}$ consisten en dos grupos. Estos valores propios asociados con los vectores de Schur

$u_1, \ldots, u_{k-1}$ serán cambiados a $\tilde{\lambda}_i = \lambda_i - \sigma_i$ y los otros no se modificarán. Si se buscan los valores propios con partes reales más grandes, entonces los desplazamientos se seleccionan de modo que λ_k se convierta en el autovalor siguiente con la parte real más grande de $\tilde{A}$. También es posible desinflar simplemente proyectando los componentes asociados con el subespacio invariante cubierto por U_{k-1}; Esto conduciría a operar con la matriz.

$$\tilde{A} = A(I - U_{k-1}U_{k-1}^*).$$

Hay que tener en cuenta que si $AU_{k-1} = U_{k-1}R_{k-1}$ es la descomposición parcial de Schur asociada con los primeros $k-1$ valores de Ritz, entonces $\tilde{A} = A - U_{k-1}R_{k-1}U_{k-1}^*$. Los valores propios asociados con los vectores Schur $u_1, \ldots, u_{k-1}$ ahora se moverán a cero.

Una mejor implementación de la deflación, que encaja bien con el procedimiento de Arnoldi, es trabajar con una sóla base $v_1, v_2, \ldots, v_m$ cuyos primeros vectores son los vectores de Schur que ya han convergido. Supongamos que $k-1$ tales vectores han convergido y llamamos $v_1, v_2, \ldots, v_{k-1}$. Luego empezamos por elegir un vector v_k que es ortogonal a $v_1, \ldots, v_{k-1}$ y de norma 1. Luego realizamos mk pasos de un procedimiento de Arnoldi en el que la ortogonalidad del vector V_j contra todos los v_i anteriores, incluyendo $v_1, \ldots, v_{k-1}$. Ésto genera una base ortogonal del subespacio.

$$\text{span}\{v_1, \ldots, v_{k-1}, v_k, Av_k, \ldots, A^{m-k}v_k\} \ .$$

Así, la dimensión de este subespacio Krylov modificado es constante e igual a m en general. A continuación se presenta un esquema de este procedimiento implícito de deflación combinado con el método Arnoldi.

Input: Matriz A, vector inicial v_1, dimensión del subespacio m y la cantidad de autovalores nev

```
k = 1
while k ≤ nev do
    for j = k to m do
        w = Av_j
        calcular un conjunto de j coeficientes h_ij tales que w =
          Σ_{i=1}^{j} h_ij v_i es ortogonal a todos los anteriores v_i, para i = 1, 2, ..., j
        h_{j=1,j} = ||w||_2
        v_{j+1} = w / h_{j+1,j}
    end
    calcular el autovector aproximado de A con el autovalor λ̃_k y su norma
    residual estimada ρ_k
    ortonormalizar este autovector sobre todos los anteriores v_j para obtener
    un vector de Schur aproximado ũ_k y definir v_k = ũ_k
    if ρ_k < tol then
        h_ik = v_i^* A v_k, i = 1, ..., k
        k = k + 1
    end
end
```

Algoritmo 5: Método explícito de Arnoldi reiniciado con deflación para NHEP

Notar que en el bucle, los vectores Schur asociados con los autovalores $\lambda_1, \ldots, \lambda_{k-1}$ no se tocarán en pasos posteriores. A veces se les denomina ẗvectores bloqueados.: De manera similar, la correspondiente matriz triangular superior correspondiente a estos vectores también está bloqueada.

$$\underbrace{[v_1, v_2, \ldots, v_{k-1}}_{Bloqueados}, \underbrace{v_k, v_{k+1}, \ldots v_m]}_{Activos}$$

Cuando converge un nuevo vector de Schur, se calcula la columna k-ésima de R asociada con este nuevo vector de base. En los pasos siguientes, los valores propios aproximados son los valores propios de la matriz $m \times m$ Hessenberg H_m definida en el algoritmo y cuya k^2 principal submatriz es triangular superior. Por ejemplo, cuando $m = 6$ y después del segundo vector de Schur, $k = 2$, ha convergido, la matriz H_m tendrá la forma

$$H_m = \begin{bmatrix} * & * & * & * & * & * \\ & * & * & * & * & * \\ & & * & * & * & * \\ & & * & * & * & * \\ & & & * & * & * \\ & & & & * & * \end{bmatrix}.$$

En los pasos subsiguientes, sólo deben considerarse los valores propios no asociados con la matriz triangular superior de 2×2.

Se puede demostrar que, en aritmética exacta, la matriz de Hessenberg H_m en el bloque inferior (2×2) es la misma matriz que se obtendría de una corrida de Arnoldi aplicada a matriz

$$\tilde{A} = (I - U_{k-1}U_{k-1}^*)A.$$

Por lo tanto, estamos proyectando implícitamente el subespacio invariante ya calculado desde el rango de A.

2.3.3.5. Reiniciado Implícito del Método de Arnoldi

El método IRAM (por sus siglas en ingles Implicitly Restarted Arnoldi Method) es quizás el algoritmo numérico más exitoso y más completo para calcular los autovectores y autovalores de una matriz cuadrada general A es el algoritmo QR implícitamente desplazado. Una de las claves para el éxito de este método es su relación con la descomposición de Schur.

$$A = UTU^*. \tag{2.10}$$

Esta descomposición bien conocida afirma que cada matriz cuadrada A es unitariamente equivalentes a una matriz triangular superior T.

El algoritmo QR produce una secuencia de transformaciones unitarias de similitud que reducen iterativamente A a la forma triangular superior. En otras palabras, calcula una descomposición de Schur. Una implementación práctica del algoritmo QR comienza con una transformación de similitud unitaria inicial de A a la forma condensada $V^*AV = H$

donde H es Hessenberg superior (" casi triangular superior ") y V es unitario. Entonces se realiza la siguiente iteración.

Input: Matriz A
1 *Factorizar* $V^*AV = H$
2 **while** *no converge* **do**
3 | *seleccionar el desplazamiento* μ
4 | $QR = H - \mu I$
5 | $H = Q^*HQ$
6 | $V = VQ$
7 **end**

Algoritmo 6: Método QR Desplazado

En este esquema, Q es unitaria y R es triangular superior (es decir, la factorización QR de $H - \mu I$). Es fácil ver que H es unitariamente equivalente a A a lo largo de esta iteración. La iteración se continúa hasta que los elementos subdiagonales de H convergen a cero, es decir, hasta que se ha obtenido (aproximadamente) una descomposición de Schur.

Si U_k representa las columnas k principales de U, y T_k el principio principal $k \times k$ submatriz de T en 2.10, entonces:

$$AU_k = U_k T_k,$$

Y nos referimos a esto como una descomposición parcial de Schur de A. Dado que hay una descomposición de Schur con los autovalores de A que aparecen en la diagonal en cualquier orden, siempre hay una descomposición parcial de Schur de A con los elementos diagonales de T_k que consisten en cualquier subconjunto especificado de k autovalores de A. Además, span(U_k) es un subespacio invariante de A para estos autovalores.

2.4. Estructura de representacion de matrices dispersas

En esta sección explicaremos las estructuras de representaciones de matrices.

A medida que este problema particular crece las matrices se vuelven más grandes y al mismo tiempo el porcentaje de no nulos de la matriz decrese. Un sistema lineal grande de la forma $\mathcal{A}x = b$ puede ser más eficientemente resuelto si los elementos que son ceros de $\mathcal{A}$ no son guardados. Los esquemas de almacenamiento dispersos asignan almacenamiento contiguo en memoria para los elementos distintos de cero de la matriz y tal vez un número limitado de ceros. Ésto, por supuesto, requiere un esquema para saber dónde encajan los elementos en la matriz completa

Hay muchos métodos para almacenar los datos (véase por ejemplo Saad [1] y Eijkhout [2]). Aquí discutiremos cuatro métodos: *almacenamiento de filas y columnas comprimidas*, *almacenamiento de filas comprimidas en bloques*, *almacenamiento en diagonal* y *almacenamiento diagonal recortado*

2.4.1. Almacenamiento de filas y columnas comprimidas

Los formatos CRS y CCS (sus siglas en ingles de Compressed Row Storage y Compressed Column Storage) son los más generales: no hacen ninguna suposición sobre la estructura de dispersión de la matriz y no almacenan elementos innecesarios. Por otra parte, no son muy eficientes, necesitando un paso de direccionamiento indirecto para cada operación escalar individual en un producto vectorial matricial o solución precondicionadora.

Este formato pone los elementos no ceros de las filas en memoria contigua. Suponiendo que tenemos una matriz dispersa no simétrica $\mathcal{A}$, creamos 3 arreglos: uno que tiene el tipo de los elementos de $\mathcal{A}$ (flotantes simples, flotantes dobles, enteros, etc.) *val*, y otros dos con tipo enteros *col_ind, row_ptr*. El arreglo *val* guarda los valores de los elementos no ceros de la matriz $\mathcal{A}$ de manera lineal por cada fila teniendo primero los elementos no ceros de la fila 1 luego los de la 2 etc. El arreglo *col_ind* guarda a que columna pertenece cada elemento de del arreglo *val* y *row_ptr* guarda intervalos de inicio y fin de cada fila en del arreglo *val*. Esto es:

$$val_k = \mathcal{A}_{i,j} \Rightarrow col\_ind_k = j$$

El arreglo *row_ptr* guarda la ubicación de *val* donde empieza la fila. Esto es:

$$val_k = \mathcal{A}_{i,j} \Rightarrow row\_ptr_i \leq k < row\_ptr_{i+1}$$

Por convención definimos $row\_ptr_{n+1} = nnz + 1$. Donde nnz es la cantidad de números no ceros de la matriz $\mathcal{A}$. La memoria necesaria para este enfoque es $\mathcal{O}(nnz + n)$

Ejemplo: considere la siguiente matriz:

$$\mathcal{A} = \begin{pmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{pmatrix} \tag{2.11}$$

En formato CRS tendría esta forma

$$val = \begin{pmatrix} 10 & -2 & 3 & 9 & 3 & 7 & 8 & 7 & 3 & 8 & 7 & 5 & 8 & 9 & 9 & 13 & 4 & 2 & -1 \end{pmatrix} \tag{2.12}$$

$$col\_ind = \begin{pmatrix} 0 & 4 & 0 & 1 & 5 & 1 & 2 & 3 & 0 & 2 & 3 & 4 & 1 & 3 & 4 & 5 & 1 & 4 & 5 \end{pmatrix} \tag{2.13}$$

$$row\_ptr = \begin{pmatrix} 1 & 3 & 6 & 9 & 13 & 17 & 20 \end{pmatrix} \tag{2.14}$$

El almacenamiento de columnas comprimidas es análogo

2.4.2. Almacenamiento de filas comprimidas en bloques.

El formato BCRS (sus siglas en ingles de Block Compressed Row Storage) es útil si la matriz rala se compone de bloques densos cuadrados de no nulos con algún patrón regular, podemos modificar el formato CRS (o CCS) para explotar tales patrones de bloque. Las matrices de bloque típicamente surgen de la discretización de ecuaciones diferenciales parciales en las que hay varios grados de libertad asociados con un punto. Luego, la partición de la matriz en bloques pequeños con un tamaño igual al número de grados de libertad, y tratar cada bloque como una matriz densa, a pesar de que puede tener algunos ceros.

Sea n_b es la dimensión de cada bloque y *nnzb* la cantidad de no ceros de cada bloque en la matriz $\mathcal{A}^{n,m}$, la cantidad de memoria es $\mathcal{O}$(nnzb).

2.4.3. Almacenamiento en diagonal.

El formato CDS (sus siglas en ingles de Compressed Diagonal Storage) es útil si la matriz $\mathcal{A}$ es una matriz de banda donde el ancho de la banda es constante de fila en fila. Entonces podemos aprovechar esta estructura en el esquema de almacenamiento almacenando subdiagonales de la matriz en ubicaciones consecutivas. No sólo podemos eliminar el vector que identifica la columna y la fila (si lo miramos como un CRS), podemos empaquetar los elementos no nulos de tal manera que el producto vectorial sea más eficiente. Este esquema de almacenamiento es particularmente útil si la matriz surge de una discretización de elementos finitos o diferencias finitas en una matriz de producto tensor.

Decimos que la matriz $\mathcal{A}^{m,n}$ es *de banda* si hay enteros no negativos p,q tal que $\mathcal{A}_{i,j} \neq 0 \Rightarrow i - p \leq j \leq i + q$. En este caso podemos poner la matriz $\mathcal{A}$ en un arreglo val(1:n, -p:q) (si no tenemos índices negativos se puede hacer un desplazamientos p de índices) la declaración con dimensiones invertidas corresponde a *LINPACK band format* [6]

Por lo general, los formatos de banda incluyen almacenar algunos ceros. El formato CDS puede incluso contener algunos elementos de matriz que no corresponden a elementos de matriz en absoluto. Consideremos la matriz no simétrica definida por:

$$\mathcal{A} = \begin{pmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{pmatrix} \tag{2.15}$$

Usando el formato CDS, alojamos la matriz $\mathcal{A}$ en un arreglo $cdsA^{6,3}$ donde las columnas estan indexadas desde -1 usando el mapeo $val_{i,j} = a_{i,i+j}$

val(:, -1)	0	3	7	8	9	2
val(:, 0)	10	9	8	7	9	-1
val(:, 1)	-3	6	7	5	13	0

Notar que los dos ceros no corresponden a un elemento existente de la matriz.

Una generalización del formato CDS más adecuada para manipular matrices dispersas generales en supercomputadoras vectoriales es discutido por Melhem en [7]. Esta variante de CDS utiliza una estructura de datos de banda para almacenar la matriz. Esta estructura

es más eficiente en el almacenamiento en el caso de variar el ancho de banda, pero hace que el producto de matriz por vector sea ligeramente más caro, ya que implica una operación de recopilación.

Como está definido en [7], una línea en $\mathcal{A}^{n,m}$ es un conjunto de posiciones S $= \{(i, \theta(i)) : i \in I \subseteq I_n\}$ donde $I_n = 1, ..., n$ y θ es una función estrictamente creciente. Específicamente:

$$(i, \theta(i)), (j, \theta(j)) \in S \Rightarrow (i < j \Rightarrow \theta(i) < \theta(j))$$

Cuando se computa el producto matriz-vector $y = \mathcal{A}x$, cada (i, $\theta_k(i)$) de $\mathcal{A}$ se multiplica por $x_{\theta_k(i)}$ y es acumulado en y_i.

2.4.4. Almacenamiento diagonal recortado.

El formato JDS (sus siglas en ingles de Jagged Diagonal Storage) puede ser útil para la implementación de métodos iterativos en procesadores paralelos y vectoriales (véase Saad [4]). Al igual que el formato Diagonal Comprimido, da una longitud de vector esencialmente del tamaño de la matriz. Es más eficiente en cuanto a espacio que CDS a costa de una operación de recopilación / dispersión.

Una forma simplificada de JDS, llamada almacenamiento de ITPACK o almacenamiento Purdue, se puede describir como sigue:

$$\begin{pmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 0 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{pmatrix} \Rightarrow \begin{pmatrix} 10 & -3 & 1 \\ 9 & 6 & -2 \\ 3 & 8 & 7 \\ 6 & 7 & 5 \\ 9 & 13 & \\ 5 & -1 & \end{pmatrix} \tag{2.16}$$

Luego las columnas se almacenan consecutivamente. Todas las filas se rellenan con ceros a la derecha para darles la misma longitud. Correspondiente al arreglo de elementos de matriz *val*(:, :), un arreglo de índices de columna, *col_ind* (:, :) también se almacena.

Está claro que los ceros de relleno en esta estructura pueden ser una desventaja, especialmente si el ancho de banda de la matriz varía fuertemente. Por lo tanto, en el formato CRS, reordenamos las filas de la matriz de forma decreciente de acuerdo con el número de elementos no ceros de una fila. Las diagonales comprimidas y permutadas se almacenan entonces en una matriz lineal. La nueva estructura de datos se denomina diagonales dentadas.

val(:, 1)	10	9	3	6	9	5
val(:, 2)	-3	6	8	7	13	-1
val(:, 3)	1	-2	7	5	0	0
val(:, 4)	0	0	0	4	0	0

Cuadro 2.1:

El número de diagonales irregulares es igual al número de no ceros en la primera fila, es decir, el mayor número de no ceros en cualquier fila de $\mathcal{A}$. La estructura de datos para representar la matriz por lo tanto consiste en un arreglo de permutación *perm* (1:

col_ind(:, 1)	1	2	1	2	5	5
col_ind(:, 2)	2	3	3	4	6	6
col_ind(:, 3)	4	5	4	5	0	0
col_ind(:, 4)	0	0	0	6	0	0

Cuadro 2.2:

jdiag	6	9	3	10	9	5;	7	6	8	-3	13	-1;	5	-2	7	1	4;
col_ind	2	2	1	1	5	5;	4	3	3	2	6	6;	5	5	4	4	6;

Cuadro 2.3:

perm	4	2	3	1	5	6

jd_ptr	1	7	13	17

Cuadro 2.4:

n) que reordena las filas, un arreglo de punto flotante $jdiag(:)$ que contiene las diagonales dentadas en sucesión, un arreglo de enteros *col_ind* (:) que contiene los índices de columna correspondientes, y finalmente un arreglo de punteros (*jd_ptr* (:)) cuyos elementos apuntan al comienzo de cada diagonal dentada. Las ventajas de JDS para las multiplicaciones matriciales son discutidas por Saad en [4].

El formato JDS para la matriz anterior en el uso de los arreglos lineales *perm*, $jdiag$, *col_ind*, jd_ptr se dá en los cuadros 2.4.

2.5. Conclusiones

En este capítulo hemos visto estructuras de datos alternativas para modelar matrices con sus ventajas y desventajas, también vimos algoritmos para calcular autovalores y autovectores de un sistema. Este conocimiento previo es necesario a la hora de decidir que estructura se utilizará, cómo manipularla, cuáles serán sus beneficios y costos, y además, qué algoritmos de cálculos de autovectores y autovalores. En este trabajo se decidió usar formato comprimido por columnas (CCS) y el método Reinicio Implicito del Método de Arnoldi (IRAM) pues es el más estable y el más robusto a propositos generales por lo que resulta razonable de elegir en términos de requerimientos de sistema. Si bien el programa retorna el sistema para que se pueda usar el módulo de resolución de autovalores que más convenga, se optó por esta alternativa utilizando el módulo ARPACK++ [16].

Capítulo 3

Implementación numérica del Método de Rayleigh-Ritz

En esta sección explicaremos cómo es la primera implementación del método para una y dos particulas, se explicarán las funciones y variables y su relación con el modelo teórico explicado en la sección 2.1.

3.1. Introdución

En mecánica cuántica el estado de una partícula está dado por una función de onda, $|\Psi(\vec{r})\rangle$, la cuál puede escribirse como combinación lineal de los autoestados del sistema $|\psi_n(\vec{r})\rangle$. Los autoestados son las soluciones de la ecuación de Schrödinger independiente del tiempo

$$\mathcal{H}|\psi_n(\vec{r})\rangle = E_n|\psi_n(\vec{r})\rangle\,, \tag{3.1}$$

dónde $\mathcal{H}$ es el Hamiltoniano del sistema y E_n es la energía asociada al n-ésimo autoestado. El operador Hamiltoiano, $\mathcal{H}$, en Ec. 3.1 es el operador de la energía del sistema. En un sistema de tres dimensiones, el Hamiltoniano en coordenadas esferícas, (r, φ, θ), es

$$\mathcal{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(r)\,, \tag{3.2}$$

$$= -\frac{\hbar^2}{2m}\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d}{dr}\right) + \frac{\hbar^2\, l(l+1)}{2m\, r^2} + V(r)\,, \tag{3.3}$$

dónde $V(r)$ es potencial del sistema que sólo depende de la coordenada radial, l es el momento angular de la partícula. Para obtener el Hamiltoniano en la Ec. 3.2 es necesario escribir el operador Laplaciano ∇^2, en coordenadas esféricas y luego, haciendo separación de variables se obtiene que la dependencia de las funciones $|\phi_n(\vec{r})\rangle$ con las variables φ y θ está dada por las funciones conocidas como armónicos esféricos, $Y_{l,m}(\theta, \varphi)$. De esta forma sólo hay que resolver una ecuación en la variable radial, r, dicha ecuación es

$$-\frac{\hbar^2}{2m}\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{dR_n(r)}{dr}\right) + \frac{\hbar^2\, l(l+1)}{2m\, r^2}\, R_n(r) + V(r)\, R_n(r) = E_n\, R_n(r)\,. \tag{3.4}$$

La Ec. 3.4 puede simplificarse haciendo la sustitución

$$\phi_n(r) = r\, R_n(r)\,, \tag{3.5}$$

de esta forma la función $\phi_n(r)$ es solución de la ecuación radial reducida

$$-\frac{1}{2}\frac{d^2\phi_n}{dr^2} + \frac{l(l+1)}{2\,r^2}\phi_n(r) + V(r)\,\phi_n(r) = E_n\,\phi_n(r)\,, \tag{3.6}$$

con las condiciones de contorno $\phi_n(r=0) = \phi_n(\infty) = 0$.

La ecuación radial Ec. 3.6 se resuelve numéricamente en un subespacio suponiendo que cualquier solución $\phi_n(r)$ puede ser aproximada por un conjunto finito de funciones. Como se explica en la sección 2.1, tal subespacio puede ser generado por la base de funciones B-spline, por lo tanto, es natural expandir las soluciones en dicha base

$$\phi_n(r) = \sum_{i=1}^{N} \alpha_i\,\varphi_i(r)\,, \tag{3.7}$$

dónde $\varphi_i(r)$ es el i-ésimo B-spline de orden k. Armar la base requiere una secuencia de knots que dependan de los siguientes parámetros: un conjunto de puntos de malla llamados secuencia de breakpoints definidos en $[0, r_{max}]$, el orden k de los polinomios y las condiciones de continuidad en cada breakpoint. Todos estos parámetros se eligen a partir de las propiedades del problema físico.

Para un determinado momento angular l, los estados atómicos (energía y función de onda) que satisfacen Ec. 3.6 se calculan resolviendo el sistema de N ecuaciones lineales obtenidas al sustituir Ec. 3.7 en Ec. 3.6 y proyectándose sobre α_i. Escrito en forma de matriz, este procedimiento es equivalente a resolver el siguiente sistema de autovalores generalizado

$$H_l\vec{\alpha} = E\,S\,\vec{\alpha}\,, \tag{3.8}$$

para E y $\{\alpha_i\}_1^N$, dónde

$$\begin{aligned}(H_l)_{ij} = &-\frac{1}{2}\int_0^{r_{max}} \varphi_i(r)\frac{d^2}{dr^2}\varphi_j(r)\,dr + \frac{l(l+1)}{2}\int_0^{r_{max}} \frac{\varphi_i(r)\,\varphi_j(r)}{r^2}\,dr \\ &+ \int_0^{r_{max}} \varphi_i(r)\,V(r)\,\varphi_j(r)\,dr \end{aligned} \tag{3.9}$$

$$(S)_{ij} = \int_0^{r_{max}} \varphi_i(r)\,\varphi_j(r)\,dr \tag{3.10}$$

La matriz de solapamiento S se origina del hecho de que los B-splines no forman un conjunto ortonormal de funciones base. Todos los elementos de las distintas matrices en las ecuaciones 3.9 se calculan numéricamente usando una cuadratura de *Gauss Lengedre* (GL). Si se consideran n puntos para evaluar la cuadratura (también conocida como cuadratura de orden n), es posible calcular en forma exacta integrales de polinomios de grado menor o igual a $2n-1$ en un intervalo cerrado. Puesto que un B-spline de orden k es un polinomio particular de grado $k-1$ en cada segmento, se ve fácilmente que los integrandos del elemento $(S)_{ij}$ y el primer término de la matriz H_l son polinomios con

máximo grado $2k-2$. La aplicación del procedimiento GL en cada segmento conduce a una evaluación numérica exacta de estas dos integrales. El caso de la energía cinética angular y el término del potencial, segundo y tercer términos de H_l en Ec. 3.9 respectivamente son ligeramente diferentes. Aquí el integrando ya no es un polinomio de algún grado sino más bien una fracción racional y la afirmación anterior no es válida. Sin embargo, se puede demostrar que un pequeño incremento en el número de puntos de la cuadratura da como resultado una convergencia inmediata a la precisión del punto flotante de la computadora. Por lo tanto, todos los elementos de la matriz se calculan de esta forma.

Otra propiedad clave directamente relacionada con el uso de la base B-spline deriva del hecho de que los B-splines son funciones compactas: un B-spline de orden k difiere de cero solamente en k segmentos sucesivos. Por lo tanto, cualquier operador local expresado en la base B-spline aparece como una matriz que tiene valores distintos de cero en una banda diagonal de ancho $2k-1$. Además el Hamiltoniano, así como la matriz de superposición son matrices simétricas. Resulta así que para un dado k, en vez de N^2 elementos de matriz sólo se necesitan calcular y almacenar Nk elementos en la memoria.

Resolver el sistema ahora es sencillo si se tiene en cuenta el hecho de que la matriz de superposición (o solapamiento) S es definida positiva. El sistema generalizado de banda simétrica se puede transformar entonces en un sistema regular con la misma estructura de banda. Una diagonalización directa del nuevo sistema proporciona todas los autovalores (que serían las autoenergías), pero este procedimiento no es capaz de recuperar las funciones propias correspondientes ya que la matriz de transformación (matriz N^2 completa) ha sido eliminada. Teniendo todos los valores propios, los vectores propios se calculan uno por uno en un segundo paso por iteración inversa cuando sea necesario. Este procedimiento nunca requiere más que tres iteraciones para un error relativo del orden la precisión de la máquina.

3.1.1. Sistema de dos partículas

Para considerar un sistema de dos partículas es necesario escribir un operador Hamiltoniano que actúe sobre las variables de ambas partículas, en este caso, el operador es

$$\mathcal{H}_{2p} = \mathcal{H}_{1p}^{(1)} + \mathcal{H}_{1p}^{(2)} + U(\vec{r}_1, \vec{r}_2)\,, \tag{3.11}$$

dónde $\mathcal{H}_{1p}^{(i)}$ es el operador Hamiltoniano que actúa sobre la partícula i ($i = 1, 2$) dado en la ecuación 3.2 y $U(\vec{r}_1, \vec{r}_2)$ es el operador que representa la interacción entre las partículas y en general es una función que depende del módulo de la diferencia entre $\vec{r}_1$ y $\vec{r}_2$, en otras palabras sólo depende de la distancia entre las partículas, es decir $U(\vec{r}_1, \vec{r}_2) = U(|\vec{r}_1 - \vec{r}_2|)$.

Para este sistema, la equación diferencial que se tiene que resolver es

$$\mathcal{H}_{2p}\psi_n(\vec{r}_1, \vec{r}_2) = E_n^{2p}\psi_n(\vec{r}_1, \vec{r}_2)\,, \tag{3.12}$$

dónde $\psi_n(r_1, r_2)$ es la función de onda del sistema de dos partículas. En particular, como las partículas son cuánticas, la función de onda tiene que satisfacer condiciones de simetría (además de las condiciones de contorno). Estas condiciones de contorno pueden ser que la función de onda sea simétrica al intercambio de coordenada, $\psi(\vec{r}_2, \vec{r}_1) = \psi(\vec{r}_1, \vec{r}_2)$, o bien antisimétrica al intercambio de coordenadas, $\psi(\vec{r}_2, \vec{r}_1) = -\psi(\vec{r}_1, \vec{r}_2)$.

Para resolver el problema usando el método variacional de Rayleigh-Ritz es necesario elegir una base de funciones para el cálculo de las matrices y pasar de una ecuación diferencial a una ecuación de autovalores y autovectores. Es recomendable que dicho conjunto de funciones cumpla con las condiciones de simetría del problema. Como estamos interesados en el caso de funciones de onda simétricas elegimos como base una combinación de B-splines que sean simétricas

$$\chi_{i,j}(r_1, r_2) = \begin{cases} \varphi_i(r_1)\,\varphi_j(r_2) & \text{si} \quad i = j \\ \frac{\varphi_i(r_1)\,\varphi_j(r_2) + \varphi_i(r_2)\,\varphi_j(r_1)}{\sqrt{2}} & \text{si} \quad i \neq j \end{cases} , \tag{3.13}$$

dónde $\varphi_i(r)$ es el i-ésimo B-spline de orden k. Las funciones definidas en Ec. 3.13 cumple con la condición de simetría $\chi_{i,j}(r_2, r_1) = \chi_{i,j}(r_1, r_2)$ para todo par de valores i, j. Para el cálculo de las matrices involucradas es más simple, primero calcular dichas matrices usando la base

$$\hat{\chi}_{i,j}(r_1, r_2) = \varphi_i(r_1)\,\varphi_j(r_2)\,, \tag{3.14}$$

y luego hacer el cambio de variable a la base simétrica de la EC 3.13. En esta base la matriz del operador Hamiltoniano queda

$$\begin{aligned} H_{i,i';j,j'} &= \langle \hat{\chi}_{i,j} | \mathcal{H}_{2p} | \hat{\chi}_{i',j'} \rangle \\ &= (H_l)_{i,i'}\, S_{j,j'} + S_{i,i'}\, (H_l)_{j,j'} + U_{i,i';j,j'}\,, \end{aligned} \tag{3.15}$$

dónde las matrices (H_l) y S están dadas por las integrales en la EC 3.9, mientras que la matriz de la interacción entre las partículas está dada por

$$\begin{aligned} U_{i,i';j,j'} &= \langle \chi_{i,j} | U(r_1, r_2) | \chi_{i',j'} \rangle \\ &= \int_0^{r_{max}} \left(\int_0^{r_{max}} \varphi_i(r_1)\,\varphi_j(r_2)\,U(r_1, r_2)\,\varphi_{i'}(r_1)\,\varphi_{j'}(r_2)\,dr_2 \right) dr_1\,. \end{aligned} \tag{3.16}$$

En general, la interacción entre las partículas es una función que depende de la distancia entre las mismas, $|\vec{r}_1 - \vec{r}_2|$, por lo tanto el cálculo de la integral en Ec. 3.16 puede ser bastante complicado. Una forma de calcular dichas integrales y que además se puede aplicar a un gran número de problemas es separar el integrando sobre la variable r_2 en dos, es decir, podemos definir las funciones

$$\begin{aligned} f_{j,j'}(r_1) &= \int_0^{r_1} \varphi_j(r_2)\,U(r_1, r_2)\,\varphi_{j'}(r_2)\,dr_2\,, \\ g_{j,j'}(r_1) &= \int_{r_1}^{r_{max}} \varphi_j(r_2)\,U(r_1, r_2)\,\varphi_{j'}(r_2)\,dr_2\,, \end{aligned} \tag{3.17}$$

las cuáles son integrales de una sóla variable y por lo tanto pueden ser calculadas aplicando el mismo método que se usa para las demás matrices. Luego, para evaluar los elementos de matriz de $U(r_1, r_2)$ hay que calcular una nueva integral, que es de la forma

$$U_{i,i';j,j'} = \int_0^{r_{max}} \varphi_i(r_1)\,f_{j,j'}(r_1)\,\varphi_{i'}(r_1)\,dr_1 + \int_0^{r_{max}} \varphi_i(r_1)\,g_{j,j'}(r_1)\,\varphi_{i'}(r_1)\,dr_1\,. \tag{3.18}$$

3.2. Implementación númerica

Este método, tanto para una partícula como para dos, tiene dos partes bien diferenciadas una es construir el sistema representado en las EC 3.8 o 3.12 y la otra es resolver dicho sistema. Para la resolución del sistema se utilizaron bibliotecas de terceros [2, 16] y para armar el sistema hay que calcular las matrices involucradas. Este cálculo está expresado en las fórmulas 3.9 y 3.15 para para una y dos partículas respectivamente. Las itegrales relacionadas se aproximaron con el método *Gauss Legendre*.

Llamemos v_0 al tercer término de la derecha la Ec. 3.9 y ke los dos primeros términos de 3.9 y s representa la matriz S en 3.10. Llamaremos INT_G el grado de cuadratura, L_INT la cantidad de intervalos, este parámetro define el tamaño y por tanto la presición del cálculo.

3.2.1. Primera implementación para una partícula

Para el armado de las matrices se puden diferenciar dos procesos o *bloques* secuenciales, el primero realiza un cálculo auxiliar y el segundo calcula las matrices de la Ec. 3.9 y 3.10 con *Cuadratura Gaussiana*.

En el primer bloque (ver algoritmo 7) se calcula: los knots de los B-splines con una distribución uniforme (ver lineas 9 hasta 22), las abscisas y pesos de la cuadratura de *Gauss-Legendre* para cada *punto de integración*, estos puntos son L_INT intervalos tomados uniformemente entre los límites r_{min} y r_{max} de las integrales de la Ec. 3.9 (ver líneas 3 hasta 8).

En el segundo bloque (ver algoritmo 8) se calculan las matrices de solapamiento s, la matriz de potencial v_e y la matriz de energía ke. Las integrales de cada punto de la matriz se realizan con la cuadratura de *Gauss-Legendre* sobre B-splines, esto se puede ver en las líneas 3 hasta 23, notar que en la línea 6 calculamos el valor de la función en ese punto (las dependencias del mismo se dan en el índice son j e i) y luego realizamos la operación con los pesos correspondiente a cada parte de la matriz. Cada valor de B-spline tiene un cierto peso dado por la cuadratura según la posición de la matriz.

Luego se integra de la misma manera con las dervidas de los B-splines (lineas 24 hasta 38) como está escrito en la Ec. 3.9. Notar que la evaluación de los B-splines (o más bien sus derivadas) son calculados dentro del ciclo que realiza la integracion por *Gauss-Legendre* por tanto esa función es evaluada más de una vez. Más adelante veremos cómo optimizamos esto con una factorización de código.

3.2.2. Primera implementación para dos partículas

El armado de las matrices para dos partículas se puede ver en tres partes (*o bloques*): la primera es calcular las matrices (H_l) y S del problema de una partícula como fué descripto más arriba, la segunda es calcular la interacción $U_{i,i';j,j'}$ y luego armar las matrices en una base simétrica según Ec. 3.13.

La Interacción (*segunda parte*) se describe en el algoritmo 9. En esta rutina se realiza el cálculo del tensor de la Ec. 3.16, integrando con *Gauss-Legendre*. El tensor $U_{i,i';j,j'}$ es calculado parcialmente en cada vuelta del ciclo de la línea 3, entonces sólo se calculan las funciones $f^{i,i'}_{j,j'}$ y $g^{i,i'}_{j,j'}$ del ciclo i, i' (ver Ec. 3.17). Como estas funciones son computadas

Result: Cálculo auxiliar x, w, t y k

1 $dr = \frac{(r_{max} - r_{min})}{L\_INT}$
2 $nk = L\_INT + 2 * KORD - 1$
3 **for** $i \in [0, L\_INT)$ **do**
4 $\quad ri = r_{min} + i * dr$
5 $\quad rf = ri + dr$
6 $\quad x[i] = abscisas\ del\ método\ de\ gauss - legendre\ en\ el\ intervalo\ [ri, rf]$
7 $\quad w[i] = pesos\ del\ método\ de\ gauss - legendre\ en\ el\ intervalo\ [ri, rf]$
8 **end**
9 $t[0] = r_{min}$
10 $k[0] = 0$
11 **for** $i \in [1, KORD - 1]$ **do**
12 $\quad t[i] = t[i-1]$
13 $\quad k[i] = k[i-1]$
14 **end**
15 **for** $i \in [KORD, KORD + L\_INT]$ **do**
16 $\quad t[i] = t[i-1] + dr$
17 $\quad k[i] = k[i-1] + 1$
18 **end**
19 **for** $i \in [KORD + L\_INT, nk]$ **do**
20 $\quad t[i] = t[i-1]$
21 $\quad k[i] = k[i-1]$
22 **end**

Algoritmo 7: Cálculo auxiliar

con el método de *Gauss-Lengre* en las líneas 8 hasta 29, se realiza un efoque similar al explicado en la sección 3.2.1 sólo hay que tener cuidado con los límites de integración. Luego se realiza el resto de la integración en las líneas 31 hasta 45 con el método de *Gauss-Legendre* tomando como función $U_{i,i'} = \sqrt{s_{i,i} * s_{i',i'}}$.

La tercera parte se trata de construir el *Hamiltoniano* y la matriz de *Solapamiento*. Para ésto se utiliza la una nueva base explicada en 3.13. Entonces si ya se tienen calculado U y S (matriz de solapamiento para una párticula) luego resta realizar el cálculo descripto en 3.15 variando los puntos η. Este cálculo es descripto en el algoritmo 10. Las condicionales que están en las líneas 12, 15, 18 y 21 indican cómo debe ser calculada la base que fue explicada en 3.13 según el caso.

Result: Cálculo de las matrices s, v_0 y ke

$nb = L\_INT + KORD - 3$ ▷ tamaño de la base
$ma = 0{,}5 * lmax * (lmax + 1)$ ▷ $lmax$ es el momento angular
for $i \in [KORD - 1, KORD + L\_INT - 1)$ **do**
  for $j \in [0, INT\_G)$ **do**
    $rr = x[k[i], j]$
    $sp =$ evaluar los b-splines en el punto rr
    for $m \in [0, KORD)$ **do**
      $im = i - KORD + m$
      if $0 \leq im < nb$ **then**
        for $n \in [0, KORD - 1]$ **do**
          $in = i - KORD + n$
          if $0 \leq in < nb$ **then**
            s_{ij} += sp[m] * sp[n] * w[k[i], j]
            $ke_{ij} += \frac{ma * sp[m] * sp[n] * w[k[i],j]}{rr * rr}$
            if $r_{min} < rr < r_{max}$ **then**
              $v_{0i,j}$ += sp[m] * sp[n] * w[k[i], j]
            end
          end
        end
      end
    end
  end
end
for $i \in [0, KORD + L\_INT - 1)$ **do**
  for $m \in [i - KORD + 1, i]$ **do**
    if $0 \leq m < nb$ **then**
      for $n \in [m, i]$ **do**
        if $0 \leq n < nb$ **then**
          for $j \in [0, INT\_G)$ **do**
            $rr = x[k[i], j]$
            $bm =$ *derivada del* $b -$ *spline en el punto* rr *en el índice* m
            $bn =$ *derivada del* $b -$ *spline en el punto* rr *en el índice* n
            $ke_{ij} += \frac{0{,}5 * w[k[i],j] * bm * bn}{me}$ ▷ me es la masa de la partícula
          end
        end
      end
    end
  end
end

Algoritmo 8: Cálculo de matrices

Result: Cálculo de la interacción Vef

$Vef = 0$ ▷ Tensor de dimensión N^4, que representa $H_{i,i';j,j'}$

$nb = L\_INT + KORD - 3$ ▷ tamaño de la base

for $i \in [KORD - 1, KORD + L\_INT - 1)$ **do**
  for $abs \in [0, INT\_G)$ **do**
    $rr_1 = x[k[i], abs]$
    $w_1 = w[k[i], abs]$
    $f = 0$ ▷ Matriz N^2
    $g = 0$ ▷ Matriz N^2
    for $i' \in [KORD - 1, KORD + L\_INT - 1)$ **do**
      for $abc' \in [0, INT\_G)$ **do**
        $rr_2 = x[k[i'], abc']$
        $w_2 = w[k[i], abs']$
        $sp =$ evaluar los b-splines en el punto rr_2
        for $m \in [0, KORD)$ **do**
          $j = i' - KORD + m$
          if $0 \leq j < nb$ **then**
            for $n \in [0, KORD)$ **do**
              $j' = i' - KORD + n$
              if $0 \leq j' < nb$ **then**
                if $rr_2 \leq rr_1$ **then**
                  $f_{j,j'}$ += $sp[m] * sp[n] * w_2 / rr_1$
                else
                  $g_{j,j'}$ += $sp[m] * sp[n] * w_2 / rr_2$
                end
              end
            end
          end
        end
      end
    end
    $sp =$ evaluar los b-splines en el punto rr_1
    for $m \in [0, KORD)$ **do**
      $im = i' - KORD + m$
      if $0 \leq im < nb$ **then**
        for $m' \in [1, KORD)$ **do**
          $im' = i - KORD + m' - 1$
          if $0 \leq im' < nb$ **then**
            for $j \in [1, nb)$ **do**
              for $j' \in [1, nb)$ **do**
                $Vef_{im,im';\ j,j'}$ += $\frac{sp[m] * sp[m'] * w_1 * (f_{j,j'} + g_{j,j'})}{\sqrt{s_{j,j} * s_{j',j'}}}$
              end
            end
          end
        end
      end
    end
  end
end

Algoritmo 9: Interacción

Data: s, v_0, ke, Vef
Result: Cálculo de la simetrización, $hsim_{eta}$, ms_{eta}

```
1  nb = L_INT + KORD − 3                                   ▷ tamaño de la base
2  mh = ke − v_0
3  i = 0
4  for η ∈ puntos η do
5      hsim = 0
6      ms = 0
7      for n ∈ [0, nb) do
8          for m ∈ [n, nb) do
9              j = 0
10             for n' ∈ [0, nb) do
11                 for m' ∈ [n', nb) do
12                     if m = n and m' = n' then
13                         hsim_{i,j} = 2 * s_{n,n'} * mh_{n,n'} + η * Vef_{n,n:n',n'}
14                         msi, j = s_{n,n'} * sn, n'
15                     else if m ≠ n and m' = n' then
16                         hsim_{i,j} = 1/√2 * (2 * s_{m,n'} * mh_{n,n'} + 2 * s_{n,n'} * mh_{m,m'} + η *
                             Vef_{m,n:n',n'} + η * Vef_{n,m:n',n'})
17                         ms_{i,j} = 2 * 1/√2 * s_{n,n'} * sm, n'
18                     else if m = n and n' ≠ m' then
19                         hsim_{i,j} = 1/√2 * (2 * s_{n,m'} * mh_{n,n'} + 2 * s_{n,n'} * mh_{n,m'} + η *
                             Vef_{n,n:n',m'} + η * Vef_{n,n:m',n'})
20                         ms_{i,j} = 2 * 1/√2 * s_{n,n'} * s_{n,m'}
21                     else
22                         hsim_{i,j} = s_{n,n'} * mh_{m,m'} + s_{n,m'} * mh_{m,n'}+
23                         s_{m,m'} * mh_{n,n'} + s_{m,n'} * mh_{n,m'}+
24                         η * 0,5 * (Vef_{n,m:n',m'} + Vef_{m,n:n',m'} + Vef_{n,m:m',n'}, Vef_{m,n:m',n'})
25                         ms_{i,j} = s_{n,n'} * s_{m,m'} + s_{n,m'} + s_{m,n'}
26                     end
27                     j = j + 1
28                 end
29             end
30             i = i + 1
31         end
32     end
33     guardar sistema(hsim, ms)
34 end
```

Algoritmo 10: Simetrización

Capítulo 4

Modelos Computacional CPU y GPU

En esta sección explicaremos las nociones preliminares necesarias sobre las arquitecturas de la CPU y GPU, relacionadas con computación de alto desempeño. Veremos un pantallazo general sobre las mismas, nos detendremos detallando solo las características relevantes para este trabajo.

4.1. CPU

Sobre la arquitectura de CPU explicaremos su jerarquía de memoria y sus tiempos de accesos en términos de ciclos.

La unidad central de procesamiento (o CPU por sus siglas en inglés: central processing unit), es el hardware dentro de la computadora que realiza las operaciones básicas aritméticas, lógicas y de entrada/salida.

La secuencia clásica de una instrucción es la siguiente: buscar la instrucción en la memoria, Decodificar la instrucción, Ejecutar la instrucción y Almacenar resultados. Los dos primeros pasos se conocen como ciclo de búsqueda. El ciclo de búsqueda procesa la instrucción que contiene el código de operación y el operando. Los pasos 3 y 4 del ciclo de instrucción se conocen como ciclo de ejecución.

4.1.1. Jerarquía de memorias

La jerarquía de memoria es la organización piramidal de la memoria en niveles que tienen las computadoras. Aquí solo abordaremos las distintas memorias representadas en la figura 4.1.

4.1.1.1. Registros

Los registros están en la cima de la jerarquía de memoria y son la manera más rápida que tiene el sistema de almacenar datos. Los registros son una memoria de alta velocidad y poca capacidad, está integrada en el microprocesador y permite guardar y acceder a valores muy usados en operaciones matemáticas. Los registros se miden por lo general por el número de bits que almacena. Tipos de registros:

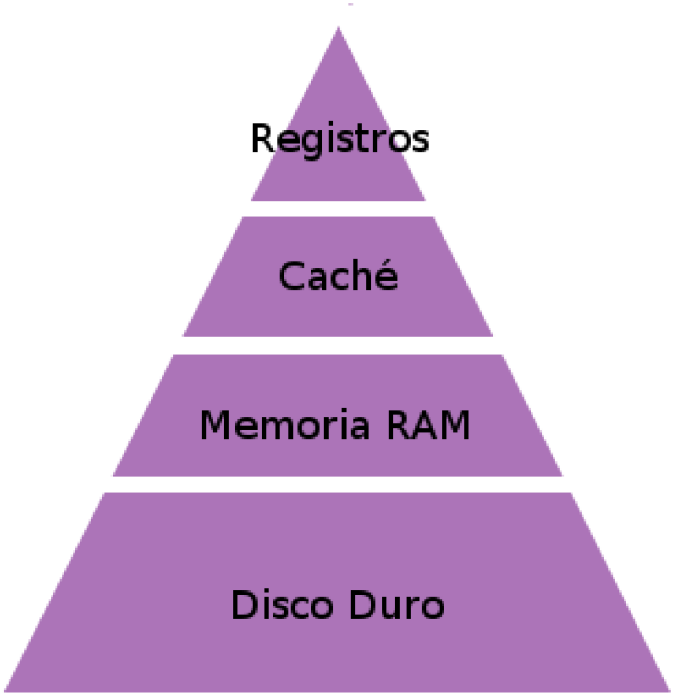


Figura 4.1: Diagrama piramidal de la jerarquía de memoria

1. De datos: usados para guardar números enteros.
2. De memoria: usados para guardar exclusivamente direcciones de memoria.
3. De propósito general: pueden guardar tanto datos como direcciones.
4. De punto flotante: usados para guardar datos en formato de flotante.
5. De propósito específico: guardan información específica del estado del sistema.
6. Constante: tiene valores creados por el hardware de sólo lectura.

4.1.1.2. Memoria Caché

Es una memoria rápida y pequeña, situada entre la memoria principal y el microprocesador, especialmente diseñada para contener información que se utiliza con frecuencia en un proceso con el fin de evitar accesos a otras memorias, reduciendo considerablemente el tiempo de acceso al ser más rápida que el resto de la memoria principal.

La memoria caché es una memoria en la que se almacena un serie de datos para su rápido acceso. La memoria caché de un microprocesador es de tipo volátil (del tipo RAM), pero de una gran velocidad. Su objetivo es almacenar una serie de instrucciones y datos a los que el microprocesador accede continuamente, con el fin de que estos accesos sean instantáneos.

Cuando la CPU necesita una palabra de memoria, se revisa la caché. Si se encuentra la palabra en caché, se lee de ahí mismo. Si la palabra direccionada de la CPU no se

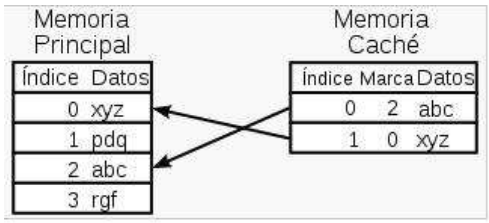


Figura 4.2: Mapeo de RAM a caché

encuentra en caché, se accede a la memoria principal para leer la palabra. Después se transfiere un bloque de palabras que contiene la palabra que se buscab de la memoria principal a la memoria caché. El tamaño de bloque puede variar de una palabra a cerca de 16 palabras adyacentes a la que se accedió. De esta manera se trasfieren algunos datos la caché para que las futuras referencias a memoria encuentren la palabra requerida en la memoria caché.

Hay tres tipos diferentes de memoria caché para microprocesadores:

1. Caché de primer nivel (L1): Integrada en el núcleo del microprocesador, trabajando a la misma velocidad que este. La cantidad de memoria caché L1 varía de un microprocesador a otro. Esta memoria suele a su vez estar dividida en dos partes dedicadas, una para instrucciones y otra para datos.

2. Caché de segundo nivel (L2): Integrada en el microprocesador, no directamente en el núcleo. Es algo más lenta que la caché L1 y suele ser mayor que dicha caché L1. Su utilización está más encaminada a programas que al sistema.

3. Caché de tercer nivel (L3): Es un tipo de memoria caché más lenta que la L2. En un principio esta caché estaba incorporada a la placa base, no al microprocesador, y su velocidad de acceso era bastante más lenta que una caché de nivel 2 o 1. Si bien sigue siendo una memoria de una gran velocidad (muy superior a la RAM y mucho más en la época en la que se utilizaba), depende de la comunicación entre el microprocesador y la placa base.

La memoria caché está estructurado por celdas, donde cada celda almacena un byte. La entidad básica de almacenamiento la conforman las filas, llamados también líneas de caché. Cuando se copia o se escribe información de la RAM por cada movimiento siempre cubre una línea de caché. La memoria caché tiene incorporado un espacio de almacenamiento llamado Tag RAM, que indica a qué parte de la RAM se halla asociada cada línea de caché, es decir, traduce una dirección de RAM en una línea de caché concreta.

4.1.1.3. Memoria RAM

La memoria de acceso aleatorio o memoria de acceso directo (Random Access Memory). Se compone de uno o más chips y se utiliza como memoria de trabajo para programas y datos. Es un tipo de memoria temporal que pierde sus datos cuando se queda sin energía (al apagar la computadora), por lo cual es una memoria volátil. Se trata de una memoria de semiconductor en la que se puede tanto leer como escribir información.

Se utiliza normalmente como memoria temporal para almacenar resultados intermedios y datos similares no permanentes. Se dicen *de acceso aleatorio* o *de acceso directo* porque los diferentes accesos son independientes entre sí. Las RAMs se dividen en:

1. Estáticas: mantiene su contenido inalterado mientras esté alimentada.

2. Dinámica: la lectura es destructiva, es decir que la información se pierde al leerla. Para evitarlo hay que restaurar la información contenida en sus celdas, operación denominada refresco.

4.1.1.4. Disco Duro

El disco duro (hard disk) es una unidad de almacenamiento magnético de la información. Es un disco metálico (normalmente de aluminio) recubierto con una capa de material magnetizable por sus dos caras (usualmente níquel). El disco duro magnético está dividido en pistas concéntricas. Cada pista se divide en igual número de bloques radiales denominados sectores. La capacidad de almacenamiento en bytes por cada pista es variable, dependiendo del tamaño de la misma y de la densidad de grabación. En todas las pistas de un mismo disco (desde las exteriores hasta las interiores) cabe la misma cantidad de información, lo que se consigue grabando con mayor densidad en las pistas interiores y menor densidad en las pistas exteriores.

Otro tipo de disco aparte del magnético y de uso masivo es la unidad de estado sólido, dispositivo de estado sólido o SSD (acrónimo inglés de Solid-State Drive). Es un tipo de dispositivo de almacenamiento de datos que utiliza memoria no volátil, como la memoria flash, para almacenar datos en lugar de los platos o discos magnéticos de las unidades de discos duros (HDD) convencionales. En comparación con los discos duros tradicionales, las unidades de estado sólido son menos sensibles a los golpes al no tener partes móviles, son prácticamente inaudibles y poseen un menor tiempo de acceso y de latencia, lo que se traduce en una mejora del rendimiento exponencial en los tiempos de carga de los sistemas operativos. En contrapartida, su vida útil es muy inferior ya que tienen un número limitado de ciclos de escritura, pudiendo producirse la pérdida absoluta de los datos de forma inesperada e irrecuperable. Los SSD hacen uso de la misma interfaz SATA que los discos duros por lo que son fácilmente intercambiables sin tener que recurrir a adaptadores o tarjetas de expansión para compatibilizarlos con el equipo.

4.1.1.5. Memoria virtual

Es un concepto que permite al software usar más memoria principal que la que realmente posee. Muchas aplicaciones requieren el acceso a más información (código y datos) que la que se puede mantener en memoria física. Esto es así sobre todo cuando el sistema operativo permite múltiples procesos y aplicaciones ejecutándose simultáneamente. Una solución al problema de necesitar mayor cantidad de memoria de la que se posee consiste en que las aplicaciones mantengan parte de su información en disco, moviéndola a la memoria principal cuando sea necesario. Aunque la memoria virtual podría estar implementada por el software del sistema operativo, en la práctica casi siempre se usa una combinación de hardware y software, dado el esfuerzo extra que implicaría para el microprocesador.

Cuando se usa Memoria Virtual, o cuando una dirección es leída o escrita por la CPU, una parte del hardware dentro de la computadora traduce las direcciones de memoria generadas por el software (direcciones virtuales) en:

1. La dirección real de memoria (la dirección de memoria física): la referencia a la memoria es completada, como si la memoria virtual no hubiera estado involucrada: el software accede donde debía y sigue ejecutando normalmente.

2. Una indicación de que la dirección de memoria deseada no se encuentra en memoria principal (llamado excepción de memoria virtual): el sistema operativo es invocado para manejar la situación y permitir que el programa siga ejecutando o aborte según sea el caso.

La memoria virtual es una técnica para proporcionar la simulación de un espacio de memoria mucho mayor que la memoria física de una máquina. Esta *ilusión* permite que los programas se ejecuten sin tener en cuenta el tamaño exacto de la memoria física. La ilusión de la memoria virtual está soportada por el mecanismo de traducción de memoria junto con una gran cantidad de almacenamiento rápido en disco duro. Así en cualquier momento el espacio de direcciones virtual hace un seguimiento de tal forma que una pequeña parte de él está en memoria real y el resto almacenado en el disco, pudiendo ser referenciado fácilmente.

4.1.2. Memorias RAM vs Cachés

A medida que la velocidad de los procesadores aumenta, el acceso a memoria es relativamente lento (en cuanto a cantidad de ciclos de CPU para hacer entrada salida). La solución al problema de tener memoria relativamente lenta es añadir almacenamiento en memoria caché y *prefetching*: la memoria caché proporciona un acceso rápido a datos utilizados frecuentemente y el *prefetching* precarga datos en cache si el patrón de acceso es predecible.

Cargar de cache toma algunos ciclos (depende de la jerarquía) y cargar desde memoria RAM toma cerca de 400 ciclos como se puede ver Cuadro 4.1. Si la CPU puede predecir los bloques de memoria que utilizará (*prefeching*) entonces la perdida de ciclos por traer memoria desde RAM es mucho menor. Por lo tanto hacer código amigable con la caché es importante en las rutinas que son *CPU bound*. El uso de patrones de acceso a la memoria predecible y el funcionamiento en trozos de datos que son más pequeños que la caché de la CPU obtendrá la mayores beneficios de las cachés modernas.

4.2. GPGPU

GPGPU es el término que se utiliza para designar las tareas de propósito general, típicamente pensadas para ser procesadas en una CPU, que aprovechan el potencial de la GPU para ejecutarse en ella. Dado que los procesadores gráficos son mucho más eficientes en cierto tipo de operaciones, los resultados se obtendrán más rápidamente.

El problema de la GPGPU es precisamente que no todas las tareas tienen que ser más eficientes en una GPU. Éstas están especializadas en tareas altamente paralelizables cuyos

Evento	Latencia	Escalado
1 Ciclo de CPU	0.3 ns	1 s
Nivel 1 acceso a cache	0.9 ns	3 s
Nivel 2 acceso a cache	2.8 ns	9 s
Nivel 3 acceso a cache	12.9 ns	43 s
Acceso a Memoria principal (DRAM, from CPU)	120 ns	6 min
Disco de Estado Sólido I/O (flash memory)	50-150 μs	2-6 días
Disco Rotacional	1-10 ms	1 - 12 meses

Cuadro 4.1: Cuadro de latencias de los diferentes accesos a memoria

algoritmos puedan subdividirse, procesarse por separado para luego unir los subresultados y tener el resultado final.

4.2.1. CUDA como modelo de programación escalable

Las arquitecturas de CPU de múltiples núcleos y GPU significaron que los chips de procesadores convencionales fueran sistemas paralelos. Más aún, su paralelismo continua escalando con la ley de Moore. El desafío es lograr construir aplicaciones que utilicen este paralelismo y que de forma transparente escalen para aprovechar el incremento en el número de núcleos. Tal como las aplicaciones de procesamiento gráfico 3D escalan su paralelismo a GPU de múltiples núcleos.

El modelo de programación paralelo de CUDA está diseñado para sobreponerse a este desafío mientras facilita el aprendizaje con la utilización del estándar de C.

Su modelo proporciona tres tipos de abstracciones: una jerarquía de grupos de hilos (threads), memoria compartida y barreras de sincronización. Éstas son utilizadas por el programador a través de un número pequeño extensiones del lenguaje C.

Estas abstracciones proveen de paralelismo de datos de grano-fino y paralelismo de hilos, mezclado en medio de un paralelismo de datos de grano-grueso y paralelismo de tareas. Esto lleva al usuario a dividir el problema en subproblemas que puedan ser solucionados independientemente en paralelo por bloques de hilos y cada problema en partes más pequeñas que puedan ser resueltos de forma cooperativa en paralelo por todas los hilos de un mismo bloque.

Esta descomposición preserva la expresividad del lenguaje permitiendo a los hilos cooperar cuando solucionan cada subproblema y, al mismo tiempo, permite la escalabilidad automáticamente. De este modo, cada bloque de hilos puede ser asignado a cualquiera de los multiprocesadores disponibles en la GPU en cualquier orden y de forma concurrente o secuencial, permitiendo que el código CUDA pueda ejecutarse en cualquier número de multiprocesadores. Ésto permite que ólo el sistema de planificación deba conocer la cantidad física de multiprocesadores, como se ilustra en la Figura 4.3

4.2.1.1. Modelo de programación

En esta Sección presentaremos los conceptos principales del modelo de programación de CUDA C. CUDA C extiende el lenguaje estándar C, permitiendo al programador definir funciones llamadas *Kernels* que, cuando son llamadas, se ejecutan N veces en paralelo por

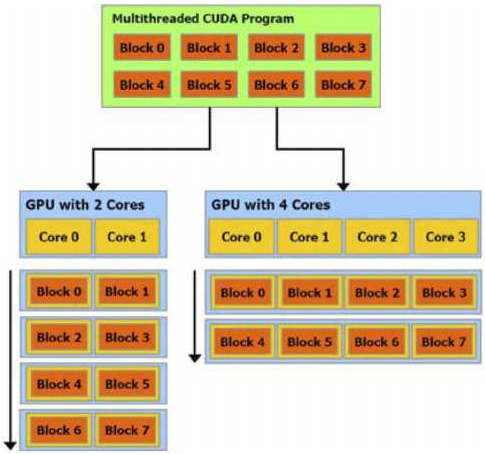


Figura 4.3: Escalabilidad Automática

N hilos de CUDA, a diferencia de sólo un hilo en una función regular de C. El programador es quién decide el valor dinámico o estático del parámetro N en el momento de ejecutar el kernel. A cada hilo que ejecuta un kernel se le asigna un identificador único el cual es accesible por el hilo dentro del kernel. Estos identificadores siguen los lineamientos de la jerarquía de hilos analizada a continuación.

Jerarquía de Hilos

Cada identificador de hilo puede ser visto como una 3-upla, por lo que cada hilo puede ser identificado utilizando un índice de una, dos o tres dimensiones formando así un bloque de hilos de una dos o tres dimensiones. Ésto provee una forma natural de mapear los identificadores de hilos con el accesos a datos. Hay un límite en el número de hilos por bloques ya que se espera que cada bloque de hilos resida en un mismo multiprocesador y debe compartir recursos de memoria limitados dentro del procesador. El número máximo de hilos por bloques es de 1024 en la arquitectura utilizada en este trabajo.

Los bloques son organizados en grillas de una, dos o tres dimensiones como se ilustra en la Figura 4.4. El número de bloques de hilos en una grilla está normalmente limitado directamente por le tamaño de los datos a procesar o el número de procesadores en el sistema.

Cada bloque dentro de una grilla puede ser identificado por un índice de una, dos o tres dimensiones (según haya sido declarado) y accesible dentro del kernel a través de una variable predefinida. Del mismo se puede acceder a las dimensiones del bloque de hilos y a la grilla de bloques.

Los hilos dentro de un bloque pueden cooperar entre ellos compartiendo datos a través de memoria compartida y sincronizando su ejecución para coordinar el accesos a esta memoria. Para que la cooperación sea eficiente, se requiere que el acceso a memoria compartida tenga baja latencia y la sincronización no tenga una gran penalización.

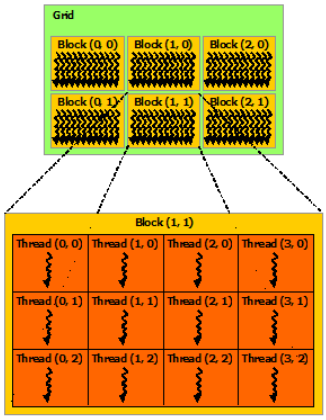


Figura 4.4: Grilla de bloques de hilos

Jerarquía de Memoria

Los hilos de CUDA pueden acceder a diferentes espacios de memoria durante su ejecución como se ilustra en la Figura 4.5. Cada hilo dispone de memoria local. Cada bloque de hilos dispone de memoria compartida visible por todos los hilos de un mismo bloque. Todos los hilos tiene acceso a la misma memoria global.

Hay adicionalmente dos memorias de sólo lectura accesible por todos los hilos: memoria constante y memoria de textura. La memoria global, memoria de textura y memoria constante están optimizadas para diferentes usos. La memoria de textura ofrece un modo de acceso y filtrado de datos para formatos de memoria específicos. No cubriremos este tipo de memoria ya que no es utilizada en el trabajo.

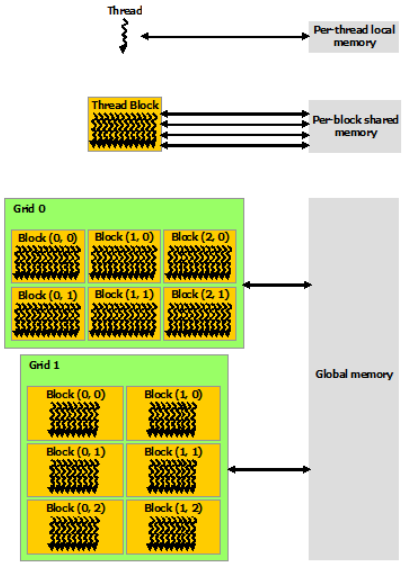


Figura 4.5: Jerarquía de memoria

Programación Heterogénea

El modelo de programación de CUDA asume que los hilos de CUDA ejecutan en un dispositivo físicamente separado que opera como un coprocesador del *host* que está ejecutando la aplicación. Por lo general (y para el análisis de nuestro trabajo) esta aplicación está escrita en C/C++ utilizando el lenguaje CUDA. El código kernel será ejecutado específicamente en la GPU y el resto del programa se ejecutará en el procesador central o CPU.

Además, el modelo computacional de CUDA asume que tanto el host como el dispositivo manejan distintos espacios de memoria referidos como *memoria de host* y *memoria de dispositivo* respectivamente. CUDA provee una API completa para manejar la memoria de dispositivo y poder ser alocada, escrita y leída por el host.

4.2.2. Implementación del hardware CUDA GPGPU

La arquitectura de NVIDA GPU está construida alrededor de un arreglo de procesadores de flujo de múltiples hilos o *Streaming Multiprocessors (SMs)*. Cuando un programa CUDA está ejecutando en host e invoca la ejecución de una grilla de kernels, los bloques de la grilla son numerados y distribuidos a los multiprocesadores disponibles para su ejecución. Los hilos de un bloque ejecutan concurrentemente en un mismo multiprocesador y múltiples bloques de hilos pueden ejecutar de forma concurrente en un mismo multiprocesador. A medida que los bloques de hilos terminan, nuevos bloques son asignados a los multiprocesadores vacantes.

Los multiprocesadores están diseñados para ejecutar cientos de hilos de forma concurrente. Para manejar este número de hilos, éstos utilizan una arquitectura llamada *SIMT (Single Instruction, Multiple Thread)*

4.2.2.1. Arquitectura SIMT

Los multiprocesadores crean, manejan, planifican y ejecutan en paralelo grupos de 32 hilos llamados *warps*. Cada hilo de un warp comienzan juntas en el mismo punto del programa, pero cada uno tiene su propio contador de instrucciones, registros de estados y son libres de ejecutar independientemente.

Cuando un multiprocesador posee uno o más bloques de hilos para ejecutar, éste parte los bloques en warps y cada warp es planificado por un planificador de warps para ser ejecutada. La forma en que los bloques son divididos en warps es siempre la misma: cada bloque contiene hilos con identificadores numéricos asignados de forma consecutiva. El primer warp contiene los hilos con identificadores 0 a 31, la segunda warp los hilos 32 a 63 y así sucesivamente.

Cada hilo dentro de un mismo warp ejecuta una misma instrucción al mismo tiempo, por lo tanto el rendimiento óptimo se consigue cuando los 32 hilos de un warp siguen el mismo camino de ejecución. Si los hilos de un warp divergen en el flujo de ejecución, la ejecución de los hilos del warp son serializados, deshabilitando los hilos que no están en el flujo de ejecución. Cuando los posibles caminos convergen, todos los hilos vuelven al mismo punto del programa. Ésto sólo ocurre entre hilos de un mismo warp. Diferentes warp ejecutan independientemente.

Si analizamos la corrección del programa, el programador puede esencialmente ignorar el comportamiento de la arquitectura SIMT. Sin embargo, se pueden conseguir sustanciales mejoras de rendimiento teniendo en cuenta la forma en que los hilos son agrupados y cómo es el comportamiento de ellos en los warps. En la práctica, esto es análogo a cómo se comporta la caché. El tamaño de caché puede ser ignorado en la corrección del diseño, pero debe ser considerado en la estructura del código para conseguir el rendimiento máximo. La arquitectura SIMT, requiere de ciertos cuidados al acceder a la memoria y manejar la divergencia de los hilos. Éstos serán analizados a más adelante.

4.2.2.2. Características del Hardware Multi-hilo

El contexto de ejecución de cada warp (contadores de programa, registros, etc.) es mantenido en la memoria interna de cada multiprocesador lo largo de la vida del warp. Ésto implica que cambiar de un contexto de ejecución a otro no tiene costo, hecho que es aprovechado para que los multiprocesadores mantengan un conjunto de warps activas para la ejecución y el planificador de ejecución del multiprocesador elija cuál es el siguiente warp a ejecutar. El modo de manejar la ejecución de las warps es una gran ventaja en el diseño de la arquitectura, permitiendo ocultar de forma óptima la latencia de lectura y escritura a memoria, siempre y cuando el multiprocesador tenga suficientes warps disponibles para la ejecución.

En particular, cada multiprocesador contiene una conjunto de registros de 32-bits que son divididos a lo largo de los warps y una caché de datos y memoria compartida que es dividida a lo largo de los bloques de hilos. El número de bloques y warps que pueden residir y ser procesados al mismo tiempo dentro de un multiprocesador para un programa dado, depende de la cantidad de registros y memoria compartida utilizada para el programa y la cantidad de registros y memoria compartida disponibles en el multiprocesador. También existe un número máximo de bloques residentes y de warps residentes en cada multiprocesador.

Para comprender este hecho, veremos un ejemplo concreto en la arquitectura específica utilizada en el trabajo. Hablamos de la arquitectura Kepler de NVIDA. En esta arquitectura, el tamaño de los warps es 32 y cada multiprocesador posee 256 KB de memoria de registros y memoria compartida programable en 16, 32 o 48 KB. Supongamos que poseemos un kernel que utiliza 25 registros locales de 32 bit y cada bloque lanzado es de 256 hilos. Cada bloque necesita de $256 \times 25 \times 4 = 25KB$ lo cuál nos indica que no puede haber más de 10 bloques simultáneamente en el mismo SM. De haberlo, el multiprocesador se quedaría sin memoria local. Recordemos que cada hilo necesita que sus valores locales persistan en memoria local a lo largo de su ejecución para permitir que el planificador los saque y ponga en ejecución rápidamente. Del mismo modo si los SM están configurados para tener 48KB de memoria compartida y cada bloque utiliza 12KB de esta memoria, no puede haber más de 4 bloques simultáneamente en el mismo SM. De estos dos parámetros analizados, para determinar en tiempo de compilación cuántos bloques pueden residir en cada SM, se concluye que el mínimo entre ambos deber ser el valor final.

De lo analizado anteriormente se desprende un valor de utilización de los multiprocesadores o *occupancy* que es el porcentaje entre la cantidad de bloques de un kernel en particular que puede manejar cada cada multiprocesador y la cantidad máxima de bloques determinados por la arquitectura. En el caso de la arquitectura Kepler el número máximo

de bloques por SM es 16. Así, Occupancy es un valor entre 0 y 1. Mientras más cerca de 1 se encuentre, no significará que el código será más eficiente ya que esto depende de la combinación de muchos factores, pero determina cuán ocupado estarán los SM, permitiendo así mejorar el ocultamiento de latencia de accesos a memoria entre otras cosas.

4.2.3. Técnicas de Rendimiento

Para lograr conseguir el máximo rendimiento de la arquitectura GPU es necesario adaptar el problema para seguir algunos lineamentos de la arquitectura. En nuestro problema trataremos de conseguir :

1. Maximizar la ejecución en paralelo para alcanzar la máxima utilización.

2. Optimizar el uso de la memoria para alcanzar el máximo ancho de banda.

Para lograr la máxima utilización debemos separar el problema en bloques lo más independientes posibles para que éstos puedan ser mapeados a diferentes componentes del sistema y mantener estos componentes lo más ocupados posible. A nivel multiprocesador, como ya explicamos, es importante que haya muchas warps activas dispuestas a ejecutar para poder ocultar la latencia de acceso a memoria. Además, es necesario que los threads de un mismo warp minimicen las bifurcaciones y las sincronización como barreras o mutex de escritura de memoria.

En cuanto a utilización de memoria, el primer paso es tratar de maximizar el rendimiento en los accesos a memoria de bajo ancho de banda, es decir, memoria que reside en el dispositivo. Las técnica más utilizada es diseñar el algoritmo para minimizar el acceso a memoria global y utilizar la memoria compartida como una caché intermedia entre la lectura - operación - escritura. El esquema básico sería :

1. Cargar los datos de memoria global a memoria local.

2. Sincronizar todas los threads del bloque de tal modo que cada thread pueda acceder a la memoria cargada por otro thread de forma segura.

3. Procesar los datos en memoria compartida.

4. Sincronizar nuevamente, si es necesario, para asegurar que todos las threads terminaron de procesar los datos.

5. Escribir los resultados nuevamente a memoria global.

Otro punto que mejora el rendimiento es seguir los patrones de accesos óptimos a memoria. Cada memoria tiene sus propias características.

La memoria global reside en memoria del dispositivo, esta memoria es accedida a través de transacciones de 16, 32 y 64 bytes. Dichas transacciones están alineadas. Cuando una warp ejecuta una instrucción que accede a memoria global, ésta genera las cantidad de transacciones necesarias dependiendo del tamaño de dato accedido de tal manera de poder satisfacer cada hilo y luego lo distribuye entre ellos. Por lo general, mientras más transacciones sean necesarias, más datos innecesarios son transferidos al warp y luego

desechados, empeorando el rendimiento. Por ello es importante que las instrucciones de acceso a memoria global sean hechas de tal forma que los datos necesarios por los hilos estén los más juntos posibles.

Los accesos a memoria local sólo ocurren para algunas variables automáticas las cuales son ubicadas en este espacio de memoria por el compilador. El espacio de memoria local reside en memoria de dispositivo, por lo tanto sus accesos tiene alta latencia y bajo ancho de banda. Además están sujetas a los mismos requerimientos de accesos que lo nombrado anteriormente en el acceso a memoria global. Ya que el acceso a esta memoria está controlada por el compilador, éste se encarga de generar los patrones de acceso que maximicen el rendimiento.

La memoria compartida reside en los multiprocesadores. Por ello los accesos a esta memoria tiene más baja latencia y más alto ancho de banda que la memoria local y la memoria global. Para maximizar el ancho de banda, la memoria compartida es dividida en módulos de igual tamaño, llamados bancos, los cuales pueden ser accedidos simultáneamente. Cualquier requerimiento de lectura o escritura realizado a n direcciones que caen en n bancos de memoria distintos pueden ser servidos simultáneamente. Del mismo modo, accesos simultáneos de varios hilos a posiciones distintas del mismo banco generan la serialización del acceso. Es importante destacar que si varios hilos acceden a la misma posición de memoria, el warp realiza una sola transacción y luego distribuye la información a todos los hilos que la requirieron.

La memoria constante y memoria de textura son memorias que residen en memoria de dispositivo, pero no analizaremos su patrón de acceso ya que este trabajo no hace uso de este tipo de memorias.

Capítulo 5

Optimización

Este trabajo tiene como objetivo la optimización de diferentes funciones y estructuras de datos de una implementación del método variacional de Rayleigh-Ritz explicado en la Sección 3, ya que esta implementación escala muy mal, necesitando gran cantidad de memoria y tiempo de procesamiento a medida que el tamaño del problema crece. A lo largo de este Capítulo mostraremos las diferentes modificaciones, tanto algorítmicas como de estructuras de datos más eficientes, para lograr que el programa tome menor tiempo y memoria, logrando una mejor escalabilidad.

5.1. Optimización en Memoria

La optimización en cuanto a almacenamiento se da en dos cambios:

- Evitar almacenar ceros en la matrices: almacenar la mínima cantidad de ceros posibles sin desmerecer la eficiencia.
- No almacenar datos que requieren unas pocas operaciones para ser recalculados: ir a buscar estos datos a memoria puede ser mas costoso que recalcularlos.

Para ésto se utilizó estructuras de matrices dispersas y se reemplazaron arreglos por funciones que devuelven el mismo valor dado un índice. Se utilizó dos tipos de estructuras: CDS explicadas en 2.4.3 y CCS explicadas en 2.4.1.

5.1.1. Matrices de Banda

Las matrices que estaban definidas en torno a la traza según la constante $KORD$ son $s, v_0, ke, f\ y\ g$ (ver Figura 5.5) donde todas son simétricas salvo f y g, por lo que se utilizó una estructura de matriz de banda. Estas matrices son comprimidas en un orden de $\mathcal{O}((L_{INT} + KORD)^2)$ a $\mathcal{O}((L_{INT} + KORD) \times KORD)$.

La variable $Vef_{i,j,k,l}$ es un tensor que puede ser comprimida en una matriz, si no se almacenan los elementos nulos, este tensor es comprimido en un orden de $\mathcal{O}((L_{INT} + KORD)^4)$ a $\mathcal{O}(((L_{INT} + KORD) \times KORD)^2)$, la distribución de no ceros de esa matriz se ve reflejado en las Figuras 5.3 y 5.4. Para mantener una interfaz limpia se realiza un redireccionamiento de índices. En esta variable el incide i y k tanto como el j y l están relacionadas de la siguiente manera: cuando los incides $|i - k| > KORD$ o $|j - l| > KORD$ $Vef_{i,j,k,l}$

vale 0, por tanto se puede aplicar un redireccionamiento análogo al de las matrices de banda.

5.1.2. Matrices de Almacenamiento de Columnas Comprimidas

Para las matrices $hsim$, ms, mv que no necesariamente están definidas sólo en torno a la traza (ver Figura 5.1) se utilizó este tipo de almacenamiento, que también es provechoso para el método de Arnoldi [16]. Estas matrices están comprimidas en un orden de $\mathcal{O}((L_{INT} + KORD)^4)$ a $\mathcal{O}((L_{INT} + KORD)^2)$

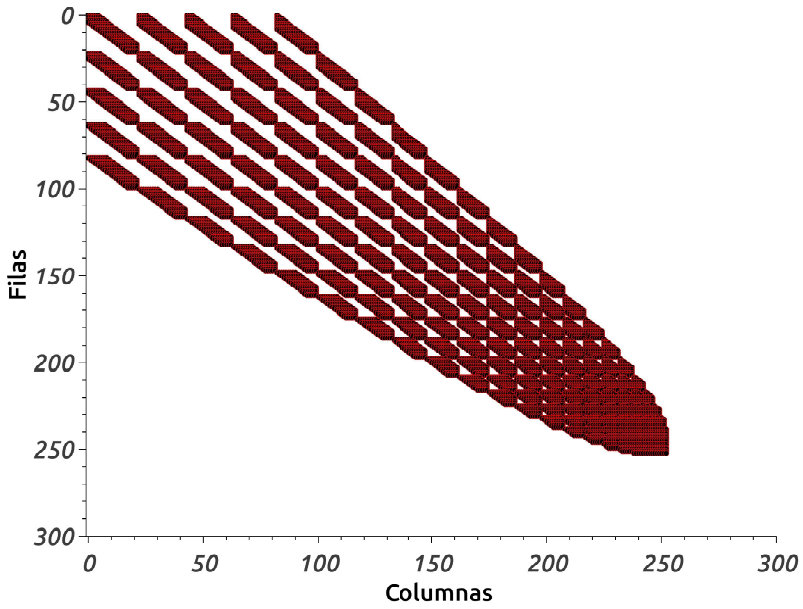


Figura 5.1: Distribución de elementos distintos de cero de las matrices $hsim$, ms, mv para $L_{INT} = 20$

5.2. Optimización en CPU

La optimización más grande se logró al reducir el espacio de cálculo de los ciclos (reduciendo las iteraciones de los ciclos), tomando sólo los valores distintos de cero de la matriz ya que éstos pueden saberse de antemano. Dado que la cantidad de los elementos no nulos es en orden menor al tamaño de los elementos nulos, permite una gran mejora en tiempo computacional, reduciendo la complejidad. Otras optimizaciones han sido por cacheo de resultados, factorización de código y cálculo de valores en vez de ser almacenados.

5.2.1. Reducción del Espacio de Cálculo

Para reducir el espacio de cálculo se ha restringido los valores de los índices a solo los lugares donde la matriz puede no ser cero. Para ello se estudió la estructura de cada

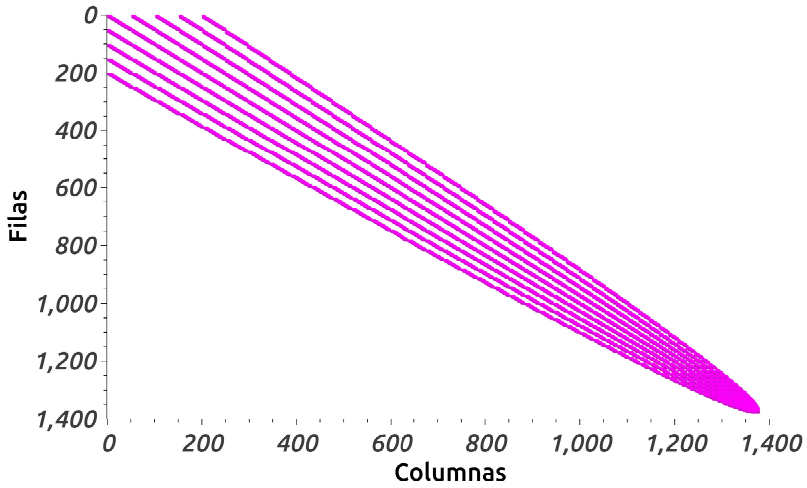


Figura 5.2: Distribución de elementos distintos de cero de las matrices $hsim$, ms, mv para $L_{INT} = 50$

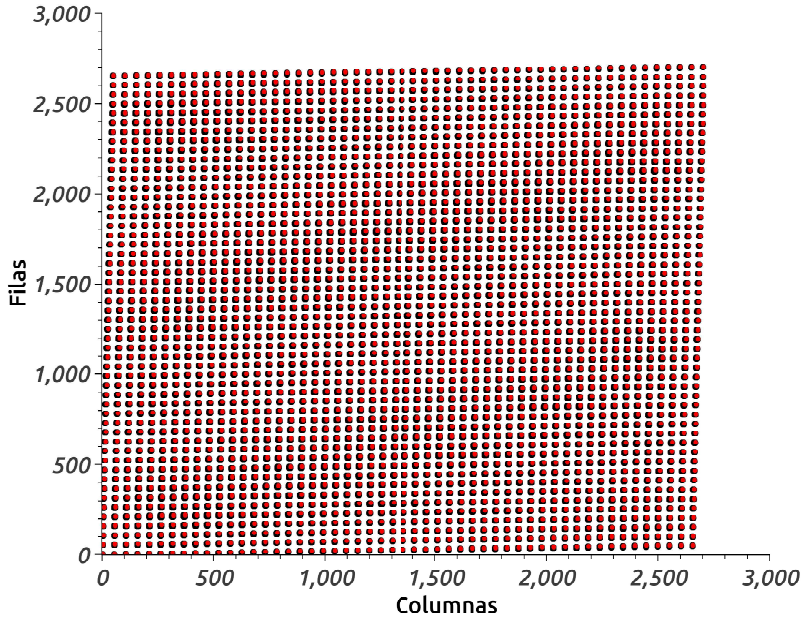


Figura 5.3: Distribución de elementos distintos de cero de la matriz Vef

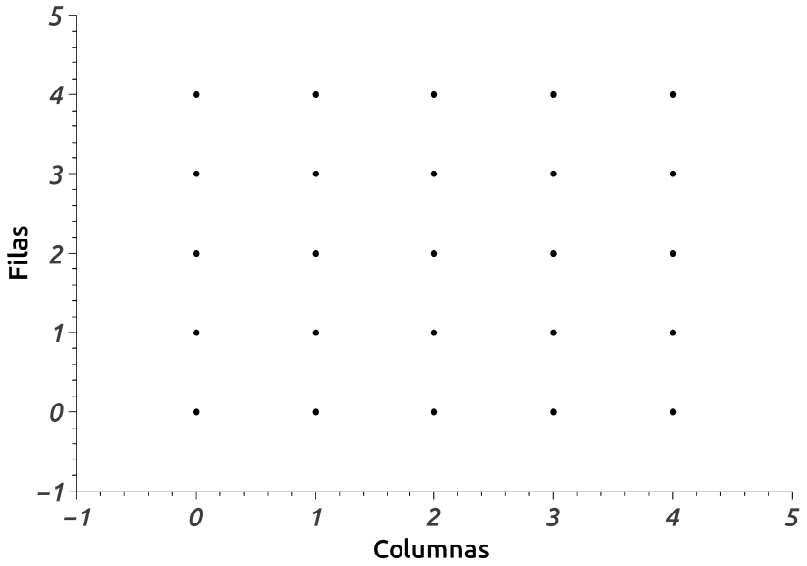


Figura 5.4: Zoom de uno de los puntos de Vef ver Figura 5.3

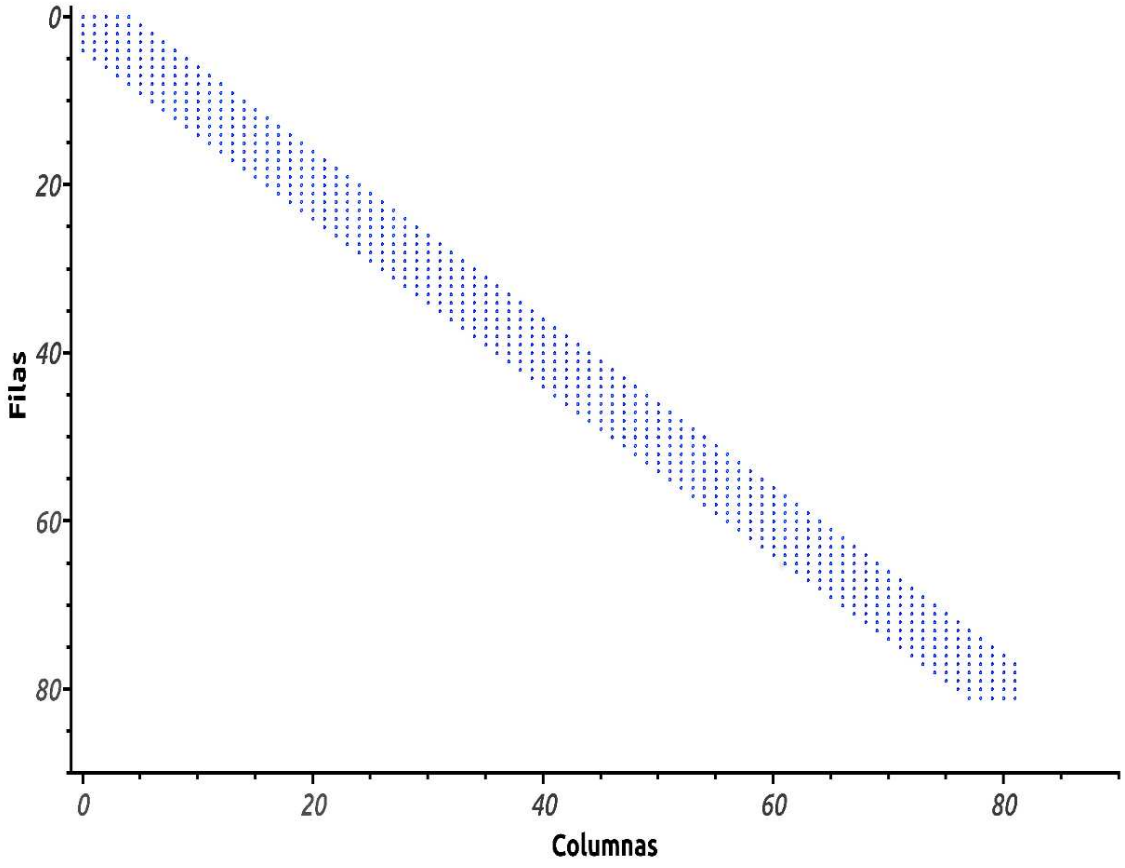


Figura 5.5: Distribución de elementos distintos de cero de la matriz s, ke, v_0, f, y g

matriz, algunas matrices son de banda y otras son solo dispersas sin ser de banda (mas bien lo que sucede es que: la banda que se debe tomar sería demasiado ancha).

5.2.1.1. Función Intersección

En esta función se pasó de tener un orden de $\mathcal{O}(L_{INT} \times INT_G \times KORD^2 \times (L_{INT} \times INT_G + (KORD + L_{INT})^2)$ a $\mathcal{O}(L_{INT} \times INT_G \times KORD^2 \times (L_{INT} \times INT_G + KORD \times (KORD + L_{INT})))$ sólo cambiando una línea, en el Algoritmo 11 podemos ver el cambio en el espacio de iteración del ciclo. Si tomamos constantes todas las variables excepto L_{INT}, que es lo que define el tamaño del problema, podemos reescribir el orden como $\mathcal{O}(L_{INT}^3)$ a $\mathcal{O}(L_{INT}^2)$

5.2.1.2. Función t y k

Ésta se calcula sin necesidad de acceder a memoria ya que es una función que tiene tres partes bien diferenciadas, dónde la primera y la última son constantes y la segunda es lineal Ver Figura 5.6. Como vimos en la subsección 4.1.2 cuando las cuentas son pocas y los datos para estas son de fácil acceso (están implícitos o recientemente usados) recalcular un valor es mejor que ir a buscarlo a memoria.

5.2.1.3. Escalas de Gauss-Legendre

El método de $Gauss-Legendre$ es calculado L_{INT} veces en la función $KNOT\_PESOS$ (ver Algoritmo 7) sin embargo este método puede ser calculado una sóla vez y luego ser escalado para cualquier intervalo [a,b]. Luego una optimización, tanto en memoria como en tiempo de ejecución, es resolver las abscisas y pesos de $Gauss - Legendre$ para [-1, 1] y realizar las funciones de escalado. Entonces podemos reemplazar el Algoritmo 7 con el Algoritmo 12 y luego escalar los $w_i^{-1,1}$ y $x_i^{-1,1}$ con las siguientes funciones descriptas en los Algoritmos 14 y 13.

El escalado se realiza de la siguiente manera: supongamos que queremos las j-ésimas abscisas y pesos del intervalo i para $i \in [0, L_{INT}]$ y $j \in [0, INT_G]$. Luego en las primeras líneas de los algoritmos 13 y 14 se calculan los límites de la integración del intervalo i y posteriormente se escala el j-ésimo valor.

5.2.1.4. Función *sener*

En el Algoritmo 15 están resaltadas las líneas donde se realizó la reducción en cuanto a iteraciones. Se han agregado dos variables extras: $jump_1, jump_2$. Éstas representan los saltos que se deben dar. Al no recorrer todo el ciclo secuencialmente, hay que ir saltando a los valores no nulos. Notar que las matrices que este algoritmo calcula tiene la estructura de la Figura 5.1. En esta función se pasó de tener un orden de $\mathcal{O}((L_{INT} + KORD)^4)$ a $\mathcal{O}((L_{INT} + KORD)^2 \times KORD^2)$.

5.2.2. Factorización de código y cacheo de resultados

Esta optimización consiste en intercambiar la anidación de ciclos para evitar recalcular una función con los mismos parámetros.

Result: Cálculo de la interacción Vef

```
1  Vef = 0                                   ▷ Tensor de dimensión N^4, que representa H_{i,i';j,j'}
   nb = L_INT + KORD − 3                                                  ▷ tamaño de la base
2  for i ∈ [KORD − 1, KORD + L_INT − 1) do
3      for abs ∈ [0, INT_G) do
4          rr_1 = x[k[i], abs]
5          w_1 = w[k[i], abs]
6          f = 0                                                          ▷ Matriz N^2
7          g = 0                                                          ▷ Matriz N^2
8          for i' ∈ [KORD − 1, KORD + L_INT − 1) do
9              for abc' ∈ [0, INT_G) do
10                 rr_2 = x[k[i'], abc']
11                 w_2 = w[k[i], abs']
12                 sp = evaluar los b-splines en el punto rr_2
13                 for m ∈ [0, KORD) do
14                     j = i' − KORD + m
15                     if 0 ≤ j < nb then
16                         for n ∈ [0, KORD) do
17                             j' = i' − KORD + n
18                             if 0 ≤ j' < nb then
19                                 if rr_2 ≤ rr_1 then
20                                     f_{j,j'} += sp[m] ∗ sp[n] ∗ w_2/rr_1
21                                 else
22                                     g_{j,j'} += sp[m] ∗ sp[n] ∗ w_2/rr_2
23                                 end
24                             end
25                         end
26                     end
27                 end
28             end
29         end
30         sp = evaluar los b-splines en el punto rr_1
31         for m ∈ [0, KORD) do
32             im = i' − KORD + m
33             if 0 ≤ im < nb then
34                 for m' ∈ [1, KORD) do
35                     im' = i − KORD + m' − 1
36                     if 0 ≤ im' < nb then
37                         for j ∈ [1, nb) do
38                             for j' ∈ [máx(n − KORD, 0), mín(nb, n + KORD + 1)) do
39                                 Vef_{im,im'; j,j'} += (sp[m]∗sp[m']∗w_1∗(f_{j,j'} + g_{j,j'})) / √(s_{j,j} ∗ s_{j',j'})
40                             end
41                         end
42                     end
43                 end
44             end
45         end
46     end
47 end
```

Algoritmo 11: Interacción

Result: Cálculo auxiliar $x_{-1,1}$, $w_{-1,1}$
1 $x^{-1,1} = abscisas\ del\ método\ de\ gauss - legendre\ en\ el\ intervalo\ [-1,1]$
2 $w^{-1,1} = pesos\ del\ método\ de\ gauss - legendre\ en\ el\ intervalo\ [-1,1]$

Algoritmo 12: Cálculo auxiliar

Result: Abscisas $x_{i,j}$
1 $dr = \frac{R\_MAX - R\_MIN}{L_{INT}}$
2 $x_1 = R\_MIN + idr$
3 $x_2 = x_1 + dr$
4 $x_m = 0{,}5 * (x_2 + x_1)$
5 $x_l = 0{,}5 * (x_2 - x_1)$
6 **if** $j \geq \frac{INT\_G+1}{2}$ **then**
7 | **return** $x_m + x_j^{-1,1} * x_l$
8 **else**
9 | **return** $x_m - x_j^{-1,1} * x_l$
10 **end**

Algoritmo 13: Valor de las abscisas j de la cuadratura del $i - esimo$ intervalo

Result: Pesos $w_{i,j}$
1 $dr = \frac{R\_MAX - R\_MIN}{L_{INT}}$
2 $x_1 = R\_MIN + idr$
3 $x_2 = x_1 + dr$
4 $x_l = 0{,}5 * (x_2 - x_1)$
5 **return** $\frac{2x_l}{w_j^{-1,1}}$

Algoritmo 14: Valor del peso j de la cuadratura del $i - esimo$ intervalo

Si bien anteriormente hablamos que reducir el uso de memoria por recalcular el valor de una función era conveniente, éste no es el caso aquí pues la función *bsplvb* y *bder* es más costosa recalcular que buscar en RAM, por ende conviene hacer cacheo de la función en vez de recalcular. Esta función requiere calcular su valor repetidas veces en los procesos *interacción* y *cálculo matrices* por lo tanto se memorizó los resultados para luego accederlos.

En el Algoritmo 16 podemos ver: cómo el ciclo del iterador j está unas líneas antes y el arreglo $bders$ guarda los de $bder$.

5.2.2.1. Cálculo de Valores versus Almacenamiento

Cuando el cálculo del valor es sencillo es mejor calcularlo nuevamente en vez de ir a buscarlo a memoria, puesto que eso produce más fallos de caché. Hay que tener en cuenta que el acceso a memoria es 100 veces más lento [23]. Tal es el caso de la función t: ésta se calcula sin necesidad de memoria ya que es una función que tiene tres partes bien diferenciadas, donde la primera y la última son constantes y la segunda es lineal. Ver Figura 5.6.

Data: s, v_0, ke, Vef
Result: Cálculo de la simetrización, $hsim_{eta}$, ms_{eta}

```
1  nb = L_INT + KORD − 3                                   ▷ tamaño de la base
2  mh = ke − v_0
3  i = 0
4  for η ∈ puntos η do
5      hsim = 0
6      ms = 0
7      for n ∈ [0, nb) do
8          for m ∈ [n, nb) do
9              jump_2 = máx(n − KORD + 1, 0)
10             j = (nb ∗ (nb + 1))/2 − ((nb − jump2) ∗ (nb − jump2 + 1))/2
11             for n' ∈ [jump_2, mín(nb, n + KORD)) do
12                 jump_1 = máx(0, n − KORD + 1)
13                 j+ = jump_1
14                 for m' ∈ [jump_1 + n', mín(nb, m + KORD)) do
15                     if m = n and m' = n' then
16                         hsim_{i,j} = 2 ∗ s_{n,n'} ∗ mh_{n,n'} + η ∗ Vef_{n,n:n',n'}
17                         msi, j = s_{n,n'} ∗ sn, n'
18                     else if m ≠ n and m' = n' then
19                         hsim_{i,j} = 1/√2 ∗ (2 ∗ s_{m,n'} ∗ mh_{n,n'} + 2 ∗ s_{n,n'} ∗ mh_{m,m'} + η ∗
                            Vef_{m,n:n',n'} + η ∗ Vef_{n,m:n',n'})
20                         ms_{i,j} = 2 ∗ 1/√2 ∗ s_{n,n'} ∗ sm, n'
21                     else if m = n and n' ≠ m' then
22                         hsim_{i,j} = 1/√2 ∗ (2 ∗ s_{n,m'} ∗ mh_{n,n'} + 2 ∗ s_{n,n'} ∗ mh_{n,m'} + η ∗
                            Vef_{n,n:n',m'} + η ∗ Vef_{n,n:m',n'})
23                         ms_{i,j} = 2 ∗ 1/√2 ∗ s_{n,n'} ∗ s_{n,m'}
24                     else
25                         hsim_{i,j} = s_{n,n'} ∗ mh_{m,m'} + s_{n,m'} ∗ mh_{m,n'}+
26                         s_{m,m'} ∗ mh_{n,n'} + s_{m,n'} ∗ mh_{n,m'}+
27                         η ∗ 0,5 ∗ (Vef_{n,m:n',m'} + Vef_{m,n:n',m'} + Vef_{n,m:m',n'}, Vef_{m,n:m',n'})
28                         ms_{i,j} = s_{n,n'} ∗ s_{m,m'} + s_{n,m'} + s_{m,n'}
29                     end
30                     j = j + 1
31                 end
32             end
33             i = i + 1
34         end
35     end
36     guardar sistema(hsim, ms)
37 end
```

Algoritmo 15: Simetrización

Result: Cálculo de las matrices s, v_0 y ke

$nb = L\_INT + KORD - 3$ ▷ tamaño de la base
$ma = 0{,}5 * lmax * (lmax + 1)$ ▷ $lmax$ es el momento angular
$bders = 0$ ▷ vector de tamaño KORD
for $i \in [KORD - 1, KORD + L\_INT - 1)$ **do**
  for $j \in [0, INT\_G)$ **do**
    $rr = x[k[i], j]$
    $sp =$ evaluar los b-splines en el punto rr
    for $m \in [0, KORD)$ **do**
      $im = i - KORD + m$
      if $0 \leq im < nb$ **then**
        for $n \in [0, KORD - 1]$ **do**
          $in = i - KORD + n$
          if $0 \leq in < nb$ **then**
            $s_{im,in}$ += sp[m] * sp[n] * w[k[i], j]
            $ke_{im,in}$ += $\frac{ma*sp[m]*sp[n]*w[k[i],j]}{rr*rr}$
            if $r_{min} < rr < r_{max}$ **then**
              $v_{0i,j}$ += sp[m] * sp[n] * w[k[i], j]
            end
          end
        end
      end
    end
    $ind = 0$
    for $m \in [\text{máx}(0, i - KORD + 1), \text{mín}(i + 1, nb))$ **do**
      $bders[ind] = derivada\ del\ b - spline\ en\ el\ punto\ rr\ en\ el\ índice\ m$
      $ind+ = 1$
    end
    for $n \in [0, KORD - 1]$ **do**
      $in = i - KORD + n$
      if $0 \leq in < nb$ **then**
        $bm = bder[m]$
        $bn = bder[n]$
        ke_{ij} += $\frac{0{,}5*w[k[i],j]*bm*bn}{me}$
      end
    end
  end
end

Algoritmo 16: Cálculo de matrices

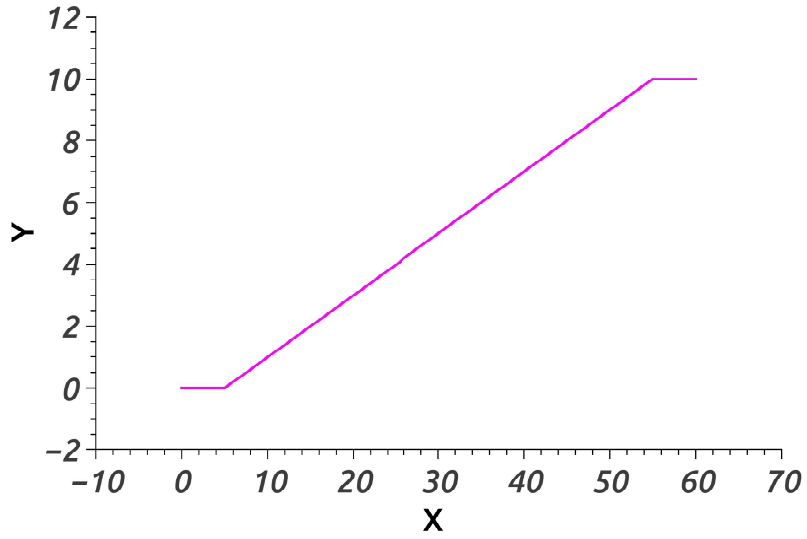


Figura 5.6: Función T Restringida para observar su forma

5.3. Optimización en GPU

Se implementó una versión híbrida (parte en GPU y parte en CPU) del método *interacción*, que es la función más computacional.

La Ec. 3.16 se puede paralelizar, estas integrales anidadas se reducen por medio del método de Gauss-Legendre a una sumatoria, donde cada término es independiente. Luego se pueden utilizar dos kernels: uno para calcular la integral interior y otra para calcular la integral externa. Como vimos en la sección 3.2.2 qué para el cálculo de la Ec. 3.16 se realiza el cálculo de dos funciones: $f_{j,j'}^{i,i'}$ y $g_{j,j'}^{i,i'}$ detallada en la Ec. 3.17. Entonces esta implementación en GPU se separó en dos kernels: uno para calcular las funciones $f_{j,j'}^{i,i'}$ y $g_{j,j'}^{i,i'}$ y otro kernel para calcular las integrales a estas dos funciones. Notar que esta idea no es diferente a la implementación secuencial, la idea de la implementación en GPU es paralelizar los *fors* involucrados en el cálculo de la cuadratura *Gauss-Legendre* de ambas integraciones, por lo tanto cada *id* de hilo y bloque representa una suma de la cuadratura.

En un sentido práctico esto es paralelizar las líneas desde 8 hasta 29 y desde 31 hasta 45 del Algoritmo 11, de esta manera cada tupla de índices de los *fors* se resuelven con bloques e hilos de la GPU. Para evitar las condiciones de carrera se utiliza las operaciones atómicas de CUDA (ver [24]), esto se debe a que varios hilos escriben en el mismo registro.

El primer kernel (ver Algoritmo 17) utiliza $(KORD^2)$ número de hilos y $(L_{INT} \times INT_G)$ cantidad de bloques. El segundo (ver Algoritmo 18) utiliza $(KORD^2 \times 2 \times (KORD + 1))$ cantidad de hilos y $(L_{INT} + KORD - 3)$ cantidad de bloques, ver algoritmo 19 para mas detalles.

Input: rr_1, w_1, f, g, x, w, Sps
Result: Kernel que calcula las funciones $f^{i,i'}_{j,j'}$ y $g^{i,i'}_{j,j'}$

$nb = L\_INT + KORD - 3$ ▷ tamaño de la base
$m = threadIdx.x$
$n = threadIdx.y$
$l = threadIdx.z$
$m' = m + m - 1$
$n' = m + n - 1$
$base = threadIdx.x$
$rr_2 = x_{base,l}$
$w_2 = w_{base,l}$
$Sp = Sps_{base,l}$
if $0 \geq n' < nb$ ***and*** $0 \geq m' < nb$ **then**
  if $rr_2 \leq rr_1$ **then**
    $f_{i,j} += \frac{Sp_m * Sp_n * w_2}{rr_1}$
  end
  else
    $g_{i,j} += \frac{Sp_m * Sp_n * w_2}{rr_2}$
  end
end

Algoritmo 17: Cálculo las funciones $f^{i,i'}_{j,j'}$ y $g^{i,i'}_{j,j'}$ con GPU

Input: w_1, f, g, $base$, i, j Sps, $sdiag$, Vef
Result: Kernel que calcula el tensor Vef

$m = threadIdx.x$
$m' = threadIdx.y$
$i' = i - KORD + m - 1$ $j' = i - KORD + mp - 1$ $n = blockIdx.x$
$n' = n - KORD + threadIdx.z$
if $0 \leq n'$ ***and*** $0 \leq i'$ ***and*** $0 \leq j$ **then**
  $Vef_{i',n;j',n'} += \frac{Sp_m + Sp_{m'} * w_1 * (f_{n,n'} + g_{n,n'})}{\sqrt{sdiag_n * sdiag_{n'}}}$
end

Algoritmo 18: Cálculo del tensor Vef con GPU

Result: Función del lado *Host* que calcula el tensor Vef

```
for k ∈ [0, L_INT) do
    for l ∈ [0, INT_G) do
        rr = x_{l,x}
        Sps_{k,l} = evaluar los b-splines en el punto rr
    end
end
for i ∈ [0, (L_INTERVALS + 2 * KORD − 3)) do
    sdiag_i = s_{i,i}
end
Copiar Sps, x, w, sdiag, a memoria de dispositivo                ▷ Contexto
Alojar memoria para d_Vef, f y g
for i ∈ [0, L_INT) do
    for j ∈ [0, INT_G) do
        rr_1 = x_{i,x}
        w_1 = w_{i+KORD,j}
        llamada al algoritmo 17
        llamada al algoritmo 18
    end
end
Recuperar el valor de d_Vef en Vef
Liberar memoria
```

Algoritmo 19: Cálculo del tensor Vef con GPU

Capítulo 6

Resultados

En este Capítulo mostraremos los tiempos y memoria requerida de las diferentes implementaciones. La implementación básica, la optimizada para CPU y la implementación con GPU sólo en la función *interacción*. Para ello se aumentará el valor de la variable L_{INT} para generar problemas de mayor tamaño.

Cada función se le medirá el tiempo individualmente, la memoria se tomará el total del programa. No se medirá el tiempo que toma en resolver el problema de autovalores ni su memoria requerida, ya que la resuelve una biblioteca de terceros y depende mucho de que biblioteca o paquete que se use. También se puede exportar las matrices para ser luego resueltos con las herramientas que se desee. Las matrices exportadas van a estar en matrices comprimidas por columnas (CCS sección 2.4.1).

6.1. Medición de rendimiento en el procesamiento

Para las pruebas sólo nos centraremos en la variable L_{INT} ya que las demás variables no influyen en el tamaño del problema, salvo INT_G y $KORD$ que usualmente no son demasiado grandes. Al comparar rendimiento, nos interesa el tiempo y la memoria que toma calcular el sistema a diagonalizar.

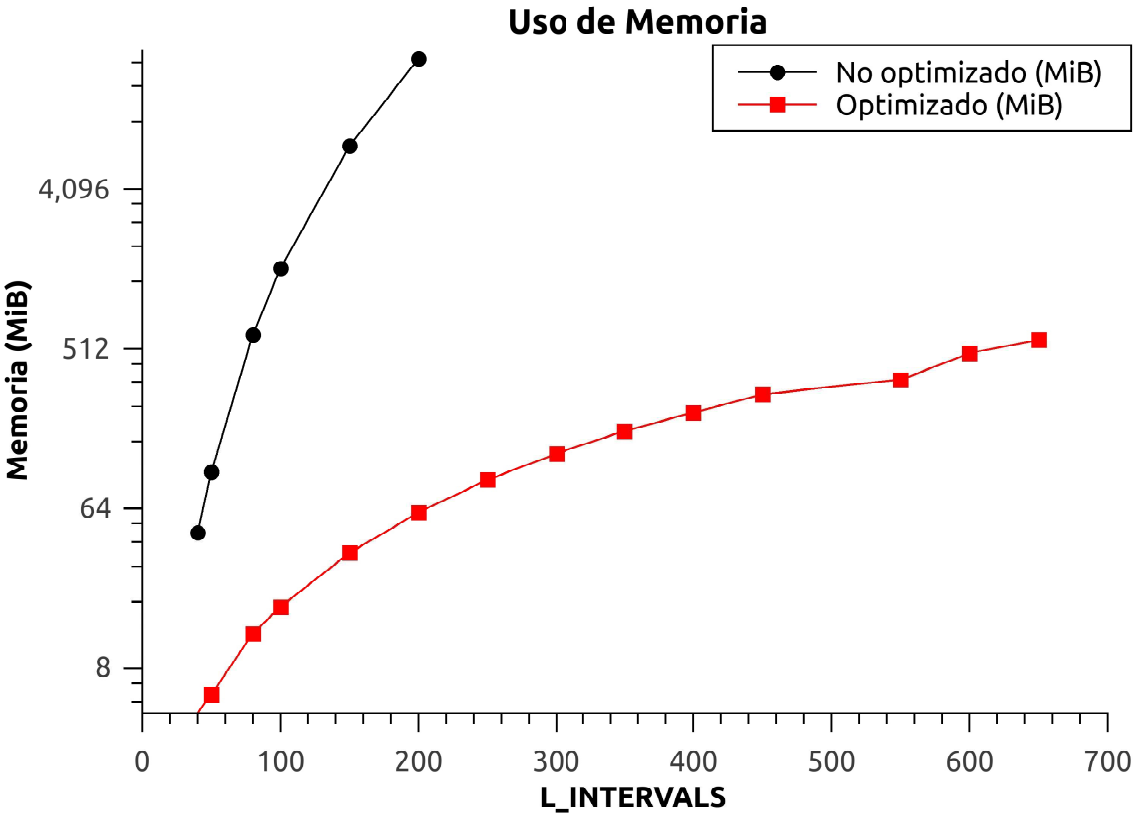
En el Cuadro 6.1 detallamos el entorno utilizado para realizar las pruebas.

CPU	
Procesador	Intel(R) Core(TM) i7 CPU 980 @ 3.33GHz
Memoria	24 GB DDR3@ 1067 MT/S
Arquitectura	Nehalem
Sistema Operativo	Debian GNU/Linux
Compilador	gcc version 7.2.0
Flags	-larpack -lblas -llapack -lgfortran -fopenmp -O3
GPU	
Version CUDA	7.5
Flags	-gencode arch=compute_30,code=sm_30 -gencode arch=compute_35,code=sm_35 -gencode arch=compute_37,code=sm_37 -gencode arch=compute_50,code=sm_50 -gencode arch=compute_52,code=sm_52 -gencode arch=compute_52,code=compute_52
Procesador	GTX 980
Memoria	4 GB DDR5
Interfaz	PCI Express 2.0
Arquitectura	**Maxwell**
Procesador	K40
Memoria	12 GB DDR5
Interfaz	PCI Express 2.0
Arquitectura	Kepler

Cuadro 6.1: Entorno

6.2. Medición en el uso de Memoria.

En el gráfico de la Figura 6.1 se puede ver cómo varía la cantidad de memoria necesaria para el cálculo del sistema en la versión no optimizada, se corta en el experimento en un $L_{INT} = 200$ pues en el experimento siguiente $L_{INT} = 250$ supera la cantidad de memoria de la computadora utilizada por tanto su cómputo se demora enormemente (por causa del *swapping*). Lo destacable del Gráfico 6.1 es cómo el crecimiento de ambas curvas es de órdenes diferentes, lo cual no guardar los ceros (o la mínima cantidad posible) ofrece una notable escalabilidad en cuanto a memoria. Esto ocurre en matrices muy ralas, como lo son en este caso.



Grágico 6.1: Pico de memoria

6.3. Medición del tiempo de ejecución

Como dijimos anteriormente el proceso del método se puede separar en cuatro partes importantes. Los cuales los dos primeros son para el problema de una partíicula y dos partículas y los otros dos son sólo para el problema de dos partículas. Como en este trabajo se empezó optimizando el problema de una partícula daremos los resultados de las optimizaciones de una partícula y luego la de dos (exclusivamente). Recordar que el proceso de dos partículas necesita la resolución del problema de una partícula.

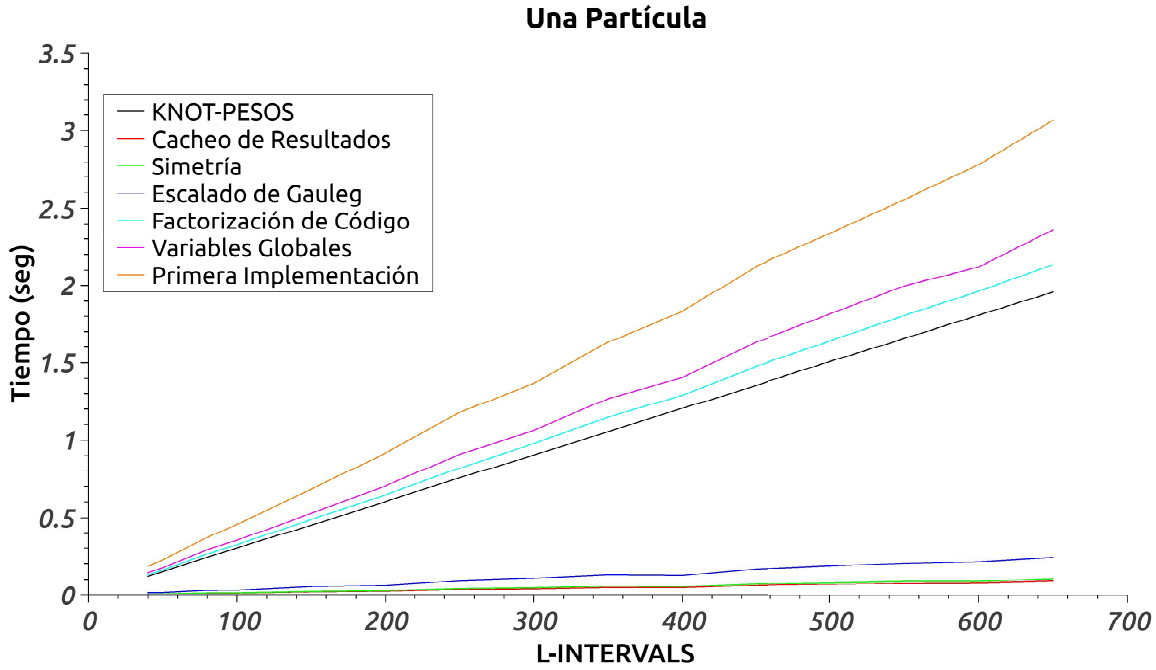
6.3.1. Optimizaciones de una Partícula.

Como se mencionó en la subsección 3.2.1 este proceso consta de dos partes: una se muestra en el Algoritmo 7 y la otra en el Algoritmo 8.

En el Gráfico 6.2 podemos ver cómo se han aplicando las diferentes forma de optimización. Notar que el gráfico tiene el tiempo que toma calcular la función auxiliar $KNOT\_PESOS$.

El primer cambio fue no pedir memoria dinámicamente, sino que declarar las variables con un tamaño constante necesario. El segundo cambio consistió en una pequeña factorización de código, para llamar menos cantidad de veces a la función que calcula el valor del B-spline en un punto dado. El tercer cambio fue la eliminación de la función

$KNOT\_PESOS$ como se explicó en 5.2.1.3. Luego se usó la simetría de las matrices para calcular sólo la mitad de la matriz. Y por último la implementación de una *lookup table* para cacheo de las derivadas de los B-splines en el punto. En el Cuadro 6.2 se muestra el *speed up* para cada valor de L_{INT} logrando un máximo de 34.7x.



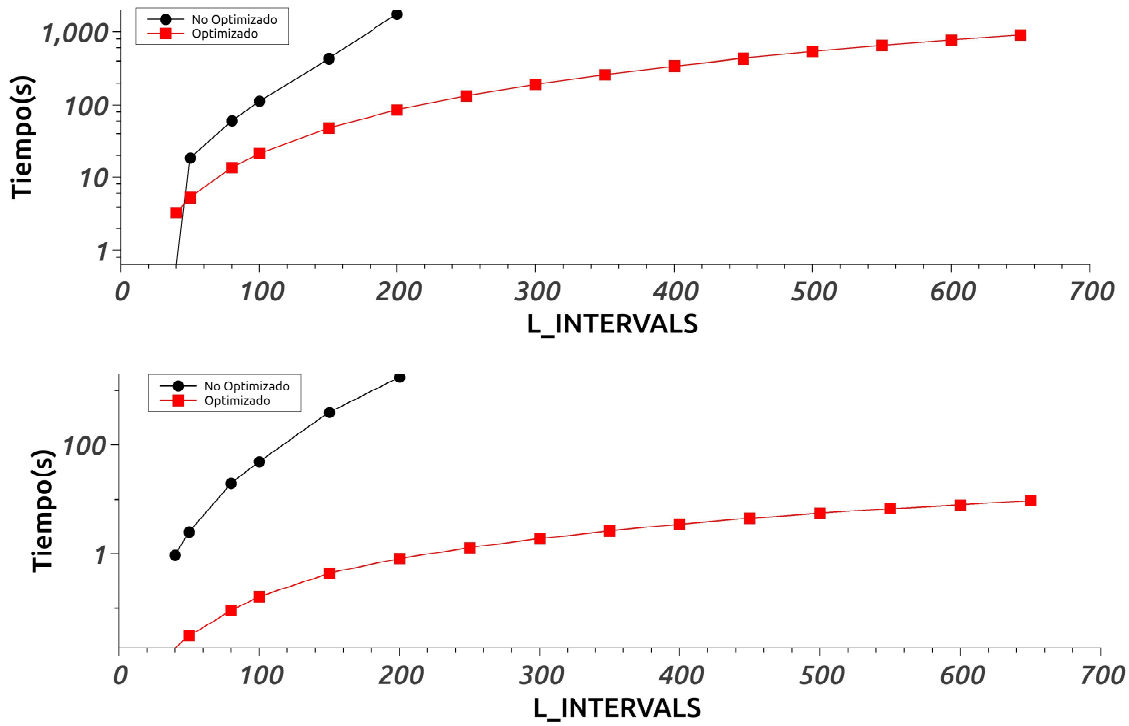
Grágico 6.2: Tiempo de ejecución

L_{INT}	Tiempo No Optimizado	Tiempo Optimizado	Speed Up
40	0.185	0.008	21.944
50	0.225	0.009	25.051
80	0.370	0.014	26.874
100	0.457	0.015	30.524
150	0.682	0.022	30.535
200	0.918	0.026	34.652
250	1.177	0.037	32.090
300	1.368	0.042	32.908
350	1.632	0.050	32.498
400	1.832	0.050	36.648
450	2.124	0.064	33.020
500	2.335	0.071	33.087
550	2.554	0.077	33.038
600	2.778	0.080	34.703
650	3.067	0.091	33.782

Cuadro 6.2: Speed Up No Optimizado vs Optimizado una partícula

6.3.2. Optimizaciones de dos Partículas

Como se explicó anteriormente este proceso cuenta con tres partes (ver sección 3.2.2), una es calcular el *problema de una partícula* (este a su vez tiene dos partes), la otra es calcular la *interacción* y la tercera realizar el proceso *sener*. Las optimizaciones en este problema son las mismas que se realizaron en el problema de una partícula, dado que las dos funciones que pertenecen exclusivamente al problema de dos partículas utilizan las matrices calculadas en el de una partícula, luego las distribuciones de los valores no nulos están acoplados, esto hace que las mismas ideas del proceso con una partícula se puedan aplicar al de dos de manera directa. Así mismo como las estructuras de datos (matrices a generar) están acopladas se realizó la reimplementación del método con las optimizaciones antes vistas. No hay pasos intermedios de mejoras, sino una sóla mejora en base al código optimizado del problema de una partícula. Como se vió el orden algorítmico es menor, ésto es consecuencia de no computar los ceros que en cantidad son de orden mayor a la cantidad de no-ceros. Haber cambiado las estructuras de datos implicó bajar uno o dos órdenes (depende de la función).

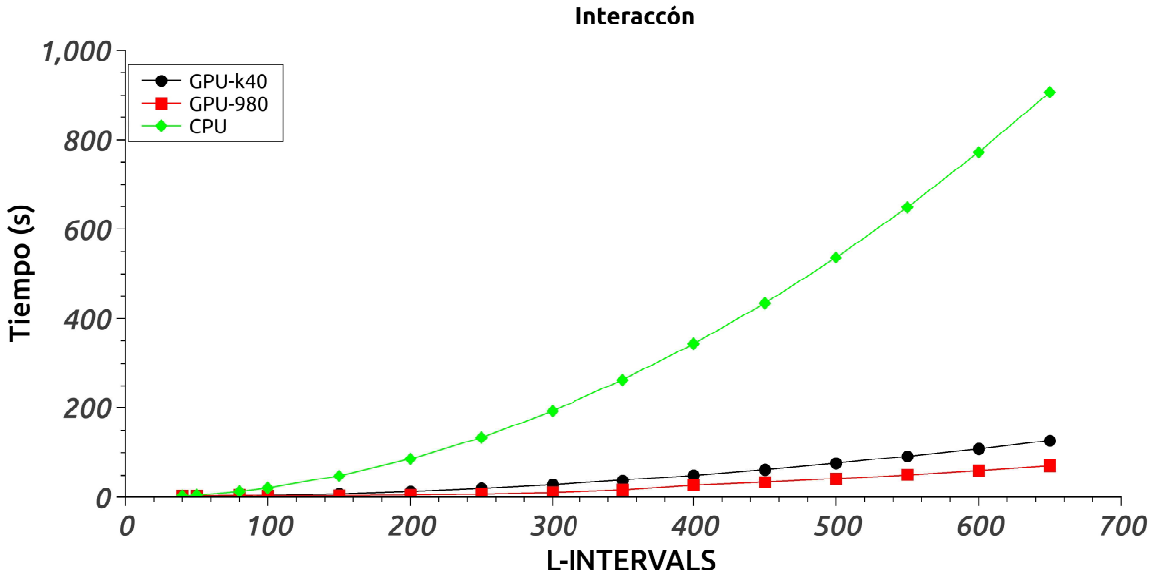


Grágico 6.3: Mediciones de tiempo de las funciones propias del problema de dos partículas: **interacción y sener** respectivamente

6.3.3. Optimización usando GPU

La optimización en GPU se realizó en el método *interacción* como fue presentado en la sección 5.3, los resultados que podemos ver en el Gráfico 6.4. Se ha conseguido

una velocidad de hasta 12.9x, como se puede ver en el Cuadro 6.3, superior a la mejor implementación en CPU. Podemos ver que la Placa de video *GTX 980* se comportó mejor que la *K40* dado que tiene mejor rendimiento en las operaciones de punto flotante de simple precición.



Grágico 6.4: Comparación entre CPU y GPU

L_{INT}	Tiempo CPU	Tiempo GTX980	Speed Up GTX980	Tiempo K40	Speed Up K40
40	3.350	4.212	0.795	2.866	1.169
50	5.265	4.024	1.308	3.205	1.643
80	13.553	5.494	2.467	4.410	3.073
100	21.201	4.288	4.944	5.164	4.106
150	48.133	4.809	10.009	8.552	5.628
200	85.695	6.169	13.891	13.550	6.325
250	133.899	7.936	16.871	20.167	6.639
300	192.871	11.364	16.972	28.515	6.764
350	262.632	16.948	15.497	38.551	6.813
400	343.160	27.332	12.555	49.341	6.955
450	434.191	34.075	12.742	62.108	6.991
500	536.205	41.716	12.854	76.332	7.025
550	648.931	50.266	12.910	92.064	7.049
600	772.392	59.990	12.875	108.469	7.121
650	906.392	70.547	12.848	126.974	7.138

Cuadro 6.3: Speed Up No Optimizado vs Optimizado dos partículas

Capítulo 7

Conclusiones

En este trabajo se presentó una manera de optimizar una implementacion del método variacional de Rayleigh-Ritz con una y dos partículas. Para su optimización se empleó: matrices ralas, factorización de código, cacheo de funciones y luego una mejora en GPU, se estudió la estructura de los diferentes tensores como también sus restricciones, sus ventajas y que operaciones se iban a realizar. Con esta información se elegió una estructura de datos más eficiente y específica. Se adaptó el código para hacer un buen uso de las estructuras.

Luego de las optimizaciones para CPU se procedió a la optimización en GPU tomando como base el código optimizado para CPU. Por otro lado este trabajo muestra que: la elección de estructuras de datos más acordes y mejorar el orden de los algoritmos son las primeras acciones a tomar, terminadas estas ya se realizan las otras mejoras (multi CPU, GPU, clusters, etc).

Bibliografía

[1] SPARSKIT *working note 50: Distributed sparse data structures for linear algebra operations*, Tech. Rep. CS 92-169, Computer Science Department, University of Tennessee, Knoxville, TN. 1992.

[2] LAPACK *A basic tool kit for sparse matrix computation, Tech. Rep. CSRD TR 1029, CSRD*, University of Illinois, Urbana, IL 1990

[3] R. B. Lehoucq & D. C. Sorensen (1996). *"Deflation Techniques for an Implicitly Restarted Arnoldi Iteration"*

[4] Krylov, *SFLubspace methods on supercomputers*, SIAM J. Sci. Statist. Comput. 10 (1989), pp. 1200-1232.

[5] I. S. DUFF, A. M. ERISMAN, AND J.K.REID, *Direct methods for sparse matrices*, Oxford University Press, London 1986

[6] J. DONGARRA, C. MOLER, J. BUNCH, AND G. STEWART *LINPACK Users' Guide, SIAM*, Philadelphia 1979.

[7] R. MELHEM *Toward efficient implementation of preconditioned conjugate gradient methods on vector supercomputers*, Internat. J. Supercomput. Appls., 1 (1987), pp. 77-98

[8] Cullum; Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations. 1.* ISBN 0-8176-3058-9.

[9] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems.* ISBN 0-470-21820-7.

[10] W. E. Arnoldi. *The principle of minimized iterations in the solution of the matrix eigenvalue problem.* Quart. Appl. Math., 9:17-29, 1951.

[11] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Halsted Press, New York, 1992

[12] B. N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, NJ, 1980. Reprinted as Classics in Applied Mathematics 20, SIAM, Philadelphia, 1997.

[13] G. Golub and C. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, Baltimore, third edition, 1996.

[14] H. F. Walker. *Implementation of the GMRES method using Householder transformations.* SIAM J. Sci. Statist. Comput., 9:152-163, 1988.

[15] *Templates for the Solution of Algebraic Eigenvalue Problems* Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst

[16] *http://www.caam.rice.edu/software/ARPACK/*

[17] Phys. Rev. 43, 830-833 (1933)

[18] D. Calvetti, L. Reichel, and D.C. Sorensen (1994). *An Implicitly Restarted Lanczos Method for Large Symmetric Eigenvalue Problems.* Electronic Transactions on Numerical Analysis 2: 1-21.

[19] J.G.F. Francis. *The QR Transformation, I.* The Computer Journal, vol. 4, no. 3, pages 265-271 (1961, received Oct 1959)

[20] Paige, C.C. *Computational variants of the Lanczos method for the eigenproblem.* J. Inst. Maths Applics 10, 373-381 (1972).

[21] J. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[22] Flynn *Some Computer Organizations and Their Effectiveness*

[23] Systems Performance: Enterprise and the Cloud *http://www.brendangregg.com/sysperfbook.html*

[24] Atomic Operation CUDA GPU *http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions*

Los abajo firmantes, miembros del Tribunal de Evaluacin de tesis, damos Fe que el presente ejemplar impreso, se corresponde con el aprobado por éste Tribunal