



**UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO**

---

INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS  
APLICADAS Y EN SISTEMAS

POSGRADO EN CIENCIA E INGENIERIA  
DE LA COMPUTACIÓN

**DIAGONALIZACIÓN DE LA MATRIZ DE  
KHON-SHAM CON TARJETAS GRÁFICAS**

**TRABAJO FINAL**

QUE PARA OBTENER EL GRADO DE:

**ESPECIALISTA EN CÓMPUTO DE  
ALTO RENDIMIENTO**

PRESENTA

**JOSÉ ANTONIO AYALA BARBOSA**

TUTOR

**DR. JOSÉ JESÚS CARLOS QUINTANAR SIERRA**



# **Agradecimientos**



# Índice

Índice .....	5
Introducción .....	7
Capítulo 1. Matriz de Khon-Sham .....	9
1.1 Ecuación de Schrödinger .....	9
1.2 Teoría del Funcional de la Densidad (DFT) .....	9
1.2.1 Ecuación de Kohn-Sham .....	10
Capítulo 2. Algoritmo de Jacobi .....	14
2.1 Forma cuadrática .....	14
2.2 Eigensistemas .....	14
2.2.1 Transformación de coordenadas .....	15
2.2.2 Eigenvalores .....	16
2.2.3 Eigenvector .....	17
Capítulo 3. Implementación en C, OpenACC y CUDA .....	18
3.1 Cómputo en GPU .....	18
3.1.1 Diferencia entre CPU y GPU .....	18
3.2 Características del equipo .....	19
3.3 Implementación en C .....	20
3.3.1 Creación de matrices .....	20
3.3.2 Funciones auxiliares .....	20
3.3.3 Driver .....	21
3.3.4 Jacobi .....	22
3.3.5 Compilación .....	23
3.4 Implementación en OpenACC .....	24
3.4.1 Estándar OpenACC .....	24
3.4.2 JacobiACC .....	26
3.4.3 Compilación .....	27
3.5 Implementación en CUDA .....	28
3.5.1 Modelo de programación en CUDA .....	28
3.5.2 Driver .....	35
3.5.3 JacobiCUDA .....	35
3.5.4 Compilación .....	36
Capítulo 4. Pruebas y resultados .....	38
4.1 Métricas de desempeño .....	38
4.1.2 Runtime .....	38

4.1.2 Cost Factor .....	38
4.1.3 Speedup Factor .....	38
4.1.3 Aceleración.....	39
2.1.4 Efficiency .....	39
4.2 Resultados .....	40
4.2.1 Resultados de Runtime .....	40
4.2.2 Resultados de Cost Factor .....	41
4.2.3 Resultados de Speed Up.....	43
4.2.3 Resultados de Aceleración .....	44
4.2.4 Resultados de Efficiency.....	46
4.3 Resultados de una matriz .....	47
4.3.1 Matriz a evaluar .....	47
4.3.2 Resultado Serial.....	48
4.3.3 Resultado OpenACC .....	49
4.3.4 Resultado CUDA .....	50
Conclusiones.....	51
Tabla de Figuras.....	53
Bibliografía.....	54
CÓDIGO FUENTE .....	56
CF 1.1 creaMatrix.c.....	56
CF 1.2 auxFuncs.h.....	57
CF 1.3 driverA.c .....	58
CF 1.4 jacobiA.h.....	59
CF 1.5 driverA.cu .....	63
CF 1.6 jacobiACUDA.h .....	65

# Introducción

Física y química computacional tratan de resolver problemas implementando métodos del análisis numérico para resolver con precisión problemas para los cuales ya existe una teoría cuantitativa [1]. Inicialmente se consideró como una subdisciplina de la física teórica, y como una herramienta que auxilia a la ciencia. Pero en la actualidad ya es considerada como una área de la física teórica. Por ejemplo en el Institute for Theoretical Physics ETH Zurich [2] es una de las áreas de investigación, al lado de materia condensada, teoría cuántica del campo, partículas elementales y teoría de la información entre otros.

Tanto en la física como en la química computacional es necesario, entre otras muchas cosas, resolver sistemas lineales de ecuaciones, integrar, por ejemplo, las ecuaciones diferenciales de Newton en física clásica [3], o resolver la ecuación de Schrödinger en mecánica cuántica etc.; y para esto se pueden aplicar algoritmos como Runge Kutta, Monte-Carlo, y también en la mecánica cuántica un problema a resolver son las matrices Hermitianas de valores y vectores propios, (eigenmatrices, eigenvalores y eigenvectores) y uno de los algoritmos de elección es el método de diagonalización de Jacobi.

El problema de Eigensistemas aparece en uno de los métodos empleados para resolver la ecuación de Schrödinger o la ecuación de Kohn-Sham. En la ecuación de Schrödinger su solución es conocida como la función de onda, mientras que la solución de la ecuación de Kohn-Sham es el valor absoluto al cuadrado de la función de onda y se interpreta en la física como la densidad electrónica.

Durante las últimas décadas ha incrementado el interés en la computación paralela, el principal objetivo de ella es ahorro de la energía y evitar el sobrecalentamiento de los circuitos. Desde una perspectiva de puros cálculos, la computación paralela puede definirse en esos muchos cálculos que se resuelven simultáneamente, y como atacan problemas muy largos y los dividen en unos mucho más pequeños que pueden ser resueltos paralela y concurrentemente [4].

Actualmente el paralelismo, por razones de ahorro de energías y por desempeño se ha convertido en la corriente que todo el mundo de la programación está siguiendo. Existen dos principales tipos de paralelismo en aplicaciones la paralelización de tareas y la paralelización de datos. La primera surge cuando hay muchas tareas que pueden operar independientemente en paralelo, por ello el paralelismo de tareas se enfoca en la distribución de las funciones en los distintos núcleos de procesamiento. La segunda entra cuando existen muchos datos que pueden ser operados al mismo tiempo, siendo mejor distribuirlos entre los diversos procesadores.

La unidad de procesamiento de gráficos (GPU) es uno de los componentes más importantes en los ordenadores modernos [5]. Es aquí donde se construyen los increíbles gráficos que podemos ver en los videojuegos modernos.

Dentro de la computación, CUDA es una plataforma de cómputo paralelo de uso general y un modelo de programación que aprovecha el motor de cómputo paralelo en las GPU de la marca NVIDIA para resolver muchos problemas computacionales complejos de una manera más eficiente. Usando CUDA, puede acceder a las tarjetas de video para el cálculo como se ha hecho tradicionalmente en la CPU.

Es inevitable no mencionar que CUDA al ofrecer un entorno de programación de bajo nivel, ha mantenido un poco alejado a los programadores que están fuera del desarrollo del cómputo científico, por ello se creó el estándar de programación OpenACC, que con solo algunas directivas se le indica al compilador que cierta sección de código podría ser paralelizable, y el este se encargará de verificar si existen dependencias de dato o si es posible realizarse.

Varias de esas aplicaciones están relacionadas con la solución de las ecuaciones de Kohn-Sham, dentro de la teoría de funcionales de la densidad, también se han creado diversas aplicaciones relacionadas con métodos basados en la función de onda que estiman la correlación electrónica o energías de ionización [6].

En marzo de 2014, el Departamento de Energía de Estados Unidos (DOE), otorgó a IBM la comisión para construir dos supercomputadoras. Sierra localizada en el Laboratorio Nacional Lawrence Livermore y Summit localizada en el Laboratorio Nacional Oak Ridge. En el caso de Summit se apostó por equiparla con nodos de alto rendimiento y tarjetas gráficas, lo cual abrió el camino a utilizar esta tecnología de forma masificada [7]. Summit representa un cambio importante con respecto a la forma en que se viene realizando el mundo del supercómputo, ya que, al combinar CPUs de alto rendimiento con GPUs de NVIDIA optimizadas para la AI, se puede ver que la computación se va decantando cada vez más por el cómputo heterogéneo.

La potencia de las GPUs se ha hecho presente con sistemas computacionales relacionadas con el modelado de átomos en moléculas [8]. Debido a esto, en esta tesina se paralelizará el método de Jacobi en tarjetas gráficas y se realizará una comparación entre su versión Secuencial realizada en lenguaje C, contra una versión paralela utilizando OpenACC y otra versión utilizando el modelo de CUDA.



# Capítulo 1. Matriz de Kohn-Sham

## 1.1 Ecuación de Schrödinger

A diferencia de la mecánica clásica que determina los elementos por su posición y velocidad, la mecánica cuántica calcula la probabilidad de encontrar partículas en una cierta posición física gracias a la ecuación de Schrödinger [9]. Pero para poder ser implementada se requiere de mucho poder de cómputo, y al ser calculada con métodos numéricos, depende de cuál se elija, los resultados que se arrojan pueden ser algo imprecisos.

La ecuación de Schrödinger considera varios aspectos para poder ser implementada, dentro de los fundamentales están:

- La existencia de un núcleo atómico, en donde se concentra la mayoría del volumen del átomo.
- Los niveles energéticos donde se distribuyen los electrones según su energía.
- La dualidad onda-partícula.
- La probabilidad de encontrar al electrón.

Aunque con la mecánica cuántica queda claro que no se puede saber dónde se encuentra un electrón, sí define la región en la que puede encontrarse en un momento dado. Cada solución de la ecuación de ondas de Schrödinger,  $\Psi$ , describe un posible estado del electrón. El cuadrado de la función de onda,  $\Psi^2$ , define la distribución de densidad electrónica alrededor del núcleo. Este concepto de densidad electrónica da la probabilidad de encontrar un electrón en una cierta región del átomo, llamada orbital atómico, concepto análogo al de órbita en el modelo de Bohr [10].

$$\frac{\partial^2 \Psi}{\partial x^2} + \frac{8\pi^2 m}{h^2} (E - V) \Psi = 0$$

$\Psi = \text{función de onda}$   
 $m = \text{masa del electrón}$   
 $h = \text{constante de Planck}$   
 $E = \text{energía total del electrón}$   
 $V = \text{energía potencial del electrón}$

## 1.2 Teoría del Funcional de la Densidad (DFT)

Es un procedimiento alternativo a la ecuación de Schrödinger que permite una descripción muy exacta de los sistemas con muchas partículas, la DFT simplifica los cálculos, dejando a un lado el uso de la función de onda en la determinación del movimiento de electrones y átomos de una molécula. En el caso de la ecuación de Schrödinger esta solución se conoce como la función de onda  $\psi$ , mientras que la solución de la ecuación Kohn-Sham [11] está dada por el valor absoluto al cuadrado de la función de la función de onda y esta cantidad se interpreta como la densidad electrónica  $|\psi|^2 = \rho$ . Una forma de obtener una solución de ambas ecuaciones es aplicando el método variacional, éste método se usa para

determinar de una manera aproximada el nivel de energía más bajo del sistema; también conocida como la energía del estado base; éste método consiste en proponer una función tentativa que depende de varios parámetros los cuales se varían hasta que se obtenga una energía mínima.

La Teoría del funcional de la densidad obtiene la energía y distribución electrónica del estado fundamental [12], trabajando con la densidad electrónica en vez de la función de ondas. Una desventaja es que, salvo los casos más simples, no se conoce de manera exacta el funcional que relaciona esta densidad electrónica con la energía del sistema. En la práctica, se usan funcionales que se han comprobado que dan buenos resultados [13].

La primera aproximación que se hace al estudiar sistemas con más de un átomo es la aproximación de Born-Oppenheimer, en la cual los electrones en una molécula se mueven en un campo eléctrico producido por los núcleos fijos. Esta aproximación puede hacerse porque al ser la masa del núcleo mayor a la de los electrones, su velocidad es menor, por lo que para él, los electrones funcionan como una nube de carga [14], y por otro lado para los electrones, los núcleos parecen estar estáticos.

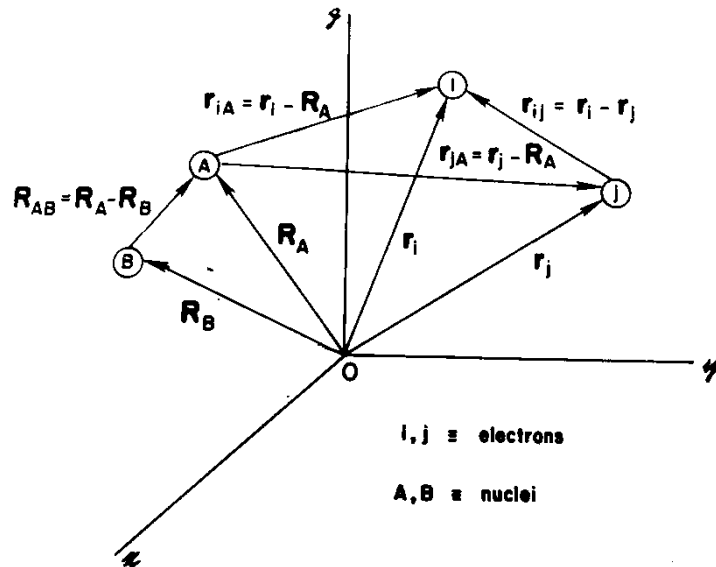


Figura 1 Sistema coordinado molecular:  $i, j$ =electrones y  $A, B$ = núcleos.

### 1.2.1 Ecuación de Kohn-Sham

La ventaja de que la densidad es una magnitud mucho más simple que la función de onda, simplificando las ecuaciones y bajando el costo computacional [15]. Por ello hasta el momento es el procedimiento preferido para abordar problemas a partir de una cierta complejidad.

La energía del sistema está dada por la suma de la energía:

$$E = E^{CE} + E^{CN} + E^{PEN} + E^{PEE} + E^{PNN}$$

$E^{CE}$  = energía cinética de los electrones (Atractivo)

$E^{CN}$  = energía cinética de los núcleos (Atractivo)

$E^{PEN}$  = energía potencial núcleo – electrón (Atractivo)

$E^{PEE}$  = energía potencial electrón – electrón (Repulsivo)

$E^{PNN}$  = energía potencial núcleo – núcleo (Repulsivo)

Más propiamente dicho:

$$\hat{H} = -\frac{1}{2} \sum_{i=1}^N \nabla_i^2 - \frac{1}{2} \sum_{A=1}^M \frac{1}{M_A} \nabla_A^2 - \sum_{i=1}^N \sum_{A=1}^M \frac{Z_A}{r_{iA}} + \sum_{i=1}^N \sum_{j>i}^N \frac{1}{r_{ii}} + \sum_{A=1}^M \sum_{B>A}^M \frac{Z_A Z_B}{r_{AB}}$$

Al considerar que los núcleos son fijos, el termino de la energía cinética de los núcleos es eliminada y el termino de interacción de núcleo - núcleo se vuelve constante, y así podemos cambiar el origen del eje y volverlo 0 para simplificar la ecuación [13].

Resultando el Hamiltoniano como:

$$\hat{H}_{elec} = -\frac{1}{2} \sum_{i=1}^N \nabla_i^2 - \sum_{i=1}^N \sum_{A=1}^M \frac{Z_A}{r_{iA}} + \sum_{i=1}^N \sum_{j>i}^N \frac{1}{r_{ii}}$$

El método de Hartree-Fock es una de las primeras implementaciones computacionales para resolver la ecuación de Schrödinger. La teoría del funcional de la densidad se desarrollo para resolver las ecuaciones de Khon-Sham. En ambos métodos se diagonalizan matrices Hermitianas.

Comparando la ecuación de Schrödinger con el DFT, puede observarse que son muy similares [12]:

HF (1928, 1930)	DFT (1964, 1965)	
$E = E[\Psi, \mathbf{R}_\alpha]$	$E = E[\rho, \mathbf{R}_\alpha]$	(1ab)
$E = \int \Psi^* [\sum_i h_i + \sum_{i>j} 1/r_{ij}] \Psi d\tau$	$E = T[\rho] + U[\rho] + E_{xc}[\rho]$	(2ab)
$\Psi =  \psi_1(1), \psi_2(2), \dots, \psi_n(n) $	$\rho(\mathbf{r}) = \sum_{occ}  \psi_i(\mathbf{r}) ^2$	(3ab)
$\partial E / \partial \Psi = 0$	$\partial E / \partial \rho = 0$	(4ab)
$[-1/2 \nabla^2 + V_C(\mathbf{r}) + \mu_x^i(\mathbf{r})] \psi_i = \epsilon_i \psi_i$	$[-1/2 \nabla^2 + V_C(\mathbf{r}) + \mu_{xc}(\mathbf{r})] \psi_i = \epsilon_i \psi_i$	(5ab)

Figura 2 Comparación entre Ecuación de Schrödinger y Ecuación de Khon-Sham.

Donde para la ecuación de Schrödinger la energía se calcula en función de la función de onda para saber la posición de los electrones. Es necesario utilizar la sumatoria de las Hermitianas más el inverso de las posiciones en  $ij$ , por la función de onda. Y la función de onda se calcula con el valor absoluto de la sumatoria cada átomo. Dándonos así el principio variacional = 0. Con ello nos da como resultante la suma de la energía cinética más el potencial colombiano más el potencial de intercambio y correlación [12].

Por el lado del DFT, la energía esta en función de la densidad y la posición de los electrones. Y la energía se calcula con la energía cinética en función de la densidad [16], más la energía potencial también en función de la densidad y energía  $E_{xc}$  (de intercambio y correlación). Es importante mencionar que la densidad esta dada en función de la posición, para obtenerla, debe realizarse la sumatoria de los niveles ocupados del valor absoluto del cuadrado de la función de onda. Y nuevamente el principio variacional es igual a 0. Entonces la ecuación del DFT resulta al sumar la energía cinética más la energía colombiana más la energía de intercambio y correlación.

Puede observarse que ambas ecuaciones dan como resultado el vector energía por la longitud de onda.

Para resolver la ecuación, en escaso se procederá a transformar la expresión polinomio a una matriz del tipo cuadrática, la cual tiene la peculiaridad de ser simétrica, por lo que es aconsejable resolverla por el método numérico de rotaciones de Jacobi. Con dicho método se podrán encontrar los valores característicos y finalmente dar solución a la ecuación de Khon-Sham.

### 1.3 Modelado molecular en GPU

El modelado molecular en GPUs es una técnica que usa el poder de procesamiento de las tarjetas gráficas para realizar simulaciones moleculares. Está mayormente utilizado para realizar cálculos y simulaciones en el campo de la fisicoquímica cuántica. Existen ya algunos sistemas que implementan esta tecnología.

#### 2.9.2.1 TeraChem

Es un software de propósito general de química cuántica, está diseñado para correr en las GPU de NVIDIA, fue desarrollado en la Universidad de Stanford por el profesor Todd Martinez, quien fue pionero en el uso de la tecnología GPU para la química computacional.

TeraChem simula la dinámica y el movimiento de las moléculas, resolviendo la ecuación electrónica de Schrödinger para determinar las fuerzas entre los átomos, lo cual es denominado como “dinámica molecular ab initio”. Algunas de sus características son:

- Códigos de energía y gradiente de Kohn-Sham basados en grid y Hartree-Fock restringidos, no restringidos y restringidos.

- Cuadrículas DFT estáticas y dinámicas.
- Teoría funcional de la densidad dependiente del tiempo (TDDFT) y CI Singles (CIS).
- La optimización puede llevarse a cabo en coordenadas cartesianas o internas como se especifica en el archivo de inicio (todas las geometrías de entrada se proporcionan en cartesianos).
- Optimización restringida con átomos congelados, longitudes de enlaces restringidos, ángulos y diedros.
- Condiciones esféricas de frontera.
- Soporte completo para GPUs de NVIDIA.

### 2.9.2.3 deMon2k

Es un paquete de software para realizar cálculos con la Teoría del Funcional Densidad, el cual utiliza las combinaciones de orbitales de tipo Gauseano para dar solución a las ecuaciones Kohn-Sham [17].

La primera versión disponible de deMon apareció en 1992 en la Université de Montréal (UdM) como un proyecto de tesis de doctorado <sup>1</sup>.

deMon significa “densité de Montréal”.

Algunas características que contiene el programa son [18]:

- Ajuste variable del potencial de Coulomb.
- Optimización geométrica y búsqueda de estado de transición.
- Simulaciones dinámicas moleculares.
- Cálculo de propiedades como polarizabilidades, hiperpolarizabilidades, RMN, espectros e intensidades IR e Raman, datos termodinámicos.
- Código paralelo (MPI).
- Interfaces para software de visualización (Molden, Molekel, Vu).
- Portabilidad a varias plataformas informáticas y sistemas operativos.

El sistema está pensado para correr en sistemas Unix y Linux, la compilación se realiza bajo FORTRAN90. El programa también está pensado para correr en paralelo utilizando MPI (Message Passing Interface), pero únicamente si se tienen las bibliotecas instaladas.

La solución que se implementará, está basada en el módulo del cálculo de la energía por teoría del DFT de este paquete.

---

<sup>1</sup> Alain St-Amant, Ph.D. Thesis, Université de Montréal, 1992.

# Capítulo 2. Algoritmo de Jacobi

## 2.1 Forma cuadrática

Todo polinomio de segundo grado homogéneo de la forma:

$$q = X'AX = \sum_{i=1}^n \sum_{j=1}^n a_{ij}x_i x_j$$

$$q(x_1, x_2, \dots, x_n) = (a_{11}x_1^2 + a_{22}x_2^2 + 2a_{12}x_1x_2 + \dots + 2a_{n-1n}x_{n-1}x_n)$$

Dicho polinomio puede expresarse de una manera matricial de la forma:

$$q(x_1, x_2, \dots, x_n) = (x_1, x_2, \dots, x_n) \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = X'AX$$

Donde la matriz A asociada a la forma cuadrática es una matriz simétrica (Hermitiana) de orden n cuyos elementos de la diagonal principal son los coeficientes de los términos cuadráticos de la expresión polinómica, y los restantes elementos de la matriz son la mitad de los coeficientes de los términos no cuadráticos de dicha expresión. La matriz con forma cuadrática es simétrica, lo que significa que los términos de productos son semejantes ( $a_{ij} = a_{ji}$ ). Esta relación entre los elementos de una y otra expresión de la forma cuadrática, permite obtener fácilmente cada una de ellas a partir de la otra [19].

Ejemplo.

Se tiene el siguiente polinomio y se expresará la matriz de forma cuadrática.

$$q = x_1^2 + 2x_2^2 - 7x_3^2 - 4x_1x_2 + 8x_1x_3$$

$$q = X' \begin{bmatrix} 1 & -2 & 4 \\ -2 & 2 & 0 \\ 4 & 0 & -7 \end{bmatrix} X$$

## 2.2 Eigensistemas

En muchos problemas en el campo de las matemáticas aplicadas, es necesario el resolver ecuaciones lineales con la forma:

$$\begin{bmatrix} a_{11} - \lambda & a_{12} & a_{13} \\ a_{21} & a_{22} - \lambda & a_{23} \\ a_{31} & a_{32} & a_{33} - \lambda \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = 0$$

o de la forma:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

donde A es una matriz de  $n \times n$  con parámetros escalares  $\lambda$  conocidos como valores característicos o eigenvalores, x es una matriz columna de variables independientes también llamado *eigenvector*.

$$[A] \cdot [x] = \lambda[x]$$

Obviamente cualquier múltiplo de un eigenvector x podrá considerarse como eigenvector, excepto para este caso el 0. El problema es encontrar  $\lambda$  y su correspondiente eigenvector.

### 2.2.1 Transformación de coordenadas

Para mostrar el siguiente concepto, tenemos una matriz x como vector columna con dos componentes  $x_1$  y  $x_2$ , si ponemos el los ejes de  $\bar{x}_1$  y  $\bar{x}_2$  con el mismo origen, tenemos que el vector  $\overline{OD}$  tiene las dos diferentes componentes mencionadas [20]. Trigonométricamente tenemos:

$$\begin{aligned} x_1 &= \bar{x}_1 \cos \theta - \bar{x}_2 \sin \theta \\ x_2 &= \bar{x}_1 \sin \theta + \bar{x}_2 \cos \theta \end{aligned}$$

o en la forma matricial:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \end{bmatrix}$$

A esto se le llama la relación de transformación, lo que da:

$$[x] = [T][\bar{x}]$$

que conecta los dos sistemas x y  $\bar{x}$ .

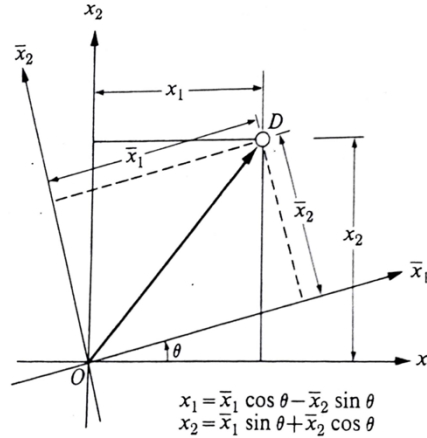


Figura 3 Giro de ejes de una matriz simétrica.

Lo importancia de la matriz T toma lugar cuando sustituimos la ecuación en la de los eigensistemas, dándonos:

$$[A][T][\bar{x}] = \lambda[T][\bar{x}]$$

pudiendo también:

$$[T]'[A][T][\bar{x}] = \lambda[T]'[T][\bar{x}]$$

recordando que  $T'T = I$ , comprobándolo con:

$$\begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

por ende deducimos que:

$$\begin{aligned} [T]'[A][T][\bar{x}] &= \lambda[I][\bar{x}] \\ &= \lambda[\bar{x}] \end{aligned}$$

### 2.2.2 Eigenvalores

Nuestro objetivo es que todos los elementos fuera de la diagonal se vuelvan cero, para ello es necesario seleccionar un ángulo de rotación que permita realizarlo.

$$\begin{aligned} B &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \\ &= \begin{bmatrix} a_{11} \cos^2 \theta + 2a_{12} \sin \theta \cos \theta + a_{22} \sin^2 \theta & a_{12}(\cos^2 \theta - \sin^2 \theta) + \sin \theta \cos \theta (a_{22} - a_{11}) \\ a_{12}(\cos^2 \theta - \sin^2 \theta) + \cos \theta \sin \theta (a_{22} - a_{11}) & a_{11} \sin^2 \theta - 2a_{12} \sin \theta \cos \theta + a_{22} \cos^2 \theta \end{bmatrix} \end{aligned}$$

Como queremos eliminar el elemento  $b_{12}$  fuera de la diagonal y teniendo en cuenta que  $b_{12}$  y  $b_{21}$  son similares:



$$a_{12}(\cos^2\theta - \sin^2\theta) + \cos\theta\sin\theta(a_{22} - a_{11}) = 0$$

por identidades trigonométricas podemos concluir:

$$\tan 2\theta = \frac{2a_{12}}{a_{11} - a_{22}}$$

Dando como resultado esperado:

$$B = \begin{bmatrix} a_{11}\cos^2\theta + 2a_{12}\sin\theta\cos\theta + a_{22}\sin^2\theta & 0 \\ 0 & a_{11}\sin^2\theta - 2a_{12}\sin\theta\cos\theta + a_{22}\sin^2\theta \end{bmatrix}$$

Con lo que al final del procedimiento podemos obtener que  $b_{11}$  y  $b_{22}$  son los eigenvalores deseados.

### 2.2.3 Eigenvector

Los eigenvectores son matrices columna  $[\bar{x}]$  de una matriz  $[A]$  a los que les corresponde un eigenvalor  $\lambda$ . Para agrupar los eigenvectores en una matriz cuadrada  $V$  únicamente se agrupan de izquierda a derecha. También es necesario agrupar los eigenvalores en una matriz diagonal  $[\lambda]$ . Así podemos expresar la igualdad:

$$[A][V] = [V][\lambda]$$

Si aplicamos un poco de algebra:

$$[V]^{-1}[A][V] = [V]^{-1}[V][\lambda]$$

Tomando en cuenta que  $[V]^{-1}[V] = [I]$ :

$$[V]^{-1}[A][V] = [\lambda]$$

Por ello podemos entender que la matriz cuadrada  $[V]$  que representa los vectores es igual a las multiplicaciones sucesivas de las matrices  $[T]$ .

$$[V] = [T_1][T_2] \cdots [T_m]$$

# Capítulo 3. Implementación en C, OpenACC y CUDA

Ahora bien, se implementará la resolución de la matriz de Khon-Sham en lenguaje C, posteriormente se realizará la programación en una tarjeta gráfica habilitada para OpenACC y para CUDA.

## 3.1 Cómputo en GPU

El computeo en GPU o también llamado GPGPU (cómputo de propósito general en unidades de procesamiento de gráficos) es utilizada para acelerar el cómputo realizado tradicionalmente por únicamente la CPU, donde la GPU actúa como un coprocesador que puede aumentar la velocidad el trabajo donde la computación es más intensiva aprovechando enorme potencia de procesamiento paralelo, mientras que el resto del código se ejecuta en la CPU [21].

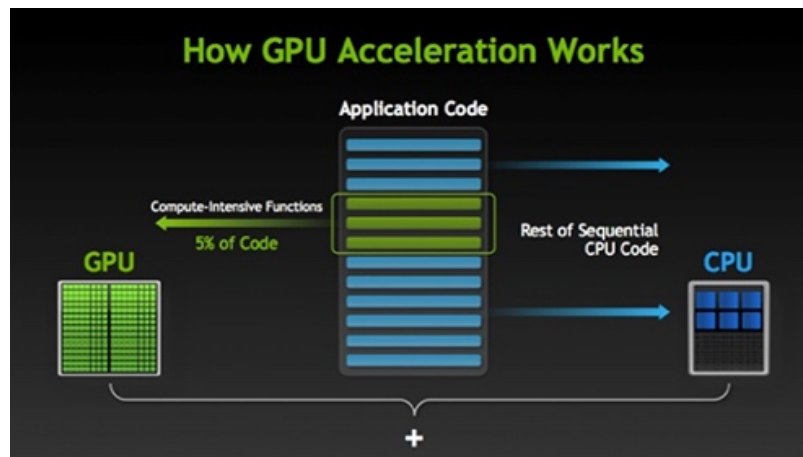


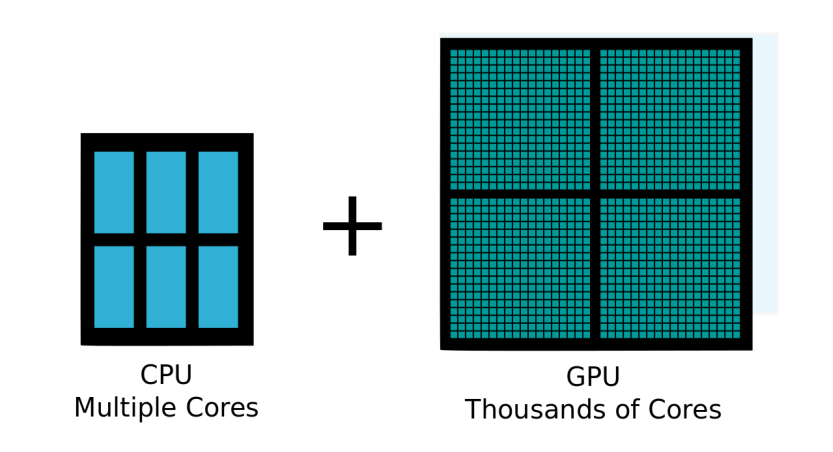
Figura 4 Aceleración de programas en GPUs.

### 3.1.1 Diferencia entre CPU y GPU

El CPU es un procesador de propósito general, lo que significa que puede hacer cualquier tipo de calculo, pero esta diseñada para realizar el procesamiento en serie. Aunque se pueden utilizar bibliotecas para realizar concurrencia y paralelismo, el hardware per se no tiene esa implementación [22].

El GPU es un procesador mucho más especializado para tareas que requieren de un alto grado de paralelismo. La tarjeta gráfica en su interior contiene miles de núcleos de procesamiento que son más pequeños y que por ende realizan un menor número de operaciones. Esto hace que la GPU esté optimizada para procesar cantidades enormes de

datos pero con programas más específicos [23]. Lo más común al utilizar la aceleración por GPU es ejecutar una misma instrucción a múltiples datos para aprovechar su arquitectura.



*Figura 5 Comparación entre Multicore y Manycore.*

Es necesario destacar que los manycore y los multicore son utilizados para etiquetar a los CPU y los GPU, pero entre ellos existen diferencias. Un core de CPU es relativamente más pesado, está diseñado para realizar un control lógico muy complejo para buscar y optimizar la ejecución secuencial de programas. En cambio un core de GPU es más ligero y está optimizado para realizar tareas de paralelismo de datos como un control lógico simple enfocándose en el throughput de los programas paralelos.

Con aplicaciones computacionales intensivas, las secciones del programa a menudo muestran una gran cantidad de paralelismo de datos. Las GPU se usan para acelerar la ejecución de esta porción de paralelismo de datos. Cuando un componente de hardware que está físicamente separado de la CPU se utiliza para acelerar secciones computacionalmente intensivas de una aplicación, se lo denomina acelerador de hardware. Se puede decir que las GPU son el ejemplo más común de un acelerador de hardware.

### 3.2 Características del equipo

Las pruebas se realizaron en el servidor NV que está instalado en la Facultad de Ciencias. Entre sus características:

Sistema Operativo	CentOS Linux release 7.3.1611
Arquitectura	64 bits
Dusco duro	1TB
Procesador	Intel(R) Core(TM) i7-7700 @ 3.60GHz
Tarjeta de video	Nvidia GeForce GTX 1060 6GB
Arquitectura de tarjeta de video	Pascal
Máximo de threads por bloque	1280
Memoria	6GB

### 3.3 Implementación en C

Ahora empezaremos la descripción de la implementación del algoritmo Jacobi para resolver la matriz de Khon-Sham en lenguaje C.

#### 3.3.1 Creación de matrices

Antes que nada se creó un programa llamado **creaMatrix.c**, el cual emula las matrices cuadráticas propias que se forman a partir de la ecuación de Khon-Sham, este programa genera archivos .txt con un formato especial, donde el primer elemento es el orden de la matriz, y los siguientes datos son los respectivos a cada posición de la matriz, y el formato del archivo a generar inicia con la cadena "matrix" y se le concatena el orden n de la matriz, concatenándosele al final la extensión ".txt". Por ejemplo:



```
3
2.0 -1.0 0.0
-1.0 2.0 -1.0
0.0 -1.0 2.0
```

Figura 6 Matriz de 3 x 3.

#### 3.3.2 Funciones auxiliares

Fue necesario crear una biblioteca llamada **auxFuncs.h** que contuviera algunas funciones auxiliares que se utilizarían a lo largo del programa completo. Esta biblioteca se basó en la que contiene el libro de Numerical Recipes [24]. Aquí se almacenan las funciones que obtienen el tiempo del sistema y la transforman a segundos, también contiene funciones que reservan memoria para vectores de elementos de tipo entero y de tipo flotante, y por último una función que en caso de existir algún problema en la asignación de la memoria dispara el código de error. Por términos de compatibilidad entre las versiones de código en las tecnologías que hacen uso de las tarjetas gráficas, las matrices se verán en forma de vectores.

##### 3.3.2.1 Matrices Vector

Lo más fácil para convertir una matriz a un arreglo es desdoblar cada renglón en serie.

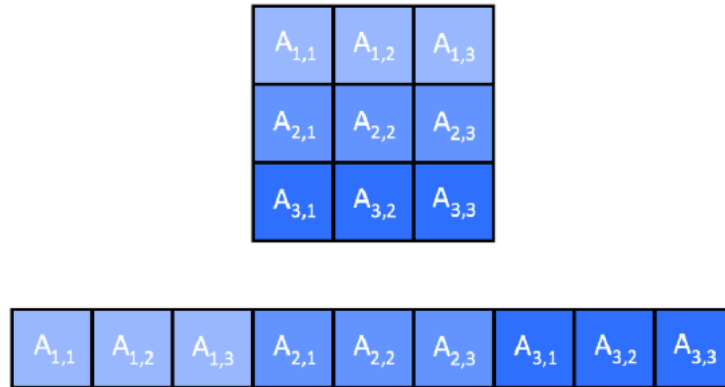


Figura 7 Conversión de matriz a arreglo.

Pasando así de  $A[i][j]$  a  $Am[i*n+j]$ , donde  $i$  representa el renglón,  $j$  la columna y  $n$  el orden de la matriz, pensando que sea cuadrada.

### 3.3.3 Driver

Para realizar las corridas del método numérico, se realizó un programa llamado **driverA.c** que contiene la función *main()*, aquí se deben agregar las bibliotecas **auxFuncs.h** y **jacobiA.h** que contienen las funciones necesarias para correr el código.

primero lee una matriz de un archivo que le fue pasado como parámetro al ejecutar el programa, dicho archivo debe contener el formato que ya sé específico. Si existe el archivo lo va leyendo y guardando en una matriz  $c$ , si no, retorna 1 y el programa termina con un error. Posteriormente al haber leído la matriz, empieza a reservar la memoria de una matriz de operación, la matriz que contendrá a los eigenvectores y un arreglo que contendrá a los eigenvalores. Para mantener la integridad de la matriz leída, se copia la información en la matriz de operación y se inicializa la estructura de eigenvectores con la matriz identidad. Para verificar que matriz es la que se va a operar, se imprime la matriz de operación.

Ahora se inicia la función *jacobiMultip()*, pero para cronometrar el tiempo que tarda en obtener las soluciones del método se obtiene el tiempo del sistema antes y después de realizar la subrutina. La función tiene como parámetros:

<i>jacobiMultip(mat,n,eval,nrot);</i>	<div style="background-color: #f0f0f0; padding: 2px;">mat -&gt; La matriz de operación.</div> <div style="background-color: #f0f0f0; padding: 2px;">n -&gt; El orden de la matriz.</div> <div style="background-color: #f0f0f0; padding: 2px;">eval -&gt; El eigenvector</div> <div style="background-color: #f0f0f0; padding: 2px;">eval -&gt; El arreglo de eigvec</div> <div style="padding: 2px;">nrot -&gt; Variable inicializada en 0 que da el numero de rotaciones que realizo el algoritmo</div>
---------------------------------------	---

Al momento de concluirse el cálculo se imprime la matriz de operación nuevamente, que ahora en su diagonal principal tendrá obligatoriamente en su diagonal principal los eigenvalores. Luego se imprime el eigenvalor con su eigenvector correspondiente.

Para finalizar el driver, se imprime tanto el número de rotaciones que se necesitaron para obtener el resultado final como el tiempo que tardo en llegar a la solución.

### 3.3.4 Jacobi

La biblioteca **jacobiA.h** contiene la función `jacobiMultip()`, la cual es el núcleo de operación del método numérico para resolver la matriz de Khon-Sham.

Dentro de esta matriz se debe reservar memoria para una matriz temporal que nos será de utilidad para auxiliar a los cálculos, una matriz *T* que será la que contendrá las transformaciones de las coordenadas y un pequeño vector que será el almacén de las coordenadas del elemento a eliminar de la matriz original.

Se empezará a iterar el método buscando el elemento máximo en el triangulo superior de la matriz y almacenando las coordenadas en el vector *piv\_element[]*. Las iteraciones las realizará hasta que se encuentre que el máximo elemento a eliminar es menor que el *threshold* (límite) recomendado por Numerical Recipies de  $1 \times 10^{-7}$  [24] o que haya superado las 50,000 iteraciones, en cualquiera de los casos termina el ciclo por que ha tomado el máximo elemento como `|0|`.

En caso de que aún existan elementos a eliminar, se realizan 5 funciones que hacen que el método numérico acerque los valores a cero. Las funciones son:

```
new_T_mat(piv_elem[0],piv_elem[1],n,mat,T,mat_temp);
```

Se recibe como parametros la posicion en *i* y en *j* del elemento máximo, el orden de la matriz, y la matriz *T* donde se guardaran las transformaciones de coordenadas. Inicializa la matriz *T* con la identidad, calcula el seno, el coseno y la tangente y los agrega en las posiciones necesarias para ir eliminando los valores de la matriz de operación.

```
mat_mult(n,eigvec,T,mat_temp);
```

Esta función multiplica el eigenvector con la matriz *T* y la asigna en la matriz auxiliar. La multiplicación de matrices que se realiza es trivial ya que se utilizan 3 for para recorrer las matrices y almacenar el producto en la tercer matriz.

```
copy_mat(n,mat_temp,eigvec);
```

Se recibe como parámetro el orden de la matriz original, la matriz auxiliar y el eigenvector. Lo que se realiza aquí es el copiado elemento a elemento de lo obtenido en la función anterior a el eigenvector para que se mantenga actualizado con los últimos valores.

```
mat_mult_tra(n,T,mat,mat_temp);
```

En esta sección es necesario realizar la premultiplicación y la postmultiplicación de la Matriz T por la matriz de operación, por lo que esta función lo que realiza es la operación de la traspuesta de T por mat, para evitar el calculo de la traspuesta, se modifica un poco la función de *mat\_mult()* y se cambia el orden de recorrido de los renglones de T, para que ahora recorra por columnas y la operación se guarda en la matriz auxiliar.

```
mat_mult(n,mat_temp,T,mat);
```

Por último se realiza la posmultiplicación de la matriz auxiliar por T y se almacena el resultado en la matriz de operación.

Lo anterior se realiza hasta que ya no exista un máximo, por lo que cuando termina el ciclo, la diagonal principal de la matriz se asigna a cada posición correspondiente de los eigenvalores.

Y termina la funcion *jacobiMultip()* devolviendonos al *main()* del driver.

### 3.3.5 Compilación

Para compilar el programa desde la terminal, es necesario unicamente tener el compilador GNU que está por defecto en los sistemas basados en UNIX.

La sintaxis de la compilacion debería ser:

```
gcc -o jacobi driverA.c
```

Y para correrlo, se le debe pasar como parametro el nombre de un archivo con la matriz a resolver.

```
./jacobi file.txt
```

Un ejemplo de la salida con una matriz de 3x3 es:

```

antonioayala — curso02@nv:~/tesina/jacobiCeroPruebas — ssh curso02@moo...
[[curso02@nv jacobiCeroPruebas]$ gcc -o jacobi driverA.c -lm -w
[[curso02@nv jacobiCeroPruebas]$ ./jacobi matrix3.txt

[2.000000][-1.000000][0.000000]
[-1.000000][2.000000][-1.000000]
[0.000000][-1.000000][2.000000]
***** Finding Eigenvalues *****

***** Eigenvalues *****

[3.414215][-0.000000][0.000000]
[-0.000000][2.000001][-0.000000]
[0.000000][-0.000000][0.585787]
eigenvalue 1, = 3.414215
eigenvector:
0.500000 0.707107 0.500000
eigenvalue 2, = 2.000001
eigenvector:
-0.707107 0.000000 0.707107
eigenvalue 3, = 0.585787
eigenvector:
0.500000 -0.707107 0.500000
Rotations: 29
Total time:0.000074 sec
[curso02@nv jacobiCeroPruebas]$

```

Figura 8 Salida de Jacobi con matriz de 3 x 3 en C.

## 3.4 Implementación en OpenACC

Se modificará el código para poder utilizar el estándar de OpenACC y paralelizar automáticamente en la tarjeta gráfica el algoritmo de Jacobi.

### 3.4.1 Estándar OpenACC

Como veremos posteriormente, CUDA al ofrecer un entorno de programación de bajo nivel, ha mantenido un poco alejado a los programadores que están fuera del desarrollo del cómputo científico, por ello se creó el estándar de programación OpenACC [25], que con solo algunas directivas indica al compilador que cierta sección de código podría ser paralelizable, y este se encargará de verificar si existen dependencias de dato o si es posible realizarse.

Al no manejarse completamente el comportamiento del programa a un bajo nivel, tiene la desventaja de que el rendimiento puede ser inferior al de CUDA, pero se acelera el desarrollo de los proyectos al no tenerse que modificar la lógica de programación.

Para poder utilizar el estándar es necesario tener el compilador PGI que tiene las bibliotecas de automatización del paralelismo.

#### 3.4.1.2 Estructura de CUDA

Consta de cuatro principales componentes: gangs, workers y vectors [26].

vectors	Es el elemento de granularidad más fina, su simil en SIMD es el thread.
workers	Son el intermediario entre gangs y vectors, permiten el mapeo de los vectors en el hardware.
gangs	Es un grupo o bloque de workers, en diferentes gangs, los workers pueden tener trabajos independientes.



#### 3.4.1.3 Paralelizar Bucles

Es necesario identificar los puntos estratégicos de paralelización como pueden ser las estructuras de bucle ya que aquí es donde se puede acelerar el procesamiento. En este caso no es necesario preocuparse del manejo de la memoria, únicamente es necesario verificar si se están obteniendo los datos correctos. La filosofía de OpenACC marca que es necesario asegurarse que primero se identifiquen los puntos estratégicos de paralelismo para después verificar cuál es la directiva más apropiada para el fragmento de código [26].

#### 3.4.1.4 Directivas

Para darle los parámetros de la configuración de threads al compilador, debe usarse *#pragma acc parallel num\_gangs(#), vector\_length(#)*, y por la versión de nuestra tarjeta gráfica usaremos 32 gangs de 64 workers para poder operar matrices de máximo 2048 renglones. Una vez dados los parámetros, debe indicarse que acción de OpenACC a utilizar. Sin profundizar mucho, la directiva más utilizada por su simplicidad es *#pragma acc loop*, al momento de identificar que una región puede ser paralelizable, se agrega la línea y automáticamente el compilador verificará si es posible realizar su paralelización.

En el ejemplo en que tenemos una multiplicación de matrices, por recomendación [26], se deben poner las directivas donde se realice el recorrido de los renglones y las columnas.

```
#pragma acc parallel num_gangs(32), vector_length(64)
{
    #pragma acc loop
    for (i = 1 ; i <= n ; i++) {
        #pragma acc loop
        for (j = 1 ; j <= n ; j++) {
            for (k = 1 ; k <= n ; k++) {
                A[i*n+j] += B[k*n+i] * C[k*n+j];
            }
        }
    }
}
```

La siguiente recomendación [26] es que si existe un bloque en donde no se realice otra operación que asignación de números sin tanta precisión como lo pueden ser los enteros, es preferible dejar que el CPU utilice sus pipelines para acelerar el código y no utilizar la directiva.

#### 3.4.1.5 Portabilidad

El estándar permite la portabilidad en el sentido que, si no se cuenta con el compilador PGI, puede utilizarse cualquier otro para C, y en el proceso de compilación simplemente se ignoraran las directivas.

### 3.4.2 JacobiACC

En la biblioteca **jacobiA.h** se buscan las posibles regiones que se puedan paralelizar.

La primera se encuentra en la función *max\_elem()* y la directiva se colocan al inicio del primer ciclo.

```
max_elem() {  
    ...  
    #pragma acc parallel num_gangs(32), vector_length(64)  
    {  
        #pragma acc loop  
        for (r = 0; r < n-1; r++)  
    }  
    ...  
}
```

La siguiente region paralelizable, se encuentra en las funciones *mat\_mult()* y *mat\_mult\_tra()*, Por lo que seguimos la recomendación y se colocan al inicio del primer y segundo for.

```
mat_mult(){  
    ...  
    #pragma acc parallel num_gangs(32), vector_length(64)  
    {  
        #pragma acc loop  
        for (i = 0 ; i < n ; i++) {  
            #pragma acc loop  
            for (j = 0 ; j < n ; j++) {  
                C[i*n+j] = 0.0;  
                for (k = 0 ; k < n ; k++)  
                    C[i*n+j] += A[i*n+k] * B[k*n+j];  
            }  
        }  
    }  
    ...  
}  
mat_mult_tra(){  
    ...  
    #pragma acc parallel num_gangs(32), vector_length(64)  
    {  
        #pragma acc loop  
        for (i = 0 ; i < n ; i++) {  
            #pragma acc loop  
            for (j = 0 ; j < n ; j++) {  
                C[i*n+j] = 0.0;  
                for (k = 0 ; k < n ; k++)  
                    C[i*n+j] += A[k*n+i] * B[k*n+j];  
            }  
        }  
    }  
}
```

```

        }
    }
    ...
}

```

Finalmente, se agrega la directiva en la función `copy_mat()` al inicio del primer `for`.

```

copy_mat(){
    ...
    #pragma acc parallel num_gangs(32), vector_length(64)
    {
        #pragma acc loop
        for (i = 0; i < n; ++i)
            for (j = 0; j < n; ++j)
                B[i*n+j]=A[i*n+j];
    }
    ...
}

```

Y así obtenemos nuestra versión de OpenACC a partir del código en C. Por lo que mantenemos el driver intacto.

### 3.4.3 Compilación

Para compilar, como ya se mencionó, es necesario tener el compilador PGI, en donde se activar la directiva de compilación `-acc` para indicar que se utilizará el estándar ahora indicaremos que automáticamente se le asigne los threads necesarios en cada matriz, utilizamos la directiva `-ta=multicore`.

La sintaxis de la compilación debería ser:

```
pgcc -acc -ta=multicore -o jacobiACC driverA.c
```

Y para correrlo, se le debe pasar como parametro el nombre de un archivo con la matriz a resolver.

```
./jacobiACC file.txt
```

Nuevamente mostrando un ejemplo con la misma matriz de 3x3:

```

[curso02@nv jacobiCeroPruebas]$ pgcc -acc -ta=multicore -o jacobiACC driverA.c -lm -w
[curso02@nv jacobiCeroPruebas]$ ./jacobiACC matrix3.txt

[2.000000][-1.000000][0.000000]
[-1.000000][2.000000][-1.000000]
[0.000000][-1.000000][2.000000]
***** Finding Eigenvalues *****

***** Eigenvalues *****

[3.414214][-0.000000][0.000000]
[-0.000000][2.000001][-0.000000]
[-0.000000][-0.000000][0.585787]
eigenvalue 1, = 3.414214
eigenvector:
0.500000 0.707107 0.500000
eigenvalue 2, = 2.000001
eigenvector:
-0.707107 -0.000000 0.707107
eigenvalue 3, = 0.585787
eigenvector:
0.500000 -0.707107 0.500000
Rotations: 29
Total time:0.009269 sec
[curso02@nv jacobiCeroPruebas]$

```

Figura 9 Salida de Jacobi con matriz de 3 x 3 en OpenACC.

## 3.5 Implementación en CUDA

### 3.5.1 Modelo de programación en CUDA

CUDA es el acrónimo en inglés de *Compute Unified Device Architecture*, el cual es una arquitectura de hardware y de software que permite ejecutar programas en las tarjetas gráficas de la marca NVIDIA [27]. Utiliza el modelo SIMT (Single Instruction, Multiple Thread) el cual está basado en el modelo SIMD [28], pero adecuado al la arquitecturas de las tarjetas gráficas.

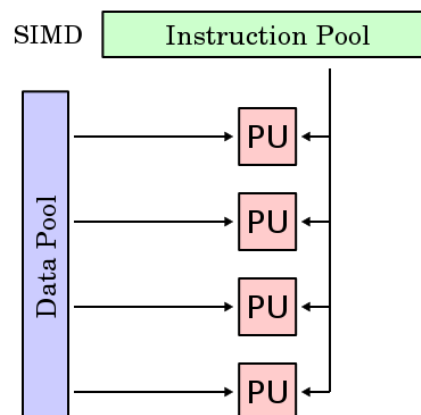


Figura 10 Diagrama de modelo SIMD.

La programación en CUDA es adecuada para resolver problemas que pueden ser expresados en paralelismo de datos, por la arquitectura que manejan.

Se explicará el caso particular de la extensión de CUDA en C, ya que es el lenguaje en el que la implementación del algoritmo se realizará.

### 3.5.1.1 Arquitectura CUDA

CUDA C es una extensión del estándar ANSI C con varios complementos del lenguaje para utilizar la programación heterogénea y también API sencillas para administrar los dispositivos, memoria y otras tareas. CUDA también es un modelo de programación escalable que permite a los programas trabajar transparentemente con un número variable de núcleos, aumentando o disminuyendo la cantidad de cores [29].

CUDA nos provee dos niveles de API para manejar el GPU y organizar los threads. El *API driver* es un API de nivel bajo y es un poco difícil de programar, pero provee un control sobre como el dispositivo (GPU) está siendo usado. El *API runtime* es un API a un nivel más alto, cada función de este nivel se rompe en operaciones más básicas del API driver.

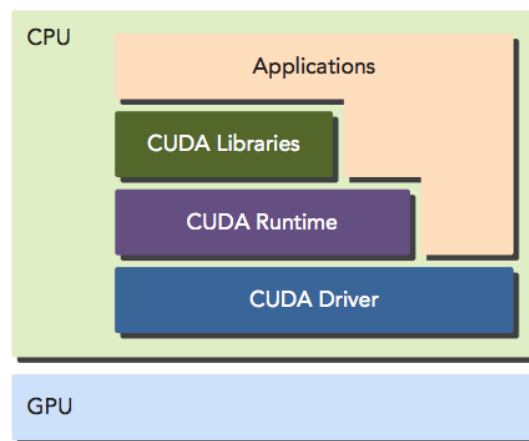


Figura 11 Arquitectura CUDA.

Un programa en CUDA consiste en la mezcla de dos códigos, el host code y el device code. El compilador de NVIDIA, *nvcc*, separa ambos códigos durante el proceso de compilación. Durante la etapa de enlace, las librerías de CUDA se agregan al procedimiento de las funciones que irán al device para poder manipular completamente la tarjeta.

El compilador *nvcc* está basado en LLVM<sup>2</sup>, por lo que el lenguaje puede ser extendido creando nuevos soportes para el GPU, eso sí, utilizando el CUDA Compiler SDK que nos comparte NVIDIA [29].

---

<sup>2</sup> LLVM (Low Level Virtual Machine) es una plataforma para desarrollar compiladores, que está diseñada para optimizar el tiempo de compilación, el tiempo de enlazado y el tiempo de ejecución en cualquier lenguaje de programación.

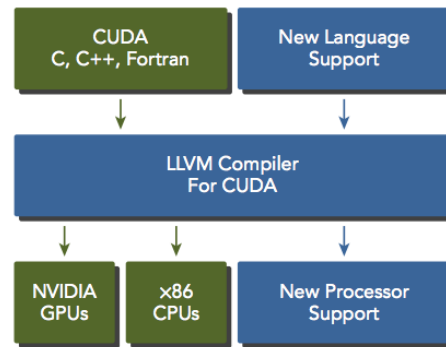


Figura 12 Jerarquía LLVM.

### 3.5.1.2 Estructura de CUDA

Consta de cuatro principales componentes: kernel, threads, bloques y grids.

kernel	Son las funciones paralelas escritas en el programa que indican que operaciones se realizaran en el GPU.
thread	Un thread de un bloque ejecuta una instancia de un kernel. Tiene su propio id dentro de su bloque, su propio PC, registros, memoria privada, entradas y salidas.
bloque	Un thread de un bloque ejecuta una instancia de un kernel. Tiene su propio id dentro de su bloque, su propio PC, registros, memoria privada, entradas y salidas.
grid	Es un arreglo de bloques de threads que ejecutan el mismo kernel, leen entradas desde memoria global, escriben resultados en memoria global, y se sincronizan entre otras llamadas a kernels.

Cada thread en CUDA tiene su propio contador de programa y registros. Todos los threads comparten un espacio de direcciones de memoria llamado “global memory”, el cual es la memoria con la que se comunican los bloques entre sí. Todos aquellos que se encuentren dentro de un bloque comparten el acceso a una memoria muy rápida llamada “shared memory”, pero el tamaño es más limitado.

NVIDIA define las características y limitaciones de CUDA con las “compute capabilities”, por ello los threads deben estar agrupados en bloques, de los cuales pueden poseer cuando mucho, en las últimas versiones, de 512 ó 1024 threads, dependiendo de la versión del CUDA. Los bloques de threads pueden ser de una, dos o tres dimensiones [29].

Hay que tener en cuenta que aunque todos los threads en cada bloque van a trabajar, lo primero que harán es localizar si se encuentran dentro de la matriz, si lo están, realizan la instrucción, pero si no, inmediatamente finalizarán.

### 3.5.1.3 Mapeo de Estructuras

Los threads son necesarios de organizar para que realicen un procesamiento ordenado y no generen condición de competencia. Por lo anteriores, es necesario diseñar la forma en la cual accederán a los datos.

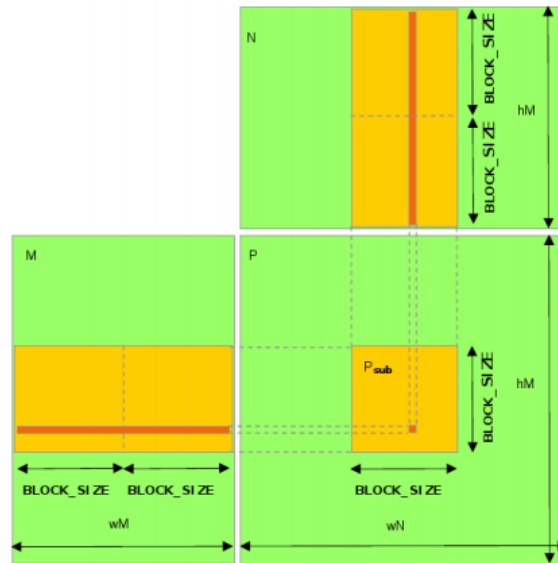


Figura 13 Mapeo de estructuras en CUDA.

Al momento de ejecutar el programa, se pueden obtener los id de las siguientes estructuras para verificar en que posición de procesamiento nos encontramos [30] [29].

gridDim.x , gridDim.y , gridDim.z	Numero de block que se encuentran a lo largo de las dimensiones x,y,z en el Grid.
blockIdx.x , blockIdx.y , blockIdx.z	El índice del bloque en el que se encuentra.
blockDim.x , blockDim.y , blockDim.z	El numero de threads que se encuentran en un bloque .
threadIdx.x , threadIdx.y , threadIdx.z	El id del thread en que se está procesando.

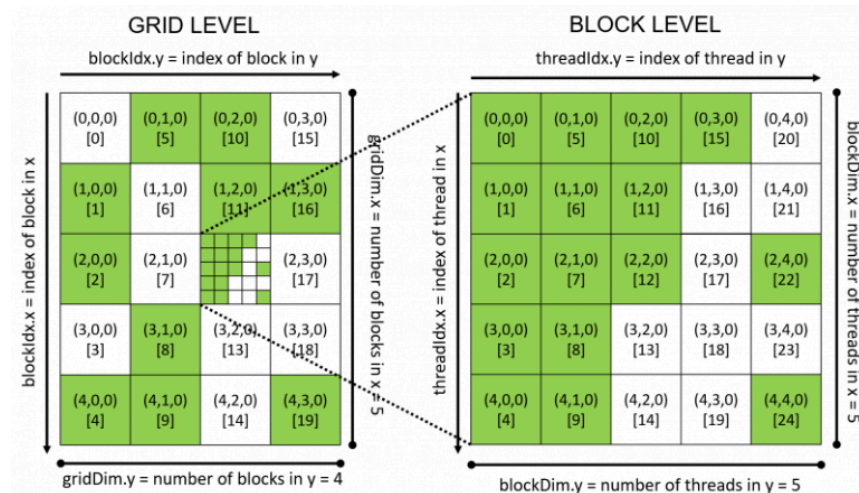


Figura 14 Dimensión de Grids y Blocks en CUDA.

#### 3.5.1.4 Kernels

El modelo de programación CUDA permite ejecutar aplicaciones en sistemas heterogéneos simplemente anotando el código con un pequeño conjunto de extensiones para el lenguaje de programación C, pero es importante decir que tanto el host como el device tienen memoria independiente, por lo que para acceder a los datos del otro es necesario enviar la información. Para ayudar a diferenciar entre que variables están en el host y en el device, se recomienda empezar con *d\_* las del segundo.

Un componente clave del modelo de programación de CUDA es el kernel, ya que se puede expresar un kernel como un programa secuencial, pero en realidad, CUDA administra la programación de ellos y los asigna a los threads [29] . La nomenclatura de un kernel es parecida a la de una función, pero hay que agregar la directiva `__global__` (doble guion bajo a cada lado. Para diferenciarlas de las funciones del host, se sugiere que empiecen con *kernel\_*. Un ejemplo de esto podría ser:

```
__global__ void kernel_helloFromGPU(argument list){}
```

Para poder lanzar el kernel, se debe utilizar la nomenclatura siguiente:

```
kernel_name <<<grid, block>>>(argument list);
```

El host puede operar independientemente del dispositivo, ya que, cuando se ha lanzado un kernel, el control se devuelve inmediatamente al host liberando a la CPU para realizar tareas adicionales complementadas por el código paralelo de datos que se ejecuta en el dispositivo [29]. El modelo de programación de CUDA es principalmente asíncrono, por lo que el cómputo del device puede superponerse con el procesamiento del dispositivo host y realizar actividades al mismo tiempo.

El proceso típico del flujo de datos de un programa en CUDA sigue el siguiente patrón:

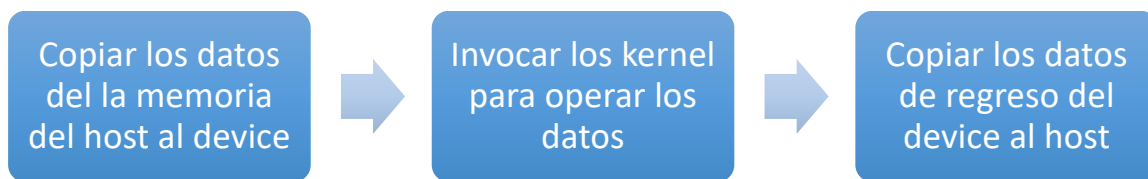


Figura 15 Pipeline de ejecución de CUDA.

#### 3.5.1.5 Manejo de memoria

Al diseñar un programa en paralelo con partición de datos, es necesario que dichos datos sean enviados a cada thread para que trabajen los datos asignados. Generalmente existen dos tipos de partición, la partición por bloques y la partición cíclica. En la partición por bloque muchos elementos consecutivos son empaquetados juntos, cada paquete es

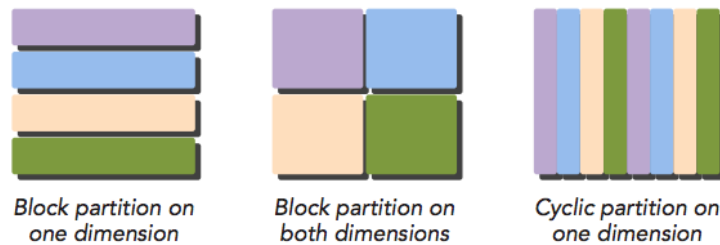


asignado a un thread en orden, por lo que cada thread procesa un paquete a la vez. En cambio la partición cíclica tiene menos datos empaquetados juntos, al terminar solicitan más trozos de información [29], resultando así en que pueden ir saltando entre los datos.



*Figura 16 Partición de datos en CUDA.*

Usualmente los datos están guardados en una dimensión, aunque el punto de vista del problema requiera una visualización multidimensional. Determinar el cómo distribuir datos entre threads está estrechamente relacionado con la forma en que los datos se almacenan físicamente, así como orden de ejecución de cada uno. La forma de organización de los threads tiene un efecto significativo en el rendimiento del programa.



*Figura 17 Partición de datos de una matriz en CUDA.*

El modelo de programación CUDA supone un sistema compuesto por un host y un dispositivo propia memoria separada entre si. El tiempo de ejecución de CUDA proporciona funciones para asignar y liberar memoria del dispositivo y transferir datos entre la memoria del host y la memoria del dispositivo.

Aquí hay una tabla que muestra las funciones correspondientes a C y a CUDA C para administrar la memoria

Funciones estándar de C	Funciones de CUDA C
<i>malloc</i>	<i>cudaMalloc</i>
<i>memcpy</i>	<i>cudaMemcpy</i>
<i>memset</i>	<i>cudaMemset</i>
<i>free</i>	<i>cudaFree</i>

La función utilizada para transferir datos entre el host y el device es `cudaMemcpy`, pero es necesario indicar cual es la fuente y el destino.

*cudaMemcpyHostToDevice*  
*cudaMemcpyDeviceToHost*

*cudaMemcpy(d\_a, h\_a, nBytes, cudaMemcpyHostToDevice);*  
*cudaMemcpy(h\_b, d\_b, nBytes, cudaMemcpyDeviceToHost);*

Esta función tiene un comportamiento síncrono, ya que el host se bloquea hasta que la función regrese una bandera de que la transferencia se realizó con éxito [29].

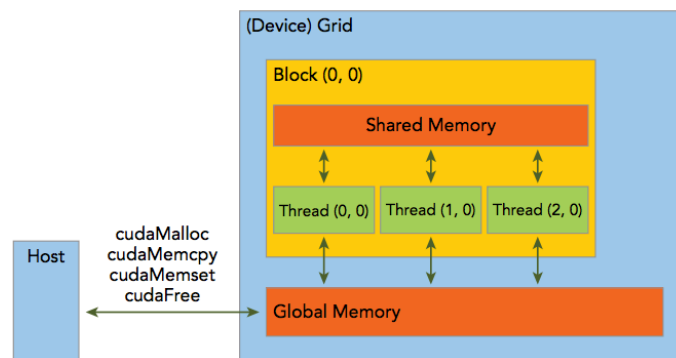


Figura 18 Transferencia de datos entre Host y Device en CUDA.

### 3.5.1.6 Trabajando con matrices

Cuando se trabaja con matrices en CUDA es necesario convertirlas a arreglos de una dimensión. Esto puede sonar un poco confuso, pero el problema es el lenguaje en sí, ya que necesita saber el número de columnas que se tendrán antes de compilar el programa, lo que hace imposible cambiarlo en la mitad del código. Por ello no podemos utilizar la notación de  $A[i][j]$  si no que debemos utilizar una manera para indexar correctamente entre renglones y columnas [29]. Como ya habíamos visto anteriormente, lo más fácil para convertir una matriz a un arreglo es desdoblarse cada renglón en serie, pasando así de  $A[i][j]$  a  $Am[i*n+j]$ , donde  $i$  representa el renglón,  $j$  la columna y  $n$  el número total de columnas.

### 3.5.1.7 Id de Threads

Al trabajar con matrices, es necesario que verifiquemos que sección de threads ingresaran a la estructura de datos para evitar duplicidad de operaciones o, al contrario, estemos fuera del rango de la matriz.

threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2							

Figura 19 Orden de ID de los threads de CUDA.

### 3.5.2 Driver

Al código del driver, únicamente se le quitará la biblioteca **jacobiA.h** y se intercambiará por **jacobiACUDA.h**, y el siguiente cambio únicamente es el de cambiar la extensión **\*.c** por **\*.cu**

### 3.5.3 JacobiCUDA

En la biblioteca que contiene la función de Jacobi es donde se realizan los cambios mayores, ya que es necesario mover la información entre el host y el device.

Por cada una de las variables que se tienen originalmente es necesario crear su homologo en el device, por lo que, por buenas practicas, se debe iniciar el nombre con `d_{variable}`. Ahora es necesario indicar las dimensiones de los grid y bloques, por las características que se tienen en la computadora, se decidió tener una dimensión de grids con 32 bloques de 64 threads cada uno, esto nos permitirá realizar matrices máximo de 2048 x 2048.

Ahora es necesario reservar la memoria en el device para las variables que se operaran, para proceder a copiar la información del host al device. Una vez superado este paso las operaciones ser realizaran dentro del device, pero serán controladas por el CPU. Las funciones traducidas a CUDA son las siguientes:

```
__global__ kernel_max_elem<<<1,1>>>(d_piv_elem,n,d_mat);
```

Ya que el elemento máximo únicamente se va a buscar en el triangulo superior, se pude utilizar un solo thread para realizar la operación.

```
__global__ kernel_set_max_elem<<<1,1>>>(d_mat,n,d_piv_elem,d_max_elem);
```

Como la operacion se realiza en el device, es necesario encontrar ese valor máximo y bajarlo al host para que el sea el que determine si ya se llegó al threshole o aun se necesita iterar.

```
__global__ kernel_new_T_mat<<<dimGrid, dimBlock>>>(d_piv_elem,n,d_mat,d_T);
```

Cuando todavia hay necesidad de aplicar el método numérico, se debe calcular la matriz de transformación de coordenadas. Donde se checa que los threads que esten fuera del rango de la matriz terminen sin afectar la operación. A cada thead se le encarga que a la columna que les tocó les asignen ceros a todos sus renglones, posteriormente el elemento que esta dentro de la diagonal principal le da el valor 1, para formar la matriz identidad. Si a el thread le toca en su columna un elemento de la transformacion de coordenadas, debe calcular el seno, coseno y tangente y lo asigna a las posiciones que necesitan ser rotadas. Dichas fuciones ahora están únicamente disponibles para ser llamadas en el device porque se les antecede la directiva `__device__`.

```

__device__ float cal_tan()
__device__ float cal_cos()
__device__ float cal_sin()

```

```

__global__ kernel_mat_mult<<<dimGrid, dimBlock>>>(n,d_eigvec,d_T,d_mat_temp);

```

En esta función, se le agrega el índice de los threads de la tarjeta, y a las columnas y el de las x a las columnas. Nuevamente se descartan los threads que caen fuera de la matriz. Por la naturaleza del índice de los threads, a cada thread se le asigna un renglón de la primera matriz a cada thread una columna de la matriz b. Y es necesario anidar un ciclo para ir recorriendo cada uno de los elementos y operarlos.

```

__global__ kernel_copy_mat<<<dimGrid, dimBlock>>>(n,d_mat_temp,d_eigvec);

```

Esta operación es trivial, ya que a cada thread se le asigna una columna de la matriz resultante y el elemento que se encuentra en la diagonal principal lo copia en el arreglo de eigenvalores.

```

__global__ kernel_mat_mult_tra<<<dimGrid, dimBlock>>>(n,d_T, d_mat, d_mat_temp);

```

Esta es la sección del código donde hay que poner más atención, ya que el algoritmo pide que se transponga la matriz T, pero para ahorrarnos operaciones, vamos a agregar un índice que nos permita recorrer ambas matrices por las columnas, pero que se conserve el orden de llenado de la matriz solución.

```

__global__ kernel_mat_mult<<<dimGrid, dimBlock>>>(n,d_mat_temp,d_T,d_mat);

```

Aquí, al igual que en el cálculo del eigenvector, se multiplica renglón por columna de lo obtenido en la función anterior por la matriz de transformación T.

```

__global__ kernel_copy_diag<<<dimGrid, dimBlock>>>(n,d_mat,d_eigval);

```

Al superar el threshold, se rompe el ciclo y se copian los elementos de la diagonal principal al arreglo de eigenvalores.

Ahora que se tienen el resultado, es necesario copiarlo de regreso al host para que se pueda mostrar en pantalla los resultados `cudaMemcpy(..., ..., ..., cudaMemcpyDeviceToHost)`. Y como ya se llegó al fin del método numérico, es necesario liberar la memoria en el device, esto con la sentencia de `cudaFree()` a cada una de las variables del dispositivo.

### 3.5.4 Compilación

Para compilar el código, es necesario tener el compilador NVCC de NVIDIA, y que los archivos de programa tengan terminación `*.cu`.

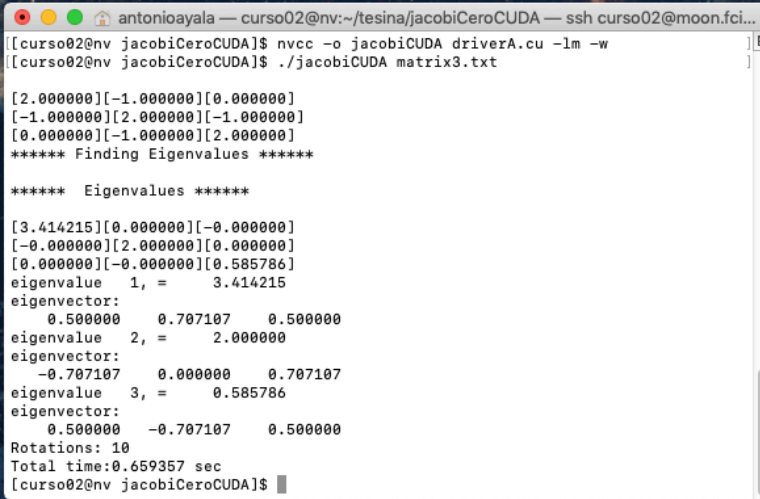
La sintaxis de la compilacion debería ser:

```
nvcc -acc -o jacobiCUDA driverA.cu
```

Y para correrlo, se le debe pasar como parametro el nombre de un archivo con la matriz a resolver.

```
./jacobiCUDA file.txt
```

Ahora mostrando un ejemplo con la misma matriz de 3x3:



```
antonioayala — curso02@nv:~/tesina/jacobiCeroCUDA — ssh curso02@moon.fci...
[curso02@nv jacobiCeroCUDA]$ nvcc -o jacobiCUDA driverA.cu -lm -w
[curso02@nv jacobiCeroCUDA]$ ./jacobiCUDA matrix3.txt

[2.000000][-1.000000][0.000000]
[-1.000000][2.000000][-1.000000]
[0.000000][-1.000000][2.000000]
***** Finding Eigenvalues *****

***** Eigenvalues *****

[3.414215][0.000000][-0.000000]
[-0.000000][2.000000][0.000000]
[0.000000][-0.000000][0.585786]
eigenvalue 1, = 3.414215
eigenvector:
0.500000 0.707107 0.500000
eigenvalue 2, = 2.000000
eigenvector:
-0.707107 0.000000 0.707107
eigenvalue 3, = 0.585786
eigenvector:
0.500000 -0.707107 0.500000
Rotations: 10
Total time:0.659357 sec
[curso02@nv jacobiCeroCUDA]$
```

Figura 20 Salida de Jacobi con matriz de 3 x 3 en CUDA.

# Capítulo 4. Pruebas y resultados

## 4.1 Métricas de desempeño

Existe una gran creciente demanda de poder de cómputo y algunas áreas que requieren gran velocidad en los cálculos mayormente están en el campo de la simulación científica u problemas de ingeniería, por ello los cálculos deben ser resueltos en un periodo “razonable” de tiempo. Aparte de la necesidad de incrementar la velocidad, el uso de varios procesadores ofrece comúnmente una solución más precisa al poder partir el problema en más subtarear.

Cuando se desarrollan soluciones en paralelo la primer pregunta que surge es “el cuán más rápido se resuelve el problema”, pero antes de saber cuanto se acelera en paralelo es necesario enfrentar el “mejor” algoritmo secuencial (en un solo procesador) para saber la aceleración y ver si compensa la inversión en el sistema paralelo, ya que si no se hace así es mejor buscar técnicas de optimización de código ahorrándose costos y tiempo.

Ahora bien, al saber que se tiene la mejor versión de la solución en secuencial podemos compararla contra la paralela. Es necesario aclarar que las mediciones no son exactas, si no un promedio ya que existen más factores que influyen como la asignación de recursos del sistema operativo o la cantidad de memoria que se posea entre otras cosas [31].

### 4.1.2 Runtime

El Runtime o tiempo de ejecución, hace referencia al tiempo que pasa entre el inicio del programa y la finalización de todos los procesos.

$$t(p)$$

### 4.1.2 Cost Factor

Representa la cantidad de trabajo realizado por el programa, ya que, aunque disminuye el tiempo al tener un sistema paralelo, los procesadores están trabajando al mismo tiempo, por ello muchas veces es necesario conocer la totalidad del costo que conlleva la utilización del algoritmo.

$$C(p) = \text{número de procesadores} * \text{runtime con } p \text{ procesadores}$$

$$C(p) = p * t_p$$

### 4.1.3 Speedup Factor

Es la medición relativa del rendimiento de un programa en paralelo, para calcularla se necesita medir el tiempo que tarda la versión secuencial entre la versión paralela. Hay que

tener en cuenta que se debe calcular el tiempo con la misma cantidad de datos a procesar para que la medición sea justa.

$$S(p) = \frac{\text{runtime en un sólo procesador}}{\text{runtime con } p \text{ procesadores}}$$

$$S(p) = \frac{t_s}{t_p}$$

#### 4.1.3 Aceleración

Es la medición de la aceleración relativa de un programa en paralelo. Es similar al Speed Up Factor, pero da el porcentaje de la aceleración con respecto al programa en un solo procesador.

$$AC(p) = \frac{\text{runtime en un sólo procesador}}{\text{runtime con } p \text{ procesadores}} - \frac{\text{runtime en un sólo procesador}}{\text{runtime en un sólo procesador}}$$

$$Ac(p) = \frac{t_s}{t_p} 100\% - 100\%$$

#### 2.1.4 Efficiency

A veces es útil conocer cuanto tarda cada procesador en realizar su tarea, regularmente se expresa en porcentaje, por ejemplo, si se obtiene el 50% significa que se están utilizando la mitad del tiempo en promedio que en la versión secuencial.

$$E = \frac{\text{runtime en un sólo procesador}}{\text{runtime con } p \text{ procesadores} \times \text{número de procesadores}}$$

$$E = \frac{t_s}{t_p * p} 100\%$$

$$E = \frac{t_s}{C(p)} 100\%$$

$$E = \frac{S(p)}{p} 100\%$$

## 4.2 Resultados

### 4.2.1 Resultados de Runtime

RUNTIME [s]			
Orden	Serial	OpenACC	CUDA
3	0.000074	0.008546	0.667713
5	0.079594	0.105867	0.667437
10	0.537204	0.277441	0.663814
16	2.272125	0.606796	0.693325
32	16.624936	6.420716	1.075765
64	127.455333	61.942851	6.336773
128	1127.73628	592.659655	87.821701
256	10858.2201	5429.11005	516.680462
512	93691.0741	46845.537	2495.80583
1024	493691.074	246845.537	12754.6959

RUNTIME [min]			
Orden	Serial	OpenACC	CUDA
3	0.000001	0.0001	0.0111
5	0.0013	0.0018	0.0111
10	0.0090	0.0046	0.0111
16	0.0379	0.0101	0.0116
32	0.2771	0.1070	0.0179
64	2.1243	1.0324	0.1056
128	18.7956	9.8777	1.4637
256	180.9703	90.4852	8.6113
512	1561.5179	780.7590	41.5968
1024	8228.1846	4114.0923	212.5783



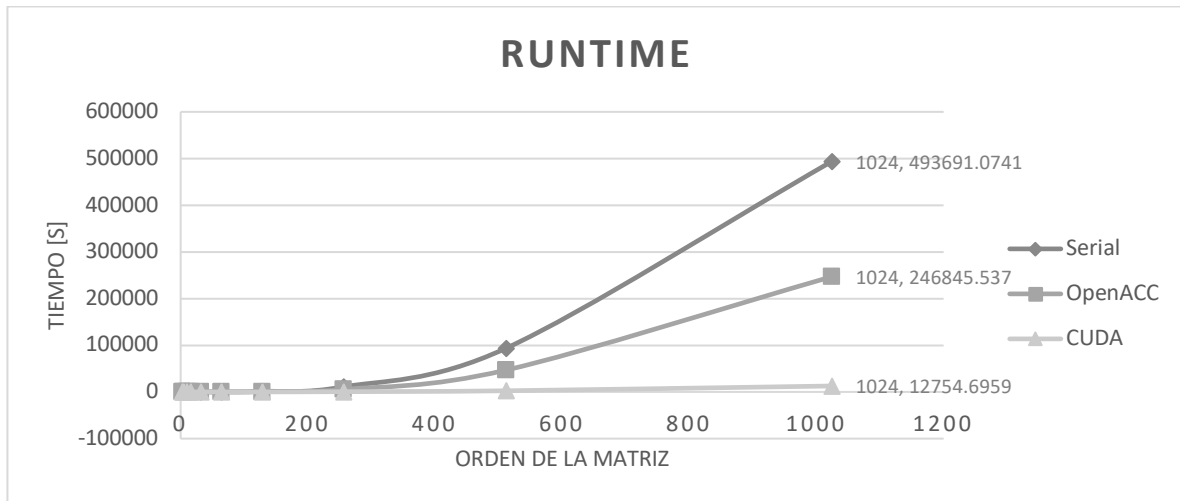


Figura 21 Gráfica de RUNTIME.

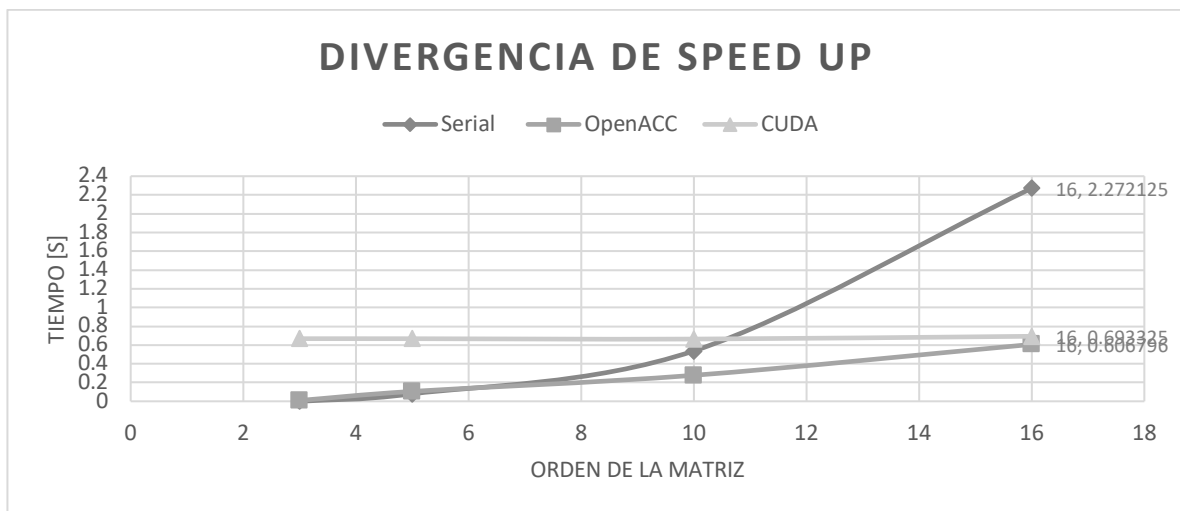


Figura 22 Gráfica del inicio de la divergencia del RUNTIME.

#### 4.2.2 Resultados de Cost Factor

COST FACTOR						
Orden	Procesadores	Serial	Procesadores	OpenACC	Procesadores	CUDA
3	1	0.000074	32	0.273472	3	2.003139
5	1	0.079594	32	3.387744	5	3.337185
10	1	0.537204	32	8.878112	10	6.63814
16	1	2.272125	32	19.417472	16	11.0932
32	1	16.624936	32	205.462912	32	34.42448
64	1	127.455333	32	1982.17123	64	405.553472
128	1	1127.73628	32	18965.109	128	11241.1777
256	1	10858.2201	32	173731.522	256	132270.198

<b>512</b>	<b>1</b>	93691.0741	32	1499057.19	512	1277852.59
<b>1024</b>	<b>1</b>	493691.074	32	7899057.19	1024	13060808.6

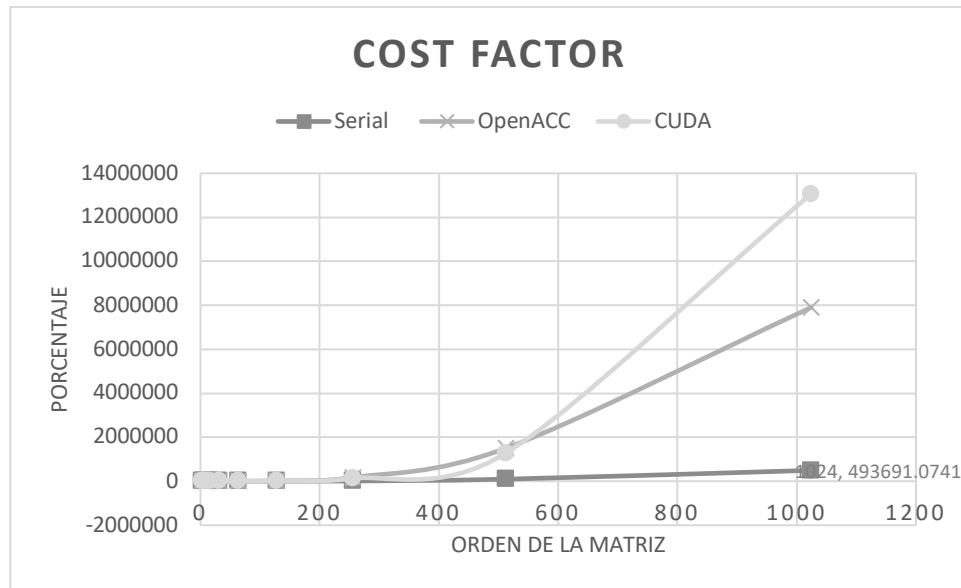


Figura 23 Factor de Costo total.

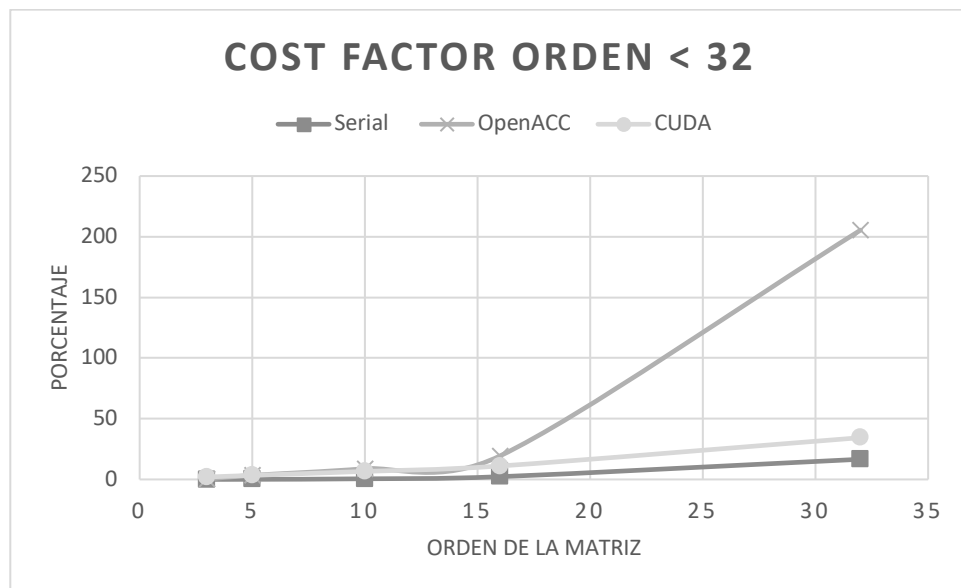


Figura 24 Factor de Costo hasta matriz de 32 x 32.

### 4.2.3 Resultados de Speed Up

Speed UP			
Orden	Serial	OpenACC	CUDA
3	100.00%	0.87%	0.01%
5	100.00%	75.18%	11.93%
10	100.00%	193.63%	80.93%
16	100.00%	374.45%	327.71%
32	100.00%	258.93%	1545.41%
64	100.00%	205.76%	2011.36%
128	100.00%	190.28%	1284.12%
256	100.00%	200.00%	2101.53%
512	100.00%	200.00%	3753.94%
1024	100.00%	200.00%	3870.66%

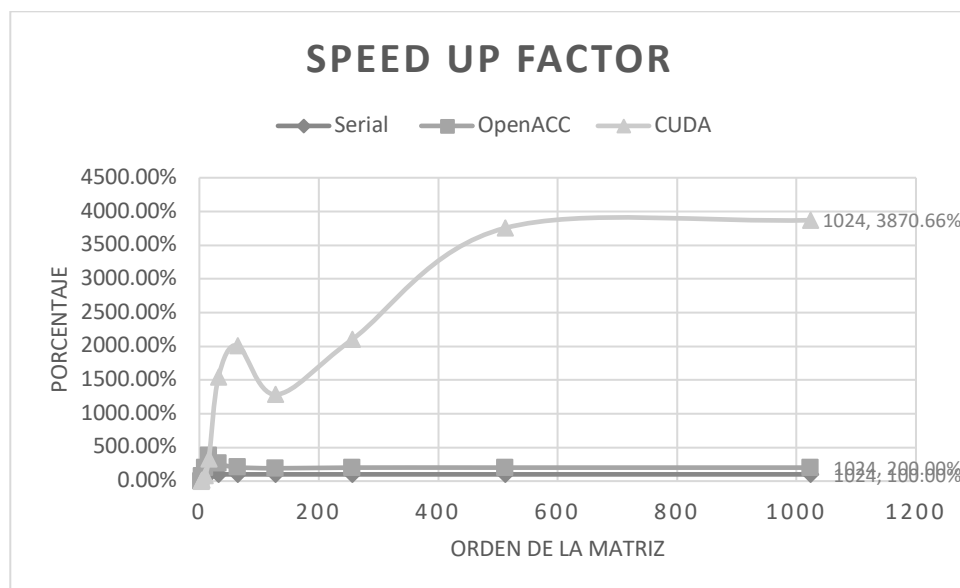


Figura 25 Speed up total.

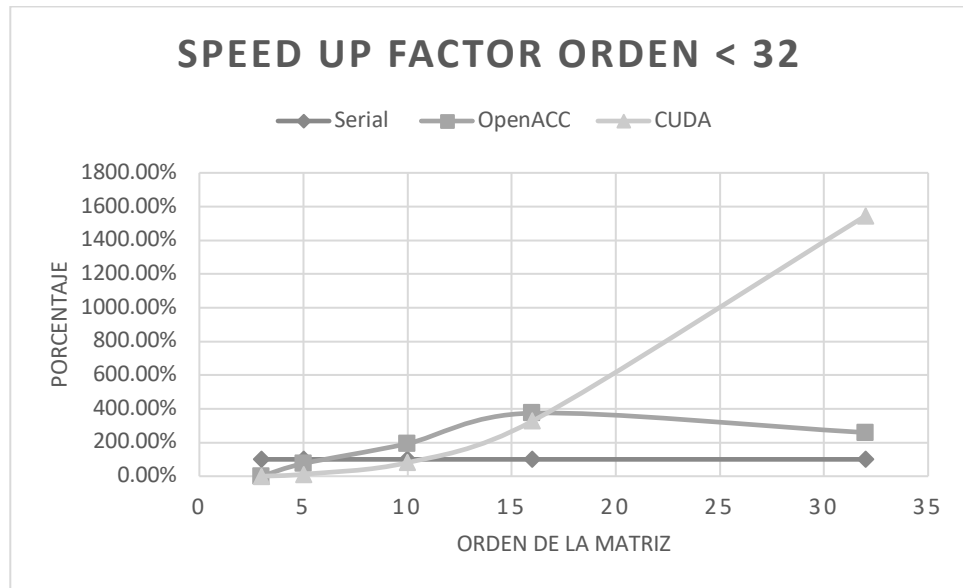


Figura 26 Speed up de las primeras matrices.

#### 4.2.3 Resultados de Aceleración

Aceleración			
Orden	Serial	OpenACC	CUDA
3	0.00%	-99.13%	-99.99%
5	0.00%	-24.82%	-88.07%
10	0.00%	93.63%	-19.07%
16	0.00%	274.45%	227.71%
32	0.00%	158.93%	1445.41%
64	0.00%	105.76%	1911.36%
128	0.00%	90.28%	1184.12%
256	0.00%	100.00%	2001.53%
512	0.00%	100.00%	3653.94%
1024	0.00%	100.00%	3770.66%

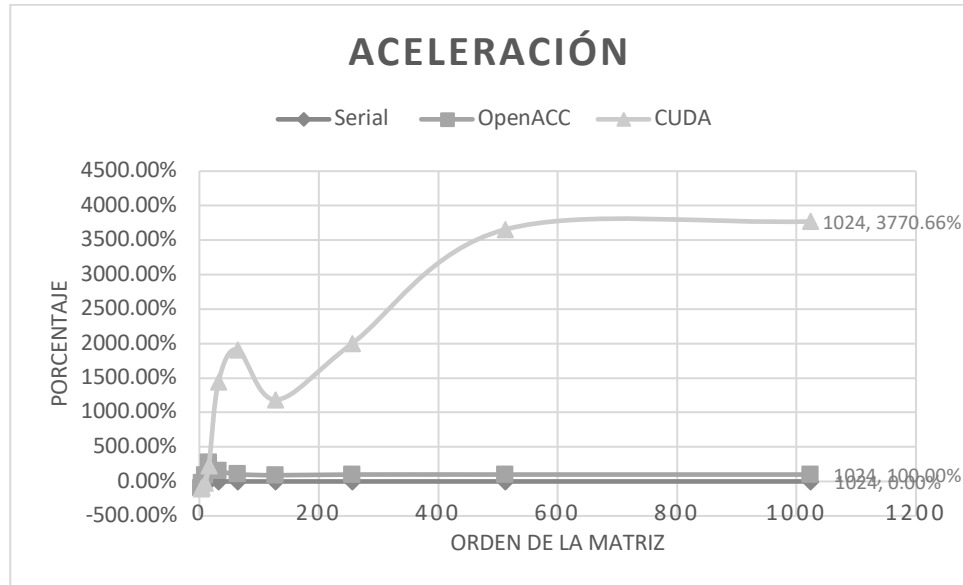


Figura 27 Aceleración total.

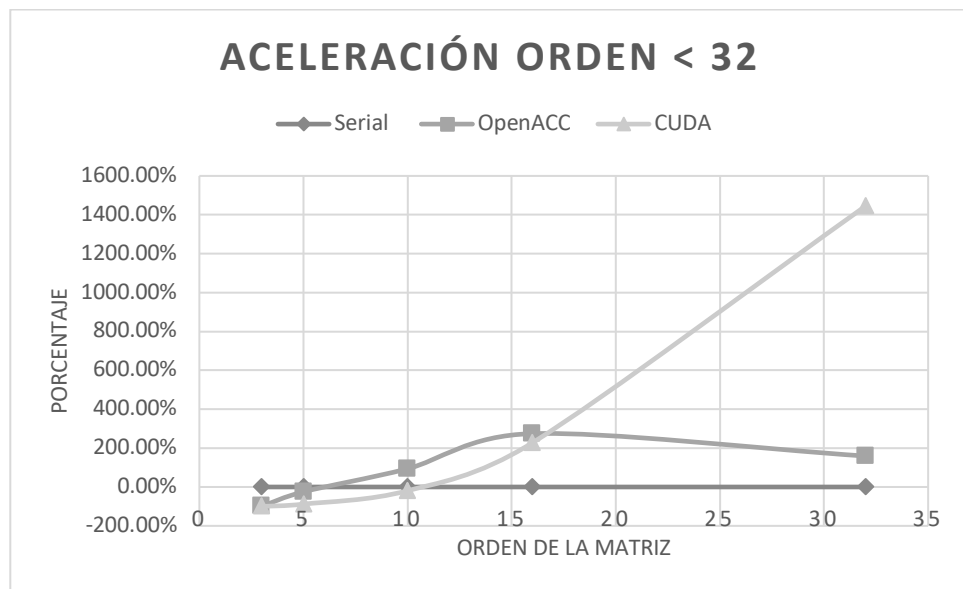


Figura 28 Aceleración de las primeras matrices.

#### 4.2.4 Resultados de Efficiency

Efficiency			
Orden	Serial	OpenACC	CUDA
3	100.00%	0.03%	0.00%
5	100.00%	2.35%	2.39%
10	100.00%	6.05%	8.09%
16	100.00%	11.70%	20.48%
32	100.00%	8.09%	48.29%
64	100.00%	6.43%	31.43%
128	100.00%	5.95%	10.03%
256	100.00%	6.25%	8.21%
512	100.00%	6.25%	7.33%
1024	100.00%	6.25%	3.78%

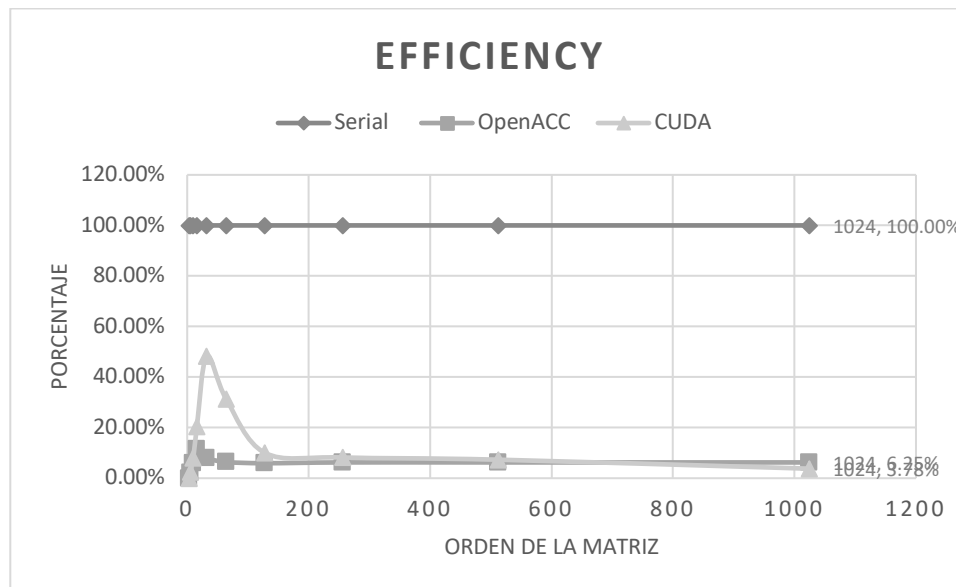


Figura 29 Eficiencia.

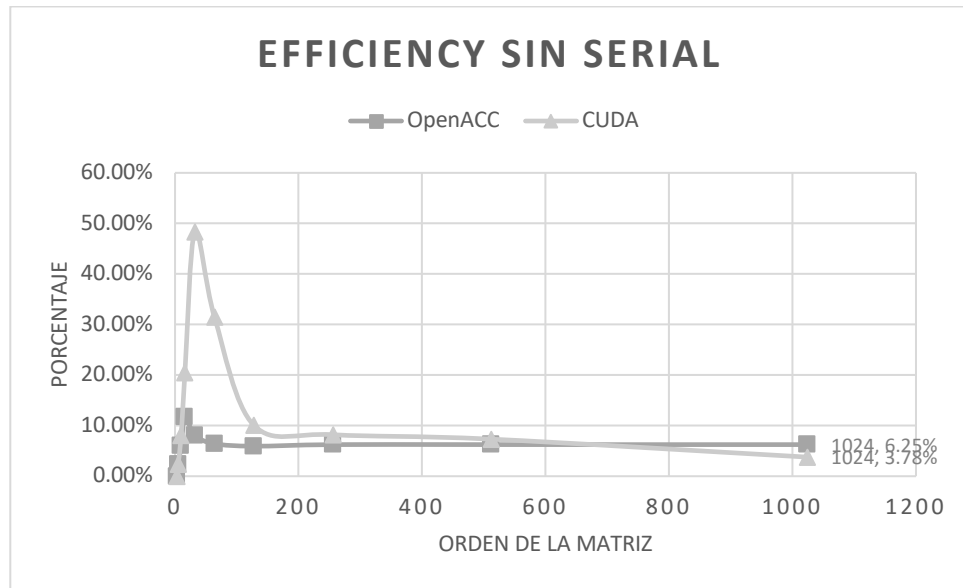


Figura 30 Eficiencia sin los valores del programa en C.

## 4.3 Resultados de una matriz

### 4.3.1 Matriz a evaluar

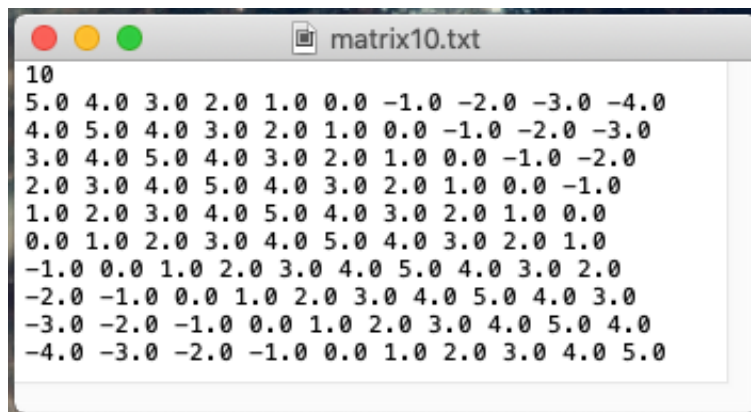


Figura 31 Matriz de 10 x 10.

### 4.3.2 Resultado Serial

```

eigenvalue 1, = 20.431740
eigenvector:
0.441217 -0.073035 0.401183 0.197568 -0.339168
0.294600 0.218292 0.397703 0.118752 0.422681

eigenvalue 2, = 20.431759
eigenvector:
0.442172 0.066901 0.076475 0.440229 0.294792
0.337777 0.185343 -0.409752 -0.249987 -0.367442

eigenvalue 3, = 2.425922
eigenvector:
0.399878 0.200240 -0.311234 0.320889 0.337301
-0.293735 -0.433588 0.088906 0.357426 0.277221

eigenvalue 4, = 2.425919
eigenvector:
0.318415 0.314004 -0.443101 -0.063785 -0.292322
-0.336648 0.319787 0.302760 -0.427282 -0.157214

eigenvalue 5, = 1.001439
eigenvector:
0.205833 0.397046 -0.208551 -0.394786 -0.337161
0.291252 0.064531 -0.440893 0.451294 0.017505

eigenvalue 6, = 1.001456
eigenvector:
0.073014 0.441227 0.196894 -0.401674 0.290013
0.338084 -0.399219 0.209093 -0.427239 0.127425

eigenvalue 7, = 0.630436
eigenvector:
-0.066882 0.442181 0.440885 -0.076840 0.339069
-0.290404 0.404236 0.197218 0.363059 -0.258254

eigenvalue 8, = 0.630389
eigenvector:
-0.200245 0.399861 0.320962 0.311894 -0.291170
-0.340303 -0.077261 -0.438853 -0.268425 0.359037

eigenvalue 9, = 0.512564
eigenvector:
-0.314002 0.318425 -0.063942 0.442968 -0.340310
0.292372 -0.311406 0.318851 0.150550 -0.421300

eigenvalue 10, = 0.512567
eigenvector:
-0.397058 0.205822 -0.395290 0.208628 0.294067
0.340206 0.445930 0.063800 -0.018577 0.441721

Rotations: 50001
Total time:0.537204 sec

```

Figura 32 Resultados de Jacobi en Serial de una matriz de 10 x 10.



### 4.3.3 Resultado OpenACC

```

eigenvalue 1, = 20.431728
eigenvector:
0.443527 -0.057232 0.412714 0.171080 -0.254867
-0.365212 -0.180204 0.415229 -0.186228 0.403589

eigenvalue 2, = 20.431734
eigenvector:
0.439526 0.082610 0.104970 0.434902 0.365096
-0.256018 -0.225998 -0.389645 0.046177 -0.443116

eigenvalue 3, = 2.425918
eigenvector:
0.392478 0.214368 -0.291076 0.340262 0.256921
0.365836 0.439985 0.045246 0.099593 0.441018

eigenvalue 4, = 2.425920
eigenvector:
0.307011 0.325193 -0.446061 -0.035359 -0.367040
0.257176 -0.287327 0.329481 -0.231835 -0.392988

eigenvalue 5, = 1.001523
eigenvector:
0.191503 0.404144 -0.233129 -0.381967 -0.256883
-0.368468 -0.104666 -0.427878 0.337426 0.300806

eigenvalue 6, = 1.001410
eigenvector:
0.057246 0.443517 0.171289 -0.412616 0.369530
-0.256010 0.415015 0.171221 -0.407108 -0.177290

eigenvalue 7, = 0.630444
eigenvector:
-0.082625 0.439515 0.435053 -0.104116 0.254766
0.369735 -0.389526 0.231106 0.438252 0.038263

eigenvalue 8, = 0.630422
eigenvector:
-0.214378 0.392498 0.339477 0.290410 -0.368977
0.253859 0.047471 -0.448548 -0.432022 0.100586

eigenvalue 9, = 0.512567
eigenvector:
-0.325181 0.307018 -0.035041 0.445832 -0.253524
-0.367644 0.333471 0.299339 0.386633 -0.225084

eigenvalue 10, = 0.512565
eigenvector:
-0.404138 0.191490 -0.381471 0.233384 0.366276
-0.253880 -0.439210 0.097502 -0.302859 0.328115

Rotations: 50001
Total time:0.277441 sec

```

Figura 33 Resultados de Jacobi en OpenACC de una matriz de 10 x 10.

#### 4.3.4 Resultado CUDA

```

eigenvalue 1, = 20.431736
eigenvector:
0.100390 -0.435800 0.324920 0.307289 0.443665
0.056224 0.269490 0.356896 -0.153416 0.420076

eigenvalue 2, = 20.431734
eigenvector:
0.230146 -0.383448 -0.057619 0.443486 0.056224
-0.443666 0.130332 -0.427801 0.016098 -0.446924

eigenvalue 3, = 2.425921
eigenvector:
0.337374 -0.293562 -0.392656 0.214060 -0.443666
-0.056224 -0.422706 0.146014 0.122798 0.430023

eigenvalue 4, = 2.425920
eigenvector:
0.411578 -0.174940 -0.403975 -0.191843 -0.056224
0.443665 0.366589 0.256150 -0.249673 -0.371030

eigenvalue 5, = 1.000001
eigenvector:
0.445493 -0.039193 -0.082246 -0.439586 0.443665
0.056224 -0.008246 -0.447137 0.352108 0.275718

eigenvalue 6, = 1.000000
eigenvector:
0.435800 0.100390 0.307290 -0.324920 0.056224
-0.443665 -0.356895 0.269491 -0.420076 -0.153416

eigenvalue 7, = 0.629809
eigenvector:
0.383449 0.230146 0.443487 0.057619 -0.443665
-0.056224 0.427801 0.130332 0.446924 0.016097

eigenvalue 8, = 0.629808
eigenvector:
0.293562 0.337374 0.214060 0.392655 -0.056224
0.443665 -0.146014 -0.422706 -0.430024 0.122798

eigenvalue 9, = 0.512543
eigenvector:
0.174940 0.411578 -0.191844 0.403975 0.443665
0.056224 -0.256151 0.366589 0.371030 -0.249673

eigenvalue 10, = 0.512543
eigenvector:
0.039193 0.445493 -0.439586 0.082246 0.056224
-0.443666 0.447137 -0.008245 -0.275718 0.352108

Rotations: 158
Total time:0.663814 sec

```

Figura 34 Resultados de Jacobi en CUDA de una matriz de 10 x 10.

# Conclusiones

El computo científico necesita un gran poder de procesamiento, sumado a que necesita que se obtenga en un periodo relativamente corto de tiempo. Por eso es necesario encontrar en las nuevas tecnologías formas que ayuden a la solución de problemas que antes era impensable resolver.

Al momento de paralelizar el trabajo empiezan a surgir ciertos errores en el calculo de resultados, ya que puede existir condicion de carrera y los threads chocan entre si, o en el caso de CUDA, al no contar con un dispositivo que este contruido para calculos de cómputo científico, los nucleos de la tarjeta gráfica tienen una presición de punto flotante menor, entonces surgen errores por el truncamiento o redondeo en dichos cálculos.

Por ejemplo en la matriz más pequeña que se probó (3x3):

Eigenvalores			
	Serial	OpenACC	CUDA
1	3.414215	3.414214	3.414215
2	2.000001	2.000001	2.000000
3	0.585787	0.585787	0.585786
nrot	29	29	10

Ahora con una de 5x5:

Eigenvalores			
	Serial	OpenACC	CUDA
1	5.630339	5.630339	5.630342
2	-0.277626	-0.277625	-0.277601
3	-0.391462	-0.391460	-0.390448
4	-0.786500	-0.786441	-0.784438
5	-4.177851	-4.177852	-4.177853
nrot	50001	50001	32

Finalmente una de 10 x 10:

Eigenvalores			
	Serial	OpenACC	CUDA
1	20.431740	20.431728	20.431736
2	20.431759	20.431734	20.431734
3	2.425922	2.425918	2.425921
4	2.425919	2.425920	2.425920
5	1.001439	1.001523	1.000001

<b>6</b>	1.001456	1.001410	1.000000
<b>7</b>	0.630436	0.630444	0.629809
<b>8</b>	0.630389	0.630422	0.629808
<b>9</b>	0.512564	0.512567	0.512543
<b>10</b>	0.512567	0.512565	0.512543
<b>nrot</b>	50001	50001	158

Se puede encontrar que a pocos elementos en las matrices, practicamente el unico resultado que varia es el que se obtiene con CUDA, pero una vez que vamos incrementando el orden de las matrices el resultado de OpenACC tambien va divergiendo de nuestro algoritmo en Serial.

Debemos ver que es lo que nos importa más, que el resultado sea presiso u obtener el resultado más rapido. En esté caso en particular, los digitos que difieren son casi insignificantes, y podemos estar dispuestos a perder esa pequeña presicion.

Pudimos observar que operar las matrices con pocos elementos en paralelo es una perdida de recursos, ya que no varía mucho el tiempo de ejecucion y en el caso de CUDA hasta aumenta considerablemente por el desplazamiento entre el host y el device. Pero ahora el caso contrario, el operar matrices muy grandes, el Runtime varía inmensamente y ahora si es evidente el ver que se compensa la perdida de presision con la de la aceleracion del calculo.

# Tabla de Figuras

Figura 1 Sistema coordinado molecular: $i, j$ =electrones y A, B= núcleos. ....	10
Figura 2 Comparación entre Ecuación de Schrödinger y Ecuación de Khon-Sham. ....	11
Figura 3 Giro de ejes de una matriz simétrica. ....	16
Figura 4 Aceleración de programas en GPUs. ....	18
Figura 5 Comparación entre Multicore y Manycore. ....	19
Figura 6 Matriz de 3 x 3. ....	20
Figura 7 Conversión de matriz a arreglo. ....	21
Figura 8 Salida de Jacobi con matriz de 3 x 3 en C. ....	24
Figura 9 Salida de Jacobi con matriz de 3 x 3 en OpenACC. ....	28
Figura 10 Diagrama de modelo SIMD. ....	28
Figura 11 Arquitectura CUDA. ....	29
Figura 12 Jerarquía LLVM. ....	30
Figura 13 Mapeo de estructuras en CUDA. ....	31
Figura 14 Dimensión de Grids y Blocks en CUDA. ....	31
Figura 15 Pipeline de ejecución de CUDA. ....	32
Figura 16 Partición de datos en CUDA. ....	33
Figura 17 Partición de datos de una matriz en CUDA. ....	33
Figura 18 Transferencia de datos entre Host y Device en CUDA. ....	34
Figura 19 Orden de ID de los threads de CUDA. ....	34
Figura 20 Salida de Jacobi con matriz de 3 x 3 en CUDA. ....	37
Figura 21 Gráfica de RUNTIME. ....	41
Figura 22 Gráfica del inicio de la divergencia del RUNTIME. ....	41
Figura 23 Factor de Costo total. ....	42
Figura 24 Factor de Costo hasta matriz de 32 x 32. ....	42
Figura 25 Speed up total. ....	43
Figura 26 Speed up de las primeras matrices. ....	44
Figura 27 Aceleración total. ....	45
Figura 28 Aceleración de las primeras matrices. ....	45
Figura 29 Eficiencia. ....	46
Figura 30 Eficiencia sin los valores del programa en C. ....	47
Figura 31 Matriz de 10 x 10. ....	47
Figura 32 Resultados de Jacobi en Serial de una matriz de 10 x 10. ....	48
Figura 33 Resultados de Jacobi en OpenACC de una matriz de 10 x 10. ....	49
Figura 34 Resultados de Jacobi en CUDA de una matriz de 10 x 10. ....	50

# Bibliografía

- [1] J. M. Thijssen, Computational Physics, New York: Cambridge University Press, 2007.
- [2] «Institute for Theoretical Physics ETH Zurich,» [En línea]. Available: <http://www.itp.phys.ethz.ch/research.html>.
- [3] [En línea]. Available: <https://chemistry.stanford.edu/research/research-areas/theoretical-chemistry>. [Último acceso: 22 junio 2018].
- [4] G. W. George Hager, Introduction to High Performance Computing for Scientists and Engineers, T. & F. Group, Ed., 2011.
- [5] [En línea]. Available: <https://www.amd.com/es-xl/products/graphics/server/gpu-compute>. [Último acceso: 23 junio 23].
- [6] [En línea]. Available: [http://www.ub.edu/web/ub/es/recerca\\_innovacio/recerca\\_a\\_la\\_UB/instituts/institutspropis/iqtcub.html](http://www.ub.edu/web/ub/es/recerca_innovacio/recerca_a_la_UB/instituts/institutspropis/iqtcub.html). [Último acceso: 22 junio 2018].
- [7] [En línea]. Available: <https://www.ibm.com/thought-leadership/summit-supercomputer/>. [Último acceso: 23 junio 2018].
- [8] [En línea]. Available: <https://chemistry.stanford.edu/research/research-areas/physical-chemistry>. [Último acceso: 22 junio 2018].
- [9] S. I. e. d. Schrödinger. [En línea]. Available: <http://www.ugr.es/~jllopez/Cap3-Sch.pdf>. [Último acceso: 14 junio 2018].
- [1] [En línea]. Available: <https://www.nucleares.unam.mx/~vieyra/node26.html>. [Último acceso: 23 junio 2018].
- [1] P. e. d. K.-S. p. s. f. f. Kohn-Sham. [En línea]. Available: [http://ricabib.cab.cnea.gov.ar/519/1/1Benitez\\_Moreno.pdf](http://ricabib.cab.cnea.gov.ar/519/1/1Benitez_Moreno.pdf). [Último acceso: 14 junio 2018].
- [1] J. K. Labanoswski, Density Functional Methods in Chemistry, New York: Springer-2] Velarg, 1991.
- [1] D. S. SHOLL, DENSITY FUNCTIONAL THEORY. A Practical Introduction., National Energy 3] Technology Laboratory: Georgia Institute of Technology, 2009.
- [1] T. d. f. d. l. densidad electrónica. [En línea]. Available: 4] <http://www.fis.cinvestav.mx/~daniel/thELA.pdf>. [Último acceso: 12 junio 2018].
- [1] E. Urriolabeitia, «La teoría del funcional de la densidad (DFT) va por el mal camino,» 5] *divulgame.org*, 9 enero 2017.
- [1] T. D. F. D. L. DENSIDAD. [En línea]. Available: 6] [http://depa.fquim.unam.mx/amyd/archivero/DensityFunctionalTheory\\_21556.pdf](http://depa.fquim.unam.mx/amyd/archivero/DensityFunctionalTheory_21556.pdf). [Último acceso: 14 junio 2018].
- [1] [En línea]. Available: [http://www.demon-software.com/public\\_html/index.html](http://www.demon-software.com/public_html/index.html). 7] [Último acceso: 23 junio 2018].

- [1] [En línea]. Available: [http://www.demon-8\] software.com/public\\_html/program.html#branches](http://www.demon-8] software.com/public_html/program.html#branches). [Último acceso: 23 junio 2018].
- [1] F. J. Ayres, MATRICES, Teoría y problemas resueltos, McGraw-Hill, 1962.
- 9]
- [2] S. K. Shan, Numerical Methods and Computers, Massachusetts: Addison-Wesley, 1966.
- 0]
- [2] [En línea]. Available: <http://la.nvidia.com/object/what-is-gpu-computing-la.html>.
- 1] [Último acceso: 23 junio 2018].
- [2] [En línea]. Available: <https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/>. [Último acceso: 12 mayo 2018].
- 2] [En línea]. Available: <https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/>. [Último acceso: 23 junio 2018].
- 3]
- [2] W. H. Press, Numerical Recipes in C. The Art of Scientific Computing, United States of America: Cambridge University Press, 2002.
- 4]
- [2] «OpenACC: El API GPGPU Post-CUDA,» 12 2011. [En línea]. Available: <https://www.fayerwayer.com/2011/12/openacc-el-api-gpgpu-post-cuda/>.
- 5]
- [2] openacc.org, OpenACC Programming and Best Practices Guide, 2015.
- 6]
- [2] «Monday-Programming-GPUs-OpenACC,» [En línea]. Available: <http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517B-Monday-Programming-GPUs-OpenACC.pdf>.
- 7]
- [2] [En línea]. Available: <https://es.wikipedia.org/wiki/SIMD>.
- 8]
- [2] J. Cheng, Professional CUDA C Programming, Indianapolis: John Wiley & Sons, 2014.
- 9]
- [3] [En línea]. Available: <https://www.pelagos-consulting.com/?p=503>.
- 0]
- [3] M. A. Barry Wilkinson, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Segunda Edición ed., Pearson, 2005.
- 1]

# CÓDIGO FUENTE

## CF 1.1 creaMatrix.c

```
1.  /*
2.   * creaMatrix.c
3.   *
4.   *
5.   * José Antonio Ayala Barbosa
6.   * Oct, 2018
7.   *
8.   *
9.  */
10.
11.     #include <stdio.h>
12.     #include <string.h>
13.     #include <stdlib.h>
14.
15.     #define NP 16
16.     int main(int argc, char const *argv[]){
17.
18.         float m[NP][NP];
19.         int i,j,k;
20.         char nombreArchivo[100];
21.         FILE *archivo;
22.         char tempChar[100];
23.
24.         for ( i = 0; i < NP; ++i)
25.         {
26.             for (j = i,k=NP/2; j < NP; ++j, k--)
27.             {
28.                 m[i][j]=m[j][i]= k;
29.             }
30.         }
31.
32.         strcpy(nombreArchivo, "matrix");
33.         sprintf(tempChar, "%d", NP);
34.         strcat(nombreArchivo, tempChar);
35.         strcat(nombreArchivo, ".txt");
36.
37.         if (fopen(nombreArchivo, "r") == NULL){ // Si no existe
archivo crea un archivo con los numeros
38.             printf("Creating File\n");
39.             archivo = fopen(nombreArchivo, "w"); // abre archivo
de escritura
40.             fprintf(archivo,"%d",NP);
41.             fprintf(archivo,"%c",'\n');
42.
43.             for (i = 0; i < NP; i++){
44.                 for (j = 0; j < NP; j++){
45.
46.                     fprintf(archivo,"%0.1f ",m[i][j]); //Escribe
numeros aleatorios
47.                 }
```



```

48.         fprintf(archivo,"%c",'\n');
49.     }
50.     fclose(archivo); // Cierra el archivo
51.
52.     printf("File created\n");
53. }
54.
55.     printf("\n");
56.     return 0;
57. }

```

## CF 1.2 auxFuncs.h

```

1.  /*
2.  * auxFuncs.h
3.  * Auxiliary functions for timing and allocating data.
4.  *
5.  * José Antonio Ayala Barbosa
6.  * Oct, 2018
7.  *
8.  *
9.  */
10.
11.     #include <stdio.h>
12.     #include <stddef.h>
13.     #include <stdlib.h>
14.     #include <sys/time.h>
15.
16.     void nrerror(char error_text[]){
17.         /* Numerical Recipes standard error handler */
18.         fprintf(stderr,"Numerical Recipes run-time error...\n");
19.         fprintf(stderr,"%s\n",error_text);
20.         fprintf(stderr,"...now exiting to system...\n");
21.         exit(1);
22.     }
23.
24.     void get_walltime_(double* wcTime) {
25.         struct timeval tp;
26.         gettimeofday(&tp, NULL);
27.         *wcTime = (double) (tp.tv_sec + tp.tv_usec/1000000.0);
28.     }
29.
30.     void get_walltime(double* wcTime) {
31.         get_walltime_(wcTime);
32.     }
33.
34.     float *vectorF(int n){
35.         float *v;
36.         v=(float *) malloc((size_t) (n*sizeof(float)));
37.         if (!v) nrerror("allocation failure in vector()");
38.         return v;
39.     }
40.
41.     int *vectorI(int n){

```

```

42.     int *v;
43.     v=(int *) malloc((size_t)(n*sizeof(int)));
44.     if (!v) nrerror("allocation failure in vector()");
45.     return v;
46. }

```

## CF 1.3 driverA.c

```

1.  /*
2.   * driverA.c
3.   * Driver for function JACOBI.
4.   *
5.   * José Antonio Ayala Barbosa
6.   * Oct, 2018
7.   *
8.   *
9.   */
10.
11.     #include <stdio.h>
12.     #include <sys/time.h>
13.     #include <stdlib.h>
14.     #include "auxFuncs.h"
15.     #include "jacobiA.h"
16.
17.     int main(int argc, char **argv)
18.     {
19.
20.         int n;
21.         char *nombreArchivo=argv[1];
22.         double S,E;
23.         int i, j, k, nrot=0;
24.         FILE *archivo;
25.         float *c,*mat,*eval,*evec;
26.
27.         if (fopen(nombreArchivo, "r") == NULL){
28.             printf("File not found\n");
29.             return 1;
30.         }else{
31.             archivo = fopen(nombreArchivo, "r");
32.             fscanf(archivo, "%d", &n);
33.             c = vectorF(n*n);
34.             for (i = 0; i < n; ++i){
35.                 for (j = 0; j < n; ++j){
36.                     fscanf(archivo, "%f", &c[i*n+j]);
37.                 }
38.             }
39.             fclose(archivo);
40.         }
41.
42.         mat=vectorF(n*n);
43.         evec=vectorF(n*n);
44.         eval=vectorF(n);
45.
46.         for (i = 0; i < n; ++i)

```

```

47.         for (j = 0; j < n; ++j)
48.             mat[i*n+j]=c[i*n+j];
49.
50.     for (i = 0; i < n; i++){
51.         for (j = 0; j < n; j++)
52.             evec[i*n+j] = 0.0;
53.         evec[i*n+i] = 1.0;
54.     }
55.
56.     for (i = 0; i < n; ++i){
57.         printf("\n");
58.         for (j = 0; j < n; ++j)
59.             printf("[%f]",mat[i*n+j]);
60.     }
61.
62.     printf("\n***** Finding Eigenvalues *****\n");
63.     get_walltime(&S);
64.
65.     jacobiMultip(mat,n,evec,eval,&nrot);
66.
67.     get_walltime(&E);
68.
69.     printf("\n***** Eigenvalues *****\n");
70.     for (i = 0; i < n; ++i){
71.         printf("\n");
72.         for (j = 0; j < n; ++j)
73.             printf("[%f]",mat[i*n+j]);
74.     }
75.     printf("\n");
76.     for (i = 0; i < n; ++i){
77.         printf("eigenvalue %3d, = %12.6f\n",i+1,eval[i]);
78.         printf("eigenvector:\n");
79.         for (j=0,k=1;j<n; j++,k++) {
80.             printf("%12.6f",evec[i*n+j]);
81.             if ((k%5) == 0) printf("\n");
82.         }
83.         printf("\n");
84.     }
85.
86.     printf("Rotations: %d\n",nrot );
87.     printf("Total time:%f sec\n", E-S);
88.
89.     free(c);
90.     free(mat);
91.     free(evec);
92.     free(eval);
93.
94.
95.     return 0;
96. }

```

## CF 1.4 jacobiA.h

```
1. #include <stdio.h>
```

```

2. #include <math.h>
3. #include <stdlib.h>
4. #include <stdbool.h>
5. #include <math.h>
6. #include "auxFuncs.h"
7.
8. void max_elem(int *piv_elem, int n, float *mat){
9.     int r, c;
10.    int max_i = 0; //first
    coordinate i
11.    int max_j = 1; //first
    coordinate j
12.
13.    #pragma acc parallel num_gangs(32), vector_length(64)
14.    {
15.        #pragma acc loop
16.        for (r = 0; r < n-1; r++)
17.            for (c = r+1; c < n; c++)
18.                if(fabs(mat[r*n+c]) > fabs(mat[max_i*n+max_j]))
19.                    max_i = r; //replace
    new coor
20.                    max_j = c;
21.            }
22.    }
23.    piv_elem[0] = max_i; //store new
    coordinates
24.    piv_elem[1] = max_j;
25.
26. }
27.
28. float cal_tan(int max_i, int max_j, float *mat, int n){
29.     float num;
30.     float den;
31.     float a1;
32.     float a2;
33.     float a3;
34.
35.     num = 2 * (mat[max_i*n+max_j]);
36.     if(mat[max_i*n+max_i] < mat[max_i*n+max_j])
37.         num = -num;
38.
39.     a1 = mat[max_i*n+max_i] - mat[max_j*n+max_j];
40.     a2 = a1*a1;
41.     a3 = 4 * mat[max_i*n+max_j]*mat[max_i*n+max_j];
42.     den = a2 + a3;
43.     den = sqrt(den);
44.     den = abs(a1) + den;
45.     return num/den;
46. }
47.
48. float cal_cos(float tang){ //cos = 1/√(1+tan^2)
49.     float cose;
50.     cose = 1 + (tang * tang);
51.     cose = sqrt(cose);
52.     cose = 1 / cose;
53.     return cose;

```

```

54.     }
55.
56.     float cal_sin(float cose, float tang){           //sin = cos*tan
57.         float sino;
58.         sino = cose*tang;
59.         return sino;
60.     }
61.
62.     void new_T_mat(int max_i, int max_j,int n,float *mat,float *T
, float *mat_temp){
63.         float tang, cose, sino;
64.         int c,r;
65.
66.         tang = cal_tan(max_i,max_j,mat,n);
67.         cose = cal_cos(tang);
68.         sino = cal_sin(cose,tang);
69.
70.         for (r = 0; r < n; r++){                     //Generate identity
matrix
71.             for (c = 0; c < n; c++)
72.                 T[r*n+c] = 0.0;
73.                 T[r*n+r] = 1.0;
74.         }
75.                                     //T Rotating matrix
76.         T[max_i*n+max_i] = cose;
77.         T[max_j*n+max_j] = cose;
78.         T[max_i*n+max_j] = -sino;                     //Element to
eliminate
79.         T[max_j*n+max_i] = sino;
80.
81.     }
82.
83.     void mat_mult(int n,float *A, float *B,float *C){
84.         int i,j,k;
85.         #pragma acc parallel num_gangs(32), vector_length(64)
86.         {
87.             #pragma acc loop
88.             for (i = 0 ; i < n ; i++){
89.                 for (j = 0 ; j < n ; j++){
90.                     C[i*n+j] = 0.0;
91.                     #pragma acc loop
92.                     for (k = 0 ; k < n ; k++){
93.                         C[i*n+j] += A[i*n+k] * B[k*n+j];
94.                     }
95.                 }
96.             }
97.         }
98.     }
99.
100.    void mat_mult_tra(int n,float *A, float *B,float *C){
101.        int i,j,k;
102.        #pragma acc parallel num_gangs(32), vector_length(64)
103.        {
104.            #pragma acc loop
105.            for (i = 0 ; i < n ; i++){
106.                for (j = 0 ; j < n ; j++){
107.                    C[i*n+j] = 0.0;

```

```

108.             #pragma acc loop
109.             for (k = 0 ; k < n ; k++ ){
110.                 C[i*n+j] += A[k*n+i] * B[k*n+j];
111.             }
112.         }
113.     }
114. }
115. }
116.
117. void copy_mat(int n,float *A, float *B){
118.     int i,j;
119.     #pragma acc parallel num_gangs(32), vector_length(64)
120.     {
121.         #pragma acc loop
122.         for (i = 0; i < n; ++i)
123.             for (j = 0; j < n; ++j)
124.                 B[i*n+j]=A[i*n+j];
125.     }
126. }
127.
128.
129. void jacobiMultip (float *mat, int n,float *eigvec, float *ei
    gval, int *nrot){
130.
131.     /*****+*****/
132.     /*
133.         //On input
134.         //mat: Contains the matrix to be diagonalized.
135.         //n: Order of matrix a.
136.
137.         //On output
138.         //eigvec: eigenvectors to be computed v
139.         //eigval: Contains the eigenvalues in ascending order
140.         //nrot: Number of Jacobi rotations.
141.     */
142.
143.     int i,j;
144.     int *piv_elem;           //Keep coordinates of an elemnt
145.
146.     bool min = false;
147.     float EPS = .0000001;
148.     float *T;                //Contains the ratation matrix
149.     float *mat_temp;         //A temporal matrix
150.
151.     mat_temp = vectorF(n*n);
152.     T = vectorF(n*n);
153.     piv_elem = vectorI(2);
154.
155.     for (*nrot = 0; min == false ; ++*nrot){
156.         max_elem(piv_elem,n,mat); //Search for max element in
157.         tringle up
158.         if(fabs(mat[piv_elem[0]*n+piv_elem[1]]) < EPS || *nrot
159.             >= 50000 ) //if max element doesnt exist more
160.             min=true;
161.     }

```

```

159.         else{
160.             new_T_mat(piv_elem[0],piv_elem[1],n,mat,T,mat_temp);
161.             //Calculate T
162.             mat_mult(n,eigvec,T,mat_temp); //Eigen
163.             vect
164.             copy_mat(n,mat_temp,eigvec);
165.             mat_mult_tra(n,T,mat,mat_temp);
166.             mat_mult(n,mat_temp,T,mat);
167.         }
168.         for (i = 0; i < n; ++i)
169.             eigval[i]=mat[i*n+i];
170.     }

```

## CF 1.5 driverA.cu

```

1.  /*
2.   * driverA.cu
3.   * Driver for function JACOBI in CUDA.
4.   *
5.   * José Antonio Ayala Barbosa
6.   * Oct, 2018
7.   *
8.   *
9.   */
10.
11.     #include <stdio.h>
12.     #include <sys/time.h>
13.     #include <stdlib.h>
14.     #include "auxFuncs.h"
15.     #include "jacobiACUDA.h"
16.
17.     int main(int argc, char **argv)
18.     {
19.
20.         int n;
21.         char *nombreArchivo=argv[1];
22.         double S,E;
23.         int i, j, k, nrot=0;
24.         FILE *archivo;
25.         float *c,*mat,*eval,*evec;
26.
27.         if (fopen(nombreArchivo, "r") == NULL){
28.             printf("File not found\n");
29.             return 1;
30.         }else{
31.             archivo = fopen(nombreArchivo, "r");
32.             fscanf(archivo, "%d", &n);
33.             c = vectorF(n*n);
34.             for (i = 0; i < n; ++i){
35.                 for (j = 0; j < n; ++j){
36.                     fscanf(archivo, "%f", &c[i*n+j]);
37.                 }

```

```

38.         }
39.         fclose(archivo);
40.     }
41.
42.     mat=vectorF(n*n);
43.     evec=vectorF(n*n);
44.     eval=vectorF(n);
45.
46.     for (i = 0; i < n; ++i)
47.         for (j = 0; j < n; ++j)
48.             mat[i*n+j]=c[i*n+j];
49.
50.     for (i = 0; i < n; i++){
51.         for (j = 0; j < n; j++)
52.             evec[i*n+j] = 0.0;
53.         evec[i*n+i] = 1.0;
54.     }
55.
56.     for (i = 0; i < n; ++i){
57.         printf("\\n");
58.         for (j = 0; j < n; ++j)
59.             printf("[%f]",mat[i*n+j]);
60.     }
61.
62.     printf("\\n***** Finding Eigenvalues *****\\n");
63.     get_walltime(&S);
64.
65.     jacobiMultip(mat,n,evec,eval,&nrot);
66.
67.     get_walltime(&E);
68.
69.     printf("\\n***** Eigenvalues *****\\n");
70.     for (i = 0; i < n; ++i){
71.         printf("\\n");
72.         for (j = 0; j < n; ++j)
73.             printf("[%f]",mat[i*n+j]);
74.     }
75.     printf("\\n");
76.     for (i = 0; i < n; ++i){
77.         printf("eigenvalue %3d, = %12.6f\\n",i+1,eval[i]);
78.         printf("eigenvector:\\n");
79.         for (j=0,k=1;j<n; j++,k++) {
80.             printf("%12.6f",evec[i*n+j]);
81.             if ((k%5) == 0) printf("\\n");
82.         }
83.         printf("\\n");
84.     }
85.
86.     printf("Rotations: %d\\n",nrot );
87.
88.     printf("Total time:%f sec\\n", E-S);
89.     free(c);
90.     free(mat);
91.     free(evec);
92.     free(eval);
93.
94.     return 0;

```



```
95.     }
```

## CF 1.6 jacobiACUDA.h

```
1.  /*
2.  * jacobiACUDA.h
3.  *
4.  *
5.  * José Antonio Ayala Barbosa
6.  * Oct, 2018
7.  *
8.  *
9.  */
10.
11.     #include <cuda.h>
12.     #include <stdio.h>
13.     #include <math.h>
14.     #include <stdlib.h>
15.     #include <stdbool.h>
16.     #include <math.h>
17.     #include "auxFuncs.h"
18.
19.     __global__ void kernel_max_elem(int *piv_elem,int n,float *mat)
20.     {
21.         int r,c;
22.         int max_i = 0;           //first coordinate i
23.         int max_j = 1;           //first coordinate j
24.         for (r = 0; r < n-1; r++)
25.             for (c = r+1; c < n; c++)
26.                 if(fabs(mat[r*n+c]) > fabs(mat[max_i*n+max_j])){
27.                     //if exists a higher element
28.                     max_i = r;           //replace new coor
29.                     max_j = c;
30.                 }
31.                 piv_elem[0] = max_i;     //store new coordenates
32.                 piv_elem[1] = max_j;
33.     }
34.
35.     __global__ void kernel_set_max_elem(float *mat,int n, int *piv_elem,float *max_elem){
36.         *max_elem=mat[piv_elem[0]*n+piv_elem[1]];
37.     }
38.
39.     __device__ float cal_tan(int max_i,int max_j,float *mat, int n){
40.         float num;
41.         float den;
42.         float a1;
43.         float a2;
44.         float a3;
45.
46.         num = 2 * (mat[max_i*n+max_j]);
```

```

47.         if(mat[max_i*n+max_i] < mat[max_i*n+max_i])
48.             num = -num;
49.
50.         a1 = mat[max_i*n+max_i] - mat[max_j*n+max_j];
51.         a2 = a1*a1;
52.         a3 = 4 * mat[max_i*n+max_j]*mat[max_i*n+max_j];
53.         den = a2 + a3;
54.         den = sqrt(den);
55.         den = abs(a1) + den;
56.         return num/den;
57.     }
58.
59.     __device__ float cal_cos(float tang){        //cos = 1/√(1+tan^2)
60.         float cose;
61.         cose = 1 + (tang * tang);
62.         cose = sqrt(cose);
63.         cose = 1 / cose;
64.         return cose;
65.     }
66.
67.     __device__ float cal_sin(float cose, float tang){        //sin
= cos*tan
68.         float sino;
69.         sino = cose*tang;
70.         return sino;
71.     }
72.
73.     __global__ void kernel_new_T_mat(int *piv_elem,int n,float *m
at,float *T){
74.         float tang, cose, sino;
75.         int c,r;
76.         int max_i = piv_elem[0];
77.         int max_j = piv_elem[1];
78.
79.         int row = blockIdx.y * blockDim.y + threadIdx.y;
80.         int col = blockIdx.x * blockDim.x + threadIdx.x;
81.
82.         if(row >= n || col >= n)
83.             return;
84.
85.         for (row = row; row < n; ++row)
86.             T[row*n+col] = 0.0;
87.         T[col*n+col]=1.0;
88.
89.         if(max_j==col){
90.             tang = cal_tan(max_i,max_j,mat,n);
91.             cose = cal_cos(tang);
92.             sino = cal_sin(cose,tang);
93.
94.             T[max_i*n+max_i] = cose;
95.             T[max_j*n+max_j] = cose;
96.             T[max_i*n+max_j] = -sino;        //Element to eliminate
97.             T[max_j*n+max_i] = sino;
98.         }
99.     }
100.

```

```

101.  __global__ void kernel_mat_mult(int n, float *A, float *B, float
    *C) {
102.
103.     float Cvalue;
104.     int row = blockIdx.y * blockDim.y + threadIdx.y;
105.     int col = blockIdx.x * blockDim.x + threadIdx.x;
106.     int e;
107.
108.     if(row >= n || col >= n)
109.         return;
110.
111.     for (row = row; row < n; ++row) {
112.         Cvalue = 0.0;
113.         for (e = 0; e < n; e++)
114.             Cvalue += (A[row * n + e]) * (B[e * n + col]);
115.         C[row * n + col] = Cvalue;
116.     }
117. }
118.
119.  __global__ void kernel_mat_mult_tra(int n, float *A, float *B,
    float *C) {
120.
121.     float Cvalue;
122.     int row = blockIdx.y * blockDim.y + threadIdx.y;
123.     int col = blockIdx.x * blockDim.x + threadIdx.x;
124.     int e, i;
125.
126.     if(row >= n || col >= n)
127.         return;
128.
129.     for (i = row; i < n; i++) {
130.         Cvalue = 0.0;
131.         for (e = 0; e < n; e++)
132.             Cvalue += (A[e * n + i]) * (B[e * n + col]);
133.         C[i * n + col] = Cvalue;
134.     }
135. }
136.
137.  __global__ void kernel_copy_mat(int n, float *A, float *B) {
138.
139.     int row = blockIdx.y * blockDim.y + threadIdx.y;
140.     int col = blockIdx.x * blockDim.x + threadIdx.x;
141.     int e;
142.
143.     if(row >= n || col >= n)
144.         return;
145.
146.     for (e = 0; e < n; e++)
147.         B[col*n+e]=A[col*n+e];
148. }
149.
150.  __global__ void kernel_copy_diag(int n, float *A, float *B) {
151.
152.     int row = blockIdx.y * blockDim.y + threadIdx.y;
153.     int col = blockIdx.x * blockDim.x + threadIdx.x;
154.
155.     if(row >= n || col >= n)

```

```

156.         return;
157.
158.         B[col]=A[col*n+col];
159.     }
160.
161.     void jacobiMultip (float *mat, int n,float *eigvec, float *ei
gval, int *nrot){
162.
163.         /*****
164.         //On input
165.         //mat: Contains the matrix to be diagonalized.
166.         //n: Order of matrix a.
167.
168.         //On output
169.         //eigvec: eigenvectors to be computed v
170.         //eigval: Contains the eigenvalues in ascending order
171.         //nrot: Number of Jacobi rotations.
172.         *****/
173.         int i,j;
174.         bool min = false;
175.         float EPS = .0000001;
176.         float *max_elem;
177.         float max_e;
178.
179.         float *d_mat;
180.         float *d_eigvec;
181.         float *d_eigval;
182.         float *d_T;
183.         float *d_mat_temp;
184.         int *d_piv_elem;
185.         float *d_max_elem;
186.
187.         size_t size = (n+1) * (n+1) * sizeof(float);
188.         dim3 dimGrid( 32 );           // 32 x 1 x 1
189.         dim3 dimBlock( 64 );          // 64 x 1 x 1
190.
191.         cudaMalloc(&d_mat, size);
192.         cudaMalloc(&d_eigvec, size);
193.         cudaMalloc(&d_eigval, n*sizeof (int));
194.         cudaMalloc(&d_T, size);
195.         cudaMalloc(&d_mat_temp, size);
196.         cudaMalloc(&d_piv_elem, 2*sizeof (int));
197.         cudaMalloc(&d_max_elem,sizeof (float));
198.
199.         cudaMemcpy(d_mat, mat, size, cudaMemcpyHostToDevice);
200.         cudaMemcpy(d_eigvec, eigvec, size, cudaMemcpyHostToDevice);
201.         cudaMemcpy(d_eigval, eigval, n*sizeof (int), cudaMemcpyHos
tToDevice);
202.
203.         for (*nrot = 0; min == false ; ++*nrot){
204.
205.             kernel_max_elem<<<1,1>>>(d_piv_elem,n,d_mat);    //Search
for max element in upper tringle
206.             kernel_set_max_elem<<<1,1>>>(d_mat,n,d_piv_elem,d_max_e
lem);
207.             cudaMemcpy(&max_e, d_max_elem, sizeof(float), cudaMemcpy
yDeviceToHost);

```

```

208.
209.             if(fabs(max_e) < EPS || *nrot >= 50000 ) //if max
element doesnt exist more
210.                 min=true;
211.
212.             else{
213.                 kernel_new_T_mat<<<dimGrid, dimBlock>>>(d_piv_elem,n
,d_mat,d_T);
214.                 kernel_mat_mult<<<dimGrid, dimBlock>>>(n,d_eigvec,d_
T,d_mat_temp);
215.                 kernel_copy_mat<<<dimGrid, dimBlock>>>(n,d_mat_temp,
d_eigvec);
216.                 kernel_mat_mult_tra<<<dimGrid, dimBlock>>>(n,d_T,d_m
at,d_mat_temp);
217.                 kernel_mat_mult<<<dimGrid, dimBlock>>>(n,d_mat_temp,
d_T,d_mat);
218.             }
219.         }
220.
221.         kernel_copy_diag<<<dimGrid, dimBlock>>>(n,d_mat,d_eigval);
222.
223.         cudaMemcpy(mat, d_mat, size, cudaMemcpyDeviceToHost);
224.         cudaMemcpy(eigvec, d_eigvec, size, cudaMemcpyDeviceToHost);
225.         cudaMemcpy(eigval,d_eigval,n*sizeof(float),cudaMemcpyDevic
eToHost);
226.
227.         cudaFree(d_mat);
228.         cudaFree(d_eigvec);
229.         cudaFree(d_eigval);
230.         cudaFree(d_T);
231.         cudaFree(d_mat_temp);
232.         cudaFree(d_piv_elem);
233.         cudaFree(d_max_elem);
234.     }

```