UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA
DE LA COMPUTACIÓN

DIAGONALIZACIÓN DE LA MATRIZ DE KHON-SHAM CON TARJETAS GRÁFICAS

TESINA

QUE PARA OPTAR POR EL GRADO DE:

ESPECIALISTA EN CÓMPUTO DE ALTO RENDIMIENTO

PRESENTA:
JOSÉ ANTONIO AYALA BARBOSA

TUTOR
DR. JOSÉ JESÚS CARLOS QUINTANAR SIERRA

Ciudad de México a 13 de diciembre de 2018

.

A mi abuela María, a mi madre, mi abuela Adela y abuelo Alfredo, a mi padre y mi familia, ya que este trabajo es la culminación de años de su apoyo, por ello es mi deseo que se sientan partícipes de este éxito en mí vida, siendo definitivamente, el triunfo es de todos.

Agradecimientos

A mi abuela María que siempre me apoyó en todo sentido, por su amor, comprensión y respeto recibido durante toda mi vida, inspirándome siempre a seguir adelante y superarme. Porque siempre han dado todo por mí sin esperar nada a cambio.

A mi padre, porque siempre he sentido su apoyo y cariño, por las enseñanzas recibidas. Por los sacrificios que ha realizado para llegar a donde estamos… como familia.

Al Ing. Heriberto, por las enseñanzas compartidas, la ayuda brindada y el impulso dado para realizar esta especialidad.

Al Dr. Carlos Quintanar, por la paciencia que tuvo, y las enseñanzas que me compartío.

A la UNAM, por todas la bondades y ayuda que me ha otrogado en toda mi formación profesional.

Índice

Índice 7

Introducción 9

Capítulo 1. Matriz de Khon-Sham 11
1.1 Ecuación de Schrödinger 11
1.2 El Método de Hartree Fock (HF) y la Teoría del Funcional de la Densidad (DFT) 13
1.2.1 Ecuación de Kohn-Sham 13
1.3 Modelado molecular en GPU 15
1.3.1 TeraChem 15
1.3.2 deMon2k 15

Capítulo 2. Algoritmo de Jacobi 17
2.1 Forma cuadrática 17
2.2 Eigensistemas 17
2.2.1 Transformación de coordenadas 18
2.2.2 Eigenvalores 19
2.2.3 Eigenvector 20
2.3 Algoritmo de Jacobi 21

Capítulo 3. Implementación en C, OpenACC y CUDA 22
3.1 Cómputo en GPGPU 22
3.1.1 Diferencia entre CPU y GPU 22
3.2 Caracteristicas del equipo 23
3.3 Implementación en C 24
3.3.1 Creación de matrices 24
3.3.2 Funciones auxiliares 24
3.3.3 Driver 25
3.3.4 Jacobi 26
3.3.5 Compilación 27
3.4 Implementación en OpenACC 28
3.4.1 Estándar OpenACC 28
3.4.2 JacobiACC 30
3.4.3 Compilación 31
3.5 Implementación en CUDA 32
3.5.1 Modelo de programación en CUDA 32
3.5.2 Driver 39

3.5.3 JacobiCUDA 39
3.5.4 Compilación 40
Capítulo 4. Pruebas y resultados 42
4.1 Métricas de desempeño 42
4.1.2 Runtime 42
4.1.2 Cost Factor 42
4.1.3 Speedup Factor 42
4.1.3 Aceleración 43
2.1.4 Efficiency 43
4.2 Resultados 44
4.2.1 Resultados de Runtime 44
4.2.2 Resultados de Cost Factor 46
4.2.3 Resultados de Speed Up 48
4.2.3 Resultados de Aceleración 49
4.2.4 Resultados de Efficiency 51
4.3 Resultados de una matriz 53
4.3.1 Matriz a evaluar 53
4.3.2 Resultado Serial 54
4.3.3 Resultado OpenACC 55
4.3.4 Resultado CUDA 56
Conclusiones 57
Tabla de Figuras 59
Bibliografía 60
CÓDIGO FUENTE 62
CF 1.1 creaMatrix.c 62
CF 1.2 auxFuncs.h 63
CF 1.3 driverA.c 64
CF 1.4 jacobiA.h 65
CF 1.5 driverA.cu 69
CF 1.6 jacobiACUDA.h 71

Introducción

Física y quimica computacional tratan de resolver problemas implementando métodos del análisis numérico para resolver con cierta 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 area de la fisica teorica. Por ejemplo en el Institute for Theoretical Physics ETH Zurich [2] es una de las areas de investigacion, 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 Kuta, Monte-Carlo. 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 Khon-Sham. En la ecuación de Schrödinger su solución es conocida como la función de onda, mientras que la solución de la ecuación de Khon-Sham es el valor absoluto al cuadrado de la función de onda y se interpreta en la física como la densidad electrónica.

Durante las ultimas décadas ha incrementado el interés en la computación paralela, el principal objetivo de ella es ahorro de la energia y evitar el sobrecalientamiento 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 grandes y se les divide en unos mucho más pequeños que pueden ser resueltos de manera paralela y concurrente [4].

Actualmente el paralelismo, por razones de ahorro de energias y por desempeño se ha convertido en la corriente que todo el mundo de la programación esta siguiendo. Existen dos principales tipos de paralelismo: 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 distribirlos 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 NVIDIA 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 éste se encargará de verificar si existen dependencias de datos 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 las 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 En el caso de Summit se apostó por equiparla con nodos de alto rendimiento y tarjetas gráficas, lo cual abrió el camino a utilizar está 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 en sistemas computacionales relacionados con el modelado de átomos en moléculas [8]. Debido a ésto, en este trabajo 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.

El objetivo del presente trabajo es el de programar el método númerico de Jacobi para resolver la matriz de Khon-Sham, y paralelizar la implementacion en tarjetas gráficas utilizando OpenAcc y CUDA. Posteriormente se comparará el desempeño de los tres programas con algunas métricas de cómputo de alto rendimiento.

Capítulo 1. Matriz de Khon-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 un volumen con una cierta posición física gracias a la ecuación de Schrödinger [9]. Pero para poder ser implementada se requiere de mucha capacidad de cómputo, y al ser calculada con métodos numéricos, la precisión depende de cuál método se elija. Por esta razón los resultados que se obtienen tendrán asociado un error.

La ecuación de Schrödinger para predecir el comportamiento de átomos y móleculas considera varios aspectos, los mas importantes son:

- La existencia de un núcleo atómico, en donde se concentra la mayoría de la masa del átomo.
- Los niveles energéticos donde se distribuyen los electrones según el principio de Aufbau.
- La interaccion electrón electrón.
- La interacción nucleo electrón.
- La dualidad onda-partícula.

Aunque con la mecánica cuántica queda claro que no se puede saber con certeza dónde se encuentra un electrón, la ecuacion de Shrödinger nos da la probabilidad de encontrarlo en un elemento de volumen permitido por el principio de incertidumbre de Heisenberg. Cada valor absoluto al cuadrado de la solución de la ecuación de Schrödinger $\Psi(\mathbf{r})$, nos dá la probabilidad de encontrar el electrón en una elemento de volumen centrado en **r**. El valor absoluto al cuadrado de la función de onda, $|\Psi(\mathbf{r})|^2$, 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], esta dado por:

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

$\Psi = función\ de\ onda$
$m = masa\ del\ electrón$
$\hbar = constante\ de\ Planck$
$E = energía\ total\ del\ electrón$
$V = energía\ potencial\ del\ electrón$

La primera aproximación que se hace al estudiar sistemas con más de un átomo es la aproximación de Born-Oppenheimer [11], 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 fungen como una nube de carga [14], y por otro lado para los electrones, los núcleos parecen estar estáticos.

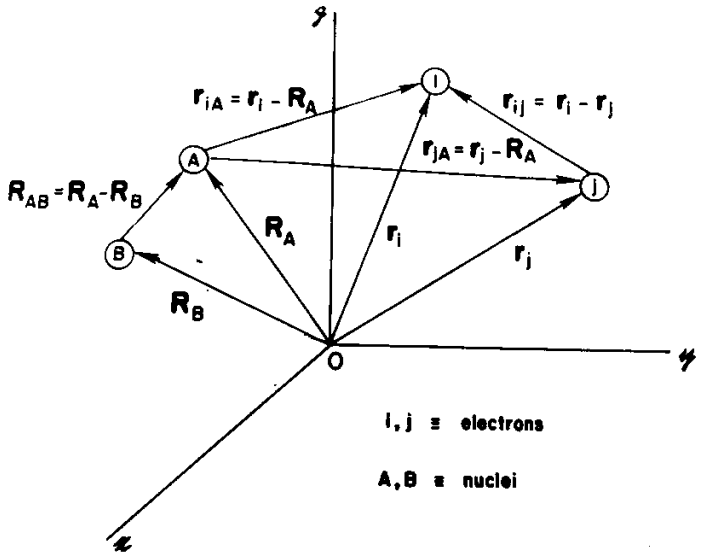

Figura 1 Sistema coordenado molecular: i,j=electrones y A, B= núcleos.

En la Figura 1, podemos observar que los vectores RA y RB son los vectores de posición de los núcleos A y B respectivamente, mientras que los vectores Ri y Rj son los vectores de posición de los electrones i y j.

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

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

$E^{CE} = energía\ cinética\ de\ los\ electrones\ (Atractivo)$
$E^{CN} = energía\ cinética\ de\ los\ núcleos\ (Atractivo)$
$E^{PEN} = energía\ potencial\ electrón - núcleo\ (Atractivo)$
$E^{PEE} = energía\ potencial\ electrón - electrón\ (Repulsivo)$
$E^{PNN} = energía\ potencial\ núcleo - núcleo\ (Repulsivo)$

La energía del sistema está dada por la suma de la energía cinética de los electrones más la energía cinética de los núcleos más la energía potencial electrón-núcleo más la energía potencial electrón-electrón más la energía potencial núcleo-núcleo, como se ve en la ecuación de la energía de un sistema poliatómico, también conocida como Hamiltoniano

El Hamiltoniano asociado se puede expresar como :

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

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 asi podemos cambiar el origen del eje y volverlo 0 para simplificar la ecuación [13].

Resultando el Hamiltoniano como:

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

1.2 El Método de Hartree Fock (HF) y la Teoría del Funcional de la Densidad (DFT)

El método de Hartree-Fock es una de las primeras implementaciones computacionales para resolver la ecuación de Schrödinger, en este método la solución se conoce como la función de onda ψ. La teoría del funcional de la densidad es un procedimiento alternativo a la solución de la ecuación de Schrödinger, la cual permite una descripción muy exacta de los sistemas con muchas partículas, en donde la ecuación a resolver es la ecuación de Khon-Sham. La DFT simplifica los cálculos reemplazando la ecuación de Hartree Fock por la ecuación de Kohn-Sham [11], la solución de esta ecuación está dada por el valor absoluto al cuadrado de la función de onda, la cual se interpreta como la densidad electrónica $|\psi|^2 = \rho$, dejando de lado el uso de la función de onda en la determinación del movimiento de los electrones y los átomos en las moléculas. Una forma de obtener la solución tanto para las ecuación de Hartee Fock como para la ecuación de Khon Sham es aplicar el método variacional. Este 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. Este método consiste en proponer una función tentativa que depende de varios parámetros los cuales se varían hasta que se obtenga una energía mínima.

La Teoria 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].

1.2.1 Ecuación de Kohn-Sham

La ventaja del método DFT es que la densidad electrónica es una magnitud mucho más simple que la función de onda, ésta simplifica las ecuaciones y baja el costo computacional

[15]. Por ello hasta el momento es el procedimiento preferido para abordar problemas que involucran muchos átomos.

La teoría del funcional de la densidad se desarrollo para resolver las ecuaciones de Khon-Sham. En ambos métodos se diagonalizan matrices Hermitianas.

Comparando las ecuaciones que aparecen en el método de Hartee Fock (HF) con con las que aparecen en el método 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)
$E = \int \Psi^* [\Sigma_i h_i + \Sigma_{i>j} 1/r_{ij}] \Psi \, d\tau$	$E = T[\rho] + U[\rho] + E_{xc}[\rho]$	(2ab)
$\Psi = \lvert\psi_1(1), \psi_2(2), \ldots, \psi_n(n)\rvert$	$\rho(\mathbf{r}) = \Sigma_{occ} \lvert\psi_i(\mathbf{r})\rvert^2$	(3ab)
$\partial E / \partial \Psi = 0$	$\partial E / \partial \rho = 0$	(4ab)
$[-1/2\nabla^2 + V_C(\mathbf{r}) + \mu^i{}_x(\mathbf{r})] \psi_i = \varepsilon_i \psi_i$	$[-1/2\nabla^2 + V_C(\mathbf{r}) + \mu_{xc}(\mathbf{r})] \psi_i = \varepsilon_i \psi_i$	(5ab)

Figura 2 Comparación entre Ecuaciónes de Hartee Fock y las Ecuaciónes de Khon-Sham.

Donde en el método de HF la energía es dependiente de la función de onda y de las posiciones de los electrones (1a), está dada por la sumatoria de los hamiltonianos de electrón independiente, más la sumatoria de las interacciones electrón-electrón (2a). La función de onda está dada por el determinante de Slater (3a). Se aplica principio variacional derivando la energía con respecto a la función de onda e igualándola a cero (4a). Finalmente tenemos la ecuación canonica de Hartree Fock para el electrón independiente (5a), la suma de la energía cinética más el potencial de coulomb más el potencial de intercambio y correlación [12].

Por el lado del DFT, la energía está en función de la densidad electrónica y de la posición de los electrones(1b). Y la energía se calcula como la suma de la energía cinética [16], más la energía potencial, mas la energía E_{xc} (de intercambio y correlación) todas ellas en función de la densidad electrónica (2b). Es importante mencionar que la densidad está dada en función de la posición, para obtenerla debe realizarse la sumatoria de la densidad electrónica $|\psi|^2$,de los niveles ocupados (3b). Nuevamente se aplica principio variacional, pero esta vez derivando la energía con respecto a la densidad ρ e igualándola a cero (4b). Finalmente, en la ecuación (5b) se presenta la ecuación de Khon-Sham para el electrón independiente, la cual es la suma de la energía cinética, más la energía coulombiana, más la energía de intercambio y correlación.

Puede observarse que ambas ecuaciones dan como resultado la energía total del sistema. Para resolver la ecuación, 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. Es mayormente utilizado para realizar cálculos y simulaciones en el campo de la fisicoquímica cuántica. Existen ya algunos sistemas que implementan esta tecnología.

1.3.1 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 y no 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.

1.3.2 deMon2k

Es un paquete de software para realizar cálculos con la Teoría del Funcional Densidad, el cual utiliza las combinaciones de orbitales de tipo Gaussiano 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 [1].

[1] Alain St-Amant, Ph.D. Thesis, Université de Montréal, 1992.

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.
- Código paralelo (MPI).
- Interfaces para software de visualización (Molden, Molekel, Vu).
- Portabilidad a varias plataformas informáticas y sistemas operativos.

El sistema esta pensado para correr en sistemas Unix y Linux, la compilación se realiza bajo FORTRAN90. El programa también esta 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.

Capítulo 2. Algoritmo de Jacobi

2.1 Forma cuadrática

Todo polinomio de segundo grado homogéneo que cumple la forma:

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

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

Puede expresarse de una manera matricial de la forma:

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

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.

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

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

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 *nxn* con parámetros escalares λ, llamados también valores característicos o eigenvalores, y x es una matriz columna de variables independientes también llamado *eigenvector*.

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

Cualquier múltiplo de un eigenvector x podrá considerarse como eigenvector, excepto para este caso el 0. El problema es encontrar λ y su correspondiente eigenvector.

2.2.1 Transformación de coordenadas

Para diagonalizar una matriz, se empieza a rotarla para ir eliminando (volviendo 0) los elementos fuera de la diagonal principal obteniendo unicamente la diagonal principal con elementos diferentes de cero.

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

$$x_1 = \overline{x_1} cos\theta - \overline{x_2} sen\theta$$
$$x_2 = \overline{x_1} sen\theta + \overline{x_2} cos\theta$$

o en la forma matricial:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} cos\theta & -sen\theta \\ sen\theta & cos\theta \end{bmatrix} \begin{bmatrix} \overline{x_1} \\ \overline{x_2} \end{bmatrix}$$

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

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

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

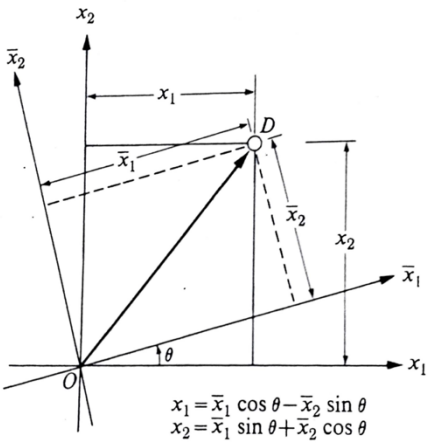

Figura 3 Giro de ejes de una matriz simétrica.

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:

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

por ende deducimos que:

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

Con estovemos que muultiplicando los eigenvalores por los eigenvectores obtenemos la matriz original.

2.2.2 Eigenvalores

Los eigenvalores o valores caracteristicos de una matriz son aquellos que se encuentran en la diagonal principal cuando todos los valores dell los triangulos superior e inferior son iguales a cero.
Para ello es necesario seleccionar un ángulo de rotación que permita realizarlo.

$$B = \begin{bmatrix} cos\theta & sen\theta \\ -sen\theta & cos\theta \end{bmatrix}\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\begin{bmatrix} cos\theta & -sen\theta \\ sen\theta & cos\theta \end{bmatrix}$$

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

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

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

por identidades trigonométricas podemos concluir:

$$tan2\theta = \frac{2a_{12}}{a_{11} - a_{22}}$$

Dando como resultado esperado:

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

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 λ. 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 $[T]$.

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

2.3 Algoritmo de Jacobi

El método de Jacobi, es un método de diagonalización de matrices simétricas. La idea principal del método es realizar rotaciones a una matriz cuyo objetivo es el anular el elemento mayor que se encuentran fuera de la diagonal principal. Y así encontrar los valores caracteristicos de la matriz [21].

Cada vez que la matriz se rota, todos los elementos que aparecen están en funcion a la cantidad que de los elementos eliminados por la función trigonométrica, por lo que el valor absoluto de los elementos fuera de la diagonal principal se va reduciendo hasta ser practicamente cero.

A cada rotación se van generando se van generando eigenvectores, donde los elementos de la diagonal principal corresponden a los eigenvalores.

Los métodos iterativos van acercandose a la solución real a medida que van realizando los ciclos, de manera que la calidad de la aproximación depende de la cantidad de iteraciones que se efectúa [22], aunque dependiendo de la mamanera de implementación del algoritmo, pueden generarse redondeos, truncamientos, y por ende el resutlado es una aproximación.

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 GPGPU

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

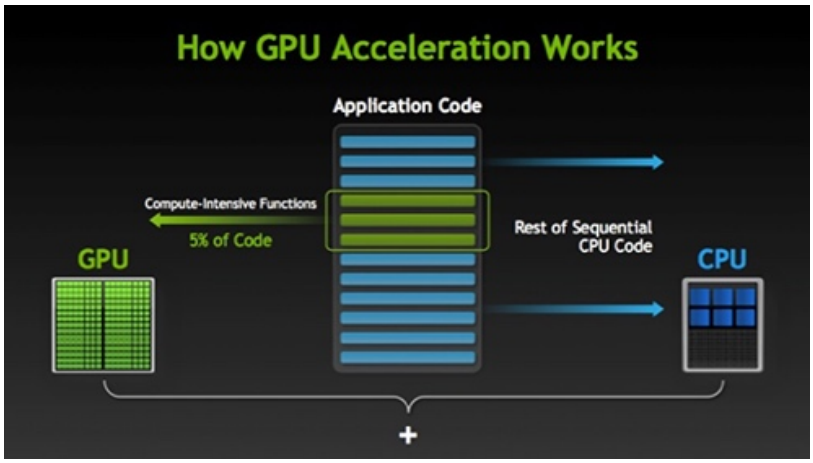

Figura 4 Aceleración de programas en GPUs.

3.1.1 Diferencia entre CPU y GPU

El CPU es un procesador de propósito general, lo que significa que puede hacer una variedad de cálculos, pero esta diseñado para realizar el procesamiento en serie, consta de pocos núcleos de proposito general. 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 desde cientos hasta 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.

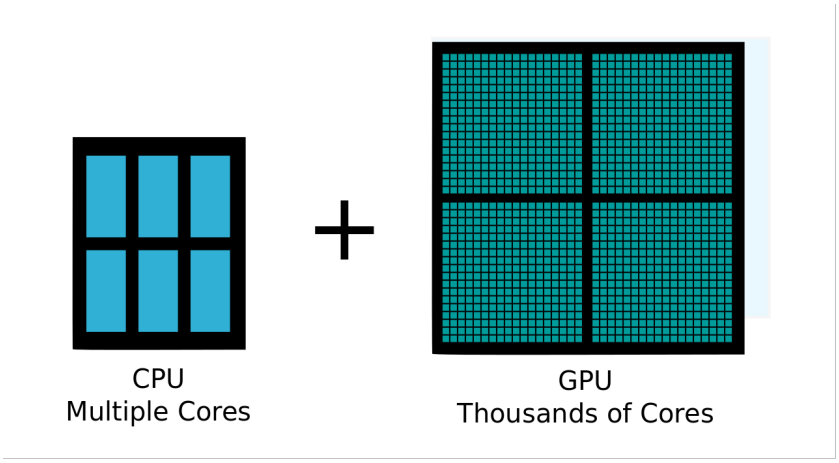

Figura 5 Comparación entre Multicore y Manycore.

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 la tasa de transferencia (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 Caracteristicas del equipo

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

Sistema Operativo	CentOS Linux release 7.3.1611
Arquitectura	64 bits
Disco 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 nucleos CUDA (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:

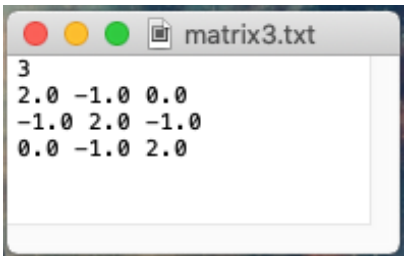

Figura 6 Matriz de 3 x 3.

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. Está 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.

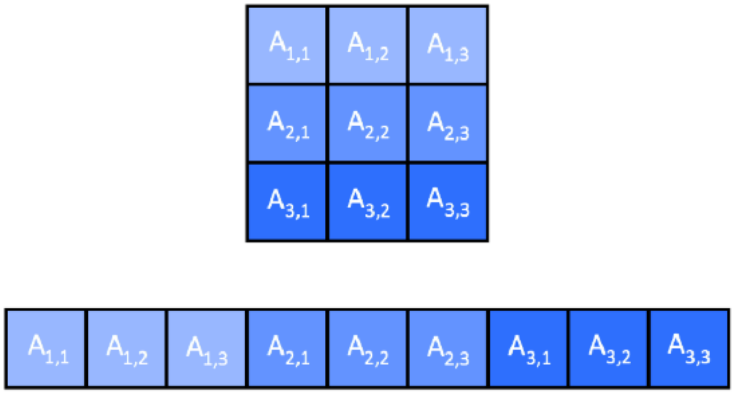

Figura 7 Conversión de matriz a arreglo.

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. Para siplificar los pasos, es recomendable iniciar *i, j =0.*

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 especifico. Si existe el archivo, se lee la matriz y se almacena en la variable 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:

jacobiMultip(mat,n,evec,eval,nrot);	mat -> La matriz de operación. n -> El orden de la matriz. evec -> El eigenvector eval -> El arreglo de eigvec nrot -> Variable inicializada en 0 que da el número de rotaciones que realizo el algoritmo

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 umbral (threshole) recomendado por Numerical Recipies de $1x10^{-7}$ [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 ciclos for para recorrer las matrices y almacenar el producto en la tercera 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 al 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 realiza 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:

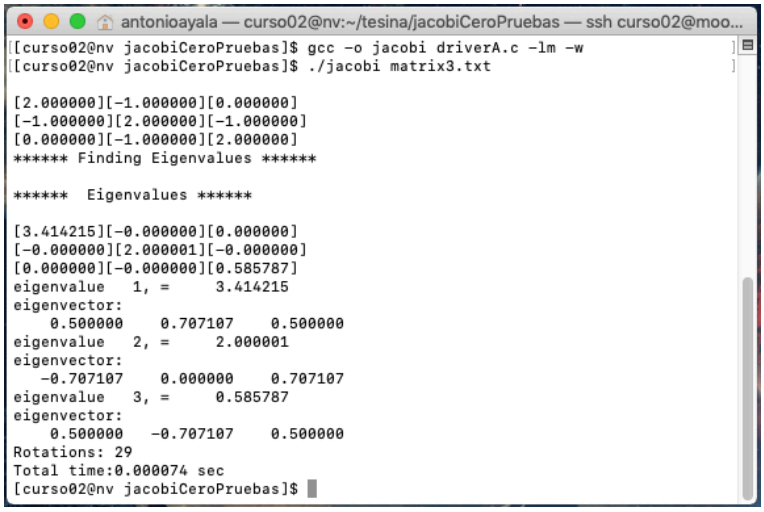

Figura 8 Salida de Jacobi con matriz de 3 x 3 en C.

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 éste se encargaría 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 OpenACC

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 vectors al compilador, debe usarse *#pragma acc parallel num_gangs(#), vector_length(#),* y por la version de nuestra tarjeta gráfica usaremos 32 gangs de 64 workers para poder operar matrices de máximo 2048 renglones. Una vez dados los parametros, 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 más 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, por lo que no se realizará ningún tipo de paralelización.

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 fucniones *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 funcion 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 compilacion 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

Un ejempo con la misma matriz de 3x3 que se utilizo en la implementación en serial:

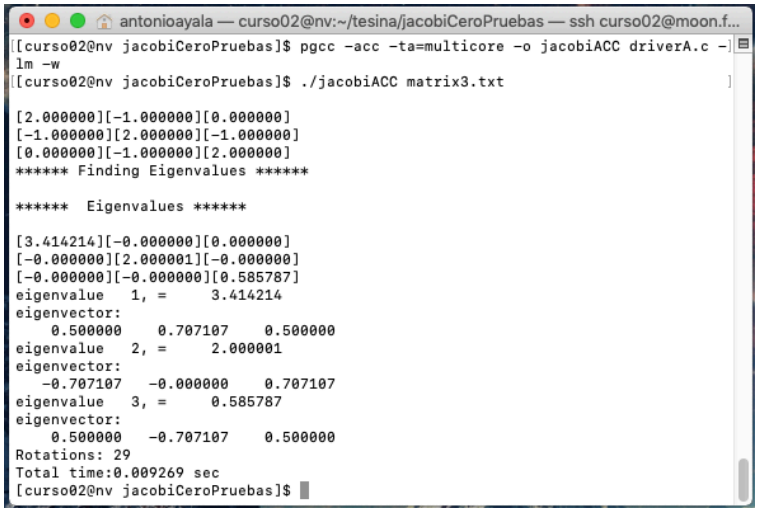

Figura 9 Salida de Jacobi con matriz de 3 x 3 en OpenACC.

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.

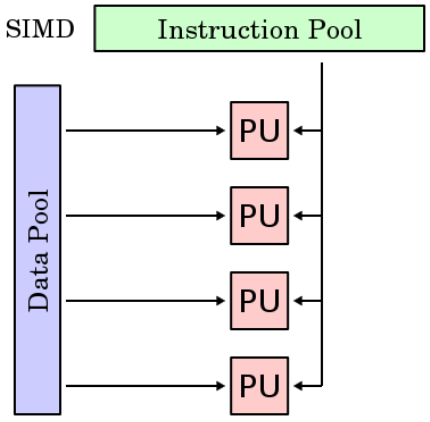

Figura 10 Diagrama de modelo SIMD.

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 esté nivel se rompe en operaciones más básicas del API driver.

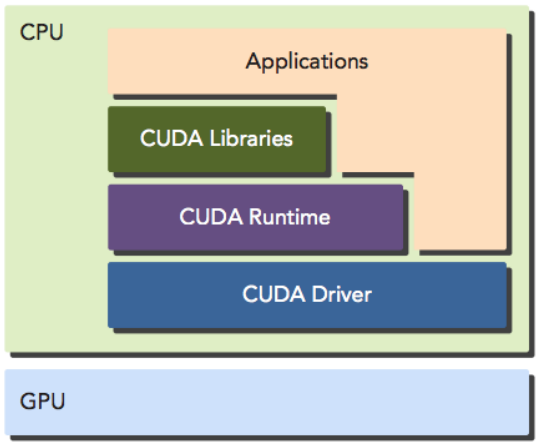

Figura 11 Arquitectura CUDA.

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 esta basado en LLVM[2], por lo que el lenguaje puede ser extendido creando nuevos soportes para el GPU, eso si, utilizando el CUDA Compiler SDK que nos comparte NVIDIA [29].

[2] 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.

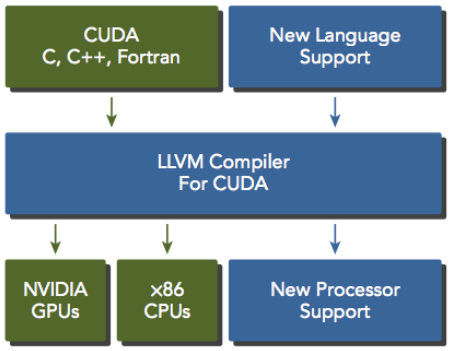

Figura 12 Jerarquía LLVM.

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 contador de programa, registros, memoria privada, entradas y salidas.
bloque	Un bloque es una agrupación de threads que utilizan memoria compartida.
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 lo bloques entre sí. Todos aquellos que se encuentren dentro de un bloque comparten el acceso a una memoria más 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

Es necesario organizar los threads de tal forma que realicen un procesamiento ordenado y no generen una condición de competencia por un recurso. Por lo anterior, es necesario diseñar la forma en la cual accederán a los datos.

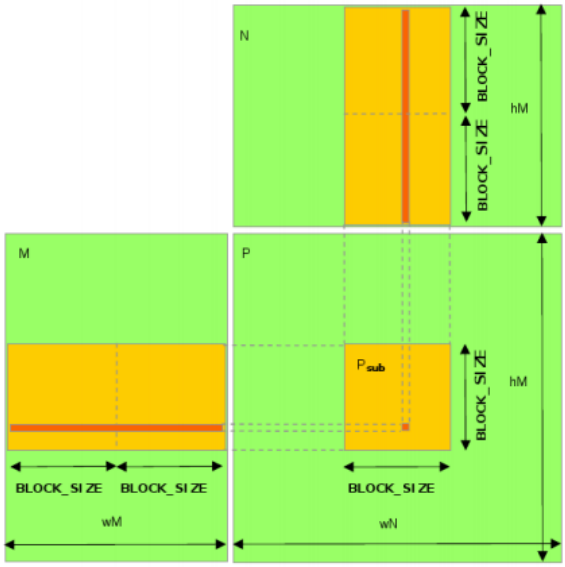

Figura 13 Mapeo de estructuras en CUDA.

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.

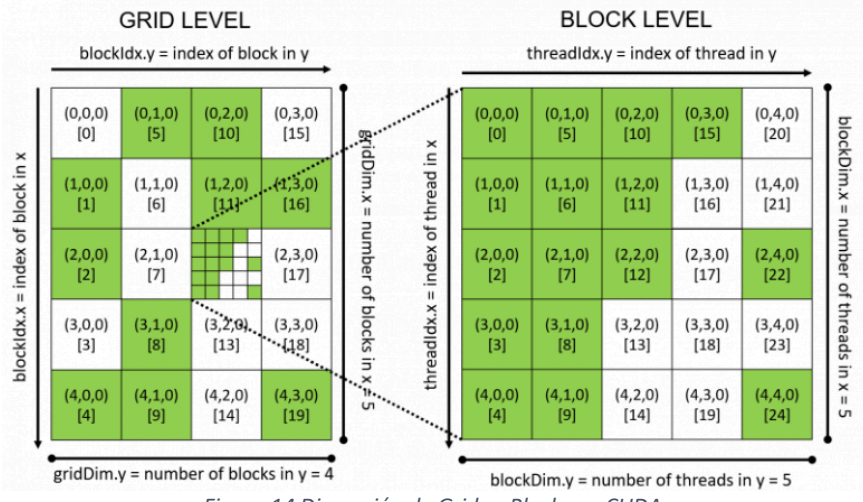

Figura 14 Dimensión de Grids y Blocks en CUDA.

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 asincrónico, 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 patrón:

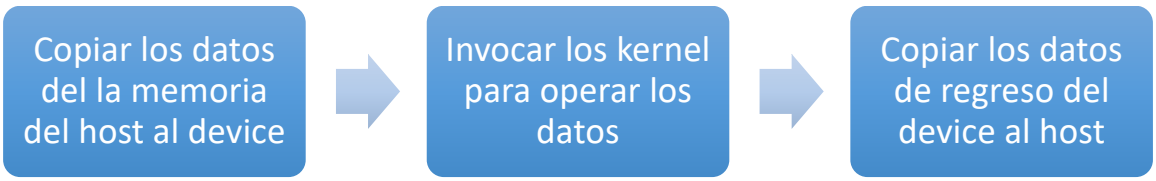

Figura 15 Pipeline de ejecución de CUDA.

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], resaultando 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.

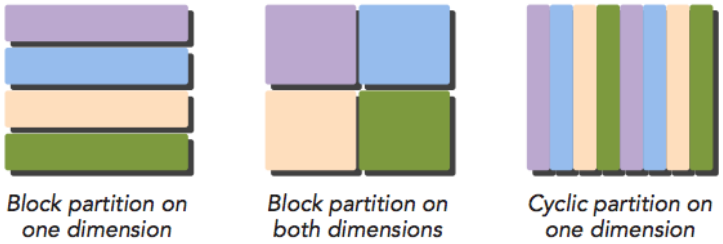

Figura 17 Partición de datos de una matriz en CUDA.

El modelo de programación CUDA supone un sistema compuesto por un host y un device con memoria propia, por lo que la información puede separada. 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**
malloc	*cudaMalloc*
memcpy	*cudaMemcpy*
memset	*cudaMemset*
free	*cudaFree*

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 realizo con éxito [29].

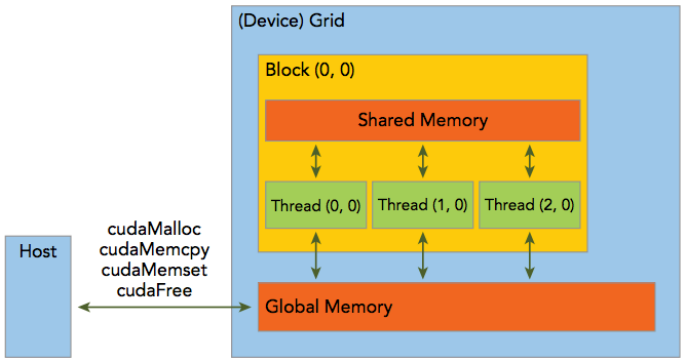

Figura 18 Transferencia de datos entre Host y Device en CUDA.

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 si, 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 desdoblar 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 numero total de columnas.

3.5.1.7 Id de Threads

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

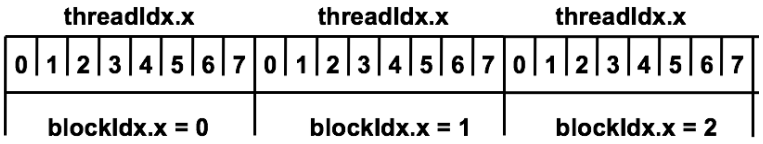

Figura 19 Orden de ID de los threads de CUDA.

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 unicamente 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 threshold 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 númerico, 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 unicamente 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 primer matriz A y uno 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 operacion es trivial, ya que a cada thread se le asigna una columna de la matiz 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 codigo 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 indice 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);

Aqui, al igual que en el calculo del eigenvector, se multiplica renglon por columna de lo obtenido en la funcion 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 ejempo con la misma matriz de 3x3:

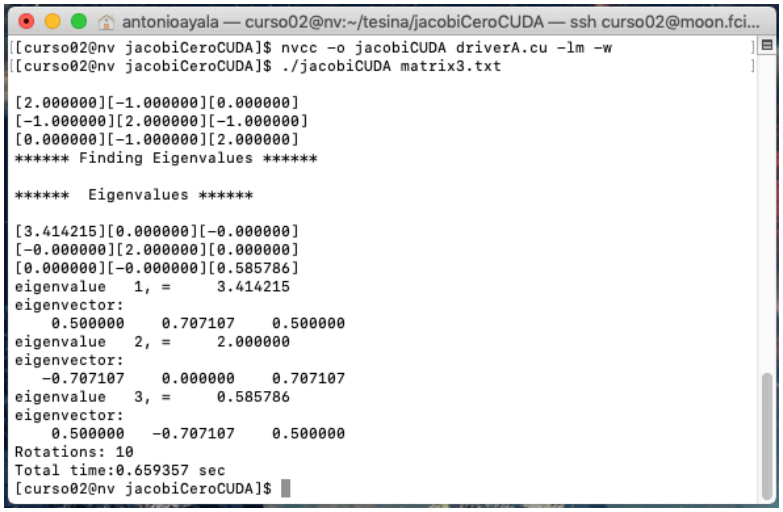

Figura 20 Salida de Jacobi con matriz de 3 x 3 en CUDA.

Capítulo 4. Pruebas y resultados

4.1 Métricas de desempeño

Existe una 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 o 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 diversas subtareas.

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.

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

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

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.

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

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

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.

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

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

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.

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

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

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

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

4.2 Resultados

4.2.1 Resultados de Runtime

RUNTIME [s]			
Orden	**Serial**	**OpenACC**	**CUDA**
3	0.000074	0.009269	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	10417.17143	5429.11005	516.680462
512	76337.37146	46845.537	2495.80583
1024	381686.8573	93691.0741	12754.6959

RUNTIME [min]			
Orden	Serial	OpenACC	CUDA
3	0.000001	0.0002	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	173.6195	90.4852	8.6113
512	1272.2895	780.7590	41.5968
1024	6361.4476	1561.5179	212.5783

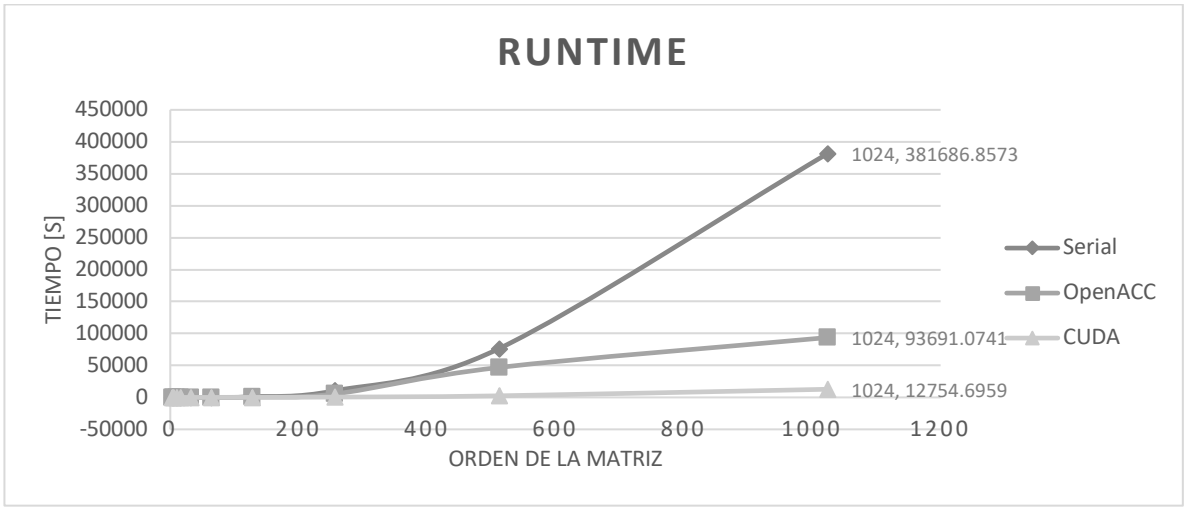

Figura 21Gráfica de RUNTIME.

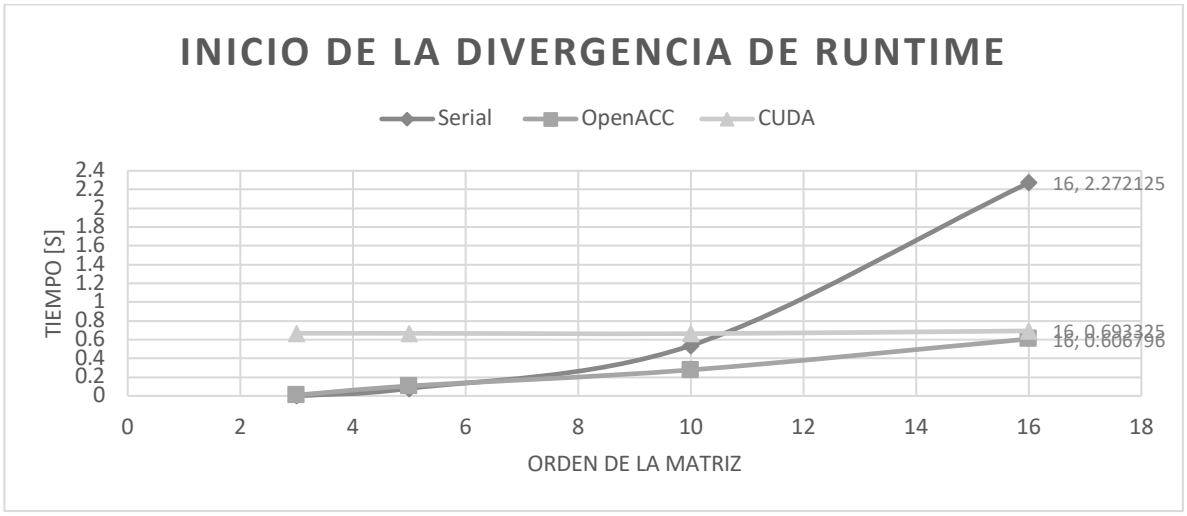

Figura 22 Gráfica del inicio de la divergencia del RUNTIME.

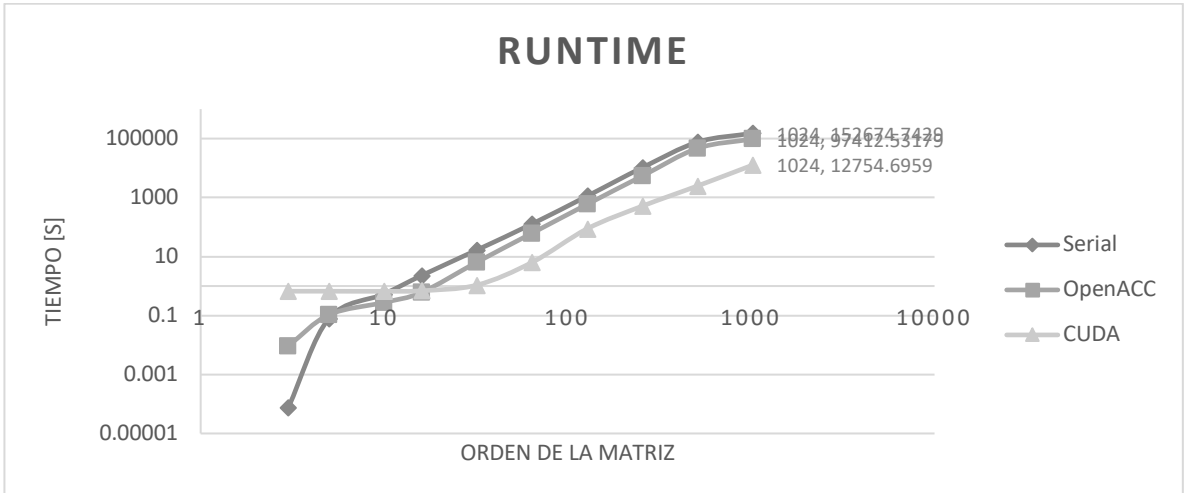

Figura 23 RUNTIME en escala logarítmica

4.2.2 Resultados de Cost Factor

COST FACTOR						
Orden	**Procesadores**	**Serial**	**Procesadores**	**OpenACC**	**Procesadores**	**CUDA**
3	1	0.000074	32	0.296608	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	10417.1714	32	173731.522	256	132270.198
512	1	76337.3715	32	1499057.19	512	1277852.59
1024	1	381686.857	32	2998114.37	1024	13060808.6

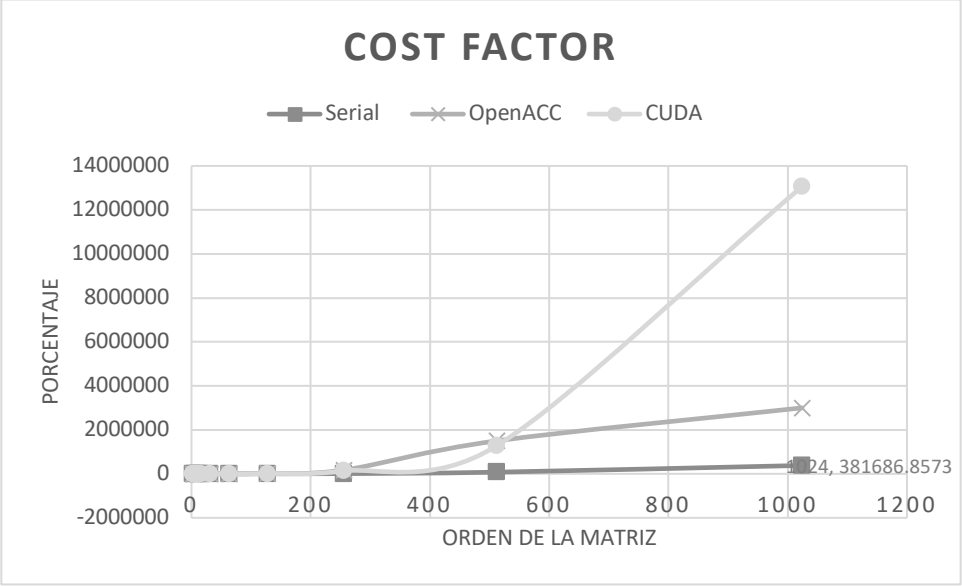

Figura 24 Factor de Costo total.

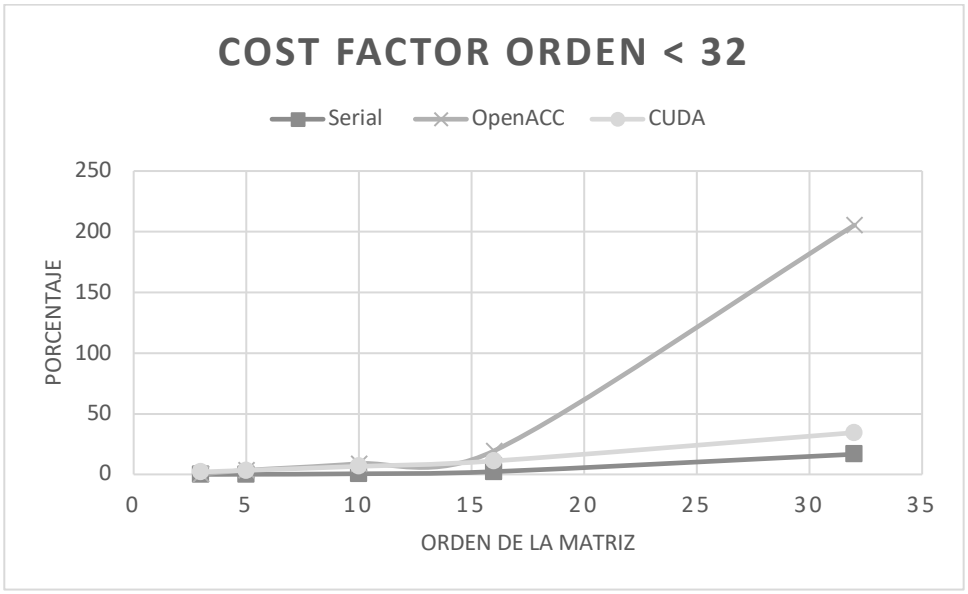

Figura 25 Factor de Costo hasta matriz de 32 x 32.

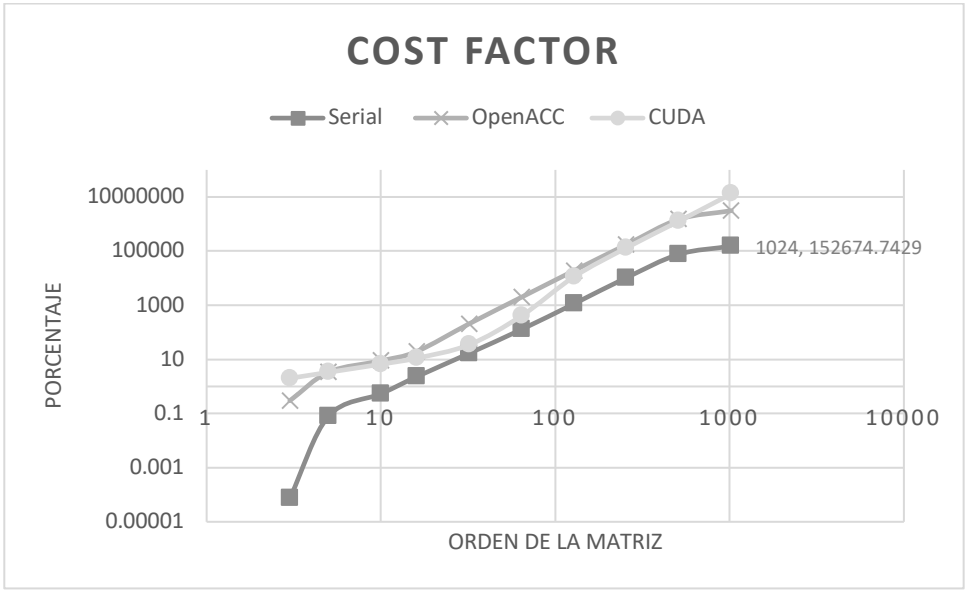

Figura 26 Factor de Costo total en escala logarítmica

4.2.3 Resultados de Speed Up

Speed UP			
Orden	1	0.0079836	0.00011083
3	1	0.75183013	0.1192532
5	1	1.93628195	0.80926886
10	1	3.74446272	3.27714275
16	1	2.58926512	15.4540592
32	1	2.05762781	20.1136025
64	1	1.90283963	12.8412029
128	1	1.91876225	20.161729
256	1	1.62955484	30.5862622
512	1	1.56730084	11.970081
1024	1	0.0079836	0.00011083

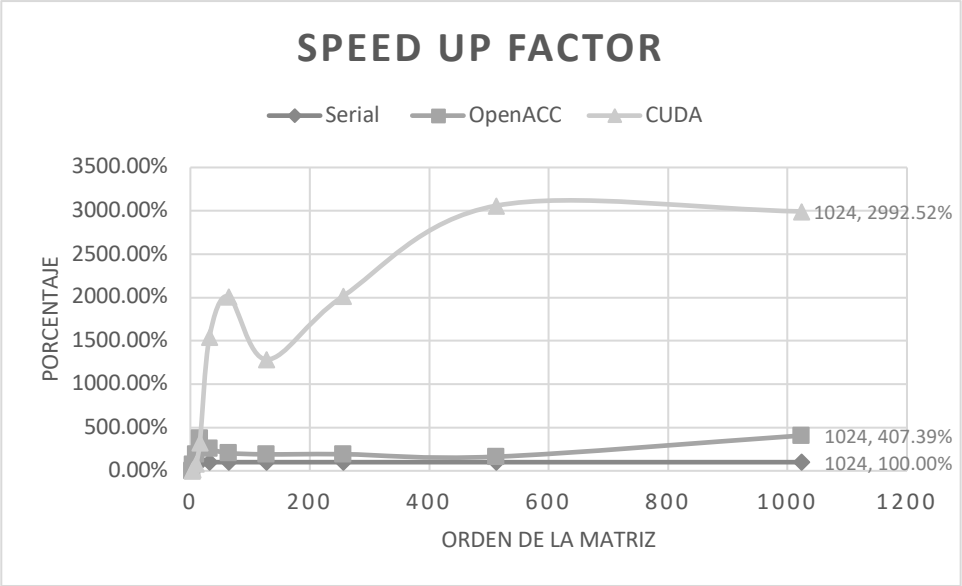

Figura 27 Speed up total.

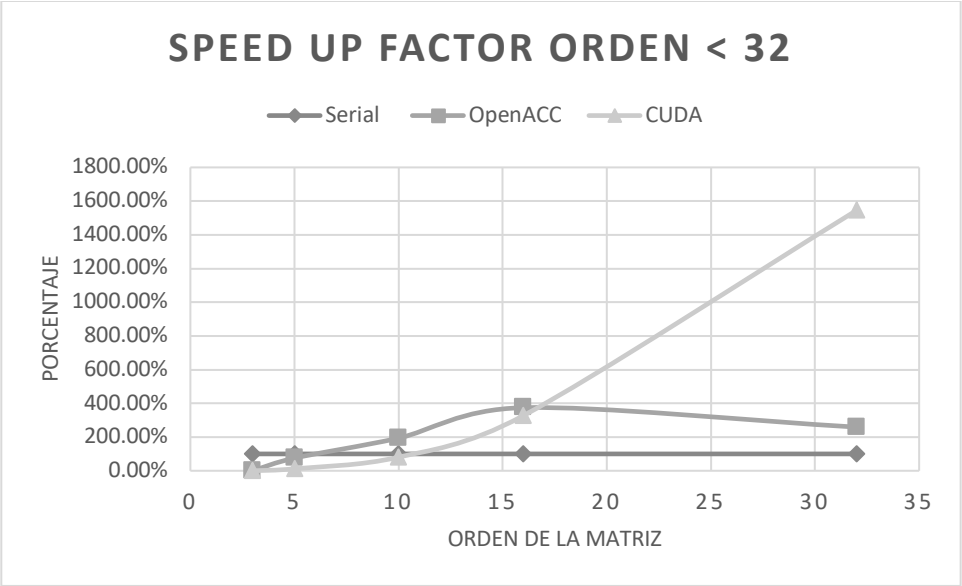

Figura 28 Speed up de las primeras matrices.

4.2.3 Resultados de Aceleración

Aceleración			
Orden	Serial	OpenACC	CUDA
3	0.00%	-99.20%	-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%	91.88%	1916.17%
512	0.00%	62.96%	2958.63%
1024	0.00%	307.39%	2892.52%

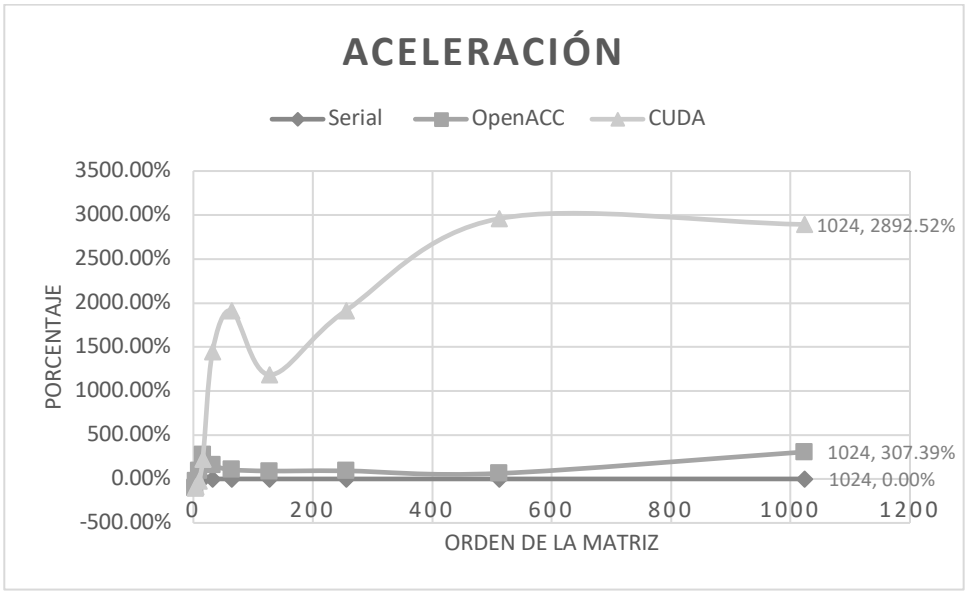

Figura 29 Aceleración total.

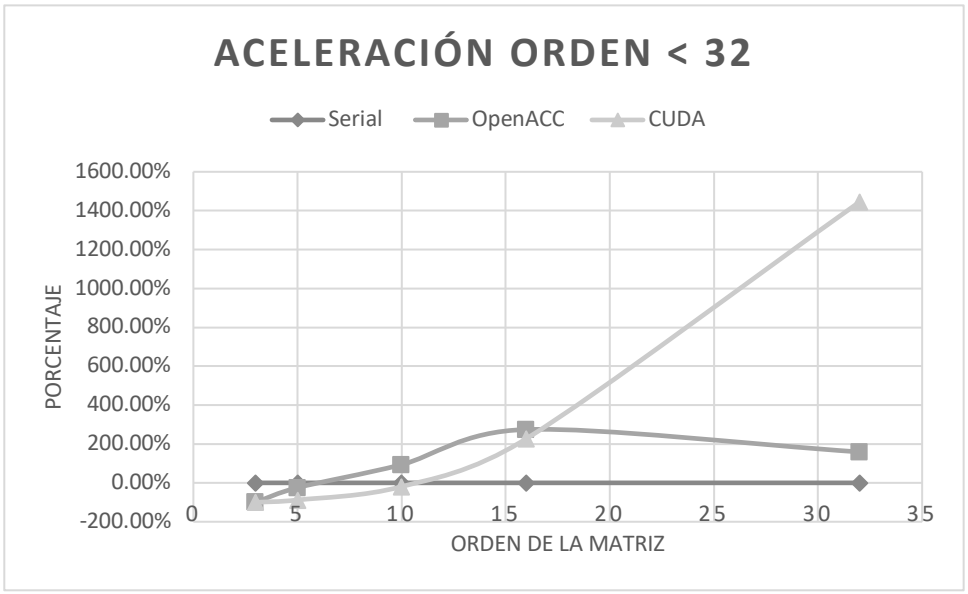

Figura 30 Aceleración de las primeras matrices.

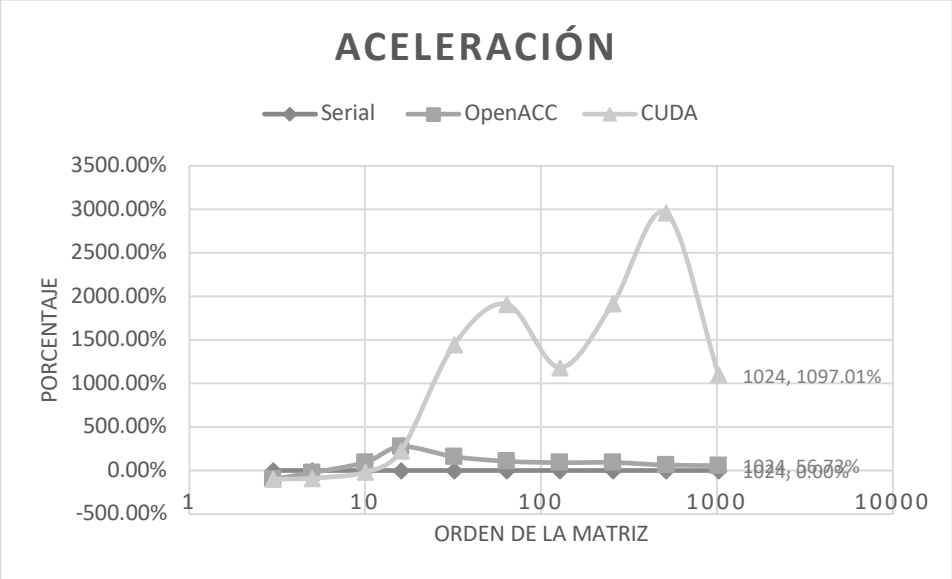

Figura 31 Aceleración de las primeras matrices con escala logarítmica.

4.2.4 Resultados de Efficiency

Efficiency			
Orden	Serial	OpenACC	CUDA
3	100.00%	0.02%	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%	6.00%	7.88%
512	100.00%	5.09%	5.97%
1024	100.00%	12.73%	2.92%

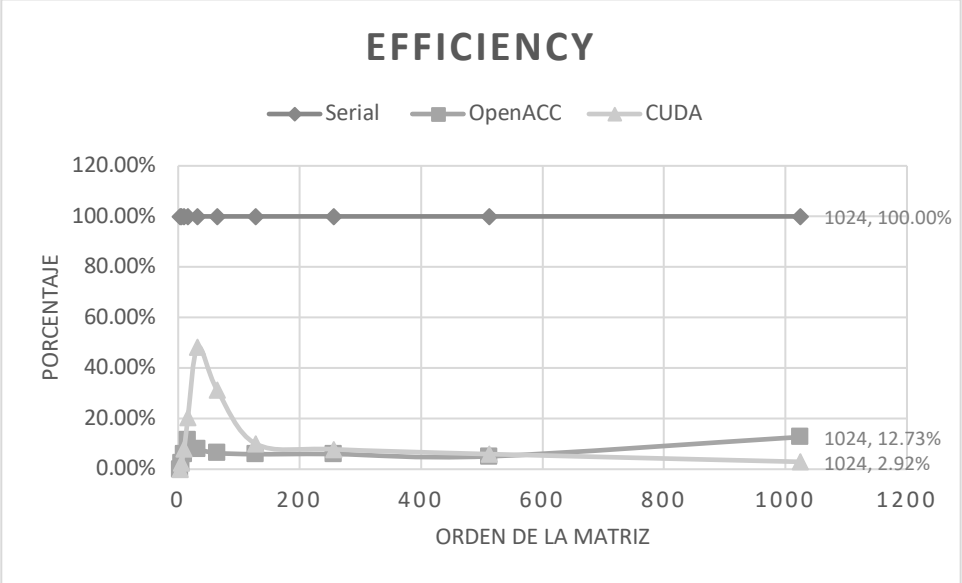

Figura 32 Eficiencia.

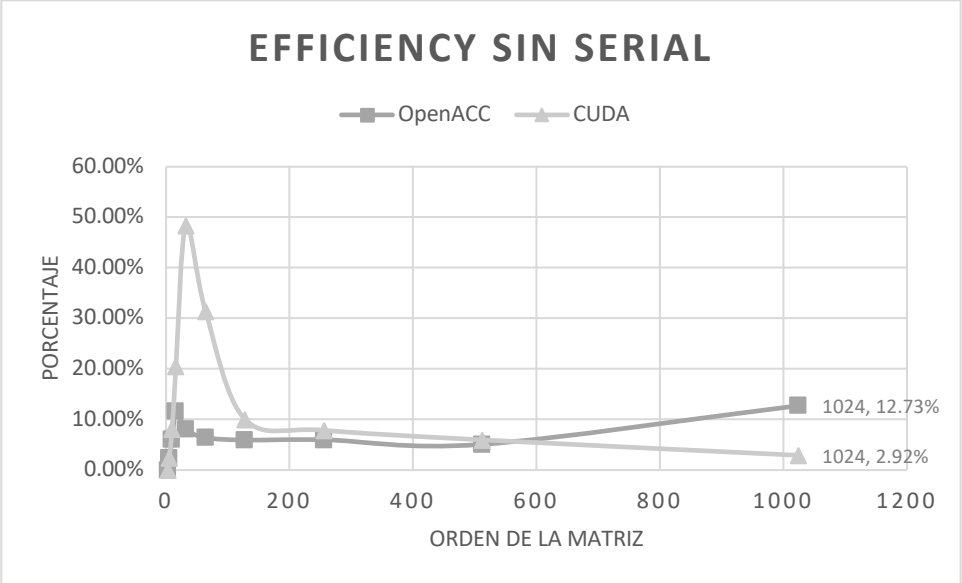

Figura 33 Eficiencia sin los valores del programa en C.

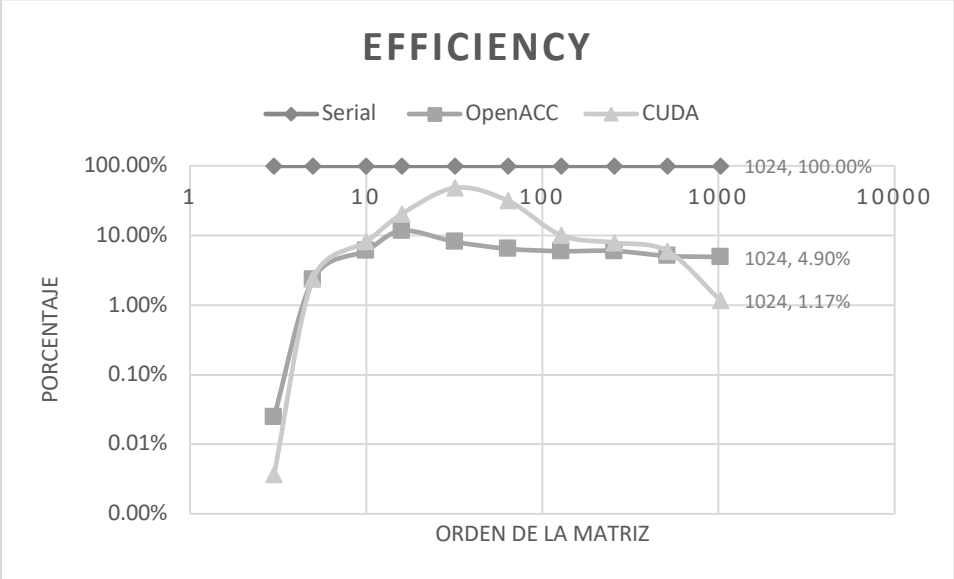

Figura 34 Eficiencia en escala logarítmica.

4.3 Resultados de una matriz

4.3.1 Matriz a evaluar

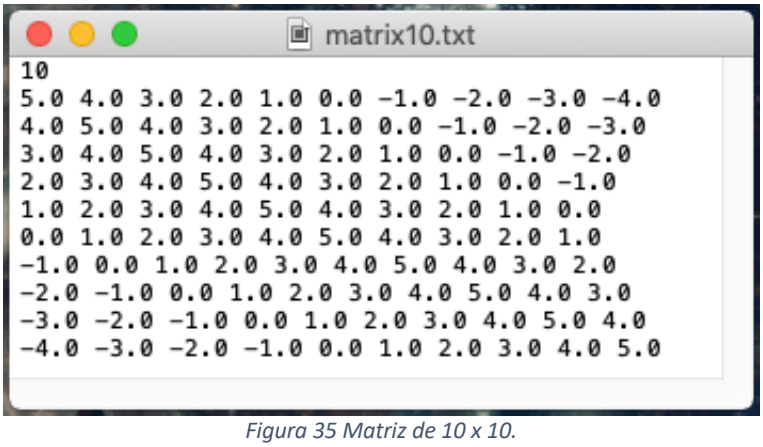

Figura 35 Matriz de 10 x 10.

4.3.2 Resultado Serial

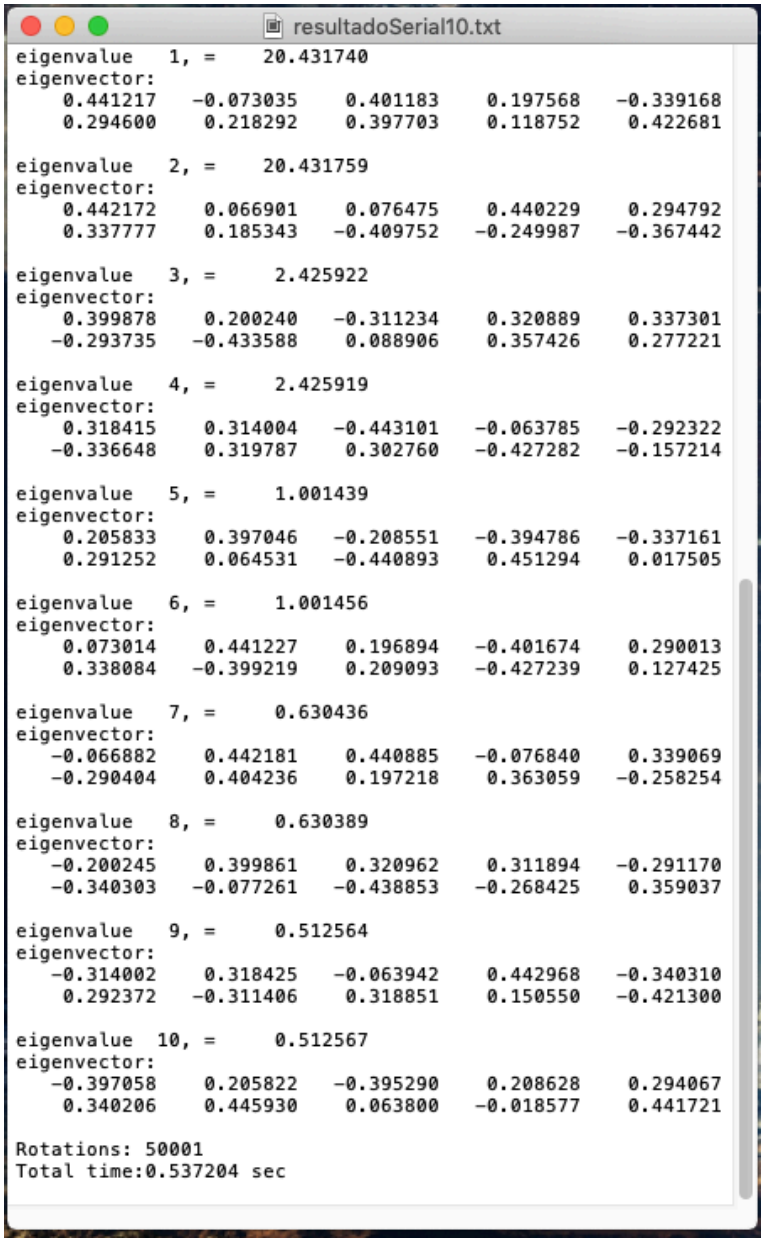

Figura 36 Resultados de Jacobi en Serial de una matriz de 10 x 10.

4.3.3 Resultado OpenACC

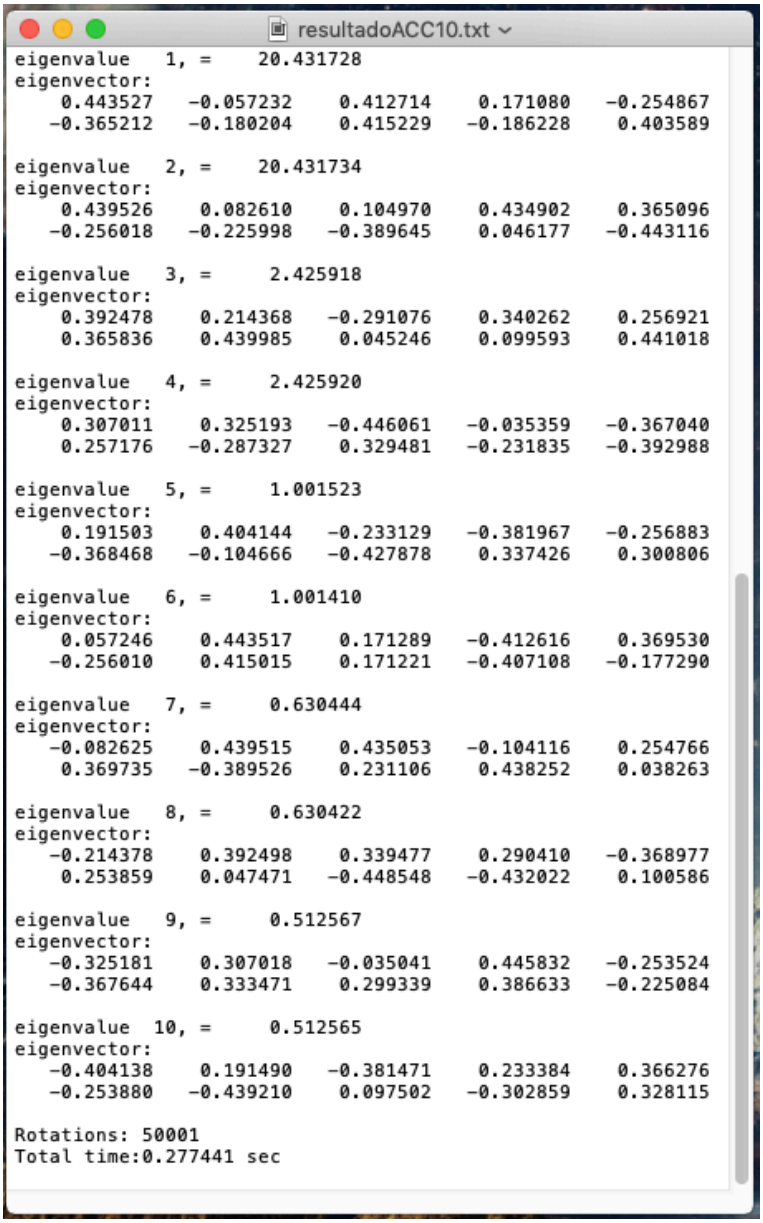

Figura 37 Resultados de Jacobi en OpenACC de una matriz de 10 x 10.

4.3.4 Resultado CUDA

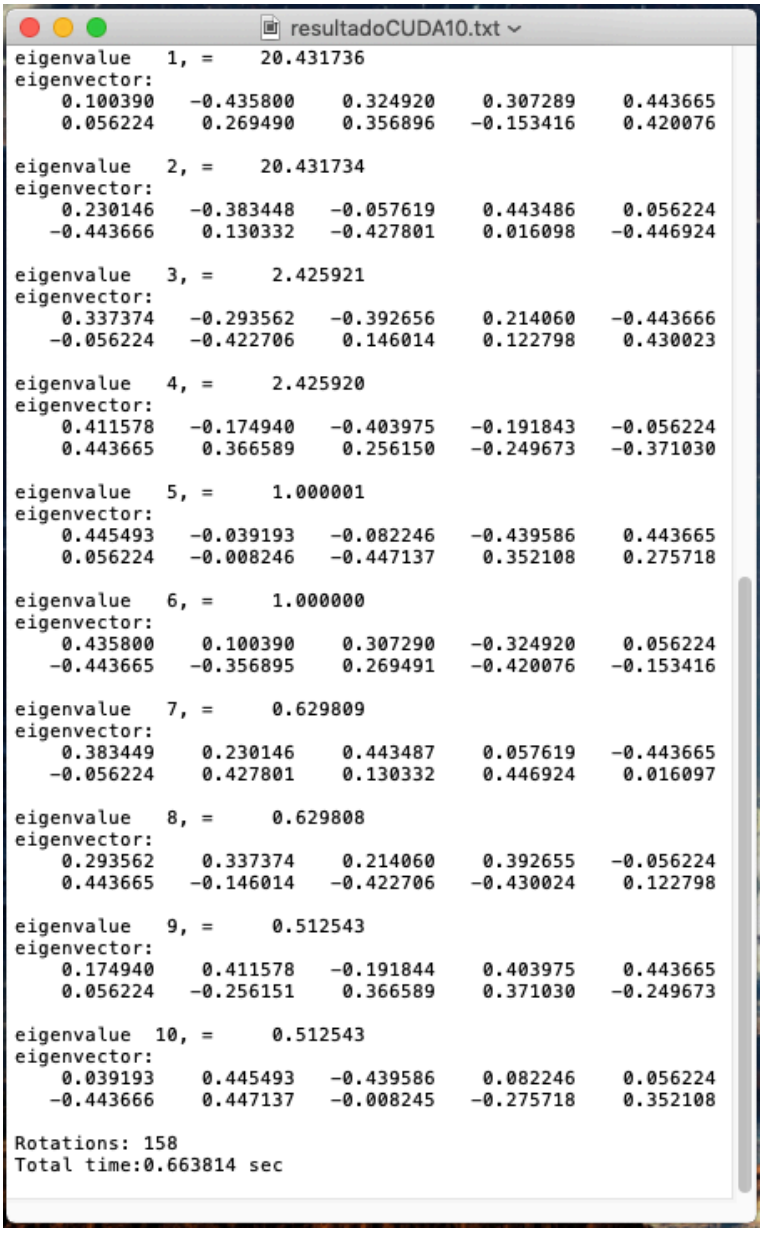

Figura 38 Resultados de Jacobi en CUDA de una matriz de 10 x 10.

Conclusiones

Normalmete los GPUs son utilizados para la computación gráfica y videojuegos, pero una alternativa para la optimización de los programas que requieren un calculo intensivo, como el que se realizó en este trabajo, es la implementación del hardware gráfico en investigaciones cientificas.

El cómputo cientifico hace uso de un gran poder de procesamiento, sumado a que necesita que se obtengan resultados en un periodo relativamente corto de tiempo. Por eso es de vital importancia encontrar en las nuevas tecnologias, formas que ayuden a la solución de problemas que antes eran impensables resolver.

Al momento de paralelizar el trabajo empiezan a surgir ciertos errores en el cálculo de resultados, ya que puede existir condicion de carrera y muchas veces los threads chocan entre si, o en el caso de CUDA, al no contar con un dispositivo que esté construido para cálculos de cómputo cientifico, los núcleos de la tarjeta gráfica tienen una precisión de punto flotante menor, entonces surgen errores de truncamiento y/o redondeo.

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

Se puede encontrar que a pocos elementos en las matrices, practicamente el único 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 también va divergiendo de nuestro algoritmo en Serial. Debemos ver qué es lo que nos importa más, obtener un resultado preciso o un resultado más rápido. En este caso en particular, los digitos que difieren son casí insignificantes, y podemos estar dispuestos a perder esa pequeña aproximación.

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 de datos entre el host y el device. Pero ahora en el caso contrario, al operar matrices muy grandes, el Runtime varía inmensamente y es evidente el ver que se compensa la pérdida de precisión con la de la aceleración del cálculo.

Considerando que, si bien, la programación serial ha estado presente desde el inicio de la computación, los problemas del campo de la ciencia y la tecnología cada día requieren de un mayor poder de procesamiento y precisión. Por ello es necesario la busqueda e implementacion tecnologias que coadyuden en las investigaciones.
EL conocer OpenACC y el entender su capacidad de paralelismo comparado con el Serial, así como las características de velocidad de CUDA darán a los investigadores la opción de cualquiera de los tres métodos de acuerdo con sus necesidades científicas.

Para un programador que se encuentra fuera del área del cómputo científico, le será fácil la implementación de OpenACC en los sistemas ya creados, ya que con pocas directivas se le indica al compilador las posibles secciones paralelizables. En cambio, CUDA requiere de un conocimiento especializado de la arquitectura de la computadora y de la tarjeta gáfica, así como para el análisis del algoritmo. Tambien es importante recordar que para utilizar el estándar CUDA, es indispensable el contar con las GPUs de la marca NVIDIA, en cambio OpenACC puede utilizar cualquier tarjeta gráfica.

Tabla de Figuras

Figura 1 Sistema coordenado molecular: i,j=electrones y A, B= núcleos. 12
Figura 2 Comparación entre Ecuaciónes de Hartee Fock y las Ecuaciónes de Khon-Sham. 14
Figura 3 Giro de ejes de una matriz simétrica. 19
Figura 4 Aceleración de programas en GPUs. 22
Figura 5 Comparación entre Multicore y Manycore. 23
Figura 6 Matriz de 3 x 3. 24
Figura 7 Conversión de matriz a arreglo. 25
Figura 8 Salida de Jacobi con matriz de 3 x 3 en C. 28
Figura 9 Salida de Jacobi con matriz de 3 x 3 en OpenACC. 32
Figura 10 Diagrama de modelo SIMD. 32
Figura 11 Arquitectura CUDA. 33
Figura 12 Jerarquía LLVM. 34
Figura 13 Mapeo de estructuras en CUDA. 35
Figura 14 Dimensión de Grids y Blocks en CUDA. 35
Figura 15 Pipeline de ejecución de CUDA. 36
Figura 16 Partición de datos en CUDA. 37
Figura 17 Partición de datos de una matriz en CUDA. 37
Figura 18 Transferencia de datos entre Host y Device en CUDA. 38
Figura 19 Orden de ID de los threads de CUDA. 38
Figura 20 Salida de Jacobi con matriz de 3 x 3 en CUDA. 41
Figura 21Gráfica de RUNTIME. 45
Figura 22 Gráfica del inicio de la divergencia del RUNTIME. 45
Figura 23 RUNTIME en escala logarítmica 45
Figura 24 Factor de Costo total. 46
Figura 25 Factor de Costo hasta matriz de 32 x 32. 47
Figura 26 Factor de Costo total en escala logarítmica 47
Figura 27 Speed up total. 48
Figura 28 Speed up de las primeras matrices. 49
Figura 29 Aceleración total. 50
Figura 30 Aceleración de las primeras matrices. 50
Figura 31 Aceleración de las primeras matrices con escala logarítmica. 51
Figura 32 Eficiencia. 52
Figura 33 Eficiencia sin los valores del programa en C. 52
Figura 34 Eficiencia en escala logarítmica. 53
Figura 35 Matriz de 10 x 10. 53
Figura 36 Resultados de Jacobi en Serial de una matriz de 10 x 10. 54
Figura 37 Resultados de Jacobi en OpenACC de una matriz de 10 x 10. 55
Figura 38 Resultados de Jacobi en CUDA de una matriz de 10 x 10. 56

Bibliografía

[1] J. M. Thijssen, Computational Physics, New York: Cambridge University Press, 2007.

[2] «Institute for Theoretical Physicas 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. [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. l. e. d. Schrödinger. [En línea]. Available: http://www.ugr.es/~jllopez/Cap3-Sch.pdf. [Último acceso: 14 junio 2018].

[10] [En línea]. Available: https://www.nucleares.unam.mx/~vieyra/node26.html. [Último acceso: 23 junio 2018].

[11] D. S. SHOLL, DENSITY FUNCTIONAL THEORY. A Practical Introduction., National Energy Technology Laboratory: Georgia Institute of Technology, 2009.

[12] T. d. f. d. l. densidad electrónica. [En línea]. Available: http://www.fis.cinvestav.mx/~daniel/thELA.pdf. [Último acceso: 12 junio 2018].

[13] 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. [Último acceso: 14 junio 2018].

[14] J. K. Labanoswski, Density Functional Methods in Chemestry, New York: Springer-Velarg, 1991.

[15] E. Urriolabeitia, «La teoría del funcional de la densidad (DFT) va por el mal camino,» *divulgame.org,* 9 enero 2017.

[16] T. D. F. D. L. DENSIDAD. [En línea]. Available: http://depa.fquim.unam.mx/amyd/archivero/DensityFunctionalTheory_21556.pdf. [Último acceso: 14 junio 2018].

[17] [En línea]. Available: http://www.demon-software.com/public_html/index.html. [Último acceso: 23 junio 2018].

[18] [En línea]. Available: http://www.demon-software.com/public_html/program.html#branches. [Último acceso: 23 junio 2018].

[19] F. J. Ayres, MATRICES, Teoria y problemas resueltos, McGraw-Hill, 1962.

[20] S. K. Shan, Numerical Methods and Computers, Massachusetts: Addison-Wesley, 1966.

[21] [En línea]. Available: https://www.uv.es/diazj/cn_tema4.pdf.

[22] «Método de Jacobi,» [En línea]. Available: https://www.academia.edu/12780782/M%C3%89TODO_DE_JACOBI.

[23] [En línea]. Available: http://la.nvidia.com/object/what-is-gpu-computing-la.html. [Último acceso: 23 junio 2018].

[24] [En línea]. Available: https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/. [Último acceso: 12 mayo 2018].

[25] [En línea]. Available: https://www.profesionalreview.com/2017/06/21/diferencia-la-cpu-la-gpu/. [Último acceso: 23 junio 2018].

[26] W. H. Press, Numerical Recipes in C. The Art of Scientific Computing, United States of America: Cambridge University Pres, 2002.

[27] «OpenACC: El API GPGPU Post-CUDA,» 12 2011. [En línea]. Available: https://www.fayerwayer.com/2011/12/openacc-el-api-gpgpu-post-cuda/.

[28] openacc.org, OpenACC Programming and Best Practices Guide, 2015.

[29] «Monday-Programming-GPUs-OpenACC,» [En línea]. Available: http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0517B-Monday-Programming-GPUs-OpenACC.pdf.

[30] [En línea]. Available: https://es.wikipedia.org/wiki/SIMD.

[31] J. Cheng, Professional CUDA C Programming, Indianapolis: Jhon Wiley & Sons, 2014.

[32] [En línea]. Available: https://www.pelagos-consulting.com/?p=503.

[33] M. A. Barry Wilkinson, Parallel Programming: Techniques and Aplications Using Networked Workstations an Parallel Computers, Segunda Edicion ed., Pearson, 2005.

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,"%.1f  ",m[i][j]); //Escribe
   numeros aleatorios
47.                 }
```

```
                    fprintf(archivo,"%c",'\n');
                }
                 fclose(archivo); // Cierra el archivo

                printf("File created\n");
        }

        printf("\n");
        return 0;
    }
```

CF 1.2 auxFuncs.h

```
/*
 * auxFuncs.h
 * Auxiliary functions for timing and allocating data.
 *
 * José Antonio Ayala Barbosa
 * Oct, 2018
 *
 *
 */

    #include <stdio.h>
    #include <stddef.h>
    #include <stdlib.h>
    #include <sys/time.h>

    void nrerror(char error_text[]){
    /* Numerical Recipes standard error handler */
       fprintf(stderr,"Numerical Recipes run-time error...\n");
       fprintf(stderr,"%s\n",error_text);
       fprintf(stderr,"...now exiting to system...\n");
       exit(1);
    }

    void get_walltime_(double* wcTime) {
       struct timeval tp;
       gettimeofday(&tp, NULL);
       *wcTime = (double)(tp.tv_sec + tp.tv_usec/1000000.0);
    }

    void get_walltime(double* wcTime) {
       get_walltime_(wcTime);
    }

    float *vectorF(int n){
       float *v;
       v=(float *) malloc((size_t)(n*sizeof(float)));
       if (!v) nrerror("allocation failure in vector()");
        return v;
    }

    int *vectorI(int n){
```

```
        int *v;
        v=(int *) malloc((size_t)(n*sizeof(int)));
        if (!v) nrerror("allocation failure in vector()");
         return v;
    }
```

CF 1.3 driverA.c

```
/*
 * driverA.c
 * Driver for function JACOBI.
 *
 * José Antonio Ayala Barbosa
 * Oct, 2018
 *
 *
 */

    #include <stdio.h>
    #include <sys/time.h>
    #include <stdlib.h>
    #include "auxFuncs.h"
    #include "jacobiA.h"

    int main(int argc, char **argv)
    {

       int n;
        char *nombreArchivo=argv[1];
        double S,E;
       int i, j, k, nrot=0;
        FILE *archivo;
        float *c,*mat,*eval,*evec;

        if (fopen(nombreArchivo, "r") == NULL){
            printf("File not found\n");
            return 1;
        }else{
            archivo = fopen(nombreArchivo, "r");
            fscanf(archivo, "%d", &n);
            c = vectorF(n*n);
            for (i = 0; i < n; ++i){
                for (j = 0; j < n; ++j){
                    fscanf(archivo, "%f", &c[i*n+j]);
                }
            }
            fclose(archivo);
        }

        mat=vectorF(n*n);
        evec=vectorF(n*n);
        eval=vectorF(n);

        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.           ("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
   coordenate i
11.         int max_j = 1;                                          //first
   coordenate 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])
   ){ //if exists a higher element
19.                       max_i = r;                            //replace
   new coor
20.                       max_j = c;
21.                 }
22.          }
23.          piv_elem[0] = max_i;                             //store new
   coordenates
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_i])
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.        //On input
133.        //mat: Contains the matrix to be diagonalized.
134.        //n: Order of matrix a.
135.
136.        //On output
137.        //eigvec: eigenvectors to be computed v
138.        //eigval: Contains the eigenvalues in ascending order
139.        //nrot: Number of Jacobi rotations.
140.     /*******************************+******************************
   */
141.
142.        int i,j;
143.        int *piv_elem;              //Keep coordenates of an elemnt
   i,j
144.        bool min = false;
145.        float EPS = .0000001;
146.        float *T;                   //Contains the ratation matrix
147.        float *mat_temp;            //A temporal matrix
148.
149.        mat_temp = vectorF(n*n);
150.        T = vectorF(n*n);
151.        piv_elem = vectorI(2);
152.
153.        for (*nrot = 0; min == false ; ++*nrot){
154.            max_elem(piv_elem,n,mat);  //Search for max element in
   tringle up
155.
156.            if(fabs(mat[piv_elem[0]*n+piv_elem[1]]) < EPS || *nrot
   >= 50000 ) //if max element doesnt exist more
157.              min=true;
158.
```

```
159.           else{
160.               new_T_mat(piv_elem[0],piv_elem[1],n,mat,T,mat_temp);
    //Calculate T
161.               mat_mult(n,eigvec,T,mat_temp);                  //Eigen
   vect
162.               copy_mat(n,mat_temp,eigvec);
163.               mat_mult_tra(n,T,mat,mat_temp);
164.              mat_mult(n,mat_temp,T,mat);
165.         }
166.
167.            for (i = 0; i < n; ++i)
168.               eigval[i]=mat[i*n+i];
169.         }
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 *ma
   t){
20.           int r,c;
21.           int max_i = 0;                          //first coordenate i
22.           int max_j = 1;                          //first coordenate j
23.
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])){
   //if exists a higher element
27.                       max_i = r;                 //replace new coor
28.                       max_j = c;
29.                 }
30.            piv_elem[0] = max_i;              //store new coordenates
31.            piv_elem[1] = max_j;
32.
33.       }
34.
35.       __global__ void kernel_set_max_elem(float *mat,int n, int *pi
   v_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,floa
   t *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), cudaMemcp
   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.     }
```