

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

José Antonio Ayala Barbosa

Tutor: Dr. José Jesús Carlos Quintanar Sierra

# Índice

1. Objetivo.
2. Matriz de Khon-Sham.
3. Algoritmo de Jacobi.
4. Implementación en C, OpenACC y CUDA.
5. Pruebas y resultados.
6. Conclusiones.

# 1. Objetivo.

- El objetivo del trabajo es programar el método numérico de Jacobi para resolver la matriz de Khon-Sham, y paralelizar la implementación en tarjetas gráficas utilizando las tecnologías de OpenACC y CUDA. Y posteriormente comparar el desempeño de los tres programas con algunas métricas de cómputo de alto desempeño.

## 2. Matriz de Khon-Sham.

- La ecuación de Khon-Sham se usa para determinar, de manera aproximada, el nivel de energía más bajo de un sistema atómico.
- El método para su obtención, consiste en proponer una función tentativa que depende de varios parámetros, entre ellos la posición de los electrones con respecto a los nucleos, los cuales se varían hasta que se tenga una energía mínima.
- La ecuación es de tipo cuadrática, por lo que puede transformarse en una matriz cuadrática para encontrar su solución.

### 3. Algoritmo de Jacobi.

- Un método numérico utilizado para resolver la matriz de Khon-Sham, es el método de Jacobi.
- Matriz Khon-Sham tiene forma cuadrática (matriz simétrica o Hermitiana):

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

$$q(x_1, x_2, \dots, x_n) = (x_1, x_2, \dots, x_n) \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = X'AX$$

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

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

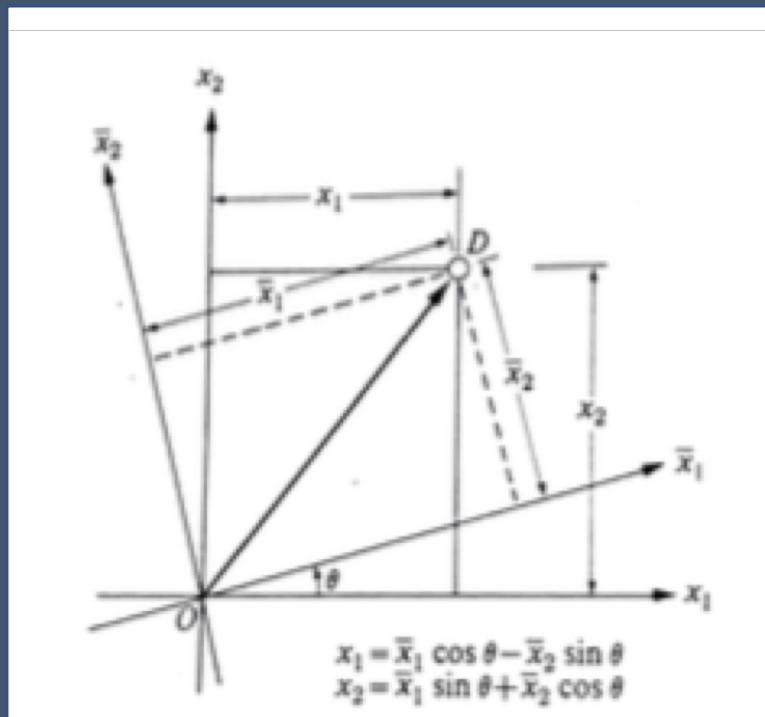
### 3. Algoritmo de Jacobi.

- Objetivo. Encontrar:
- Los **eigenvalores**, tambien llamados valores caracteristicos de la matriz, son los escalares alojados en la diagonal principal (en caso de la matriz de Khon-Sham, estos escalares son las energías de los electrones del sistema atómico).
- y los **eigenvectores**, vectores columna a los que les corresponde un eigenvalor, (representan los estados en los que están los electrones).

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

### 3. Algoritmo de Jacobi.

- Diagonalización.
- El método de Jacobi elimina los elementos fuera de la diagonal de una matriz simétrica o Hermitiana, rotando los ejes de la matriz.



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

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

### 3. Algoritmo de Jacobi.

- Obtención de Eigenvalores:

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

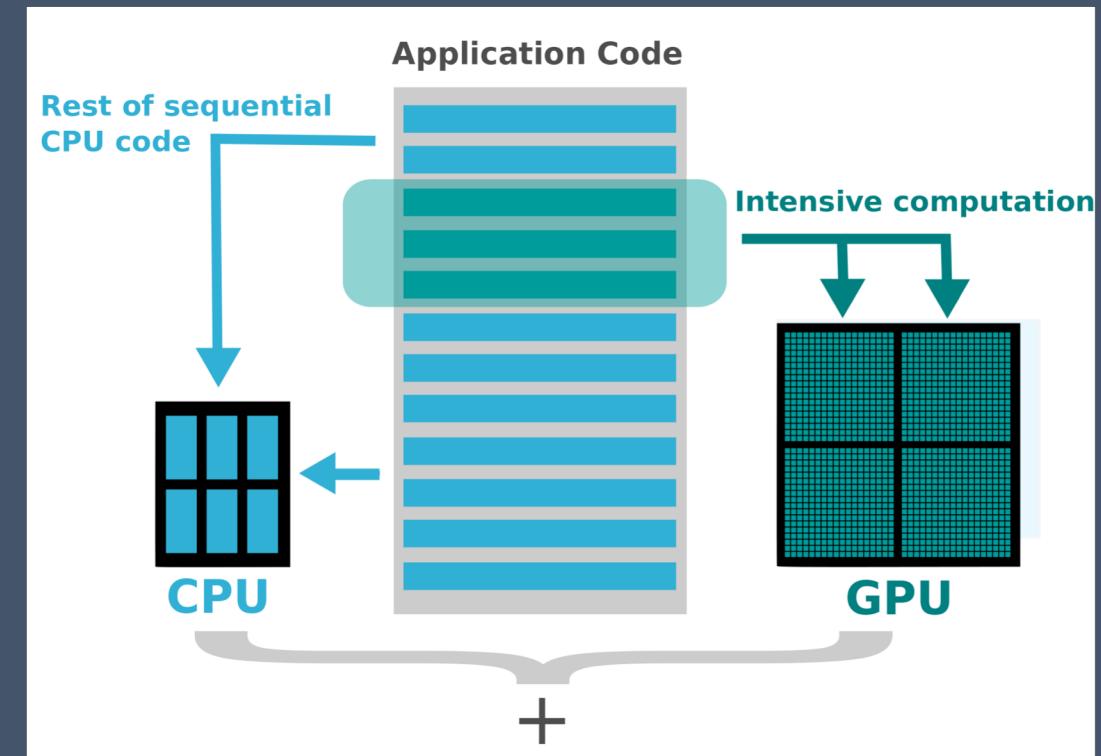
$$= \begin{bmatrix} a_{11}\cos^2\theta + 2a_{12}\sin\theta\cos\theta + a_{22}\sin^2\theta & a_{12}(\cos^2\theta - \sin^2\theta) + \sin\theta\cos\theta(a_{22} - a_{11}) \\ a_{12}(\cos^2\theta - \sin^2\theta) + \cos\theta\sin\theta(a_{22} - a_{11}) & a_{11}\sin^2\theta - 2a_{12}\sin\theta\cos\theta + a_{22}\cos^2\theta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

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

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

## 4. Implementación

- Cómputo en GPGPU. (Cómputo de propósito general en unidades de procesamiento de gráficos).
- Con aplicaciones computacionales intensivas, las secciones del programa a menudo muestran una gran cantidad de paralelismo de datos.
- Es posible delegar al GPU para acelerar el cálculo de dichas secciones.



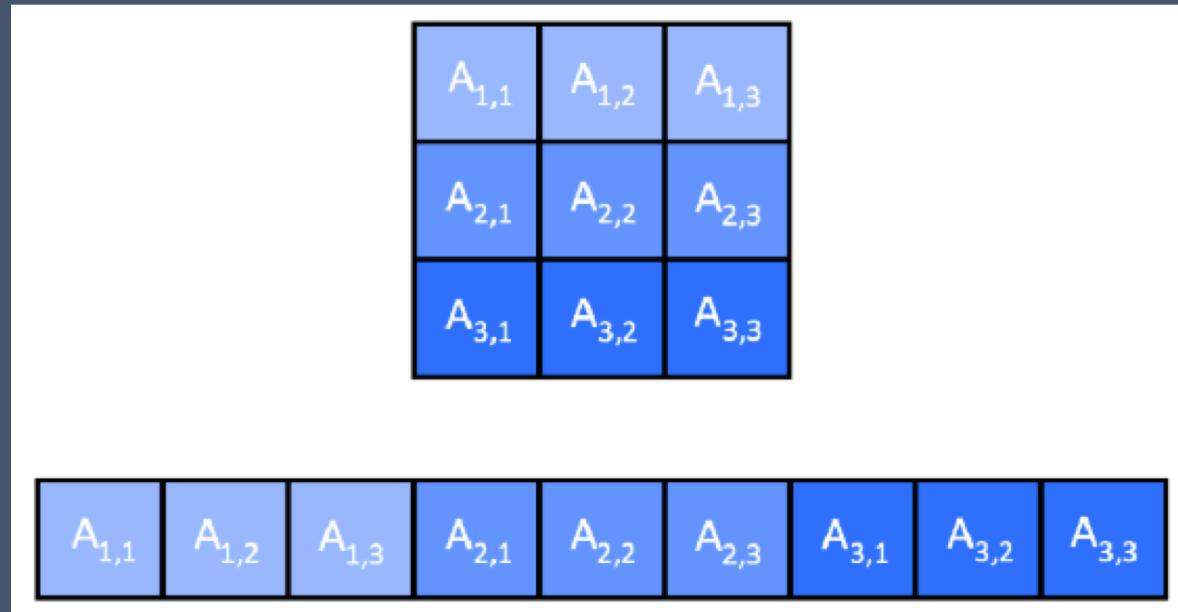
# 4. Implementación.

- Características del equipo:

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

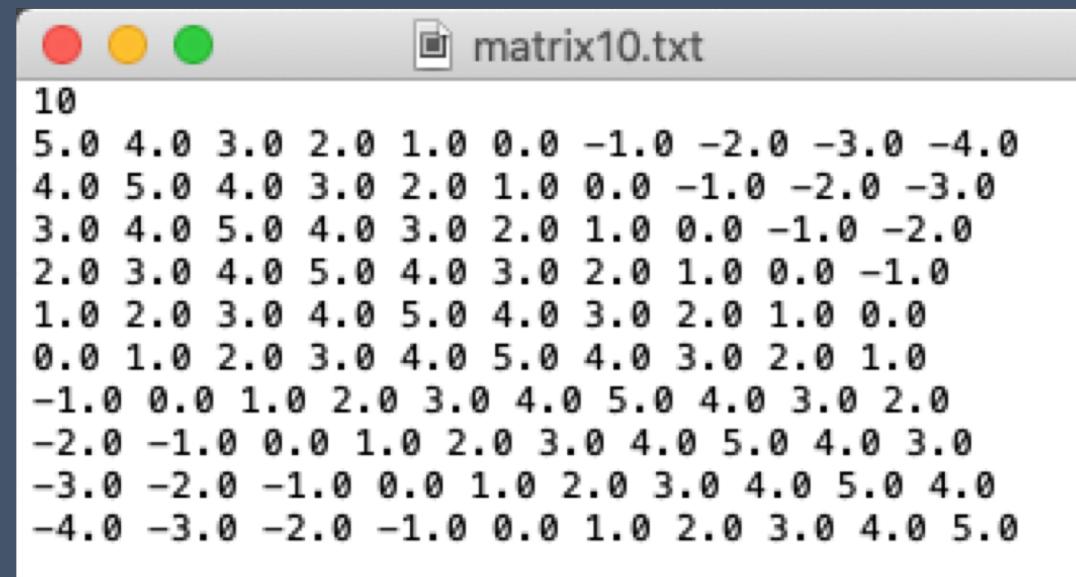
## 4. Implementación.

- Matrices Vector: Ya que, como se verá más adelante, en la implementación en CUDA es conveniente que las matrices sean declaradas como arreglos unidimensionales. Todas las implementaciones fueron realizadas así.
- Pasando así de  $A[i][j]$  a  $A[m][i*n+j]$ .



## 4. Implementación en C.

- Funciones auxiliares.
- creaMatrix.c
- Emula matrices cuadráticas, que siguen la forma que las de la ecuación de Khon-Sham. Genera archivos con extensión “.txt”.



A screenshot of a terminal window titled "matrix10.txt". The window contains a 10x10 matrix of floating-point numbers. The matrix is symmetric and follows a specific pattern where the diagonal elements are 5.0, the super-diagonal elements are 4.0, and the sub-diagonal elements are -1.0. All other elements are 0.0. The matrix is as follows:

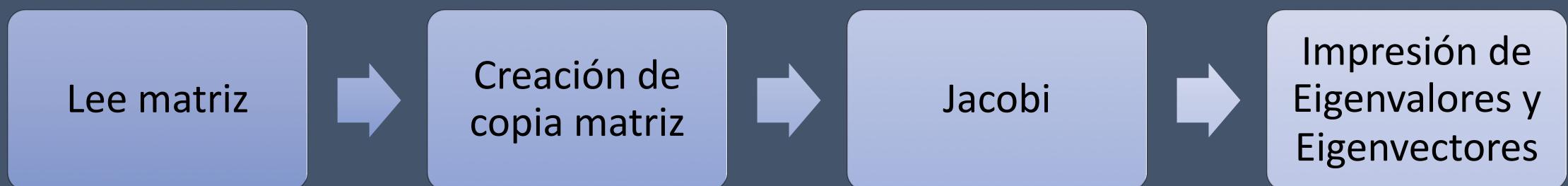
5.0	4.0	3.0	2.0	1.0	0.0	-1.0	-2.0	-3.0	-4.0
4.0	5.0	4.0	3.0	2.0	1.0	0.0	-1.0	-2.0	-3.0
3.0	4.0	5.0	4.0	3.0	2.0	1.0	0.0	-1.0	-2.0
2.0	3.0	4.0	5.0	4.0	3.0	2.0	1.0	0.0	-1.0
1.0	2.0	3.0	4.0	5.0	4.0	3.0	2.0	1.0	0.0
0.0	1.0	2.0	3.0	4.0	5.0	4.0	3.0	2.0	1.0
-1.0	0.0	1.0	2.0	3.0	4.0	5.0	4.0	3.0	2.0
-2.0	-1.0	0.0	1.0	2.0	3.0	4.0	5.0	4.0	3.0
-3.0	-2.0	-1.0	0.0	1.0	2.0	3.0	4.0	5.0	4.0
-4.0	-3.0	-2.0	-1.0	0.0	1.0	2.0	3.0	4.0	5.0

## 4. Implementación en C.

- Funciones auxiliares.
- auxFuncs.h
- Fue necesario crear una biblioteca llamada **auxFuncs.h** que contuviera algunas funciones auxiliares que se utilizarían a lo largo de los programas.
  - Funciones para obtener el tiempo del Sistema.
  - Funciones para reservar memoria.
  - Funciones de ayuda, en caso que ocurra algun error en la asignación de memoria.
- Esta biblioteca se basó en la que contiene el libro de Numerical Recipes

## 4. Implementación en C.

- Funciones auxiliares.
- driverA.c
- Contiene la función main(), y llama a las bibliotecas auxFunc.h y jacobiA.h
- Flujo del programa:



## 4. Implementación en C.

- Funciones Jacobi.
- jacobiA.c
- Contiene la función jacobiMultip(), la cual es el núcleo de operación del método numérico para resolver la matriz de Khon-Sham.
  - Se empezará a iterar el método buscando el elemento máximo en el triángulo superior de la matriz.
  - Las iteraciones las realizará hasta que
    - Se encuentre que el máximo elemento a eliminar es menor que el umbral (threshold) recomendado por Numerical Recipes de  $1 \times 10^{-7}$  [1].
    - O que se hayan superado las 50,000 iteraciones,
  - En cualquiera de los casos termina el ciclo por que ha tomado el máximo elemento como  $|0|$ .

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

## 4. Implementación en C.

- Funciones Jacobi.
- *jacobiMultip(mat,n,evec,eval,nrot);*

**jacobiMultip(mat,n,evec,eval,nrot);**

**mat -> La matriz de operación.**

**n -> El orden de la matriz.**

**evec -> El eigenvector**

**eval -> El arreglo de eigval**

**nrot -> Variable inicializada en 0 que da el número de rotaciones que realizo el algoritmo**

# 4. Implementación en C.

Crear matriz de Rotación

- *max\_elem(piv\_elem,n,mat);*
- *new\_T\_mat(piv\_elem[0],piv\_elem[1],n,mat,T,mat\_temp);*

Multiplicación de  
Eigenvector

- *mat\_mult(n,eigvec,T,mat\_temp);*
- *copy\_mat(n,mat\_temp,eigvec);*

Premultiplicación de  
matriz de rotaciones

- *mat\_mult\_tra(n,T,mat,mat\_temp);*

Postmultiplicación de  
matriz de rotaciones

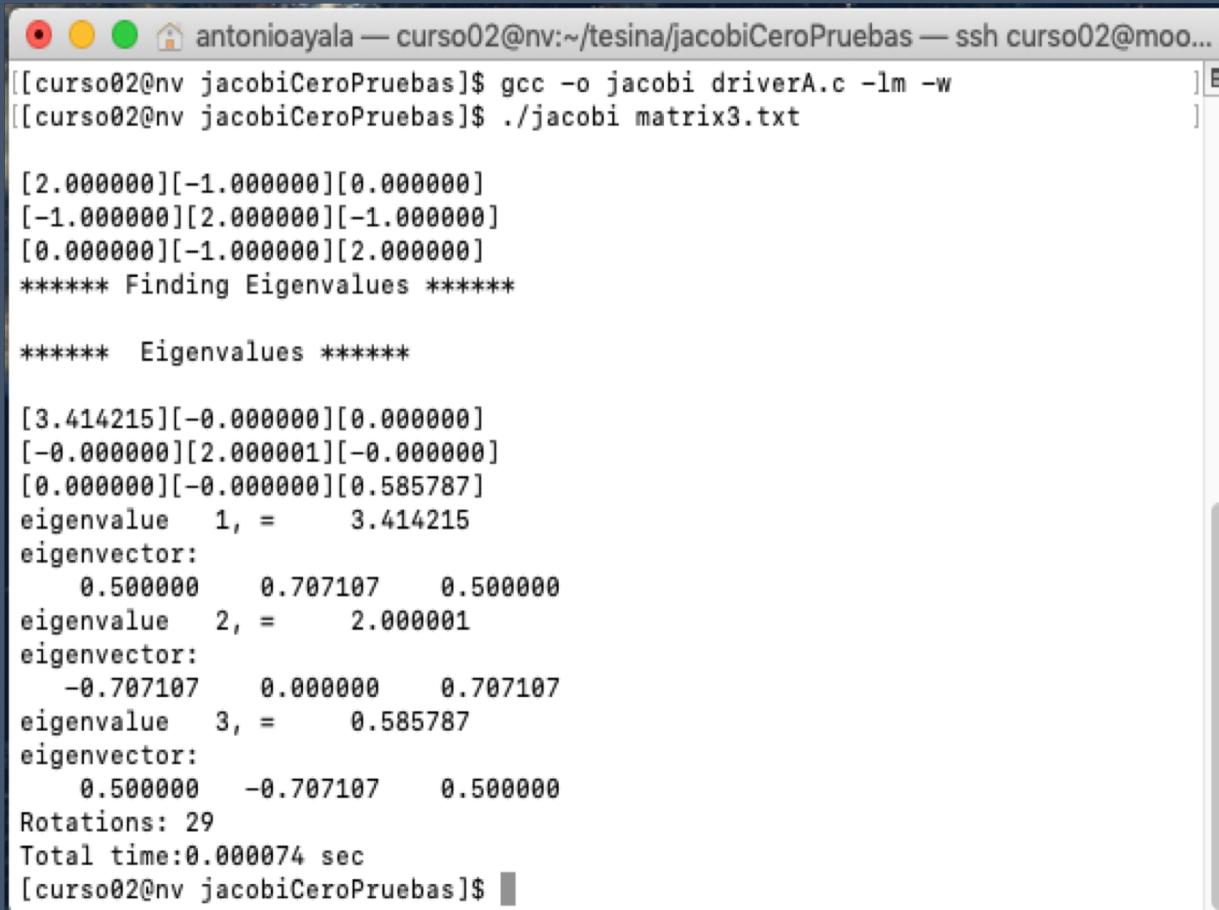
- *mat\_mult(n,mat\_temp,T,mat);*

Guardar eigenvalores

- *eigval[i]=mat[i\*n+i];*

# 4. Implementación en C.

- Referencia código serial.
- Compilación y ejecución.



The screenshot shows a terminal window with the following text output:

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

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

***** Eigenvalues *****

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

# 4. Implementación en OpenACC.

- Introducción a OpenACC.
- El estandar de programación permite que con algunas directivas se le indique 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.
- Para poder utilizar el estándar es necesario tener el compilador PGI que tiene las bibliotecas de automatización del paralelismo.
- El estandar puede utilizarse en tarjetas gráficas de cualquier marca.

# 4. Implementación en OpenACC.

- Directivas.
- Para darle los parámetros de la configuración de bloques al compilador, debe usarse:
- *Vectors*: Es el elemento de granularidad más fina, (*core o thread*).
- *Gangs*: Es un grupo o bloque de vectors.

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

# 4. Implementación en OpenACC.

- Las posibles regiones donde existe paralelismo:

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

```
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];  
    }  
    ...  
}
```

```
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];  
            }  
        }  
    }  
    ...  
}
```

```
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];  
            }  
        }  
    }  
    ...  
}
```

# 4. Implementación en OpenACC.

- Compilación y ejecución.

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

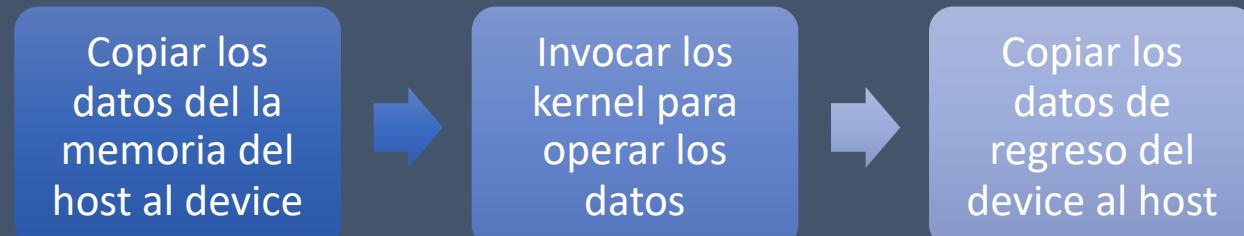
[2.000000][-1.000000][0.000000]
[-1.000000][2.000000][-1.000000]
[0.000000][-1.000000][2.000000]
***** Finding Eigenvalues *****

***** Eigenvalues *****

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

## 4. Implementación en CUDA.

- Introducción a CUDA.
- Arquitectura de hardware y de software que permite ejecutar programas en las tarjetas gráficas de la marca NVIDIA.
- Un programa en CUDA consiste en la mezcla de dos códigos, el host code (CPU) y el device code (GPU). El compilador de NVIDIA, *nvcc*, separa ambos códigos durante el proceso de compilación.
- Funciones que se realizan en GPU(conveniente llamarlas *kernel\_nombre de func*).

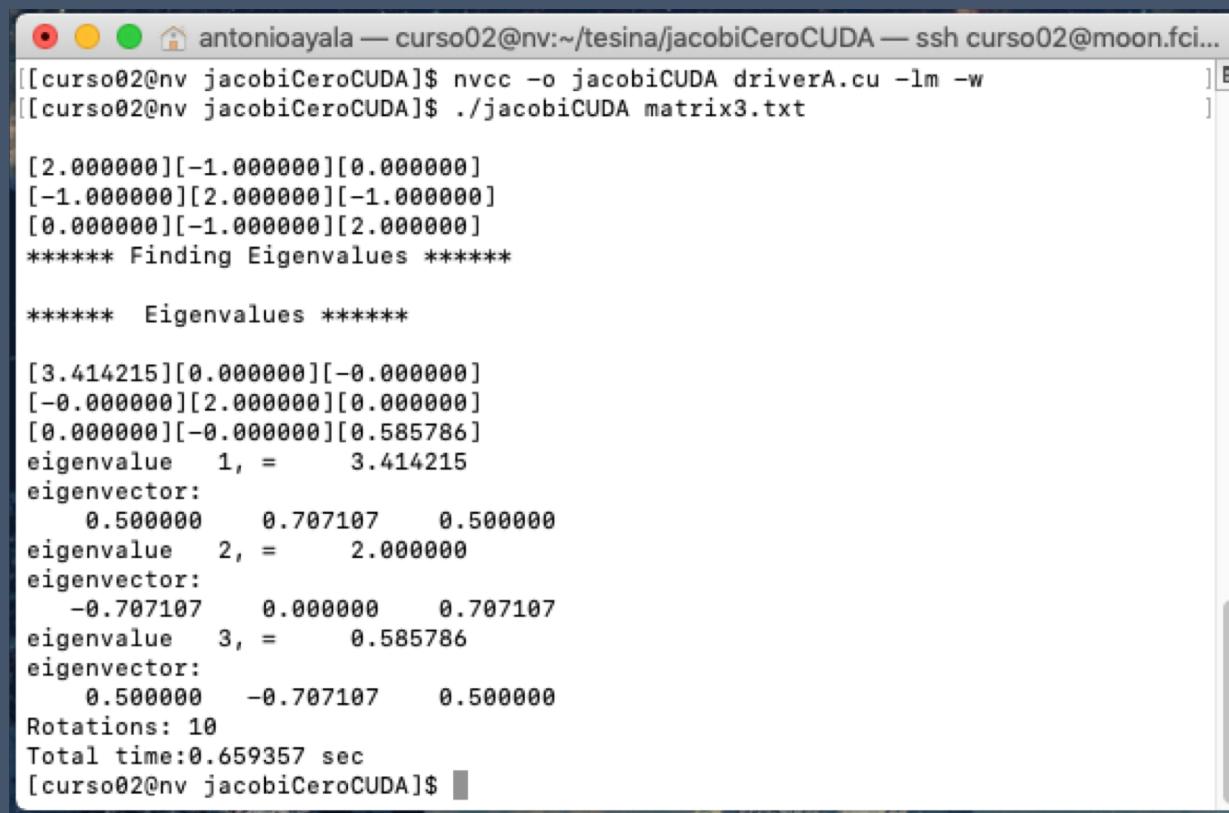


## 4. Implementación en CUDA.

- 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**.
- Thread: Ejecuta una instancia de un kernel.
- Bloque: Agrupación de threads que utilizan memoria compartida.
- Ejemplo de una función:
  - *`__global__ void kernel_helloFromGPU(argument list){}`*
- *Ejemplo de cómo llamarla:*
  - *`kernel_name <<<#blocks, #threads>>>(argument list);`*

# 4. Implementación en CUDA.

- Compilación y ejecución.



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

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

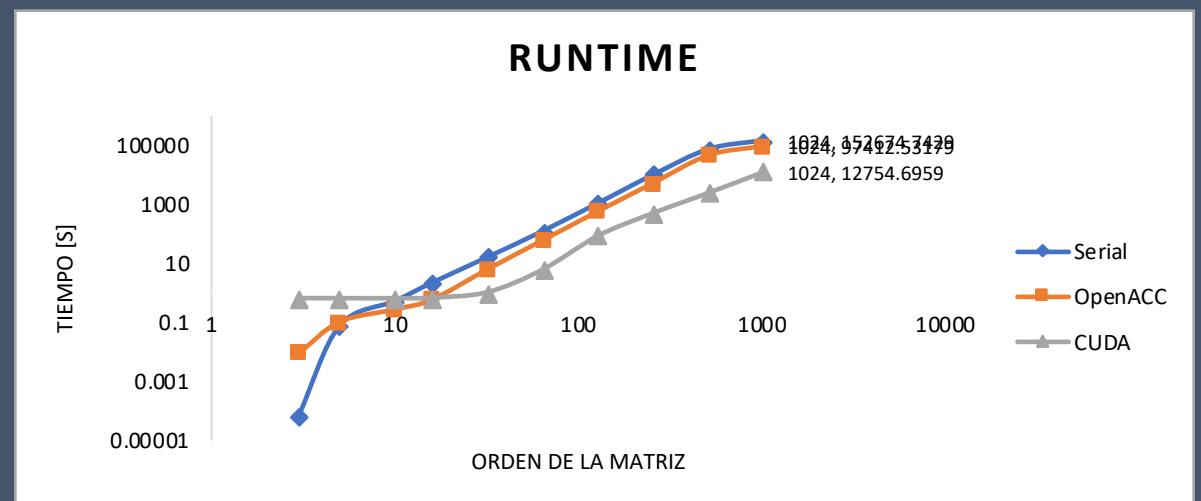
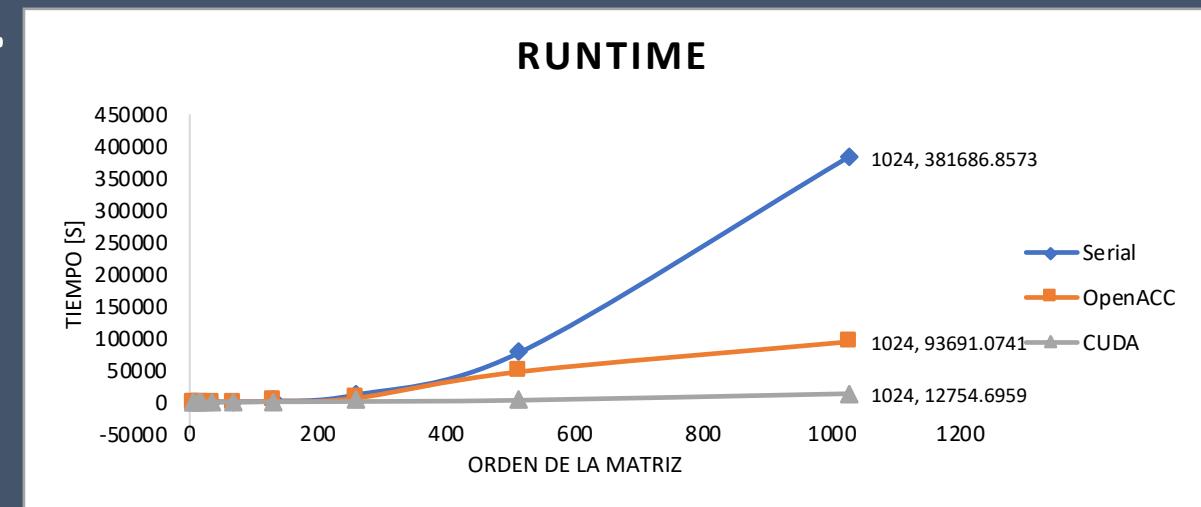
***** Eigenvalues *****

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

# 5. Pruebas y resultados.

## 1. Runtime. Tiempo de ejecución. $t(p)$ .

Orden	RUNTIME [s]		
	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



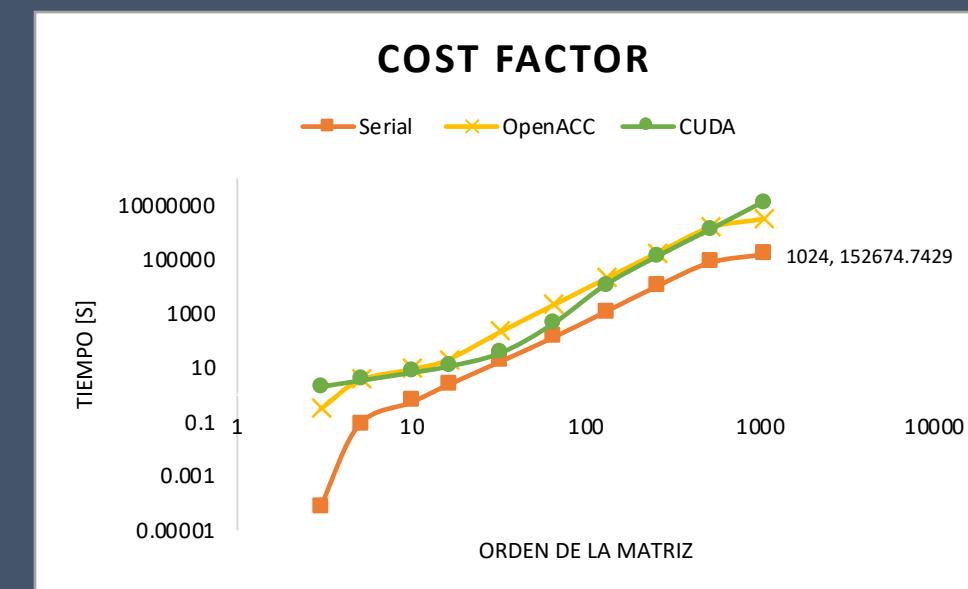
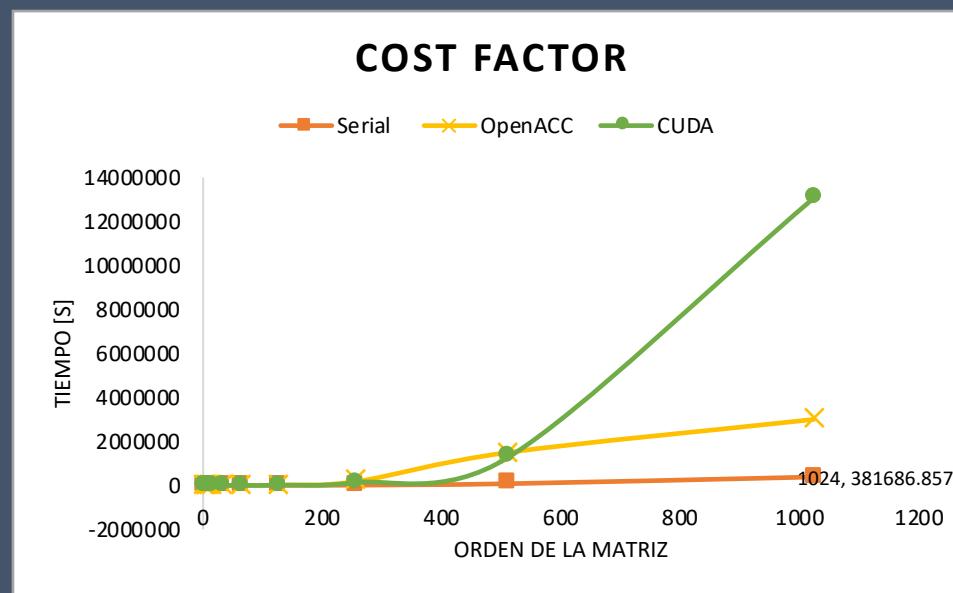
# 5. Pruebas y resultados.

**2. Cost Factor.** Cantidad de trabajo realizado por el programa.

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

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

COST FACTOR						
Orden	Proc	Serial	Proc	OpenACC	Proc	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



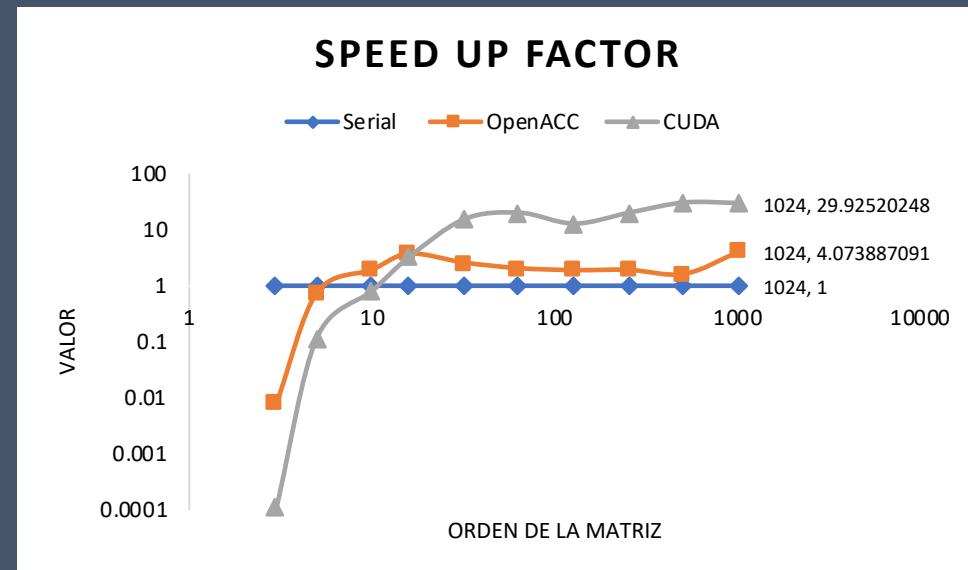
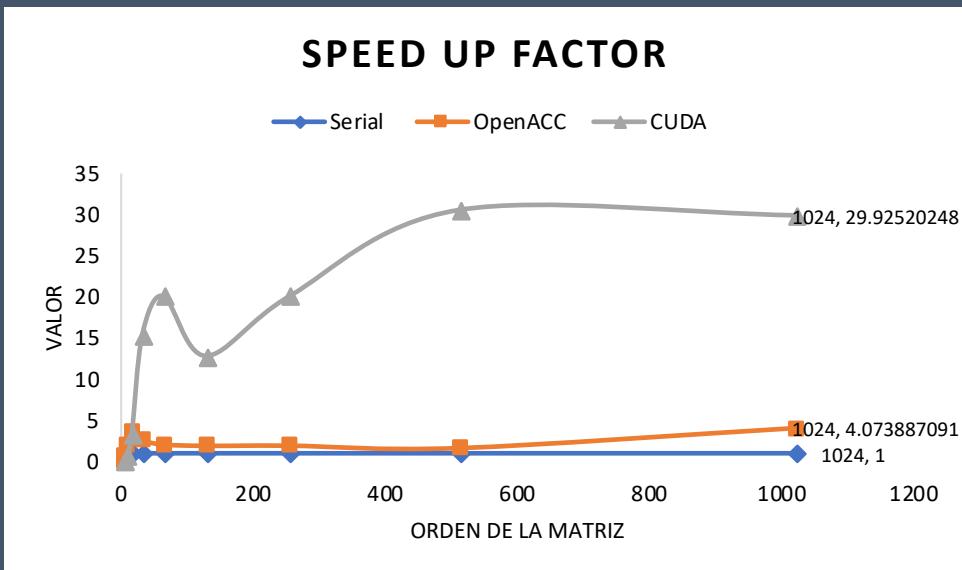
# 5. Pruebas y resultados.

**3. Speedup Factor.** Medición relativa del rendimiento de un programa en paralelo.

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

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

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

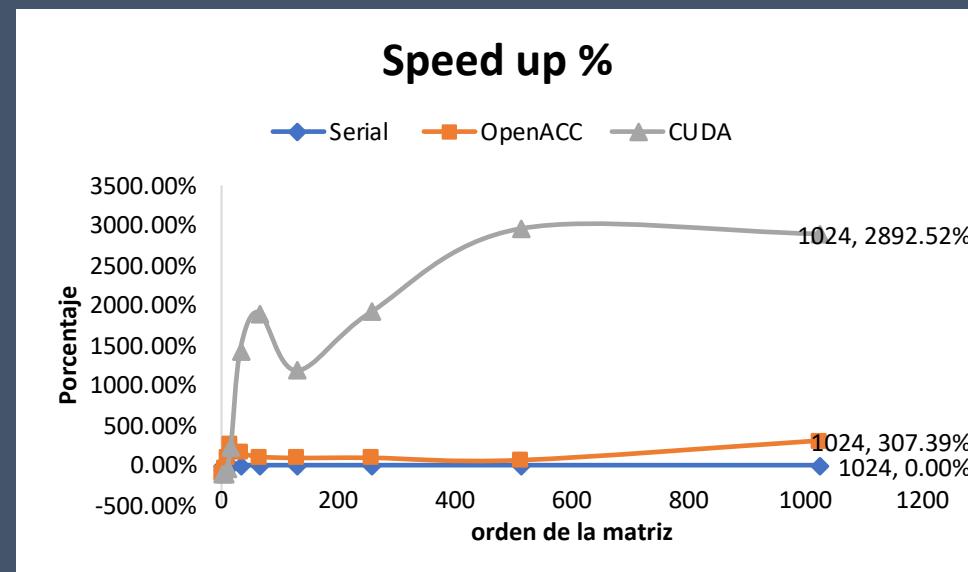
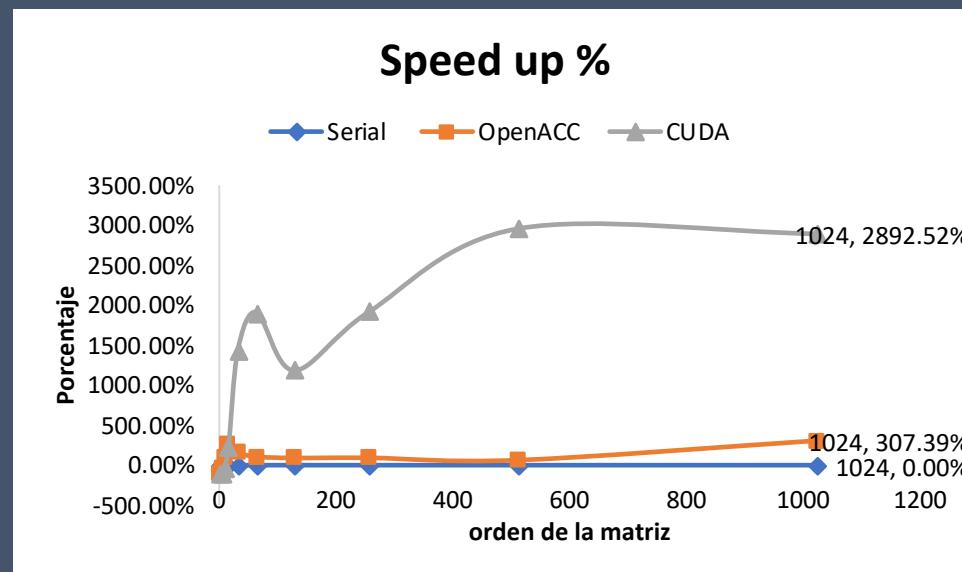


# 5. Pruebas y resultados.

**4. Speedup en porcentaje.** Da el porcentaje del Speedup con respecto al programa en un solo procesador.

$$AC(p) = \frac{\text{runtime en un procesador}}{\text{runtime con } p \text{ procesadores}} - \frac{\text{runtime en un procesador}}{\text{runtime en un procesador}}$$
$$Ac(p) = \frac{t_s}{t_p} 100\% - 100\%$$

Orden	Speedup %		
	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%	56.73%	1097.01%



# 5. Pruebas y resultados.

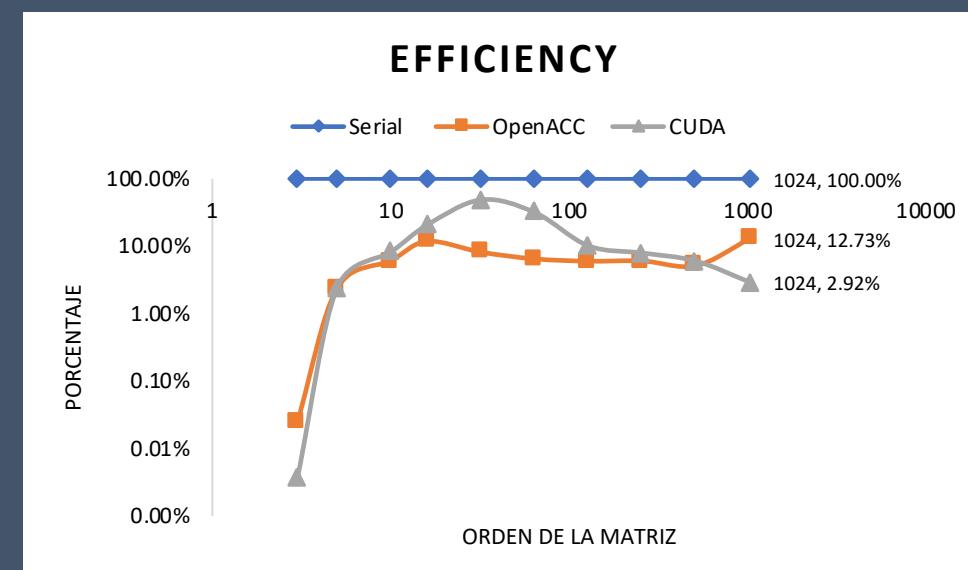
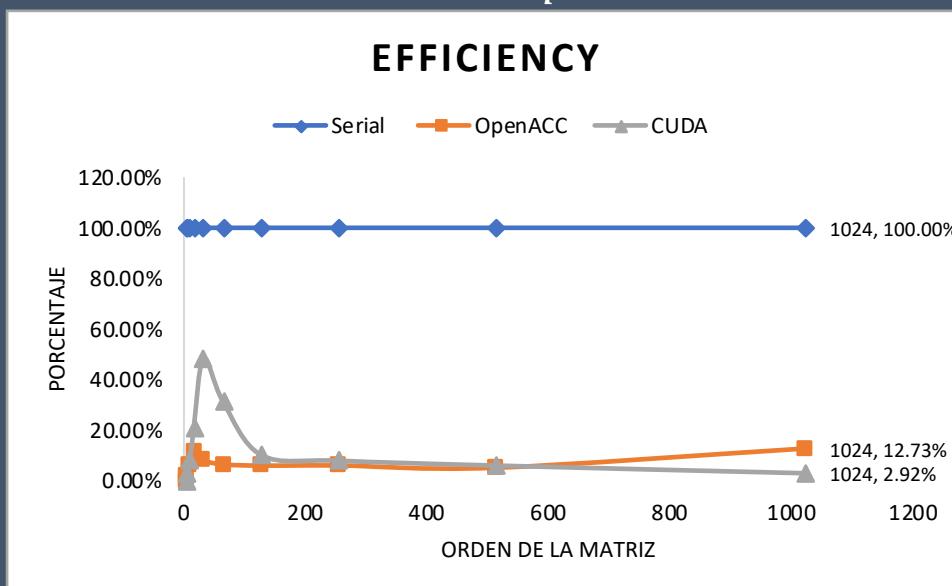
**5. Efficiency.** Tiempo tarda cada procesador en realizar su tarea.

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

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

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

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%



# 6. Conclusiones.

- Al momento de paralelizar el trabajo empiezan a surgir ciertos errores en el cálculo de resultados.
  - truncamiento
  - redondeo
- Al no contar con un dispositivo que esté construido para cálculos de **cómputo científico (Tesla)**, los núcleos de la tarjeta gráfica tienen una precisión de punto flotante menor.

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

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

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

## 6. Conclusiones.

- 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 gráfica, así como para el análisis del algoritmo.
- Para utilizar el estándar CUDA, es indispensable el contar con las GPUs de la marca NVIDIA, en cambio OpenACC puede utilizar una tarjeta gráfica de cualquier marca.

### 3. Algoritmo de Jacobi.

- Obtencion de eigenvectores. (Multiplicaciones sucesivas de matriz de rotación).

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

Si aplicamos un poco de álgebra:

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

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  $[T]$ .

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

# 5. Pruebas y resultados.

**resultadoSerial10.txt**

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

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

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

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

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

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

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

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

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

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

Rotations: 50001
Total time: 0.537204 sec
```

**resultadoACC10.txt**

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

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

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

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

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

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

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

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

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

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

Rotations: 50001
Total time: 0.277441 sec
```

**resultadoCUDA10.txt**

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

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

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

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

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

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

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

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

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

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

Rotations: 158
Total time: 0.663814 sec
```

# 5. Pruebas y resultados.

- Métricas de desempeño:

1. **Runtime.** Tiempo de ejecución.

- $t(p)$ .

2. **Cost Factor.** Cantidad de trabajo realizado por el programa.

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

- $C(p) = p * t_p$

3. **Speedup Factor.** Medición relativa del rendimiento de un programa en paralelo.

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

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

4. **Aceleración.** similar al Speedup, pero da el porcentaje de la aceleración con respecto al programa en un solo procesador.

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

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

5. **Efficiency.** Tiempo tarda cada procesador en realizar su tarea.

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

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

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

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