Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Diagonalización de matrices a través del método de Givens con múltiples tarjetas de GPUs

Tesis que presenta

Miguel Tapia Romero

para obtener el Grado de

Maestro en Ciencias en Computación

Director de Tesis

Dr. Amilcar Meneses Viveros

México, Distrito Federal **Noviembre, 2013**

Resumen

La diagonalización de matrices es uno de los problemas de algebra lineal que mayor aplicación tiene en áreas como la física y química cuántica. Especialmente cuando se trata de diagonalizar matrices de alta dimensionalidad. Lo que generalmente se busca en este tipo de problemas es encontrar los valores y vectores propios de una matriz, ya que presentan información sobre cantidades observables que tienen interpretaciones físicas como energía, velocidad o momento, por mencionar algunas.

Existen 2 métodos principales para la diagonalización, usando un método iterativo o usando un método directo. En esta tesis nos enfocaremos en el método iterativo. Este método tiene una alta complejidad respecto al uso de CPU y de memoria, debido a que se debe de calcular la facorización QR. La factorización QR representa un procedimiento pesado para una computadora, ademas se debe realizar muchas veces. Para resolver este problema se han generado diversos algoritmos paralelos.

Estos algoritmos paralelos se ejecutan correctamente con tamaños de matrices que se adapten a las capacidades de la maquina donde se ejecute. Esto ha originado que en matrices de alta dimensionalidad, se presenten problemas como el espacio en memoria, la correctitud del algoritmo y los tiempos de ejecución aumentan. Para matrices de alta dimensionalidad es necesario adaptar estos algoritmos para funcionar en clusters de computadoras. También se pueden ocupar un conjunto de bibliotecas especializadas en explotar los recursos de los clusters como son MKL, ACML o IMSL.

En los últimos años el uso de las tarjetas graficas ha crecido, esto debido a que en comparación con el CPU, estas ofrecen mayor computación por segundo (Flops/s) a un costo energético menor (Flops/watt). Por lo que las GPUs se han empezado a integrar como una infraestructura común en clusters. Sin embargo, no existen muchos algoritmos que utilicen las ventajas de los clusters de GPUs.

En esta tesis se propone implementar un algoritmo paralelo heterogéneo de diagonalización utilizando el método de Givens para la factorización QR. Nuestro algoritmo se implementó utilizando una arquitectura multi-CPU/GPU con el fin de obtener las ventajas energeticas que ofrecen los GPUs. Nuestra implementación utiliza todas las ventajas de un clúster con muchas tarjetas de GPUs y maneja la memoria con el fin

de evitar cuellos de botella.

Abstract

It can be said, in the case of high-dimensional matrices, that the diagonalization is one of the most useful problems within linear algebra due to its applicability into fields as physics and chemistry. Here, the main task is to find the eigenvalues and eigenvectors of a matrix, since they represent measures, which can lead to quantities that have physical interpretations, for instance, energy, speed, time, etcetera.

There exist two different methods to compute the diagonalization of a matrix: (i) the direct method and (ii) the iterative method. In this work we mainly focus in the iterative method. Such a method has a high complexity regarding the use of CPU and memory, since one has to compute the QR-factorization. The QR-factorization represents a heavy procedure for the computer and it has to be done several times. In order to overcome this problem, the parallel algorithms have been developed.

Parallel algorithms works properly to solve matrices whose size is adapted to the computer features where they are executed. Nevertheless, when one wants to manage high dimensional matrices, several problems arise, i.e. the lack of space to store the data, the correctness of the algorithm and the execution time increases. As a possible solution to treat a high dimensional matrix is to adapt these algorithms in order to work on clusters. Another way to exploit the resources of a cluster is to use libraries such as MKL, ACML or ISML.

Nowadays, the use of graphical cards has grown. This growth is due to the fact that GPUs can offer higher computation power per second (Flops/s) with a lower energetic cost (Flops/watts) than CPUs. The GPUs now have been integrated into clusters. However, still missing algorithms that take advantage of these GPUs.

In this work, we propose a heterogeneous parallel diagonalization algorithm using Givens method to perform the QR factorization. Our algorithm is implemented on a multi-CPU/GPU architecture in order to obtain the energetic advantages offered by GPUs. The proposed method takes advantage of having many GPU card and manages the memory in order to avoid bottlenecks.

Agradecimientos

Primero que nada quiero darle gracias a Dios por permitirme estar aquí y vivir todos los momentos que tuve durante mi estancia en esta maestría y mas aun por permitirme terminarla.

A mis padres, Miguel Tapia Ramírez y Concepción Romero Gómez, porque gracias a su cariño, guía y apoyo he llegado a realizar uno de los anhelos más grandes de mi vida, fruto del inmenso apoyo, amor y confianza que en mí se depositó y con los cuales he logrado terminar mis estudios de maestría que constituyen el legado más grande que pudiera recibir y por lo cual les agradeceré eternamente.

A mi hermano, como un testimonio de gratitud por haber significado la inspiración que necesitaba para terminar mi carrera profesional, prometiendo superación y éxitos sin fin, para devolver el apoyo brindado, y la mejor de las ayudas que puede haber.

A mi novia Melina, porque eres de esa clase de personas que todo lo comprendes y dan lo mejor de si mismos sin esperar nada a cambio, porque sabes escuchar y brindar ayuda cuando es necesario y sobre todo por el amor que siempre me ha demostrado. Te amo.

A mi familia, en especial a mi tía Mireya que ha sido como una segunda madre para mi, a mi abuelita Concepción que se que le hubiera gustado en estos momentos conmigo y a todos mis demás familiares porque gracias a su apoyo y consejos he llegado a realizar una de mis más grandes metas, la cual constituye la herencia más valiosa que pudiera recibir.

Al Dr. Amilcar, por su generosidad al brindarme la oportunidad de recurrir a su capacidad y experiencia científica en un marco de confianza, afecto y amistad, fundamentales para la concreción de este trabajo. A los profesores del Departamento por todos sus conocimientos compartidos.

A todo el personal del departamento en especial a Sofí por su apoyo y ayuda mostrado hacia mi persona y hacia mis compañeros, por el tiempo que me dedico y por esas buenas platicas que tuvimos.

Agradezco al CONACYT por el apoyo económico brindado durante mis estudios de maestría y al CINVESTAV por brindarme una agradable estancia y por ser esa

institución de enorme calidad, de la cual me siento orgulloso de haber pertenecido.

Quisiera hacer extensiva mi gratitud a mis compañeros del Departamento de Computación, por esos momentos de apoyo, de unión y esas largas noches en las que compartíamos desvelos y en ocasiones sufrimientos a causa de la maestría; por enseñarme el valor del trabajo en equipo, y por todos aquellos momentos que compartimos juntos.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

Existe un gran numero de problemas matemáticos cuya representación puede llevarse a un problema algebraico de valores propios, esto ocurre a que las soluciones de dichos problemas se llevan a cabo mediante el uso de operadores lineales [1].

Los operadores lineales pueden representarse en forma matricial, por lo que el problema se traduce en encontrar los valores propios de una matriz:

$$Ax = \lambda x.$$

La forma tradicional para obtener los valores propios de una matriz A es encontrar la matriz D diagonal similar:

$$P^{-1}AP = D.$$

Un ejemplo que se aplica a este concepto es el problema de los n cuerpos, el cual puede ser definido como el estudio de los efectos de interacción entre los cuerpos sobre el comportamiento colectivo de los n cuerpos. Este problema se puede resolver utilizando la ecuación de Schrödinger de la siguiente forma:

$$\frac{\delta^2\Psi(x)}{\delta x^2} + V(x)\Psi(X) = \lambda\Psi(x),$$

donde

$$\frac{\delta^2\Psi(x)}{\delta x^2} = \sum_{i=1,j=1,i\neq j}^{N} x_i x_j. \tag{1.1}$$

De la ecuación (1.1) podemos obtener una matriz de la siguiente forma:

$$\begin{pmatrix} 1 & x_1x_2 & x_1x_3 & \cdots & x_1x_n \\ x_1x_2 & 1 & x_2x_3 & \cdots & x_2x_n \\ x_1x_3 & x_2x_3 & 1 & \cdots & x_3x_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_1x_n & x_2x_n & x_3x_n & \cdots & 1 \end{pmatrix}.$$

Para poder resolver este problema es necesario obtener los valores propios, lo que se traduce en realizar la diagonalización de las matrices obtenidas. Otros problemas de como este se encuentran en el trabajo de Tisseur y Meerbergen [1].

1.1. Planteamiento del problema

La diagonalización de matrices es de las tareas que consume más tiempo y recursos, por lo que estas tareas se ejecutan en las supercomputadoras. Lo que hace a la diagonalización una tarea lenta, es una de las operaciones que realiza, la factorización de matrices, que es la operación que mas consume de la diagonalización.

Actualmente todos los algoritmos de diagonalización de matrices, trabajan bien en clusters de computadoras con arquitecturas multi-cores. Sin embargo utilizar este tipo de arquitecturas es costoso, dado que los gastos de equipo y energía son muy altos [2]. Además del espacio que llegan a ocupar es grande.

Se desea aprovechar las ventajas de las Unidades Graficas de Procesamiento (GPU, por sus siglas en inglés), debido a que nos permiten más operaciones que una Unidad Central de Procesamiento (CPU, por sus siglas en inglés) en cuestión de Flops/s y Flops/watt [3, 4], además su costo es menor [2] y por ende su mantenimiento también, así como su consumo energético es reducido.

Cualquier algoritmo que quiera aprovechar la tecnología de los GPU tiene diversas restricciones:

- Se debe programar con un modelo SPMD[1].
- Se debe de evitar la sincronización de hilos.

[1]Del ingles, Single Process Multiple Data, “un proceso, múltiples datos", es una técnica empleada para conseguir paralelismo a nivel de datos.

- La memoria esta delimitada dependiendo de la tarjeta de GPU.

El hecho de que las GPU trabajen con SIMD[2], nos indica que solo podemos aplicar una instrucción a muchos datos, entonces debemos de buscar un método de factorización que utilice este tipo de paralelización, por lo que de los métodos que se estudiaron, se eligió el método de Givens, ya que es el que nos ofrece una paralelización de tipo SPMD.

Evitar la sincronización de hilos es importante en las tarjetas de GPU ya que, al manejar una gran cantidad de hilos, si estos se llegaran a sincronizar en algún punto entonces lo más seguro es que se ocasionaría un cuello de botella, lo que provocaría que tal vez la implementación paralela tarde más que una implementación secuencial.

En cuanto a la memoria, actualmente el máximo con la que cuenta un tarjeta de GPU es de 6 GB. El algoritmo para la diagonalización necesita 3 matrices, A, Q y R, para llevarse a cabo. Como observamos en la tabla 1.1 el tamaño máximo de las matrices para datos de tipo *float* sería de 20,000×20,000, y de tipo *double* el tamaño máximo seria 10,000×10,000. Por lo que si se desea realizar la diagonalización con matrices de mayor tamaño[3].

Para solucionar el problema de memoria, tenemos que implementar un algoritmo escalable para múltiples tarjetas de GPU. Este algoritmo tendría que utilizar programación basada en la Interfaz de Paso de Mensajes (MPI, por sus siglas en inglés) para la comunicación entre CPU y programación a través de la Arquitectura Unificada de Dispositivos de Cómputo (CUDA, por sus siglas en inglés) para utilizar las GPU. En la figura 1.1 podemos observar que un CPU puede tener asociadas dos tarjetas de GPU. La comunicación entre estas dos tarjetas es rápida, dado que están asociadas al mismo CPU, pero si quisiéramos comunicar 2 tarjetas asociadas a 2 CPU diferentes surge un problema, por lo que debemos establecer una política para la eficiente comunicación entre tarjetas. Se propone utilizar MPI para comunicar los 2 CPU y así poder transmitir información entre tarjetas asociadas a 2 CPU diferentes.

[2]Del ingles, Single Insctruction Multiple Data, en español una instrucción, “múltiples datos".

[3]Muchos de los problemas que se realizan en clusters van de matrices de 5,000×5,000 a 100,000×100,000.

Tabla 1.1: Memoria necesaria para almacenar matrices simétricas.

Tamaño de matriz ($n \times n$)	**Memoria necesaria** (***float***)	**Memoria necesaria** (***double***)
5,000×5,000	95.36 MB	190.73 MB
6,000×6,000	137.32 MB	274.65 MB
7,000×7,000	186.92 MB	373.84 MB
8,000×8,000	244.14 MB	488.28 MB
9,000×9,000	308.99 MB	617.98 MB
10,000×10,000	381.46 MB	762.93 MB
20,000×20,000	1.49 GB	2.98 GB
30,000×30,000	3.35 GB	6.70 GB
40,000×40,000	5.96 GB	11.92 GB
50,000×50,000	9.31 GB	18.62 GB
100,000×100,000	37.25 GB	74.50 GB

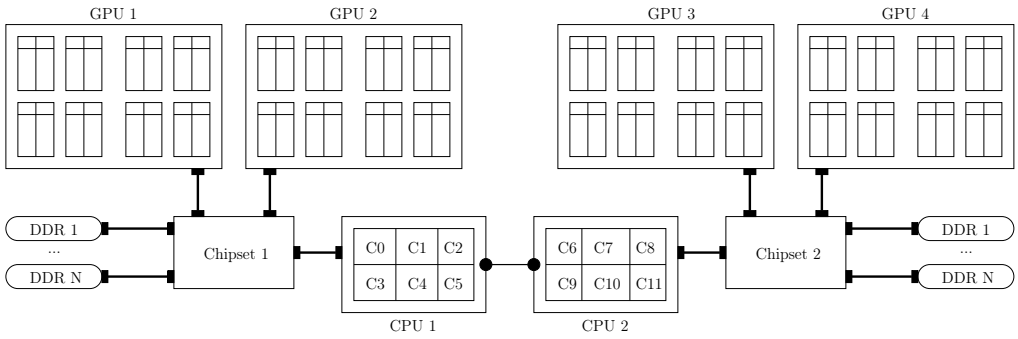

Figura 1.1: Arquitectura multi-CPU/GPU

Pero aún tenemos un problema de particionamiento de datos, sobre todo de la factorización, ya que existen diferentes métodos para hacerse. Algunos de estos métodos no son paralelizables o bien son parcialmente paralelizables, por lo que se buscó el método que ayudara más a la programación paralela. Para esta tesis, se usará el método de Givens, ya que se ha observado que es un método donde se puede realizar un particionamiento de datos, apto para la política seleccionada de comunicación entre tarjetas de GPU.

La selección de una buena estrategia de particionamiento de datos y una correcta política de comunicación entre tarjetas de GPU eficiente, no solo nos permitirá el uso de múltiples tarjetas de GPU, también nos evitará la sincronización entre hilos y entre tarjetas; esto es útil ya que no tendríamos que lidiar con cuellos de botellas.

1.2. Justificación

Se busca generar una implementación de la diagonalización donde la operación más costosa, la factorización, esté implementada de forma masivamente paralela en una arquitectura multi-GPU. Se busca una forma de comunicar las tarjetas de GPU, ya que el costo de manejo de memoria y la comunicación entre estas depende enteramente del programador, porque no existe un estándar para implementar bajo arquitecturas multi-GPU. También se buscan estrategias de particionamiento de datos entre tarjetas de GPU, para evitar demasiados envíos y recepciones de información. Utilizar el método de Givens para la factorización, ya que es un método SPMD. Se desea utilizar el algoritmo QR para la diagonalizacion de matrices, ya que el un metodo directo creemos que tiene pocas ventajas para multiples tarjetas de GPUs.

1.3. Objetivos del trabajo

El objetivo general de esta tesis es implementar la factorización de matrices masivamente paralela para poder ser utilizada en un algoritmo de diagonalización de matrices.

Los objetivos particulares de esta tesis son:

1. Implementar el método de Givens masivamente paralelo.
2. Implementar la diagonalización de matrices para múltiples tarjetas de GPU.
3. Realizar pruebas de estabilidad con los diferentes tipos de matrices y diferentes tipos de datos (*float* y *double*).
4. Comparar el rendimiento de la biblioteca propuesta con otras bibliotecas principalmente con Math Kernel Library (MKL) de Intel.

5. Definir La Interfaz de Programación de Aplicación (API, por sus siglas en ingles) de acuerdo a MKL (Blas, Lapack, Scalapck).

6. Definir estrategias de particionamiento de datos.

7. Definir política de asignación de tarjetas de GPU.

8. Buscar estrategias de organización de los múltiples trabajos.

1.4. Contribuciones y resultados esperados

Los resultados y contribuciones que se esperan obtener de esta tesis son los siguientes:

1. Implementación heterogénea para la diagonalización de matrices, escalable a nivel de tarjetas de GPU.

2. Estrategias de particionamiento de memoria entre tarjetas de GPU.

3. Factorización de Givens masivamente paralela.

4. Método paralelo para determinar cuándo una matriz es o no diagonal.

5. Evaluaciones de factibilidad para el tamaño máximo de matrices.

6. Política de asignación de tarjetas de GPU para el método de Givens.

7. Resultados del manejo de memoria entre multiples tarjetas.

1.5. Organización de la tesis

La presente tesis se organiza en seis capítulos con la siguiente estructura:

- **Capítulo 2:** se describe los algoritmos principales para la factorización QR y el algoritmo QR para la diagonalización de matrices.

- **Capítulo 3:** se explica los algoritmos del Capítulo 2 en una forma paralela y se mencionan algunas estrategias de particionamiento.

- **Capítulo 4:** se detalla la tecnología de los clusters basados en GPU y lo que son los algoritmos heterogéneos.
- **Capítulo 5:** se presentan las etapas del diseño y la implementación para llevar a cabo la implementación del proyecto propuesto en esta tesis.
- **Capítulo 6:** se muestran las pruebas, la descripción de estas y la infraestructura que se utilizo al realizar las pruebas.
- **Capítulo 7:** se dan a conocer los resultados alcanzados, las conclusiones que obtuvieron de la presente tesis, así como trabajos que pueden darse en un futuro.

Capítulo 2

Marco Teórico

En el presente capítulo se exponen a fondo los algoritmos de factorización más comunes para obtener las matrices Q y R, así como el algoritmo que se utiliza para la diagonalización de matrices. En la sección 2.1 se describe el algoritmo QR para la diagonalización de matrices y en la sección 2.2 se explican los diferentes algoritmos para factorizar matrices.

2.1. Algoritmo QR

En álgebra lineal una matriz cuadrada $A \in \mathbb{R}^{n\times n}$ se dice que es diagonalizable si es semejante a una matriz diagonal, es decir, si mediante un cambio de base puede reducirse a una forma diagonal. En este caso, la matriz podrá descomponerse de la forma $A = PDP^{-1}$ donde P es una matriz invertible, cuyos vectores columna son vectores propios de A y D es una matriz diagonal formada por los valores propios de A.

Si la matriz P es ortogonal se dice entonces que la matriz A es diagonalizable ortogonalmente, pudiendo escribirse como $A = PDP^T$. El teorema espectral garantiza que cualquier matriz cuadrada simétrica con coeficientes reales es ortogonalmente diagonalizable. En este caso, P está formada por una base ortonormal de vectores propios de la matriz siendo los valores propios reales. La matriz P es por tanto ortogonal y los vectores filas de P^{-1} son los vectores columnas de P [5].

El algoritmo QR[1] es usado para calcular el conjunto completo de valores y vectores propios de una matriz simétrica, dado que al final del algoritmo obtenemos una matriz diagonal D y una matriz P ortogonal, donde los valores de la matriz D son los valores propios y P contiene a los vectores propios. Considere la matriz A a la cual se quiere calcular los valores y vectores propios. El algoritmo básico QR empieza con $A_0 = A$ y se genera una secuencia de matrices (A_j) por la ecuación (2.1).

$$A_{m-1} = Q_m R_m \qquad R_m Q_m = A_m. \tag{2.1}$$

Esto es, A_{m-1} se factoriza en Q_m y R_m, donde Q_m es ortogonal y R_m es una triangular superior con valores positivos en su diagonal principal. Esas matrices son únicas. Las matrices son multiplicadas en el orden inverso para obtener A_m. Se puede observar que $A_m = Q_m^{-1} A_m Q_m$ por las transformaciones de similitud. Se puede ver claramente en el algoritmo 1 los pasos para obtener la matriz diagonal D, donde uno de los pasos esenciales del algoritmo se encuentra en la línea 4, que es la factorización.

Algoritmo 1 Algoritmo QR

Entrada: Matriz Cuadrada Simétrica A.

Salida: Matriz Diagonal D, Matriz Ortogonal P.

1: $D \leftarrow A$
2: $P \leftarrow 1$
3: **mientras** D no es diagonal **hacer**
4: $\quad D \rightarrow QR$
5: $\quad P \leftarrow PQ$
6: $\quad D \leftarrow RQ$
7: **termina mientras**

El algoritmo QR puede cambiar dependiendo del método de factorización que se seleccione. A continuación se muestran los métodos de factorización mas comunes.

[1]La factorización QR y el algoritmo QR son 2 cosas diferentes. El algoritmo QR, es un procedimiento iterativo para encontrar los valores y vectores propios y esta basado en la factorización QR, la cual es un procedimiento directamente relacionado al proceso Gram-Schmidt [6].

2.2. Algoritmos de factorización de matrices

En esta sección hablaremos de los métodos más comunes conocidos para factorizar una matriz, explicando más a fondo el proceso de cada uno de ellos. En la sección 2.2.1 se explica el método de Givens, el cual es el que se utilizó en esta tesis como método de factorización de matrices, en la sección 2.2.2 se expone el método de Gram-Schmidt que es el más utilizado para la factorización QR, en las secciones 2.2.3 y 2.2.4 se comentan las factorizaciones LU y Cholesky respectivamente y en la sección 2.2.5 se ilustra el método de Householder utilizado para la factorización QR.

2.2.1. Método de Givens

Las reflexiones de Householder son útiles para introducir ceros a gran escala, por ejemplo, la aniquilación de todos excepto el primer componente de un vector. Sin embargo, en cálculos donde es necesario los elementos cero más selectivamente, las rotaciones de Givens son la transformación a elegir. Estas correcciones tienen la forma que se puede observar en la ecuación (2.2).

$$G(i,k,\theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & -s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix} \\ \begin{matrix} \\ \\ i \\ \\ k \\ \\ \\ \end{matrix} \quad , \tag{2.2}$$

donde $c = \cos(\theta)$ y $s = \text{sen}(\theta)$ para algún θ. Rotaciones de Givens son claramente ortogonales.

La pre multiplicación por $G(i,k,\theta)^T$, equivale a una rotación en sentido contrario del reloj de θ radianes en la coordenada (i,k). En efecto, si $x \in \mathbb{R}^n$ y $y = G(i,k,\theta)^T x$, entonces

$$y_j = \begin{cases} cx_i - sx_k & j = i \\ sx_i + cx_k & j = k \\ x_j & j \neq i, k \end{cases} . \tag{2.3}$$

De la ecuación (2.3) es claro que podemos forzar y_k a ser cero ajustando los valores de c y s como se muestra en la ecuación (2.4).

$$c = \frac{x_i}{\sqrt{x_i^2 + x_k^2}} \qquad s = \frac{-x_k}{\sqrt{x_i^2 + x_k^2}} . \tag{2.4}$$

Así, es una cuestión simple volver cero una entrada específica en un vector usando la rotación de Givens. El algoritmo 2 muestra como calcular los valores de c y s.

Algoritmo 2 Givens

Entrada: Valores escalares a y b.

Salida: Valores c y s.

1: **si** $b = 0$ **entonces**
2: $c = 1$; $s = 0$.
3: **si no**
4: **si** $|b| > |a|$ **entonces**
5: $r = -a/b$; $s = 1/\sqrt{1 + r^2}$; $c = sr$.
6: **si no**
7: $r = -b/a$; $c = 1/\sqrt{1 + r^2}$; $s = cr$.
8: **termina si**
9: **termina si**

Las rotaciones de Givens se pueden usar para calcular la factorización QR, el siguiente caso de una matriz de 4×3 ilustra la idea general:

$$\begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(1,2)}$$

$$\begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \end{bmatrix} \xrightarrow{(3,4)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{(2,3)} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \\ 0 & 0 & \times \end{bmatrix} \xrightarrow{(3,4)} R.$$

Se puede ver cuales son los 2 vectores que definen las rotaciones de Givens. Claramente, si G_j denota la j-ésima rotación de Givens en la reducción, entonces $Q^T A = R$ es una triangular superior donde $Q = G_1 \cdots G_t$ y t es el número total de rotaciones. De manera general para un m y un n tenemos el algoritmo 3, donde la función **givens** se resuelve con el algoritmo 2 [7].

Algoritmo 3 Givens QR

Entrada: Matriz $A \in \mathbb{R}^{m \times n}$ con $m \geq n$.

Salida: Matriz A con $Q^T A = R$, donde R es triangular superior y Q es ortogonal.

1: **para** $j = 1 : n$ **hacer**
2: **para** $i = m : -1 : j + 1$ **hacer**
3: $[c, s] =$**givens**$(A(i-1, j), A(i, j))$
4: $A(i-1 : i, j : n) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T A(i-1 : i, j : n)$
5: **termina para**
6: **termina para**

El algoritmo QR utilizando el método de Givens, no cambia en esencia ya que podemos obtener la matriz Q como vimos anteriormente.

Algoritmo 4 Algoritmo QR con el método de Givens

Entrada: Matriz Cuadrada Simétrica A.

Salida: Matriz Diagonal D, Matriz Ortogonal P.

1: $D \leftarrow A$
2: $P \leftarrow 1$
3: **mientras** D no es diagonal **hacer**
4: **givens**$(D) \rightarrow QR$
5: $P \leftarrow PQ$
6: $D \leftarrow RQ$
7: **termina mientras**

2.2.2. Factorización QR y Proceso de Gram-Schmidt

Para cada matriz invertible $A \in \mathbb{R}^{n\times n}$, existe una única matriz ortogonal Q y una única matriz triangular superior R con valores positivos en la diagonal, tal que:

$$A = QR.$$

Tradicionalmente la factorización QR se resuelve con el método de Gram-Schmidt, pero existen muchos más métodos para resolverla.

La factorización QR es el camino completo del método de Gram-Schmidt, dado que las columnas de $Q = (q_1|q_2|\dots|q_n)$ son el resultado de aplicar el proceso de Gram-Schmidt a las columnas de $A = (a_1|a_2|\dots|a_n)$ y R esta dada por:

$$R = \begin{pmatrix} v_1 & q_1^T a_2 & q_1^T a_3 & \cdots & q_1^T a_n \\ 0 & v_2 & q_2^T a_3 & \cdots & q_2^T a_n \\ 0 & 0 & v_3 & \cdots & q_3^T a_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & v_n \end{pmatrix},$$

donde $v_1 = \parallel a_1 \parallel$ y $v_k = \parallel a_k - \sum_{i=1}^{k-1} \langle q_i|a_k\rangle q_i \parallel$ para $k > 1$.

Si $\mathcal{B} = x_1, x_2, \dots, x_n$ es una base para un producto punto general del espacio $\mathcal{S}$, entonces el método de Gram-Schmidt ésta definido por las ecuaciones (2.5) y (2.6).

$$u_1 = \frac{x_1}{\parallel x_1 \parallel}\ . \tag{2.5}$$

$$u_k = \frac{x_k - \sum_{i=1}^{k-1} \langle u_i|x_k\rangle u_i}{\parallel x_k - \sum_{i=1}^{k-1} \langle u_i|x_k\rangle u_i \parallel} \text{ para } k = 2, \dots, n. \tag{2.6}$$

El algoritmo 5 es la clásica implementación del procedimiento Gram-Schmidt [8].

Algoritmo 5 Gram-Schmidt

1: **para** $k = 1$ **hacer**
2: $\quad u_1 \leftarrow \dfrac{x_1}{\| x_1 \|}$
3: **termina para**
4: **para** $k > 1$ **hacer**
5: $\quad u_k \leftarrow x_k - \displaystyle\sum_{i=1}^{k-1} (u_i^* x_k) u_i$
6: $\quad u_k \leftarrow \dfrac{u_k}{\| u_k \|}$
7: **termina para**

2.2.3. Factorización LU

Supongamos que podemos escribir la matriz A como el producto de dos matrices,

$$L \cdot U = A, \tag{2.7}$$

donde L es una matriz triangular inferior y U es una matriz triangular superior. Para el caso de una matriz A de 4×4, por ejemplo, la ecuación (2.7) se vería de la forma (2.8).

$$\begin{bmatrix} l_{00} & 0 & 0 & 0 \\ l_{10} & l_{11} & 0 & 0 \\ l_{20} & l_{21} & l_{22} & 0 \\ l_{30} & l_{31} & l_{32} & l_{33} \end{bmatrix} \cdot \begin{bmatrix} u_{00} & u_{01} & u_{02} & u_{03} \\ 0 & u_{11} & u_{12} & u_{13} \\ 0 & 0 & u_{22} & u_{23} \\ 0 & 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix}. \tag{2.8}$$

Pero ¿Cómo podemos obtener L y U dado A?. Primero obtenemos el i, j-ésimo componente de las ecuaciones (2.7) y (2.8). Ese valor siempre es una suma que comienza con:

$$l_{i0} u_{0j} + \cdots = a_{ij}.$$

El número de términos en la suma depende, sin embargo, de cual de los dos números es el menor si i o j. De hecho tenemos tres casos,

$$i < j : l_{i0} u_{0j} + l_{i1} u_{1j} + \cdots + l_{ii} u_{ij} = a_{ij}, \tag{2.9}$$

$$i = j : l_{i0} u_{0j} + l_{i1} u_{1j} + \cdots + l_{ii} u_{jj} = a_{ij}, \tag{2.10}$$

$$i > j : l_{i0}u_{0j} + l_{i1}u_{1j} + \cdots + l_{ij}u_{jj} = a_{ij}. \tag{2.11}$$

Las ecuaciones (2.9) - (2.11), son n^2 ecuaciones, para las $n^2 + n$ l's y u's desconocidas. Ya que el número de variables desconocidas es mayor al número de ecuaciones, esto nos invita a especificar n de las variables desconocidas arbitrariamente y después intentar resolver las otras. De hecho, es posible tomar:

$$l_{ii} \equiv 1 \qquad i = 0, \ldots, n-1. \tag{2.12}$$

Un procedimiento que utilizaremos es el algoritmo Crout's, el cual trivialmente resuelve el conjunto de $n^2 + n$ ecuaciones (2.9) - (2.12) para todas las l's y u's solo acomodando las ecuaciones en cierto orden, el orden es:

- Fijar $l_{ii} = 1$, $i = 0, \ldots, n-1$ (ecuación (2.12)).
- Para cada $j = 0, 1, 2, \ldots, n-1$ hacer dos procedimientos: Primero, para $i = 0, 1, \ldots, j$, usar (2.9), (2.10), y (2.12) para resolver u_{ij}, haciendo

$$u_{ij} = a_{kj} - \sum_{k=0}^{i-1} l_{ik}u_{kj} \quad . \tag{2.13}$$

Cuando $i = 0$ en (2.13) el termino de la suma es cero. Segundo, para $i = j+1, j+2, \ldots, n-1$ usar (2.11) para resolver l_{ij}, realizando:

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=0}^{j-1} l_{ik}u_{kj} \right). \tag{2.14}$$

Asegurarse de realizar ambos procesos antes de seguir con el siguiente j [9].

Realizar el algoritmo QR con la factorización LU no nos proporciona los vectores propios ya que ni L ni U son ortogonales.

2.2.4. Factorización de Cholesky

Sea A una matriz definida positiva[2]. Entonces A se puede factorizar exactamente en una forma en el producto:

$$A = RR^T, \tag{2.15}$$

[2]Si una matriz $A \in \mathbb{R}^{n \times n}$, es real y simétrica y también satisface la propiedad: $x^T Ax > 0$ para todo $x \in \mathbb{R}^n$ distinto de cero, entonces A se dice que es definida positiva.

Algoritmo 6 Algoritmo QR con la factorización LU

Entrada: Matriz Cuadrada Simétrica A.

Salida: Matriz Diagonal D.

1: $D \leftarrow A$

2: **mientras** D no es diagonal **hacer**

3: $\quad D \rightarrow LU$

4: $\quad D \leftarrow UL$

5: **termina mientras**

donde R es triangular inferior y todas las entradas de su diagonal principal son positivas. R es llamada el factor de Cholesky.

De la ecuación (2.15) podemos escribir la factorización en detalle y estudiarla:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} r_{11} & 0 & 0 & \cdots & 0 \\ r_{21} & r_{22} & 0 & \cdots & 0 \\ r_{31} & r_{32} & r_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & r_{n3} & \cdots & r_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{21} & r_{31} & \cdots & r_{n1} \\ 0 & r_{22} & r_{32} & \cdots & r_{n2} \\ 0 & 0 & r_{33} & \cdots & r_{n3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & r_{nn} \end{bmatrix}. \tag{2.16}$$

Ahora veremos como calcular la j-ésima columna de R, asumiendo que ya calculamos las primeras $j-1$ columnas. Ya que solo los primeros j valores en la j-ésima columna de R^T son distintos de cero,

$$a_{ij} = r_{i1}r_{j1} + r_{i2}r_{j2} + \cdots + r_{i,j-1}r_{j,j-1} + r_{ij}r_{jj}. \tag{2.17}$$

Todas las entradas de R que aparecen en (2.17) están en las primeras j columnas de R. Teniendo en cuenta que las primeras $j-1$ columnas se conoces, podemos observar que los únicos valores en (2.17) que no se conocen son r_{ij} y rjj. Tomando $i = j$ en (2.17), tenemos:

$$a_{jj} = r_{j1}^2 + r_{j2}^2 + \cdots + r_{j,j-1}^2 + r_{jj}^2 \quad ,$$

que se puede resolver por r_{jj}:

$$r_{jj} = +\sqrt{a_{jj} - \sum_{k=1}^{j-1} r_{jk}^2} \quad . \tag{2.18}$$

Nótese que la raíz cuadrada positiva es la correcta. Ahora que tenemos r_{jj}, podemos usar (2.17) para resolver r_{ij}:

$$r_{ij} = \frac{1}{r_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} r_{ik} r_{jk} \right) \qquad i = j, \ldots, n. \tag{2.19}$$

No debemos calcular r_{ij} para $i < j$ porque estas entradas son cero.

Las ecuaciones (2.18) y (2.19) nos indican como calcular R. El algoritmo que se acaba de desarrollar es llamado el método de Cholesky [6].

Similar a la factorización LU, si deseamos utilizar la factorización de Cholesky solo podremos obtener los valores propios.

Algoritmo 7 Algoritmo QR con la factorización Cholesky

Entrada: Matriz Cuadrada Simétrica A.

Salida: Matriz Diagonal D.

1: $D \leftarrow A$
2: **mientras** D no es diagonal **hacer**
3: $\quad D \rightarrow RR^T$
4: $\quad D \leftarrow R^T R$
5: **termina mientras**

2.2.5. Método de Householder

Sea $v \in \mathbb{R}^n$ diferente de cero. Y una matriz P de $n \times n$ de la siguiente forma:

$$P = I - \frac{2}{v^T v} v v^T$$

esto se llama *reflexión de Householder*. El vector v se llama *vector Householder*. Si un vector x es multiplicado por P, entonces este se proyecta en el hiperplano $\text{span}\{v\}^\perp$.

En particular, supongamos que tenemos $x \in \mathbb{R}^n$ y $x \neq 0$ y queremos Px sea un múltiplo de e_1, donde e_1 pertence a la base canónica de $\mathbb{R}^n$, $Px = \gamma x$. Notese que:

$$Px = \left(I - \frac{2vv^T}{v^T v} \right) x = x - \frac{2v^T x}{v^T v} v$$

y $Px \in \text{span}\{e_1\}$ implica que $v \in \text{span}\{x, e_1\}$. Fijando $v = x + \alpha e_1$ nos da:

$$v^T x = x^T x + \alpha x_1$$

y

$$v^T v = x^T x + 2\alpha x_1 + \alpha^2,$$

por consiguiente

$$Px = \left(1 - 2\frac{x^T x + \alpha x_1}{x^T x + 2\alpha x_1 + \alpha^2}\right) x - 2\alpha \frac{v^T x}{v^T v} e_1.$$

A fín de que el coeficiente de x sea cero, fijamos $\alpha = \pm \parallel x \parallel_2$ para luego

$$v = x \pm \parallel x \parallel_2 e_1 \Rightarrow Px = \left(I - \frac{2vv^T}{v^T v}\right) x = \mp \parallel x \parallel_2 e_1. \tag{2.20}$$

Es esta simple determinación de v que hace que la reflexión de Householder sea tan útil. Existen diversos detalles importantes asociados con la determinación de la matriz de Householder, es decir, la determinación del vector de Householder. Uno concierne en elegir el signo de v en (2.20). Fijando

$$v_1 = x_1 - \parallel x \parallel_2$$

tiene la propiedad que Px es un múltiplo positivo de e_1. Pero eso es peligroso si x es un múltiplo positivo cercano de e_1, ya que una cancelación podría ocurrir. Sin embargo, la formula

$$v_1 = x_1 - \parallel x \parallel_2 = \frac{x_1^2 - \parallel x \parallel_2^2}{x_1 + \parallel x \parallel_2} = \frac{-(x_2^2 + ... + x_n^2)}{x_1 + \parallel x \parallel_2}$$

no sufre de este defecto en el caso donde $x_1 > 0$. En la practica, es de ayuda normalizar el vector de Householder tal que $v(1) = 1$. Esto permite el almacenamiento de $v(2:n)$ donde los ceros se han introducido en x, es decir, $x(2:n)$. Nos referimos a $v(2:n)$ como la parte esencial del vector de Householder. Tomando en cuenta que $\beta = 2/v^T v$ y dejando que **length**(x) sea la dimensión del vector, podemos obtener el algoritmo 8.

Algoritmo 8 Cálculo del vector de Householder

Entrada: $x \in \mathbb{R}^n$

Salida: $v \in \mathbb{R}^n$ y $\beta \in \mathbb{R}$

$n = \textbf{lenght}(x)$
$\sigma = x(2:n)^T x(2:n)$
$v = \begin{bmatrix} 1 \\ x(2:n) \end{bmatrix}$
si $\sigma = 0$ **entonces**
$\beta = 0$
si no
$\mu = \sqrt{x(1)^2 + \sigma}$
si $x(1) \leq 0$ **entonces**
$v(1) = x(1) - \mu$
si no
$v(1) = -\sigma/(x(1) + \mu)$
termina si
$\beta = 2v(1)^2/(\sigma + v(1)^2)$
$v = v/v(1)$
termina si

Ahora utilizaremos las transformaciones de Householder para resolver la factorización QR. La esencia de este algoritmo se puede cubrir con un pequeño ejemplo: Supongamos que tenemos una matriz de 6 filas y 5 columnas, y asumamos que las matrices de Householder H_1 y H_2 ya se calcularon entonces:

$$H_2 H_1 A = \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \\ 0 & 0 & \boxtimes & \times & \times \end{bmatrix}.$$

Concentrándonos en la entradas remarcadas, determinamos una matriz de Househol-

der $\tilde{H}_3 \in \mathbb{R}^{4\times 4}$ tal que:

$$\tilde{H}_3 \begin{bmatrix} \boxtimes \\ \boxtimes \\ \boxtimes \\ \boxtimes \end{bmatrix} = \begin{bmatrix} \times \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Si $H_3 = \text{diag}(I_2, \tilde{H}_3)$ entonces:

$$H_3 H_2 H_1 A = \begin{bmatrix} \times & \times & \times & \times & \times \\ 0 & \times & \times & \times & \times \\ 0 & 0 & \times & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times & \times \end{bmatrix}$$

Despues de n pasos obtenemos una matriz triangular superior $H_n H_{n-1} \cdots H_1 A = R$ y fijando $Q = H_1 \cdots H_n$ obtenemos $A = QR$. El algoritmo 9 nos desglosa el procedimiento para utilizar las transformaciones de Householder para calcular la factorización QR.

Algoritmo 9 Householder QR

Entrada: $A \in \mathbb{R}^{m\times n}$

Salida: Encontrar las matrices de Householder $Q = H_1 \cdots H_n$ donde A se sobrescribe como nos muestra la ecuación (2.21)

para $j = 1 : n$ **hacer**
  $[v, \beta] =$ **house**$(A(j : m, j))$
  $A(j : m, j : n) = (I_{m-j+1} - \beta v v^T) A(j : m, j : n)$
  si $j < m$ **entonces**
    $A(j + 1 : m, j) = v(2 : m - j + 1)$
  termina si
termina para

donde **house** se resuelve utilizando el algoritmo 8. Del algoritmo 9 podemos obtener:

$$A = \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ v_2^{(1)} & r_{22} & r_{23} & r_{24} & r_{25} \\ v_3^{(1)} & v_3^{(2)} & r_{33} & r_{34} & r_{35} \\ v_4^{(1)} & v_4^{(2)} & v_4^{(3)} & r_{44} & r_{45} \\ v_5^{(1)} & v_5^{(1)} & v_5^{(3)} & v_5^{(4)} & r_{55} \\ v_6^{(1)} & v_6^{(1)} & v_6^{(3)} & v_6^{(4)} & v_6^{(5)} \end{bmatrix}, \tag{2.21}$$

donde los valores r son los valores de la matriz triangular superior R y los valores v son los vectores de Householder que se calcularon [7].

Así como el método de Givens, el método de Householder no cambia el algoritmo QR, dado que las dos matrices Q y R se pueden obtener, por lo que se puede calcular tanto valores propios como vectores propios.

Algoritmo 10 Algoritmo QR con el método de Householder

Entrada: Matriz Cuadrada Simétrica A.

Salida: Matriz Diagonal D, Matriz Ortogonal P.

1: $D \leftarrow A$
2: $P \leftarrow 1$
3: **mientras** D no es diagonal **hacer**
4: **householder**$(D) \rightarrow QR$
5: $P \leftarrow PQ$
6: $D \leftarrow RQ$
7: **termina mientras**

2.2.6. Análisis de los algoritmos de factorización

Después de haber estudiado todos los algoritmos de factorización debemos elegir el mejor algoritmo para lo que se desea realizar en esta tesis, por lo que buscamos el algoritmo que tenga menos dependencias de un paso anterior necesita, que se pueda aplicar a un modelo SPMD y que se puede utilizar correctamente en el algoritmo QR para obtener tanto valores como vectores propios.

De los algoritmos expuestos, el método de Gram-Schmidt lo descartamos dado que calcular la u_i depende del calculo de las u anteriores.

La factorización LU y Cholesky las descartamos, porque como vimos al utilizarlas en el algoritmo QR no podemos calcular los vectores propios, además de que la factorización LU cada calculo de l y u son completamente dependientes de algún calculo anterior, aunque si es posible paralelizar como observaremos en el próximo capitulo. También la factorización Cholesky solo funciona para un tipo de matriz en especifico, por lo que buscamos un método que trabaje con cualquier tipo de matriz.

Por lo que nos quedan 2 métodos, Householder y Givens, los 2 métodos nos ofrecen la posibilidad de utilizar el algoritmo QR completamente. Householder es un método que se puede aplicar con un modelo SPMD, pero en cuestión de memoria su manejo es complicado para poderlo utilizar en las GPU.

En conclusión, el método que observamos se adapta mejor a la programación en GPU es el método de Givens, ya que es un método que podemos adaptar al modelo SPMD, el manejo de memoria en la GPU se puede hacer con algunas modificaciones y nos permite trabajar el algoritmo QR para obtener valores y vectores propios.

Capítulo 3

Estrategias de Particionamiento de Datos y Paralelización

En este capítulo se explicaran algunas formas de paralelización de algunos métodos vistos en el capítulo 2, así como una estrategia de particionamiento de datos. En la sección 3.1 se expone más a fondo el por qué y cómo se debe de llevar a cabo una partición de datos dependiendo de la arquitectura de memoria con la que se cuente. En las secciones 3.2 se muestran algunas de las formas de paralelizar el método de Givens, la factorización LU y la factorización de Cholesky.

3.1. Estrategia de particionamiento de Datos

Se puede comenzar con un análisis para determinar si una estrategia de partición de datos es adecuada si el problema presenta las siguientes características:

- si gran parte del cómputo intensivo está centrado a lo largo del procesamiento de una estructura de datos muy grande. Como bien podría ser una matriz,
- si operaciones similares se están aplicando a diferentes partes de la estructura, de tal forma que podrían llevarse a cabo casi de manera independiente.

Bajo un esquema de memoria compartida la partición de datos puede hacerse a través de la partición de tareas. Sin embargo, cuando la memoria está distribuida debe hacerse distribución no automatizada, dado que debe orientarse el paso de mensajes.

Cuando se maneja esta estrategia de paralelización, no se deben analizar demasiado las tareas, pero si las estructuras principales de que definen la tarea, a fin de identificar en qué casos pueden ser particionadas.

Cuando se vaya a dividir una estructura, debe tomarse en cuenta la granularidad del problema, a fin de que la eficiencia sea adecuada, es decir, que la carga de trabajo por tarea sea lo suficientemente rentable para compensar el *overhead*.

En cuanto a la flexibilidad, debe asegurarse de que el problema puede dividirse lo suficiente para abarcar un amplio número de computadoras.

Existen diferentes técnicas para manejar datos para sistemas multihilos en los GPU, estas técnicas se basan: en que los datos son entregados eficientemente por la DRAM, en usar el almacenamiento en chip con el fin de aprovechar los accesos a la DRAM y en reducir el uso del ancho de banda en los algoritmos [10]:

La transformación del diseño de datos A veces direccionar estructuras de datos multidimensionales no es fácil y el hecho de que un grupo de hilos quieran acceder en paralelo a un campo en común de múltiples elementos, provoca pasos largos en el acceso a los datos. Por lo que se recomienda pasar de arreglos de estructuras de datos a estructura de arreglos de datos.

Transformación de dispersión a reunión En muchas aplicaciones los datos obtenidos son una combinación de contribuciones de cada uno de los elementos de entrada. Una salida utilizando dispersión trabaja pobremente mientras el paralelismo crece, debido a que los acceso a la salida son continuos. Por lo que es importante invertirlo a una salida de tipo reunión lo que resulta en lecturas al mismo tiempo que es algo que el hardware maneja mejor.

Tiling *Tiling* es quizá la técnica más aplicada y entendida para el uso de una jerarquía de memoria. El concepto es utilizar conjuntos de datos más pequeños para que quepan en el almacenamiento, más rápido, del *on-chip*, mientras que el sistema los procesa. Aunque la técnica es fundamentalmente la misma en código secuencial, aunque aplicada a las iteraciones en lugar de hilos, una implementación paralela de esta técnica requiere una atención especial.

Privatización Es la transformación que toma algunos datos que fueron alguna vez común o compartido entre tareas paralelas y los duplica para que así hilos paralelos puedan tener una copia privada con la cual operar. Los hilos individualmente actualizan los resultados privados, y luego los combinan en un grupo de resultados colectivo que son finalmente combinados con todos los grupos en los resultados globales.

Estructura de datos de Agrupación Realizar una operación de reunión es difícil sin método para determinar que entradas contribuyen a la localización. Sin embargo, es benéfico primero crear un mapa de las localidades de salida a un subconjunto de los datos de entrada que tal vez afecten las localidades de salida, reduciendo las lecturas redundante de los datos. A esta creación de estructura de datos se le llama agrupación.

Compactación Es una técnica que coordina las tareas paralelas para que dinámicamente determinen las localidades de salida sin introducir ningún hoyo. La coordinación dinámica acarrea algo de *overhead*, pero las mejoras en el uso de memoria compensan el *overhead*. Un desarrollador puede implementar la compactación como un kernel separado o, preferentemente, lo puede integrar directamente en el kernel producido.

3.1.1. Análisis de técnicas del manejo de datos

De las 6 técnicas antes mencionadas, solo se ocuparan dos, ya que el problema no se adapta a las demás, no hay una transformación de datos, porque no se considera utilizar una estructura. La técnica de dispersión a reunión no aplica para esta tesis ya que toda la matriz será copiada a la memoria de la GPU. La agrupación y la compactación no son usadas ya que no creamos o determinamos las salidas.

Las técnicas que se usaron en esta tesis son: la de privatización dado que cada hilo trabaja con sus datos privados y luego realiza cálculos y al final guarda el resultado en una localidad de memoria y así todos los hilos hasta que terminen el trabajo; la otra técnica utilizada es la de *tiling*, ya que en la implementación multi-GPU para dividir la matriz para poder utilizar mas de una GPU.

3.2. Algoritmos paralelos de factorización de matrices

En esta sección se describen algunas formas de paralelización de algunos algoritmos mencionados en la sección anterior. En la sección 3.2.1 exponemos algunas formas de paralelizar el método de Givens. En las secciones 3.2.2 y 3.2.3, se describen los grafos de dependencia de las factorizaciones LU y Cholesky, así como los pasos a seguir para que se realicen de forma paralela

3.2.1. Paralelización del método de Givens

Como se observó en la sección 2.2.1 las rotaciones de Givens solo afectan 2 filas de una matriz, de modo que se puede aprovechar esta característica para poder realizar rotaciones en paralelo.

Por lo que en varios artículos se han propuesto diferentes formas de paralelizar el método de Givens para realizar la factorización QR [11, 12].

En [11] se muestran:

- El esquema Sameh y Kuck que calcula arriba de n rotaciones de Givens simultáneamente mejor descrito en [13]:

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 5 & \times & \times & \times & \times & \times \\ 4 & 6 & \times & \times & \times & \times \\ 3 & 5 & 7 & \times & \times & \times \\ 2 & 4 & 6 & 8 & \times & \times \\ 1 & 3 & 5 & 7 & 9 & \times \end{pmatrix}$$

Figura 3.1: Esquema Sameh and Kuck.

- Una modificación de Sahem y Kuck:

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 9 & \times & \times & \times & \times & \times \\ 7 & 11 & \times & \times & \times & \times \\ 5 & 9 & 13 & \times & \times & \times \\ 3 & 7 & 11 & 15 & \times & \times \\ 1 & 5 & 9 & 13 & 17 & \times \end{pmatrix}$$

Figura 3.2: Esquema Modificado de Sameh and Kuck.

- Dos secuencias de Givens *Pipeline*, la primera (PGS-1) requiere aproximadamente $n^2/2$ procesadores. Y la segunda (PGS-2) requiere n procesadores:

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 5 & \times & \times & \times & \times & \times \\ 4 & 7 & \times & \times & \times & \times \\ 3 & 6 & 9 & \times & \times & \times \\ 2 & 5 & 8 & 11 & \times & \times \\ 1 & 4 & 7 & 10 & 13 & \times \end{pmatrix}$$

Figura 3.3: PGS-1.

$$\begin{pmatrix} \times & \times & \times & \times & \times & \times \\ 6 & \times & \times & \times & \times & \times \\ 4 & 11 & \times & \times & \times & \times \\ 3 & 6 & 10 & \times & \times & \times \\ 2 & 5 & 9 & 14 & \times & \times \\ 1 & 3 & 6 & 10 & 15 & \times \end{pmatrix}$$

Figura 3.4: PGS-2.

Y así como estas formas de paralelización se encuentran mas en la literatura. Pero no hay un estándar establecido para utilizar el método de Givens de forma paralela para calcular la factorización QR, por lo que cada programador lo paraleliza de la forma en que mas se adapte a la infraestructura y lenguaje se vaya a usar.

3.2.2. Paralelización de la factorización LU

Uno de los beneficios de la factorización LU es que la matriz resultante se puede almacenar en la misma matriz de entrada, esto con el fin de evitar desperdiciar memoria.

En la sección 2.2.3 se muestra que existen 2 ecuaciones necesarias para llevar a cabo esta factorización. Al realizar un análisis un poco más detallado, se puede desarrollar el grafo de dependencias de datos, tal como muestra la figura 3.5.

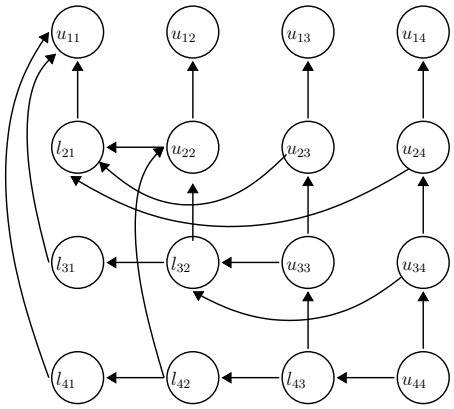

Figura 3.5: Grafo de dependencias de la factorización LU

El grafo nos indica que un elemento de la matriz U requiere necesariamente el cálculo previo del elemento superior a este y los elementos de L que se encuentran a la izquierda de este.

También se aprecia que un elemento de L requiere del cálculo de los elementos de L a la izquierda de este y del elemento superior de U a este.

Existen varias secuencias de pasos que permiten calcular varios elementos en paralelo y además cumplen con las dependencias mostradas en la figura 3.5.

Una de estas secuencias se puede apreciar en la figura 3.6, que consiste en realizar operaciones en elementos por diagonales.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 5 & 6 & 7 & 8 \\ 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 6 & 7 & 8 & 9 & 10 \\ 6 & 7 & 8 & 9 & 10 & 11 \end{pmatrix}$$

Figura 3.6: Paralelización de la factorización LU

Se puede notar que se puede ocupar un hilo por fila o por columna como se desee. Para paralelizar este proceso se requieren n hilos y $2n-1$ pasos para una matriz $A \in \mathbb{R}^{n \times n}$.

3.2.3. Paralelización de la factorización de Cholesky

Al ser un caso especial de la factorización LU, la factorización de Cholesky y LU tienen muchas similitudes, la forma de particionamiento es una de estas. En la sección 2.2.4 se observa que para realizar esta factorización se tienen 2 ecuaciones primordiales, por lo que al analizarlas obtenemos el grafo de dependencias que se muestra en la figura 3.7.

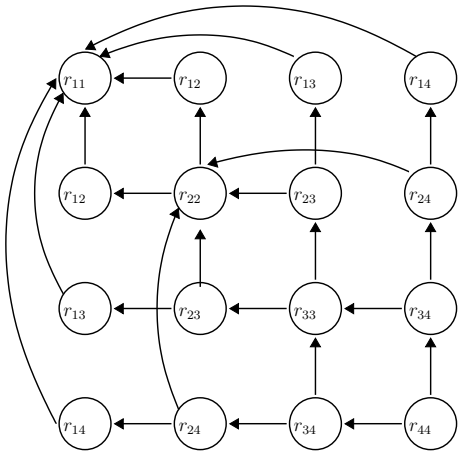

Figura 3.7: Grafo de dependencias de la factorización de Cholesky

Cabe aclarar que basta con calcular la parte triangular superior o inferior de la matriz para obtener la factorización.

Para obtener los elementos de la diagonal principal de R, se necesitan los elementos a la izquierda o superiores de estos, dependiendo si se calcula la triangular superior o inferior.

Para los demás elementos de R, se debe calcular primero el elemento de la diagonal principal superior a este y los elementos a la izquierda si se quiere calcular la triangular inferior, para la triangular superior se debe calcular primero el elemento de la diagonal principal a la izquierda y los elementos superiores.

En la figura 3.8 se muestra la secuencia de pasos para obtener la matriz R.

$$\begin{pmatrix} 1 & 2 & 2 & 2 & 2 & 2 \\ 2 & 3 & 4 & 4 & 4 & 4 \\ 2 & 4 & 5 & 6 & 6 & 6 \\ 2 & 4 & 6 & 7 & 8 & 8 \\ 2 & 4 & 6 & 8 & 9 & 10 \\ 2 & 4 & 6 & 8 & 10 & 11 \end{pmatrix}$$

Figura 3.8: Paralelización de la factorización de Cholesky

Así como en la factorización LU se requieren n hilos y $2n-1$ pasos para una matriz $A \in \mathbb{R}^{n \times n}$ para realizar la factorización de Cholesky.

3.2.4. Análisis de los algoritmos paralelos

De los algoritmos en forma paralela expuestos, se comprobó lo que se explicó en la sección anterior acerca de la dependencia de los cálculos anteriores de l o u para la factorización LU, lo cual no es adaptable a un modelo SPMD.

La factorización de Cholesky, es un método que se puede llevar a las tarjetas de GPU, pero sigue siendo un método que trabaja sobre un tipo de matriz en especifico, además de que no nos permite calcular los vectores propios.

Por lo que, para esta tesis el método de Givens es el que mejor se adapta a un modelo SPMD, ya que cada hilo puede trabajar sobre las filas de cada columna (columna = hilo), y haciendo unas modificaciones al método podemos evitar la sincronización.

Capítulo 4

Tecnologías para implementar algoritmos heterogéneos

La mayoría de arquitecturas paralelas especializadas para calculo matemático en la ultima década han sido de naturaleza homogénea. No obstante, la computación heterogénea se esta convirtiendo en una alternativa viable a los (generalmente caros) sistemas paralelos de altas prestaciones. En particular, las redes heterogéneas de computadores permiten utilizar recursos ya existentes con escalabilidad incremental de componentes, y con posibilidad de evaluar el rendimiento de forma aislada.

Si se desea aprovechar las ventajas de la computación heterogénea, se deben implementar algoritmos que utilicen diferentes tecnologías como son MPI,CUDA y OpenMP, entre otras. En este capitulo se describirán algunas tecnologías que se pueden ocupar para implementar algoritmos heterogéneos.

Daremos una pequeña introducción a OpenMP en la sección 4.1, en la sección 4.2 explicaremos lo que es CUDA, como funciona y para que se utiliza, así como en la sección 4.3 explicaremos que es, como funciona y algunas funciones principales de MPI y en la sección 4.4 expondremos las ventajas de utilizar CUDA y MPI para escalar a múltiples tarjetas de GPU, así como los problemas que se pueden encontrar al combinar estas 2 tecnologías.

4.1. OpenMP

OpenMP es una API definida conjuntamente por un grupo de los principales proveedores de hardware y software. OpenMP provee un modelo portable y escalable para los desarrolladores de aplicaciones paralelas de memoria compartida. La API es compatible con C/C++ y Fortran en una amplia variedad de arquitecturas. La API se compone de tres componentes primarios:

- Directivas del compilador.
- Biblioteca con rutinas para la ejecución.
- Variables de ambiente.

Algunas de las metas de OpenMP son:

- Estandarización:
  - Provee un estándar sobre una variedad de arquitecturas o plataformas de memoria compartida.
  - Conjuntamente definido y aprobado por un grupo de los principales proveedores de hardware y software.
- Claro y directo:
  - Establece un simple y limitado conjunto de directivas para la programación en máquinas de memoria compartida.
  - Puede implementarse un paralelismo significativo solo usando 3 o 4 directivas.
- Fácil de usar:
  - Proporciona la capacidad de paralelizar gradualmente un programa serial, a diferencia de las bibliotecas de paso de mensaje que típicamente requieren un todo o nada.
  - Proveen la capacidad de implementar paralelismo de grano fino o grueso.

- Portabilidad:
  - La API esta especificada para C/C++ y Fortran.
  - Esta implementado para casi todas las plataformas incluyendo Unix y Windows.

4.1.1. Modelo de Programación de OpenMP

OpenMP esta diseñado para maquinas multi-core con memoria compartida. La arquitectura puede ser memoria compartida tipo UMA[1] o NUMA[2].

OpenMP logra el paralelismo exclusivamente a través del uso de hilos, un hilo de ejecución es la unidad mas pequeña que se puede planificar por el sistema operativo. Los hilos existen con los recursos de un proceso solo. Típicamente, el numero de hilos corresponde al numero de cores en la maquina.

Es un modelo de programación explicito, ofreciendo al programado control total sobre la paralelización. La paralelización puede ser tan simple como tomar un programa serial e insertar directivas del compilador, o tan complejo como insertar subrutinas para establecer múltiples niveles de paralelismo, bloqueos e incluso bloqueos anidados.

OpenMP usa un modelo de ejecución paralelo de bifurca y une, todo programa OpenMP comienza como un proceso simple: el hilo maestro. El hilo maestro ejecuta secuencialmente hasta la primera región paralela es encontrada. Después se bifurca, el hilo maestro crea un equipo de hilos paralelos. Las declaraciones en el programa que son encerradas por la región paralela se ejecutan en paralelo por el equipo de hilos. Luego unimos, cuando el equipo de hilos termina las declaraciones en la región paralela, se sincronizan y terminan, dejando solo al hilo maestro. El numero de regiones paralelas y los hilos que las comprenden son arbitrarios.

La mayoría del paralelismo se especifica por el uso de directivas del compilador que están incrustadas en el código C/C++ o Fortran. La API permite poner regiones paralelas dentro de otras regiones. También permite al ambiente de ejecución que dinámicamente alteren el numero de hilos usados para ejecutar regiones paralelas

[1]Del inglés Uniform Memory Access, en español “acceso a memoria uniforme".
[2]Del inglés Non-Uniform Memory Access, en español “acceso a memoria no uniforme".

[14].

4.2. CUDA

CUDA es una nueva arquitectura de software y hardware para la emisión y gestión de cálculos en la GPU como un dispositivo para realizar cálculos sobre datos de forma paralela sin la necesidad de enviarlos a una API de gráficos. El mecanismo multitareas del sistema operativo es responsable del manejo de acceso a las GPU por diversas CUDA y aplicaciones graficas corriendo concurrentemente.

El paquete de software de CUDA se compone de varias capas como se observa en la figura 4.1: un controlador de hardware, una API y su tiempo de ejecución, y dos capas superiores de bibliotecas matemáticas para uso común, CUFFT[3] y CUBLAS[4]. El hardware se diseño para soportar controladores ligeros y capas de tiempo de ejecución, resultan en un alto rendimiento.

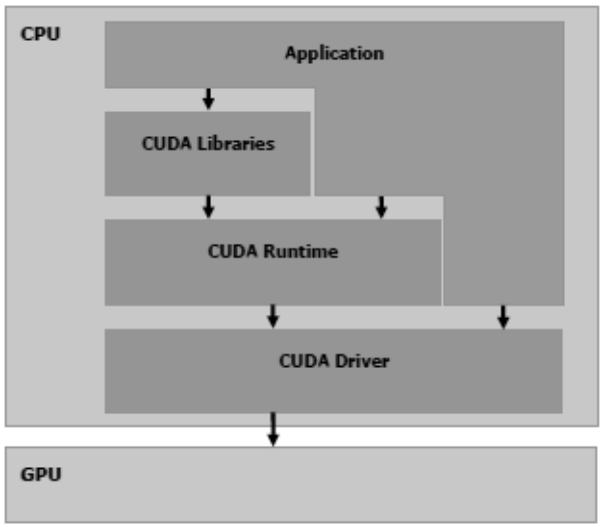

Figura 4.1: Paquete de software de CUDA.

El API de CUDA abarca una extensión del lenguaje de programación C para una curva de aprendizaje mínima.

CUDA proporciona un direccionamiento general de memoria DRAM[5] como muestra la figura 4.2 para una programación mas flexible: ambas dispersión y reunión

[3]NVIDIA CUDA Fast Fourier Transform library.
[4]NVIDIA CUDA Basic Linear Algebra Subroutines.
[5]Dynamic Random Access Memory

operaciones de memoria. Desde una perspectiva de programación, esto se traduce en la habilidad de leer y escribir datos a cualquier dirección en la DRAM, justo como en un CPU.

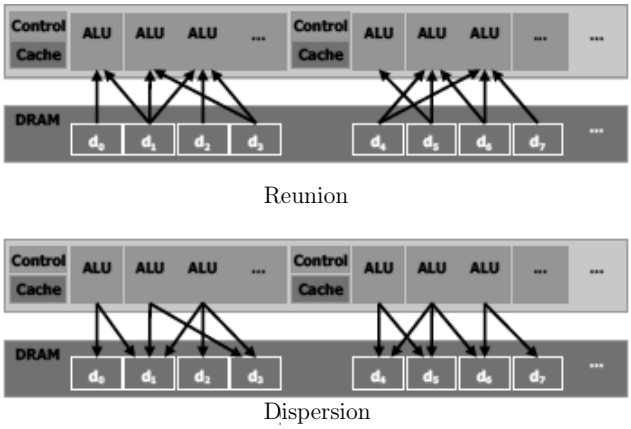

Figura 4.2: Las operación en memoria: dispersión y reunión.

CUDA cuenta con memoria compartida de datos en paralelo en el cache o en el chip con accesos de lectura y escritura generales muy rápidos, los threads usan los datos compartidos entre cada uno. Como se muestra en la figura 4.3, las aplicaciones pueden tomar ventajas de esto minimizando las búsquedas y los idas y vueltas de la DRAM y mas aun dependiendo lo menos posible del ancho de banda de la memoria DRAM.

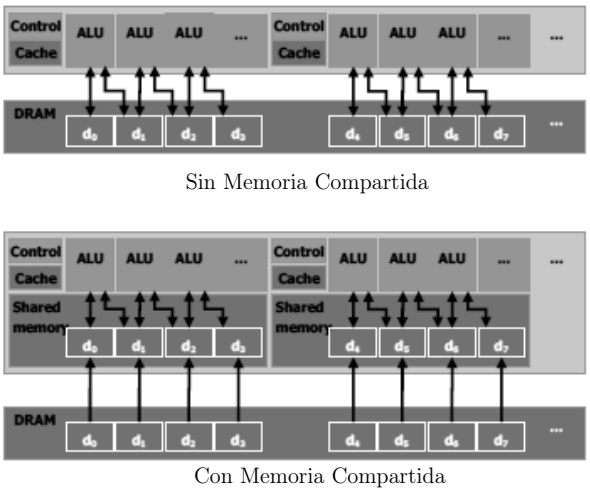

Figura 4.3: Memoria compartida acerca los datos a las unidades aritméticas lógicas (ALU, por sus siglas en ingles).

4.2.1. Un coprocesador vectorial

Cuando programamos por medio de CUDA, la GPU se ve como un *dispositivo de computo* capaz de ejecutar una gran cantidad de hilos en paralelo. Esta opera como un coprocesador para el CPU principal o *anfitrión*. En otras palabras, datos paralelos, porciones de computo intensivo de aplicaciones corriendo en el anfitrión que son descargadas en el dispositivo.

Mas preciso, una porción de una aplicación que se ejecuta muchas veces, pero independientemente en diferentes datos, puede ser aislada en una función que es ejecutada en el dispositivo en muchos hilos diferentes. Para este efecto, esta función es compilada para el conjunto de instrucciones del dispositivo, y el programa resultante, llamado textitkernel, es descargado al dispositivo.

Pero el anfitrión y el dispositivo mantienen su propio DRAM, llamado como *memoria del anfitrión* y *memoria del dispositivo*, respectivamente. Uno puede copiar información de un DRAM al otro mediante de llamadas del API optimizadas que utilicen el motor Acceso Directo a Memoria (DMA, por sus siglas en ingles) de alto rendimiento del dispositivo.

4.2.2. Grupos de hilos

Los grupos de hilos que ejecutan un *kernel* se organizan como una red de bloques de hilos como se describen en las secciones 4.2.2.1 y 4.2.2.2 y que se muestra en la figura 4.4.

4.2.2.1. Bloques de hilos

Un bloque de hilos es un grupo de hilos que pueden cooperar entre si por la eficiente compartición de datos debido a la memoria compartida y sincronizando las ejecuciones para coordinar los accesos a memoria. Mas precisamente, uno puede especificar los puntos de sincronización en el *kernel*, donde los hilos en un bloque son suspendido mientras todos llegan al punto de sincronización.

Cada hilo se identifica por su *ID de hilo*, el cual es el numero de hilo dentro del bloque. Para ayudar con el complejo direccionamiento basado en el ID del hilo, una aplicación puede especificar un bloque como un arreglo de dos o tres dimensiones

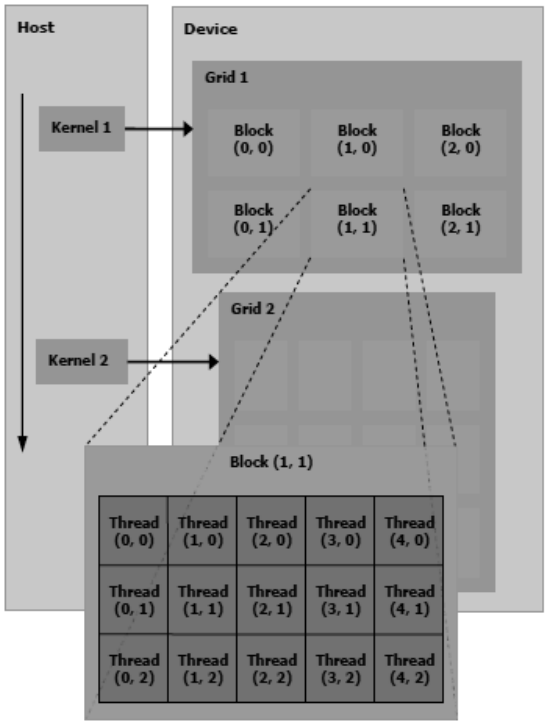

Figura 4.4: Grupo de hilos.

de tamaño arbitrario e identificar cada hilo usando un índice de 2 o 3 componentes respectivamente. Para un bloque bidimensional de tamaño (D_x, D_y), el ID de un hilo de índice (x, y) es $(x + yD_x)$ y para un bloque de tridimensional de tamaño (D_x, D_y, D_z), el Id de un hilo con índice (x, y, z) es $(x + yD_x + zD_xD_y)$.

4.2.2.2. Red de bloques de hilos

Existe un numero máximo de hilos limitado que cada bloque puede contener. Sin embargo, bloques de la misma dimensionalidad y tamaño que ejecuten el mismo *kernel* pueden ser agrupados juntos en una red de bloques, por lo que el numero total de hilos que se pueden lanzar en una sola invocación de *kernel* es mucho mas grande. Esto viene a expensas de la reducción de cooperación de hilos, porque los hilos en diferentes bloques de la misma red no pueden comunicarse y sincronizarse entre estos. Este modelo permite a los *kernels* ejecutarse eficientemente sin recompilarse en varios dispositivos con diferentes capacidades paralelas: Un dispositivo podría ejecutar todos los bloques de la red secuencialmente si este tiene muy pocas capacidades paralelas, o

en paralelo si cuenta con muchas capacidades paralelas, o usualmente una combinación de las dos.

Cada bloque se identifica por su *ID de bloque*, el cual es el numero del bloque dentro de su red. Para ayudar con el complejo direccionamiento basado en el ID de bloque, una aplicación puede especificar una red como un arreglo bidimensional de tamaño arbitrario e identificar cada bloque usando un índice de 2 componentes. Para una red bidimensional del tamaño (D_x, D_y) el ID del bloque con índice (x, y) es $(x + yD_x)$.

4.2.3. Modelo de Memoria

Un hilo que se ejecuta en el dispositivo tiene solo acceso a la memoria DRAM y del chip del dispositivo por los siguientes espacios de memoria, como se expone en la figura 4.5:

- lectura y escritura por hilo *registros*,
- lectura y escritura por hilo *memoria local*,
- lectura y escritura por bloque *memoria compartida*,
- lectura y escritura por red *memoria global*,
- lectura y escritura por red *memoria constante*,
- lectura y escritura por red *memoria de textura*.

Los espacios de memoria global, constante y de textura pueden ser leídos o escritos por el anfitrión y son continuos a través de las llamadas a *kernel* por la misma aplicación.

Los espacios de memoria global, constante y de textura son optimizados para diferentes usos de memoria. La memoria de textura también ofrece diferentes modos de direccionamiento, así como de filtrado de datos.

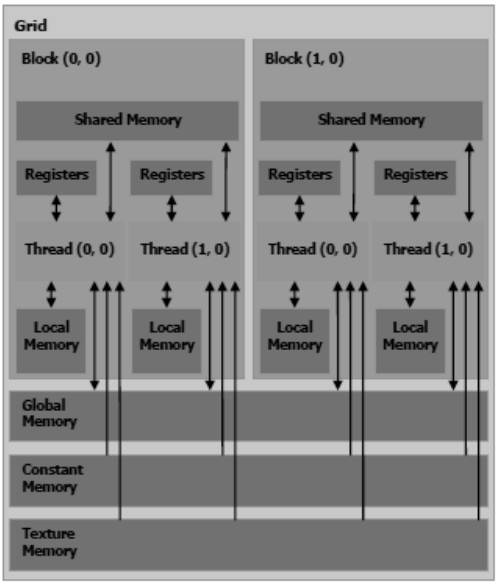

Figura 4.5: Modelo de Memoria.

4.2.4. Implementación de Hardware: Un conjunto de multiprocesadores SIMD

El dispositivo esta implementado como un conjunto de *multiprocesadores* como ilustra la figura 4.6. Cada multiprocesador tiene una arquitectura SIMD: A cualquier ciclo de reloj dado, cada procesador del multiprocesador ejecuta la misma instrucción, pero opera en diferentes datos.

Cada multiprocesadores tiene una memoria en chip de los cuatro diferentes tipos:

- un conjunto local de *registros* de 32 bits por procesador,
- un cache de datos o *memoria compartida* paralela que es compartida por todos los procesados e implementa el espacio de memoria compartido,
- un *cache constante* solo de lectura que es compartido por todos los procesadores y aumenta la velocidad de lectura del espacio de memoria constante, el cual es implementado como una región solo de lectura de la memoria del dispositivo,
- un *cache de textura* solo de lectura que es compartido por todos los procesadores y aumenta la velocidad de lectura del espacio de memoria de textura, el cual es implementado como una región solo de lectura de la memoria del dispositivo.

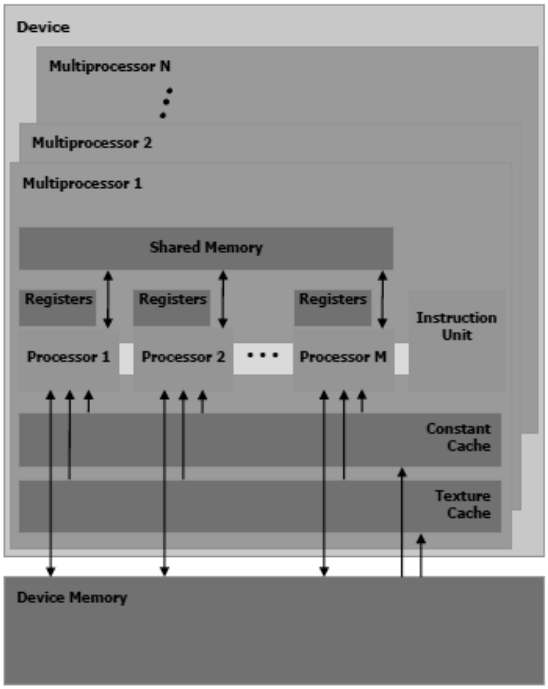

Figura 4.6: Modelo de Hardware.

Los espacios de memoria local y global están implementados como regiones de lectura y escritura de la memoria del dispositivo y no están en cache.

Cada multiprocesador accesa al cache de textura vía la *unidad de textura* que implementa los varios modos de direccionamiento y filtrado de datos.

4.2.5. Modelo de Ejecución

Una red de hilos se ejecuta en el dispositivo ejecutado uno o mas bloques en cada multiprocesador usando cortes de tiempo: Cada bloque se parte en grupos SIMD de hilos llamados *warp*, cada uno de estos *warps* contiene el mismo numero de hilos, llamado *warp size*, y se ejecutan por el multiprocesador en una manera SIMD; un *planificador de hilos* periódicamente cambia de un *warp* a otro para maximizar el uso de los recursos computacionales del dispositivo. Un *half-warp* es cualquier de los dos la primera o segunda mitad de un *warp*.

La manera en que un bloque se divide en *warp* es siempre la misma; cada *warp* contiene hilos consecutivos, con crecientes IDs con el primer *warp* teniendo el hilo 0.

Un bloque se procesa por solamente un multiprocesador, por lo que el espacio de memoria compartida reside en la memoria compartida del chip, conduciendo a accesos de memoria muy rápidos. Los registros del multiprocesador son alojados entre los hilos del bloque. Si el numero de registros usado por hilo multiplicado por el numero de hilos en el bloque es mayor al numero total de registros por multiprocesador, el bloque no puede ser ejecutado y el correspondiente *kernel* fallara al lanzarse.

Varios bloques pueden ser procesador por el mismo multiprocesador concurrentemente alojando los registros y memoria compartido entre los bloques.

La orden de emisión de los *warps* dentro de un bloque no esta definida, pero la ejecución puede ser sincronizada, para coordinar los accesos a memoria global y compartida. Si la instrucción ejecutada por el *warp* escribe a la misma dirección de la memoria global o compartida por mas de uno de los hilos del *warp*, cuantas escrituras ocurren a esa dirección y el orden en el que ocurren no esta definido, pero una de las escrituras se garantiza que sucederá.

La orden de emisión de los bloques dentro de una red de bloques de hilos no esta definida y no un mecanismo de sincronización entre bloques, así los hilos de dos diferentes bloques de la misma red no puede comunicarse de manera segura entre estos mediante la memoria global durante la ejecución de la red [15].

4.3. MPI

Muchos de los conceptos que encontramos en la computación paralela son conceptos análogos en áreas sociales como la administración de empresas. La idea de mucha gente trabajando junta por una meta es similar a muchos procesos trabajando juntos para alcanzar una solución. La idea de querer partir el trabajo para que todos los procesos estén ocupados y ninguno permanezca ocioso es similar a esperar mantener a un equipo ocupado, sin nadie sentado alrededor esperando por la información de alguien mas. Desde esta perspectiva, observamos que la computación paralela es una extensión natural del concepto divide y vencerás, es decir, comenzamos con un problema que queremos resolver, después tenemos los recursos disponibles que podemos usar para resolver el problema (que en el caso de la computación seria el numero de procesadores que podemos usar), y después intentamos partir el problema en piezas manejables que

pueden ser realizadas por cada persona del equipo.

El dificultad mas común de la gente con este concepto no es el componente de divide y conquistaras; mas comúnmente, estamos mas cómodos con el concepto de partir un problema en subproblemas con los cuales trabajar. Sin embargo, puede ser difícil (1) observar como un problema en especifico se puede partir eficientemente para ser resuelto en paralelo y (2) entender como podemos hacer que las computadoras trabajen en paralelo.

La premisa detrás de MPI es que múltiples procesos paralelos trabajen concurrentemente para alcanzar un a mee usando mensajes como medio de comunicación entre cada uno. Esta idea se ilustra en la figura 4.7. Múltiples procesos MPI pueden correr en diferentes procesadores, y estos procesos comunicarse por medio de la infraestructura que provee MPI. Como usuarios, no necesitamos saber la implementación de esta infraestructura; necesitamos solo saber como tomar ventaja de esta. Con este fin, casi todo en MPI se puede resumir en una simple idea “Mensaje Enviado - Mensaje Recibido".

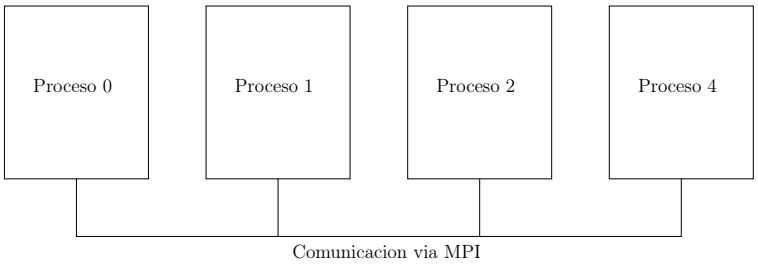

Figura 4.7: Esquemática de los procesos MPI trabajando juntos.

La interfaz de programación de MPI es una colección de funciones. MPI es una biblioteca de funciones designadas para manejar los aspectos esenciales del paso de mensajes en la arquitectura en la cual se quiere ejecutar. Hay ciertas cosas que hay q tener en cuenta al hacer un programa que utilice MPI:

- debemos incluir *mpi.h*. Esto nos provee todas las declaraciones de todas las funciones de MPI,

- debemos tener un comienzo y un final. El inicio es en la forma de una llamada MPI_Init(), la cual indica al sistema operativo que es un programa de MPI y permite al sistema operativo hacer la inicialización necesaria. El final se realiza

con una llamada a la función MPI_Finalize, la cual indica al sistema operativo que limpie todo lo relacionado a MPI,

- si el programa es demasiado paralelo, entonces las operaciones que se realicen entre la inicialización y finalización de MPI no involucra comunicación.

Hay dos importantes comando que se utilizan comúnmente en MPI:

- MPI_Comm_rank(MPI_comm comm, int* result).
- MPI_Comm_size(MPI_comm comm, int* size).

La primera de las dos funciones nos provee el identificador del proceso (que es un entero en el rango de 0 a $p-1$, donde p es el numero de procesos que se están ejecutando). La segunda función nos provee con el numero total de procesos que se han alojado. El argumento comm es llamado comunicado y en esencia es una designación para la colección de los procesos que se pueden comunicar entre ellos. MPI tiene la funcionalidad de permitir al usuario especificar varios comunicadores.

Si se ejecuta un programa con $p = 8$ procesos, la funciona MPI_Comm_size nos pasaría el numero 8 como el total de procesos ejecutándose y MPI_Comm_rank nos retornaría un numero 0, 1, ..., 7 que denotan cual proceso es que se esta ejecutando en ese momento.

Hay que hacer una observación critica: Cuando se esta ejecutando MPI, todos los proceso ejecutan el mismo código binario y por ende todos los procesos están corriendo el mismo código, por lo que hay dos cosas que distinguen el programa paralelo:

- cada proceso usa su identificador para determinar cual parte de las instrucciones del algoritmo realizara el proceso,
- los procesos se comunican entre toso para cumplir la tarea final.

Aun cuando cada proceso recibe una copia idéntica del conjunto de instrucciones para ser ejecutadas, esto no implica que todos los procesos van a ejecutar las misma instrucciones. Porque cada proceso es capaz de obtener su identificador, entonces cada proceso puede determinar que parte del código debe correr. Esto se logra utilizando la sentencia **if**. El código que debe ejecutarse en un proceso en particular debe colocarse

dentro de la sentencia **if**, el cual verificara el identificador de proceso. Si el código no contiene ninguna sentencia **if**, entonces el código será ejecutado por todos los procesos [16].

Para la comunicación de procesos que es una de las características principales de MPI se utilizan dos funciones principalmente, aunque existen muchas otras, estas son MPI_Send y MPI_Recv. En general funciona enviando un mensaje de cierto tipo de dato de un proceso a otro utilizando los identificadores de procesos, algunas cosas que hay que considerar de estas funciones son:

- en general, el mensaje del que envía y el que recibe deben ser del mismo tipo de dato y deben de tener el mismo tamaño,
- el proceso que envía debe especificar a cual proceso será enviado el mensaje.

4.4. CUDA y MPI

En algunos casos es necesario juntar las 2 tecnologías antes expuestas, ya que las tarjetas graficas tienen una memoria limitada. Por lo que en problemas donde la cantidad de memoria que se necesita es muy grande, trabajar con una sola GPU es imposible, por lo que tenemos que utilizar mas de una GPU.

Actualmente no existe una forma de comunicar dos GPU, por lo que se deben de buscar formas para comunicar estas GPU. Es aquí donde podemos utilizar MPI, ya que esto nos permite levantar diferentes procesos, para que cada proceso utilice una GPU y así poder utilizar mas de una GPU al mismo tiempo [17].

Pero hay un problema ya que no existe un estándar para el uso de las GPU y MPI, lo que ocasiona que cada programa es responsabilidad del programador, por lo que:

- los movimientos de datos son controladores por el usuario,
- el usuario tiene todo el control del programa,
- la sincronización esta a cargo del programador.

Dentro del programa hay dos actores principales el CPU lanzado por MPI y el GPU que funciona gracias a CUDA, las funciones principales de cada actor son las siguientes:

- responsabilidades del CPU:
  - copiar los datos a la memoria de la GPU,
  - copiar los datos de la memoria de la GPU,
  - iniciar las transferencias de MPI.
- Mientras que el GPU solo hará el trabajo que el CPU le indique.

Algunos otros problemas que surgen con la combinación de MPI y CUDA:

- es en la compilación al utilizar estas dos tecnologías ya que hay que buscar las librerías que necesita cada una de las tecnologías y juntarlas,
- después de compilar el programa, en la ejecución de este como saber si se esta ejecutando en el GPU,
- y por ultimo como depurar los programas, por donde empezar la depuración.

De estos problemas algunos se pueden resolver, pero hay algunos que aun no se conoce como solucionarlos actualmente:

- el primer problema se resuelve con conocer la dirección de las librerías necesarias y colocar toda la información de la compilación en una Makefile[6],
- el segundo problema relacionado con la ejecución, se resuelve realizando pruebas de los dispositivos para conocer la información que estos arrojan,
- y del problema de depuración aun no se conoce soluciones.

En conclusión, el hecho de utilizar ambas tecnologías juntas (MPI y CUDA) puede ocasionar muchos problemas, pero al utilizarlos de forma correcta podemos utilizar el poder de mas de un GPU paralelamente, con el fin de poder trabajar sobre problemas que ocupen una cantidad de memoria mayor al de una GPU.

[6]Los makefiles son los ficheros de texto que utiliza make para llevar la gestión de la compilación de programas.

Capítulo 5

Diseño e Implementación

En este capítulo se explicara el cómo y que se hizo la implementación paralela de algunos de los algoritmos mencionados en los capítulos anteriores utilizando las tecnologías de CUDA (en algunos casos solamente se utilizó esta) y de MPI. En la sección 5.1 se explica el proceso que se llevó a cabo para poder implementar de manera paralela, si bien no el algoritmo QR, si algunas de sus partes y en la sección 5.2 se describe como se paralelizo el método de giben para múltiples GPU.

5.1. Paralelización del algoritmo QR

Del algoritmo 1 en la seccion 2.1 podemos encontrar las operaciones que se pueden paralelizar. Estas operaciones están definidas en la línea 3, donde se revisa si una matriz es o no diagonal. En la línea 4, la factorización de matrices. Y en las líneas 5 y 6, la multiplicación de matrices.

5.1.1. Determinar si una matriz es diagonal

Para poder paralelizar esta operación, aprovechamos el comportamiento de la matriz D en la iteración. Es decir, al realizar el proceso de diagonalización a una matriz simétrica, se van eliminando los valores simétricamente de las diagonales más lejanas a la diagonal principal en cada paso de la iteración, hasta obtener una matriz diagonal. Por lo que la manera de revisar si la matriz es o no diagonal, basta con verificar si lo valores de la diagonal inmediatamente inferior o superior cualquiera se quiera es

diferente a algún ϵ que corresponde al error, en la figura 5.1 se observa las diagonales a revisar.

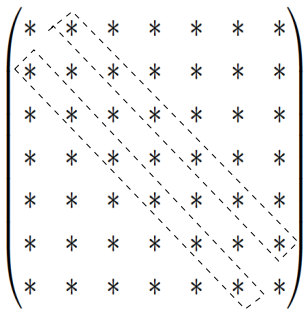

Figura 5.1: Diagonales a revisar de la matriz.

Entonces para revisar si una matriz es o no diagonal, debe estar la matriz en la memoria global de la tarjeta gráfica de GPU y cada hilo en la tarjeta revisa el valor $a_{i,i-1}$ o $a_{i,i+1}$ donde i es el número de hilo, y si alguno de los hilos encuentra que el valor buscado es mayor al ϵ establecido, entonces se dice que la matriz no es diagonal, pero si después de revisar toda la diagonal inferior o superior a la diagonal principal ninguno de los hilos encuentra un valor mayor a ϵ entonces podemos decir que la matriz es diagonal.

5.1.2. Multiplicación de matrices

Este tipo de operación ya ha sido estudiado mucho en la programación paralela con tarjetas gráficas de GPU, por lo que no se implementó esta operación, solo se utilizó la biblioteca de NVIDIA CUBLAS, la cual ya tiene definida la multiplicación de matrices de manera óptima.

5.1.3. Factorización de matrices

La factorización de matrices dentro del algoritmo de diagonalización es una operación importante, ya que depende del tipo de matriz, la arquitectura donde se implementara, el método que se seleccione, además de que es una operación que consume mucha memoria y mucho CPU, por lo que es esencial poder paralelizar esta operación.

Hemos seleccionado el método de Givens para la factorización, ya que es el método que mejor se adapta a la arquitectura con memoria compartida, que es el modelo de

memoria para los GPU, además de que podemos evitar la sincronización de hilos con una buena implementación.

Paralelizar el método de Givens se basa en realizar las rotaciones de Givens a 2 filas, esto quiere decir que cada vez que se realice la rotación de Givens, todos los valores de estas 2 filas se modificaran, nos podemos dar cuenta que los cálculos por cada columna son independientes, por lo que se puede utilizar un hilo por cada columna.

La implementación que se llevó a cabo involucra una sincronización de trabajo entre el CPU y el GPU, por lo que la implementación se vuelve completamente heterogénea.

En el algoritmo 11 se puede observar los pasos que realiza el CPU. Las operaciones descritas en las líneas 5 y 6 se ejecutan en las tarjetas gráficas de GPU, donde el CPU las ejecuta a través de los kernels.

Algoritmo 11 Factorización QR con rotaciones de Givens

Entrada: Matriz Cuadrada Simétrica $A \in \mathbb{R}^{n \times n}$, Matriz Identidad $I \in \mathbb{R}^{n \times n}$.

Salida: Matriz Triangular Superior $R \in \mathbb{R}^{n \times n}$, Matriz Ortogonal $Q \in \mathbb{R}^{n \times n}$.

1: $R \leftarrow A$.
2: $Q \leftarrow I$.
3: Copiar R e Q a memoria de la tarjeta gráfica de GPU.
4: **para** $i = 0, 1, \ldots, n-1$. **hacer**
5: Obtener la columna i de R.
6: Aplicar las rotaciones de Givens a la columna i de R y Q.
7: **termina para**
8: Copiar R y Q de la tarjeta gráfica de GPU al CPU.

5.1.3.1. Obtener la columna i de R

Al trabajar con el método de Givens sobre 2 filas, hay que esperar a que la fila superior cambie todos su valores para poder aplicar las rotaciones a esta fila y a la superior, lo que conlleva en la paralelización a una sincronización de hilos, ya que los hilos deben de esperar a que termine el hilo que está trabajando sobre la columna en cuestión.

Para evitar la sincronización se decidió que cada hilo hiciera el calculo de los

valores c y s para realizar las rotaciones en la columna que le toca. Por esto, se decidió almacenar la columna a la que se aplicaran las rotaciones y así cada hilo podría obtener los valores de esta columna y después utilizarlos para calcular el c y el s y así evitar la sincronización.

Obtener y guardar la columna i de R, línea 5 del algoritmo 11, en el kernel cada hilo obtiene un valor de la columna i de R y lo almacena un arreglo que se encuentra en la memoria global de la tarjeta, esto se logra con ayuda de su identificador. Por lo que en una sola llamada a este kernel se almacenan todos los valores de la columna requerida.

5.1.3.2. Aplicar las rotaciones de Givens a la columna i de R y Q

Para aplicar las rotaciones de Givens a las columnas se sigue el algoritmo 12 por cada columna, como se explicó antes una rotación de Givens afecta a 2 filas, por lo que cada hilo ejecuta el mismo proceso, lo único que cambia es que cada hilo trabaja sobre una columna diferente, por lo que al final de llamar una vez el algoritmo 12 modificaremos a 0's los valores de una columna bajo la diagonal principal y después de llamar $n-1$ veces el algoritmo tendremos la matriz triangular superior R y Q^T.

La matriz Q^T se obtiene al aplicar las rotaciones de Givens a la matriz identidad, ya que:

$$A = QR \Rightarrow Q^T A = R \Rightarrow Q^T = G_n \dots G_1 I,$$

donde $Q^T = Q^{-1}$ ya que la matriz Q de la factorización QR es una matriz ortogonal.

En la figura 5.2 podemos observar cómo funciona un llamado al kernel para aplicar las rotaciones de Givens a una columna, en este caso es para la primera columna.

Algoritmo 12 Calcular rotaciones de Givens por columna

Entrada: Matriz Simétrica $R \in \mathbb{R}^{n \times n}$, Matriz Identidad $Q \in \mathbb{R}^{n \times n}$, Columna obtenida de R, $L \in \mathbb{R}^n$ e identificador *col* de la columna a aplicar rotaciones de Givens.

Salida: Matriz R y Q con rotaciones de Givens aplicadas a la columna *col*.

1: $i =$ identificador de hilo.
2: $j = n$.
3: **mientras** l_j sea 0 **hacer**
4: $\quad j = j - 1$
5: **termina mientras**
6: $tempj = j$
7: **mientras** $i < n$ **hacer**
8: $\quad \mu_i' = \sqrt{{l_{j-1}}^2 + {l_j}^2}$.
9: $\quad c = \frac{l_{j-1}}{\mu_i'}$ y $s = \frac{l_j}{\mu_i'}$.
10: $\quad$ **mientras** $j > col$ **hacer**
11: $\quad\quad \mu_i = r_{j-1,i}$ y $\nu_i = r_{j,i}$.
12: $\quad\quad \alpha_i = q_{j-1,i}$ y $\beta_i = q_{j,i}$.
13: $\quad\quad r_{j-1,i} = c\mu_i + s\nu_i$ y $r_{j,i} = -s\mu_i + c\nu_i$.
14: $\quad\quad q_{j-1,i} = c\alpha_i + s\beta_i$ y $q_{j,i} = -s\alpha_i + c\beta_i$.
15: $\quad\quad j = j - 1$.
16: $\quad\quad a = \mu_i'$.
17: $\quad\quad \mu_i' = \sqrt{{l_{j-1}}^2 + a^2}$.
18: $\quad\quad c = \frac{l_{j-1}}{\mu_i'}$ y $s = \frac{a}{\mu_i'}$.
19: $\quad$ **termina mientras**
20: $\quad j = tempj$
21: $\quad$ recorrer i al final del bloque
22: **termina mientras**

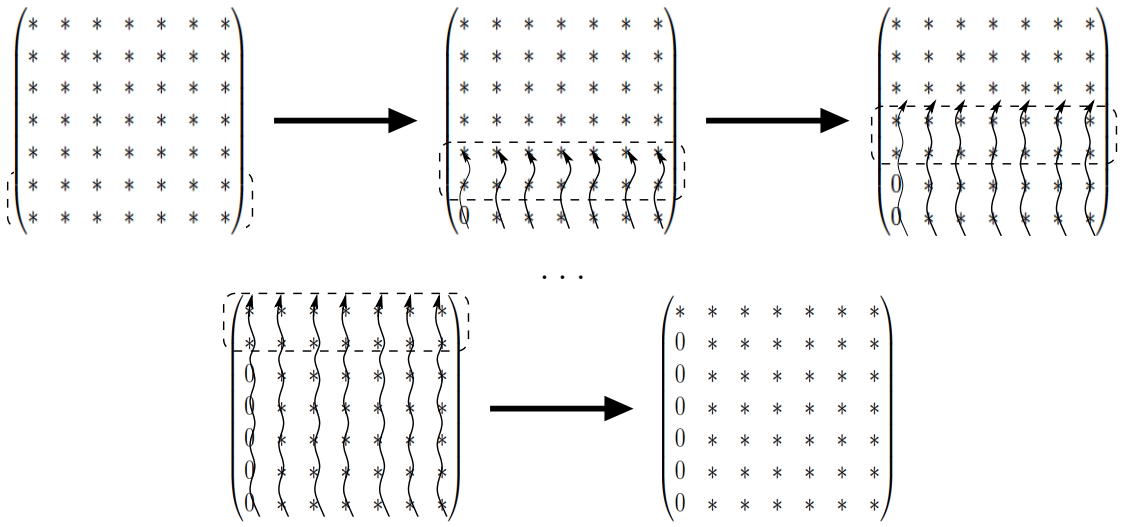

Figura 5.2: Primer llamado al kernel para aplicar rotaciones de Givens a la primera columna.

Por lo que después de $n-1$ llamados obtendremos R, en la figura 5.3 se puede ver el proceso completo para obtener la matriz R, el mismo proceso se aplica a la matriz identidad para obtener Q^T.

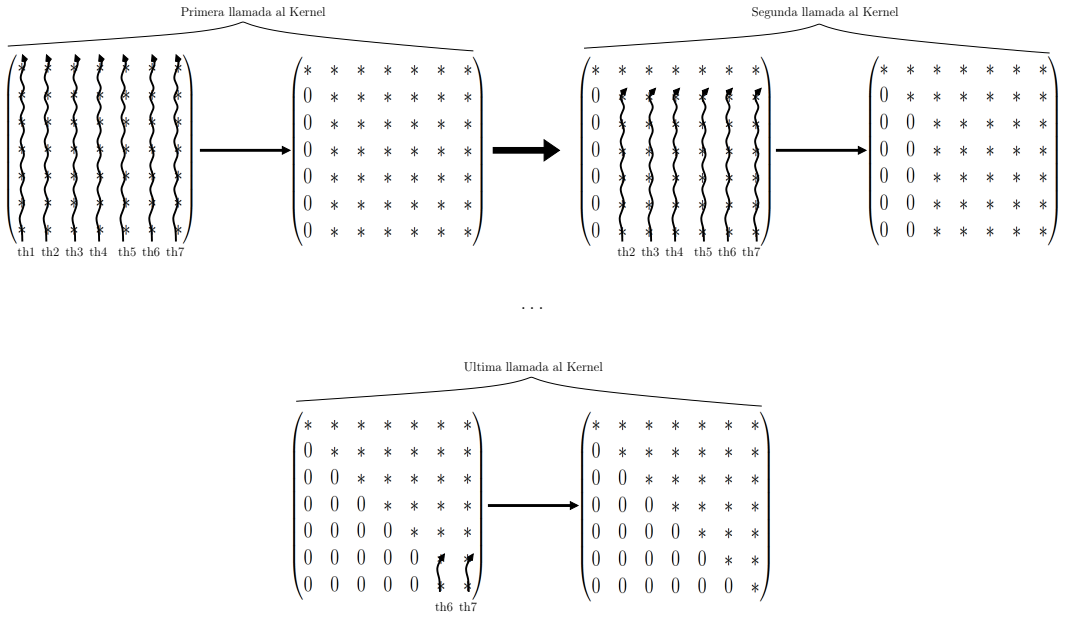

Figura 5.3: Proceso para obtener la matriz triangular superior R con rotaciones de Givens.

En las figuras antes mencionadas también se puede apreciar que conforme va

avanzando el proceso hay algunos hilos que se quedan sin trabajar, efectivamente esto sucede para los últimos pasos. Pero cuando el número de hilos es menor al número de columnas de la matriz, los hilos se recorren.

Por ejemplo, se quiere aplicar el método de Givens a una matriz $M \in \mathbb{R}^{7\times 7}$ y se trabajara con 3 hilos, entonces el primer hilo aplicara las rotaciones de Givens a la primera columna, después de terminar pasara a la columna 1+ *número de hilos* es decir a la columna 4 para aplicar las rotaciones a esta columna y así lo harán los demás hilos, siempre que sea posible, todo esto hasta que toda la matriz es cubierta.

Así no queda ninguna columna sin que se le apliquen las rotaciones de Givens y no hay necesidad de tener el mismo número de hilos que de columnas, algo que sirve de mucho ya que en cualquier implementación no se pueden tener un número ilimitado de hilos.

5.2. Método de Givens para múltiples GPU

Como describimos en el planteamiento del problema, uno de los puntos débiles de utilizar tarjetas de GPUs es la memoria, ya que la tarjeta con más memoria RAM disponible es la Tesla K10 con 8 GB [18]. Entonces, si queremos ocupar más memoria de la que cuenta nuestra tarjeta de GPUs, tenemos que buscar la forma de poder utilizar más de una tarjeta, con el fin de trabajar con conjuntos de datos más grandes.

Existen algunas estrategias para utilizar más de una tarjeta de GPUs; una es utilizar una tecnología que comienza a surgir y que el mismo NVIDIA creó, se denomina *GPUDirect*, pero como se menciona es una tecnología nueva por lo que no todas las tarjetas la soportan, por lo que esta no es una opción; otra es que un programa cambie entre tarjetas en una misma computadora, el problema es que solo funcionaria para una computadora pero no para un cluster de tarjetas de GPUs; y la opción que se seleccionó, utilizar la tecnología MPI que nos permite generar diferentes procesos por cada nodo, donde a cada proceso se le asignara una partición de los datos y así trabajar en la tarjeta que se le indique con los datos asignados.

El siguiente paso era seleccionar la estrategia con la cual se van a utilizar múltiples tarjetas de GPUs, ahora procedemos a seleccionar una forma de particionar la información, como primera opción se quería particionar los datos por columnas como

muestra la figura 5.4, ya que el método de Givens como hemos visto trabaja por columnas, pero no podemos particionar por columnas debido a la manera de almacenar los datos. Por lo que después de revisar el método de Givens se observó que, se pueden aplicar las rotaciones de Givens por fila en vez de columna y en lugar de obtener la matriz R, obtendríamos R^T. Para el particionamiento de datos, después de observar que se puede aplicar las rotaciones a las filas, se estudió si se podía particionar la matriz por filas como muestra la figura 5.5 y se llegó a la conclusión que era posible.

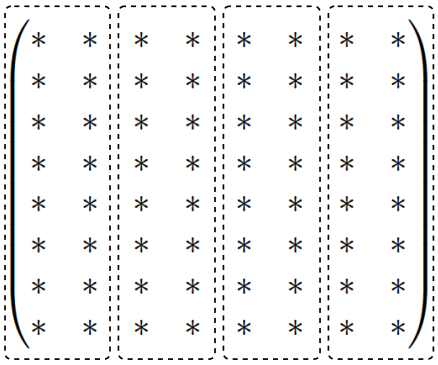

Figura 5.4: Particionamiento por columnas

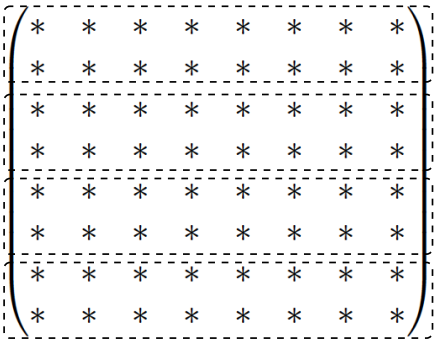

Figura 5.5: Particionamiento por filas

Posterior al estudio se llegó a la conclusión de utilizar MPI con CUDA para utilizar múltiples de GPUs y la estrategia de particionamiento seria por filas en lugar de columnas.

Ya seleccionado el método para utilizar múltiples GPUs y la estrategia de particionamiento se procede a implementar el algoritmo para múltiples GPUs.

Como se explicó en la sección 4.3, MPI requiere de que se inicialice y se finalice con dos funciones respectivamente (*MPI_Init* y *MPI_Finalize*), aunque esto no quiere decir que solo se creen los procesos desde que se llaman a estas funciones, los procesos se crean en el momento de ejecutar el programa con el comando *mpirun*, y se crean el número de procesos que se indiquen al ejecutar el programa.

Por lo que la lectura y escritura de información la debe realizar un solo proceso desde el principio, con el fin de evitar que todos los procesos realicen estas operaciones al mismo tiempo.

Después de inicializar MPI, se deben obtener el número de procesos que se utilizaran, este número debe ser un número que divida a la n de una matriz $A \in \mathbb{R}^{n \times n}$, para después particionar los datos en partes iguales que se le asignaran a cada proceso.

En seguida cada proceso aplicara el método de Givens a la partición que se le asigno, uno de los puntos importantes para poder aplicar las rotaciones de Givens, es la obtención de la columna a la que se le aplicaran las rotaciones como se observa en la sección 5.1.3.1.

Luego se deben reunir los datos de cada proceso en un solo conjunto de datos para poder mostrar los resultados, en este caso las matrices Q y R^T.

En el algoritmo 13 podemos observar con más detalle los pasos que sigue cada proceso creado para poder aplicar el método de Givens en múltiples GPUs, siempre y cuando el número de procesos divida al tamaño de la matriz.

Analizando el algoritmo 13, hay algunas líneas interesantes. En la línea 7, se selecciona a que tarjeta estará asignado cada proceso, la función en CUDA encargada de seleccionar la tarjeta es *cudaSetDevice()*, hay que tener en cuenta que un procesador solo puede tener 2 tarjetas asociadas, por lo que para asignar la tarjeta a cada proceso se realiza una operación modulo para obtener el identificador de la tarjeta:

$$idGPU = idPro \mod 2$$

El numero 2 está relacionado al número de tarjetas que puede tener cada procesador. Por ejemplo si tenemos cuatro procesos, el proceso 0, 1, 2 y 3, entonces a los procesos 0 y 2, se les asignara tarjeta 0 y a los procesos 1 y 3, la tarjeta 1. Y así cada proceso llamara la función *cudaSetDevice(idGPU)* para asignar la tarjeta con la cual se estará trabajando.

Otra línea interesante es la 8, donde se dividen las matrices R y Q por filas de tal manera que a cada proceso le corresponda un mismo número de filas, para esto, primero calculamos el número total de elementos con los que cuenta cada matriz y esto lo dividimos entre el número de procesos, de aquí la importancia que el número de procesos divida al número de columnas y filas de la matriz. Ya con el número obtenido de la siguiente ecuación:

$$tamDiv = \frac{n * n}{np}$$

donde $tamDiv$ es el número de elementos que se le asignara a cada proceso, n corresponde a $A \in \mathbb{R}^{n \times n}$ y np es el número de procesos.

Algoritmo 13 Factorización QR con rotaciones de Givens para múltiples GPUs

Entrada: Matriz Cuadrada Simétrica $A \in \mathbb{R}^{n \times n}$, np = número de procesos.

Salida: Matriz Triangular Superior $R \in \mathbb{R}^{n \times n}$ transpuesta, Matriz Ortogonal $Q \in \mathbb{R}^{n \times n}$.

1: Iniciar MPI.
2: id = número de proceso.
3: **si** $id == 0$ **entonces**
4: $R \leftarrow A$.
5: $Q \leftarrow I$.
6: **termina si**
7: Seleccionar a que tarjeta se asociara este proceso.
8: Dividir R y Q en particiones iguales y enviar a todos los procesos del $id = 0$.
9: Copiar mi partición de R y Q a la tarjeta gráfica de GPUs.
10: **para** $i = 0, 1, \ldots, n-1$. **hacer**
11: **si** La fila i a la que se desea aplicar la rotación de Givens pertenece a la partición de R de id **entonces**
12: Obtener la fila i de la partición de R de id.
13: **termina si**
14: id envía la fila obtenida a todos los demás procesos.
15: Esperar a que todos los procesos hayan recibido la fila.
16: Copiar la fila a la tarjeta de GPUs.
17: Aplicar las rotaciones de Givens a las particiones de R y Q.
18: **termina para**
19: Copiar mi partición de R y Q de la tarjeta gráfica de GPUs al CPU.
20: Unir las particiones para obtener R^T y Q.
21: Finalizar MPI.

Después se utiliza una función de MPI para dividir un conjunto de datos en partes iguales entre los diferentes procesos, esta función es *MPI_ Scatter* y para juntar de nuevo los datos ocupamos la función *MPI_ Gather* que se puede observar en la línea 20 del algoritmo.

Las líneas más importantes se describirán más a fondo en las próximas secciones, estas son: las líneas 11 y 12 que muestran como obtener la fila necesaria de un proceso en específico y la más importante la línea 17 donde se aplican las rotaciones de Givens a la partición de las matrices de cada proceso.

5.2.1. Obtener la fila requerida de la partición *R* que la contiene

Como se explicó en la sección 5.1.3.1, el fin de obtener la fila a la que se le aplicaran las rotaciones de Givens, es evitar sincronización entre hilos, y en el caso de la implementación multi-GPU, evitar la sincronización entre tarjetas, ya que una sincronización entre tarjetas no es posible. Solo existen sincronización de hilos y bloques de una misma tarjeta, por lo que una sincronización entre tarjetas implicaría, envió de datos con MPI, lo que retrasaría el tiempo de ejecución y más aún en estos problemas que requieren muchas iteraciones.

Después de entender el motivo de la obtención de la fila a la que se le aplicaran las rotaciones de Givens, ahora nos enfocaremos en saber en qué partición se encuentra, dado que en un principio dividimos a la matriz en partes iguales, necesitamos encontrar en que partición se encuentra la fila, por lo que se utilizaron algunas variables para poder manejar los índices. A continuación se muestra el algoritmo 14 que detalla de la línea 11 a la línea 14 del algoritmo 13.
donde j indica que fila debemos obtener dentro de cada partición, k nos indica a que partición debemos ir para obtener la fila, d es el número de fila que cada partición tiene, n es el número de filas que tiene la matriz original, np es el numero proceso que se generaron que es equivalente al número de particiones que se realizaron de la matriz y id es el identificador de cada proceso.

Este proceso se realiza n veces como observamos en el algoritmo 13, donde j y k se inicializan en 0 antes de iniciar el ciclo *for*, así podremos obtener la fila necesaria en el momento que se requiera y nos asegura recorrer todas las filas.

Algoritmo 14 Obtener la fila requerida de la partición R que la contiene de las múltiples GPUs

Entrada: j, k, $d = n/np$ y id.

1: **si** $id == k$ **entonces**
2: Obtener la fila j de la partición R de id.
3: **termina si**
4: Enviar la fila de id a los demás procesos.
5: $j = j + 1$
6: **si** $j == d$ **entonces**
7: $j = 0$
8: $k = k + 1$.
9: **termina si**

5.2.2. Aplicar las rotaciones de Givens a la fila i con las múltiples GPU

Como sabemos aplicar las rotaciones de Givens a una fila afectara las filas inferiores también, por eso es necesario que todos los procesos tengan los datos de la fila a la que se le aplicaran las rotaciones de Givens, para así poder modificar los datos de cada partición de R y Q.

El proceso es muy similar al expuesto en la sección 5.1.3.2, aunque tiene algunos cambios para poder aplicar las rotaciones a cada partición. En el algoritmo 15 podemos ver los pasos que se deben de aplicar a cada partición de R y Q para obtener la partición con las rotaciones aplicadas.

Como podemos observar el algoritmo 15 es muy parecido al algoritmo 12, con algunos cambios muy importantes.

Un cambio que se puede observar, es en los índices de los valores de las particiones de R y Q, como ahora trabajamos sobre filas, cada hilo debe cambiar los valores de una fila en específico, sin este cambio no podríamos realizar la factorización QR con múltiples tarjetas de GPUs.

Otro cambio se observa en la línea 7, ya que en el algoritmo de una tarjeta tiene que recorrer toda la matriz por columnas, pero en este algoritmo tiene que recorrer solo el número de filas con las que cuenta cada partición, este número está representando

Algoritmo 15 Calcular rotaciones de Givens por fila en cada tarjeta de GPUs

Entrada: Particiones de R y Q, columna obtenida L, identificador fil de la fila a aplicar rotaciones de Givens, tamaño de matriz n, numero de proceso np

Salida: Particiones de R y Q con rotaciones de Givens aplicadas a la fila fil.

1: $i =$ identificador de hilo.
2: $j = n$.
3: **mientras** l_j sea 0 **hacer**
4: $\quad j = j - 1$
5: **termina mientras**
6: $tempj = j$
7: **mientras** $i < n/np$ **hacer**
8: $\quad \mu'_i = \sqrt{{l_{j-1}}^2 + {l_j}^2}$.
9: $\quad c = \frac{l_{j-1}}{\mu'_i}$ y $s = \frac{l_j}{\mu'_i}$.
10: $\quad$ **mientras** $j > fil$ **hacer**
11: $\quad\quad \mu_i = r_{i,j-1}$ y $\nu_i = r_{i,j}$.
12: $\quad\quad \alpha_i = q_{i,j-1}$ y $\beta_i = q_{i,j}$.
13: $\quad\quad r_{i,j-1} = c\mu_i + s\nu_i$ y $r_{i,j} = -s\mu_i + c\nu_i$.
14: $\quad\quad q_{i,j-1} = c\alpha_i + s\beta_i$ y $q_{i,j} = -s\alpha_i + c\beta_i$.
15: $\quad\quad j = j - 1$.
16: $\quad\quad a = \mu'_i$.
17: $\quad\quad \mu'_i = \sqrt{{l_{j-1}}^2 + a^2}$.
18: $\quad\quad c = \frac{l_{j-1}}{\mu'_i}$ y $s = \frac{a}{\mu'_i}$.
19: $\quad$ **termina mientras**
20: $\quad j = tempj$
21: $\quad$ recorrer i al final del bloque
22: **termina mientras**

por n/np, si no hiciéramos este cambio empezaríamos a acceder a memoria que no corresponde con la partición seleccionada, ocasionando problemas de memoria.

El algoritmo 15, hace referencia a una llamada a kernel que realiza cada una de las tarjetas, lo importante para que este algoritmo funcione es tener la fila a la que se le aplicaran las rotaciones en memoria, ya que sin ella el algoritmo no funcionaria, o solo funcionaria en la tarjeta donde se tengan los datos de la fila. En la figura 5.6 se puede observar que pasa en las particiones dentro de una llamada al kernel.

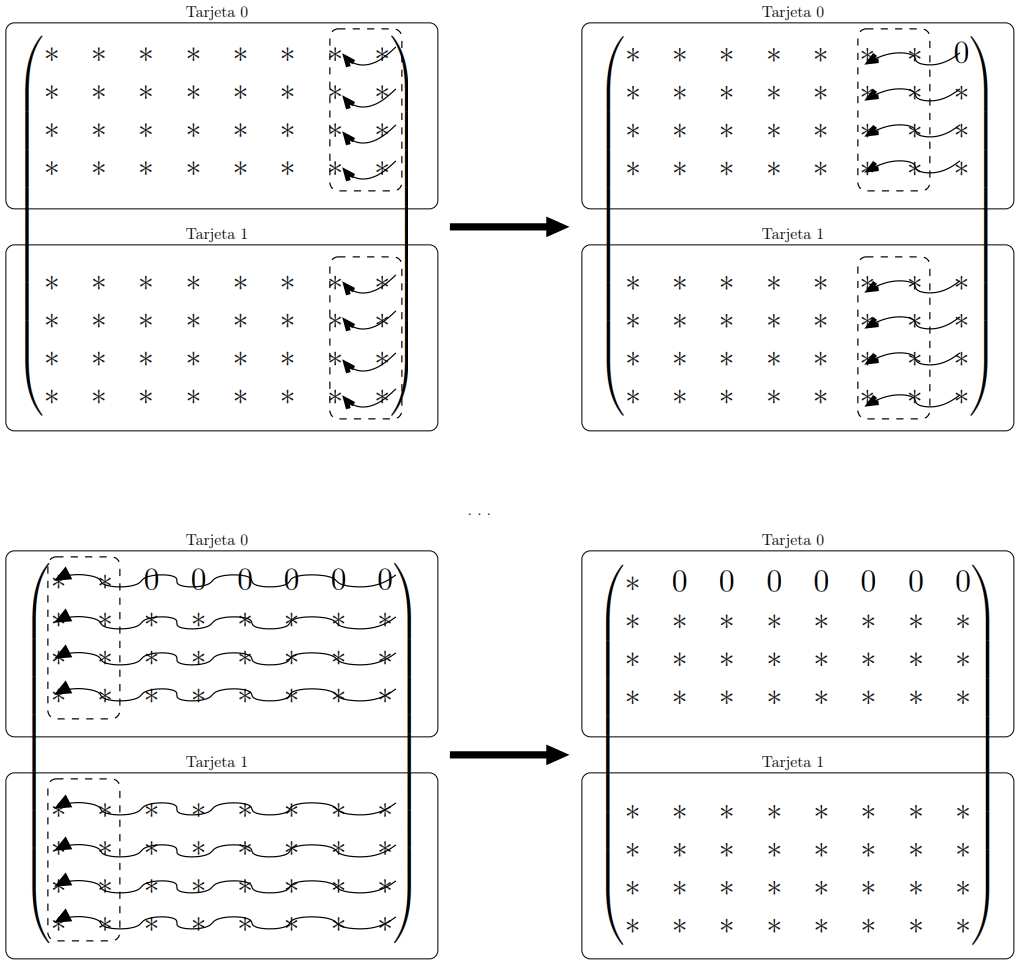

Figura 5.6: Proceso que realizan 2 tarjetas en una llamada a kernel.

El hecho de que las tarjetas realicen los mismos pasos, no quiere decir que todas terminen al mismo tiempo, de ahí la importación de que los procesos que ejecutan las funciones en las tarjetas esperen hasta que todos los procesos lleguen a un punto antes de llamar al kernel de las rotaciones de Givens.

Como en el método de una sola tarjeta, para terminar el método de Givens para múltiples tarjetas de GPUs por completo, cada tarjeta debe de hacer n llamadas al kernel antes explicado. En la figura 5.7 se puede revisar a grandes rasgos como funciona todo el método.

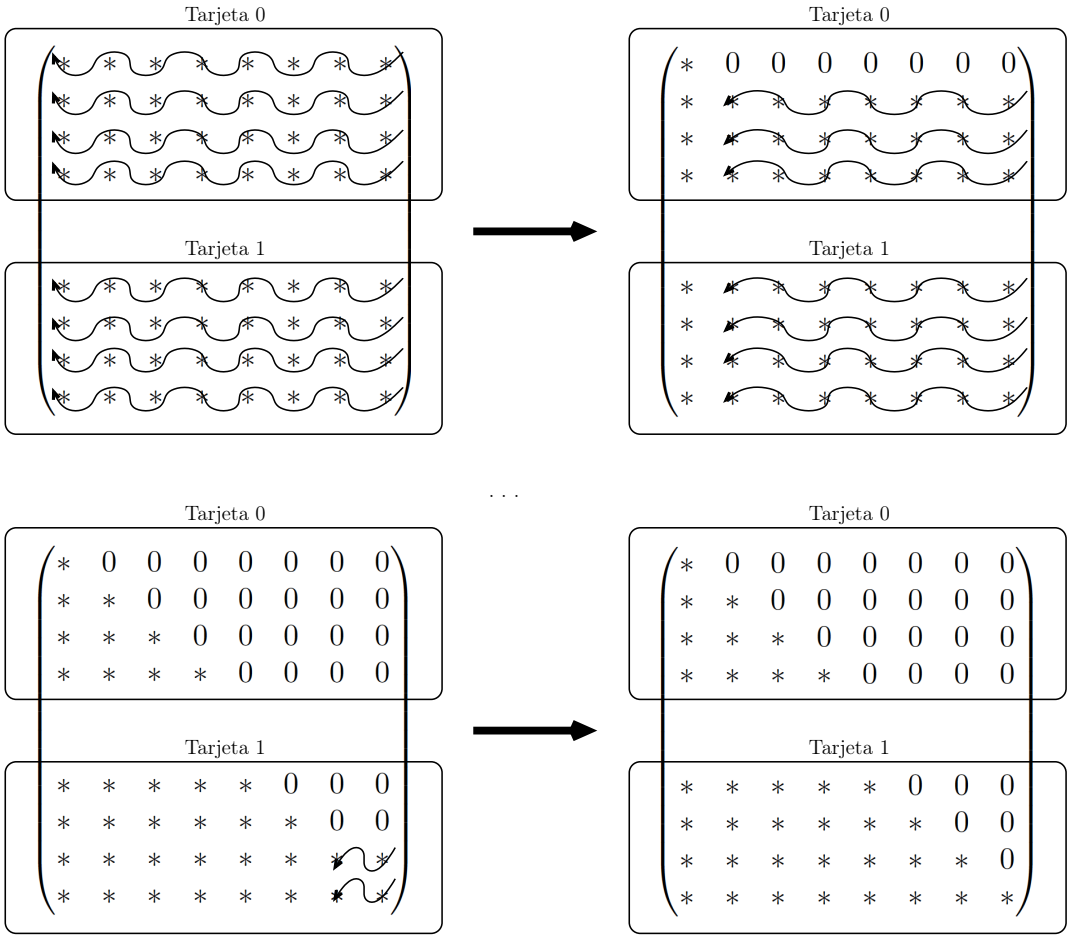

Figura 5.7: Método de Givens para 2 tarjetas de GPUs en la matriz R.

Se pude observar en la figura que después de la mitad del proceso una de las tarjetas, la que tiene la parte superior de la matriz, ya no realiza ninguna operación, pero esto no es así, ya que si bien no trabaja en la partición de R, en la partición de Q si trabaja, ya que llamada al kernel cambia la partición de Q. En la figura 5.8 se observa como aun después de la mitad del proceso siguen trabajando las dos tarjetas.

Similar al método en una sola tarjeta, si el número de filas de cada partición es mayor al número de hilos, entonces los hilos comienzan a recorrerse para cubrir por completo las particiones. Así no es necesario que tengamos en mismo número de hilos

que de filas por partición, esto aún toma más importancia ya que se puede dar el caso de que las dos tarjetas sin diferentes, por lo que no podemos estar cambiando el número de hilos en la ejecución del programa.

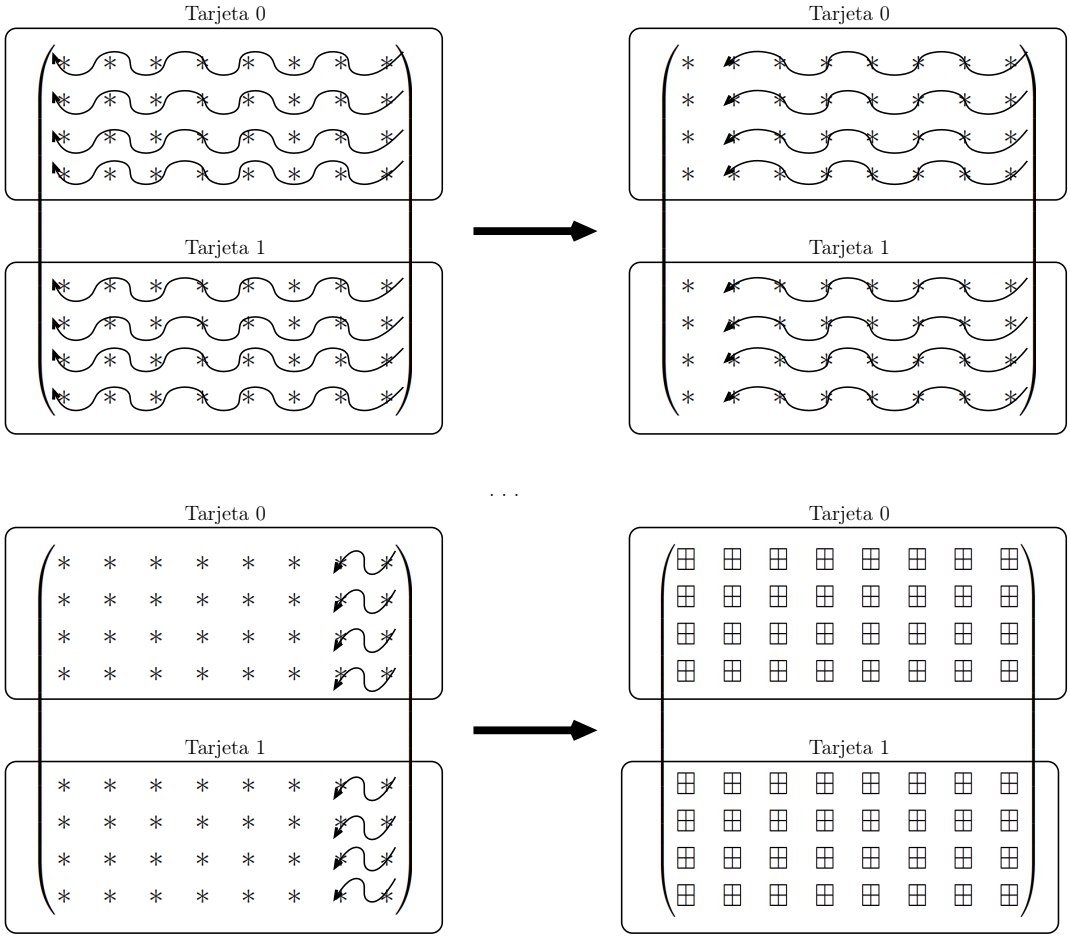

Figura 5.8: Método de Givens para 2 tarjetas de GPUs en la matriz Q.

Capítulo 6

Pruebas

En este capitulo explicaremos las pruebas realizadas de los algoritmos expuestos en el capitulo anterior, así como las comparaciones que se realizaron con un método secuencial de la factorización QR con el método de Givens, y comparaciones con la biblioteca MKL. También se describirá la infraestructura que se utilizo para poder llevar a cabo las pruebas.

6.1. Infraestructura

El conjunto de hardware utilizado en esta tesis es el siguiente:

- Se utilizaron 2 procesadores **Intel Xeon X5675** [19] para la parte de pruebas de MKL. Cada procesador cuenta con:
    - Numero de Cores: 6.
    - Velocidad de reloj: 3.06 GHz.
    - Ancho de Banda de Memoria: 32 GB/s.
- Y para la parte de pruebas con CUDA se utilizaron 3 tarjetas graficas de GPUs diferentes, que se describirán a continuación:
    - 2 Tarjetas **NVIDIA Tesla K20X** [20] con las siguientes especificaciones cada una:

- Memoria RAM: 6 GB.
- CUDA Cores: 2688.
- Velocidad de reloj por core: 732 MHz.
- Ancho de Banda de Memoria: 250 GB/s.

- 1 Tarjeta **NVIDIA Tesla C2070** [21] con las siguientes especificaciones cada una:
  - Memoria RAM: 6 GB.
  - CUDA Cores: 448.
  - Velocidad de reloj por core: 1.15 GHz
  - Ancho de Banda de Memoria: 144 GB/s.
- 1 Tarjeta **NVIDIA GeForce GTX 460** [22] con las siguientes especificaciones cada una:
  - Memoria RAM: 1 GB.
  - CUDA Cores: 336.
  - Velocidad de reloj por core: 1.53 GHz.
  - Ancho de Banda de Memoria: 115.2 GB/s.
  - Cuenta con *overclocking*

6.2. Pruebas de la factorización QR con el método de Givens y de la diagonalizacion en una sola tarjeta

Se realizaron 2 pruebas diferentes del algoritmo. Lo que cambia en las pruebas son el tipo de dato, en una prueba se utiliza precisión sencilla y en la otra precisión doble. Así mismo, se realizaron pruebas con precisión sencilla y doble de MKL, para poder obtener las dos matrices Q y R de la factorización QR con MKL, se utilizan 2 funciones que son: ***?geqrf***[1], la cual nos permite obtener la matriz R^T y la función

[1]Donde el signo ***?*** cambia por una ***s*** si es precisión sencilla o una ***d*** si es precisión doble.

?orgqr que se debe de utilizar después de la función *?geqrf* para obtener la matriz Q^T, las pruebas de MKL se ejecutaron en los 2 procesadores Intel Xeon X5675.

En cuanto a las matrices de pruebas, se utilizaron matrices tridiagonales, pentadiagonales y heptadiagonales. Con precisión sencilla se utilizaron matrices desde 5000×5000 hasta 20000×20000, que es el tamaño máximo que procesaron las tarjetas Tesla K20X y C2070, mientras que la tarjeta GeForce el tamaño máximo que proceso fue de 10000×10000. Utilizando precisión doble[2] las tarjetas Tesla llegaron a un máximo de 16000×16000 y la GeForce fue de 8000×8000. En cuanto a la biblioteca MKL el tamaño máximo de matriz depende de la memoria disponible en la computadora que se realicen las pruebas.

6.2.1. Comparación de tiempos de ejecución entre un método secuencial y un paralelo

En la figura 6.1 podemos observar la diferencia de tiempos de ejecución (seg.) entre un método secuencial y un paralelo, aplicados a una matriz densa simétrica. No se hacen comparaciones con matrices de otros tipos ya que el método secuencial no cambiaria sus tiempos de ejecución porque siempre realizaría los mismos pasos.

En la tabla 6.1 podemos ver mas detalladamente la diferencia de tiempos de ejecución. Como podemos observar el método paralelo en la tarjeta Tesla C2070 tiene una aceleración de 5x a comparación del método secuencial. La tarjeta GeForce GTX 640 tiene una aceleración de 2x con respecto a la tarjeta Tesla C2070 y de 8x con respecto al método secuencial. Y el que menor tiempo de ejecución realizo en todos los tamaños fue la tarjeta Tesla K20X con una aceleración de 22x contra el método secuencial, de 3x con respecto a la tarjeta Tesla C2070 y de 2x contra la GeForce GTX 640.

En conclusión, utilizar CUDA para paralelizar un método secuencial, en algunos casos, como este, mejora considerablemente el tiempo de ejecución.

[2]No todas las tarjetas de NVIDIA soportan la precisión doble.

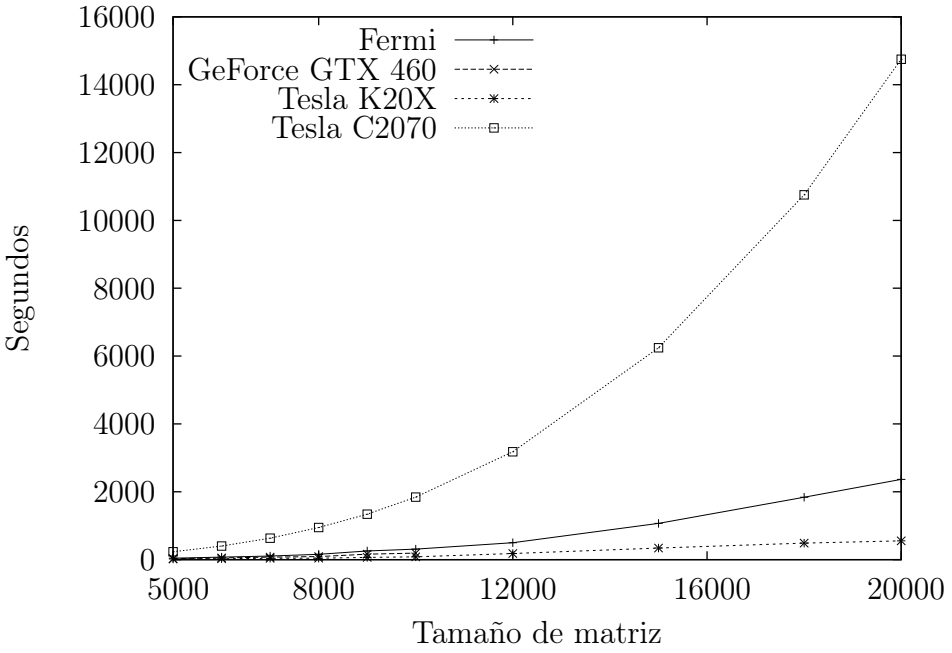

Figura 6.1: Comparación de tiempos entre un algoritmo secuencial y un paralelo.

Tabla 6.1: Tiempos de ejecución de la factorización QR de matrices densas simétricas entre un método paralelo y uno secuencial.

	Tiempos de ejecución (seg.)			
Tamaño de matriz	**Secuencial**	**C2070**	**GTX 640**	**K20X**
5000	231.3800375	42.3356905	26.82115675	19.815541
6000	400.3917195	75.89001675	51.1115865	26.80524767
7000	633.6835165	107.9252775	76.0930965	38.76723133
8000	946.367037	159.0360155	89.91713225	43.24792
9000	1342.084252	257.1983995	160.8427493	68.576285
10000	1841.883651	310.9070375	193.3412333	82.548421
12000	3180.501266	499.4751063	-	181.6179507
15000	6246.546066	1070.635072	-	342.0601173
18000	10754.48158	1840.087913	-	487.6856317
20000	14751.39375	2366.96302	-	552.654966

6.2.2. Comparación de tiempos de ejecución entre MKL y CUDA

En esta sección compararemos los tiempos de ejecución de las diferentes tarjetas contra la biblioteca MKL utilizando 6 y 12 hilos respectivamente. Los tipos de matrices serán los antes expuesto (tridiagonal, pentadiagonal y heptadiagonal).

6.2.2.1. Precisión Sencilla

Como se hacen comparaciones con 3 tipos diferentes de matrices tenemos 3 figuras para visualizar los resultados de cada comparación. En la figura 6.2 podemos observar los tiempos de ejecución de cada una de los algoritmos que se utilizaron con matrices tridiagonales, en la figura 6.3 con matrices pentadiagonales y por ultimo en la figura 6.4 con matrices heptadiagonales.

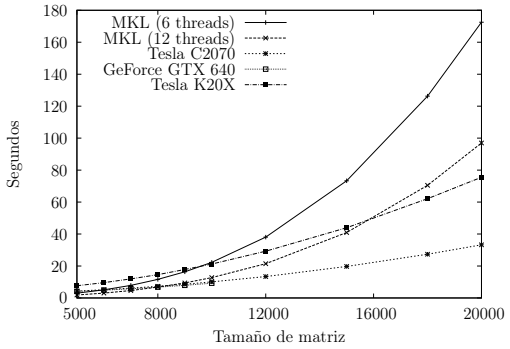

Figura 6.2: Comparación de tiempos de ejecución con matrices tridiagonales de precisión sencilla.

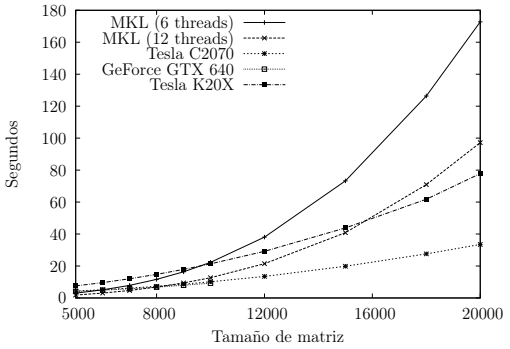

Figura 6.3: Comparación de tiempos de ejecución con matrices pentadiagonales de precisión sencilla.

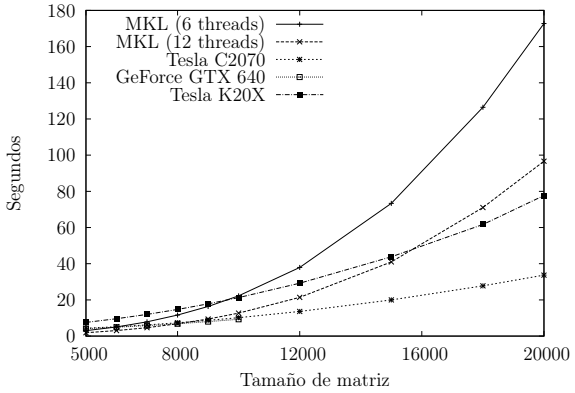

Tabla 6.2: Tiempos de ejecución con matrices tridiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)				
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070**	**GTX 640**	**K20X**
5000	3.0045	1.8481	4.3077	4.1303	7.6315
6000	5.0694	3.0561	5.1580	4.8519	9.5890
7000	7.8770	4.6502	6.1510	5.7411	11.9269
8000	11.6177	6.7874	7.2756	6.7445	14.5967
9000	16.3681	9.4189	8.5882	7.9250	17.8038
10000	22.2853	12.7146	10.0688	9.2263	21.2093
12000	38.0549	21.4291	13.4137	-	29.2787
15000	73.2948	40.9591	19.7237	-	43.9365
18000	126.2861	70.4938	27.3739	-	62.1312
20000	172.5314	96.9515	33.2559	-	75.5187

Figura 6.4: Comparación de tiempos de ejecución con matrices heptadiagonales de precisión sencilla.

En las tablas siguientes podemos ver los tiempos mas a fondo para poder hacer un análisis de que tanto es mejor el método paralelo a comparación de la biblioteca MKL.

En las tablas 6.2, 6.3 y 6.4, podemos observar que no hay mucha diferencia de tiempo entre los diferentes tipos de matrices. Al observar los datos mas a fondo calculamos la aceleración que cada una de las tarjetas tiene sobre MKL.

La tarjeta K20X, que es la que menor aceleración presenta, tiene una aceleración de 1.2x sobre MKL con 6 hilos, que comienza desde un tamaño de matriz de 12000 y que aumenta conforme al tamaño, llegando a una aceleración de 2.2x en el tamaño máximo soportado. La tarjeta GTX 640 comienza a acelerar en el tamaño de matriz de 7000 con 1.4x y en su tamaño máximo tiene una aceleración de 2.4x. Por ultimo la tarjeta que mejor rendimiento tuvo, la tarjeta C2070, comienza la aceleración en el tamaño de matriz 7000 con 1.2x y en el tamaño máximo llega a 5.1x de aceleración.

Tabla 6.3: Tiempos de ejecución con matrices pentadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)				
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070**	**GTX 640**	**K20X**
5000	3.0063	1.8320	4.2963	4.1129	7.5805
6000	5.0634	3.0542	5.1577	4.8690	9.5908
7000	7.8828	4.6900	6.1472	5.7613	12.0301
8000	11.609	6.7897	7.3053	6.7805	14.7572
9000	16.3434	9.4163	8.6348	7.9772	17.8517
10000	22.2779	12.6622	10.1132	9.2527	21.3425
12000	38.0562	21.4619	13.4933	-	29.1947
15000	73.2686	40.8407	19.8355	-	43.8219
18000	126.3450	70.9044	27.5742	-	61.7743
20000	172.7621	97.1986	33.4897	-	77.8357

Tabla 6.4: Tiempos de ejecución con matrices heptadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)				
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070**	**GTX 640**	**K20X**
5000	3.0051	1.8548	4.3150	4.1276	7.5679
6000	5.0574	3.0540	5.1811	4.8931	9.5833
7000	7.8871	4.6640	6.1752	5.7757	11.9670
8000	11.6219	6.7953	7.3455	6.8234	14.7088
9000	16.3615	9.4289	8.6898	7.9916	17.8126
10000	22.2499	12.7019	10.1533	9.2894	21.2515
12000	37.9285	21.3790	13.5988	-	29.2172
15000	73.3422	40.9672	19.9869	-	43.8386
18000	126.4499	71.0379	27.7713	-	61.6807
20000	172.7288	96.6412	33.6774	-	77.6769

Ahora comparando con MKL con 12 hilos, la tarjeta K20x comienza a presentar mejores resultados hasta un tamaño de 18000 con 1.15x y llegando a 1.24x en el tamaño máximo soportado. La tarjeta GTX 640 empieza con mejores resultados en un tamaño de matriz de 9000 con 1.17x y el tamaño máximo soportado de 1.36x de aceleración. La tarjeta C2070 comienza a presentar mejoras con respecto a MKL en el tamaño de matriz de 10000 con una aceleración 1.2x y llegando a una aceleración de 2.8x en el tamaño máximo soportado.

6.2.2.2. Precisión Doble

Como en la sección anterior se cuenta con 3 graficas diferentes representadas en las figuras 6.7 para tridiagonales, 6.5 para pentadiagonales y 6.6 para heptadiagonales.

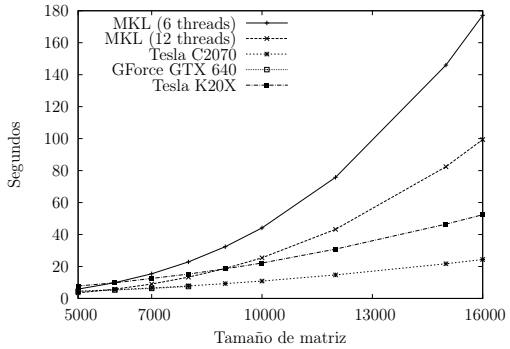

Figura 6.5: Comparación de tiempos de ejecución con matrices pentadiagonales de precisión doble.

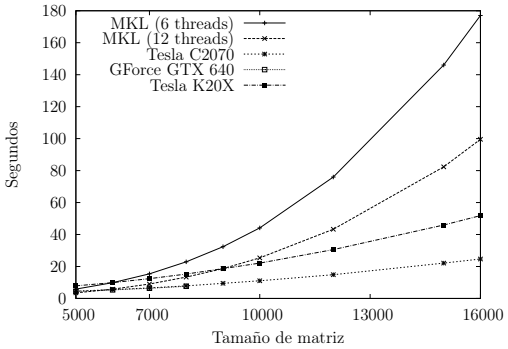

Figura 6.6: Comparación de tiempos de ejecución con matrices heptadiagonales de precisión doble.

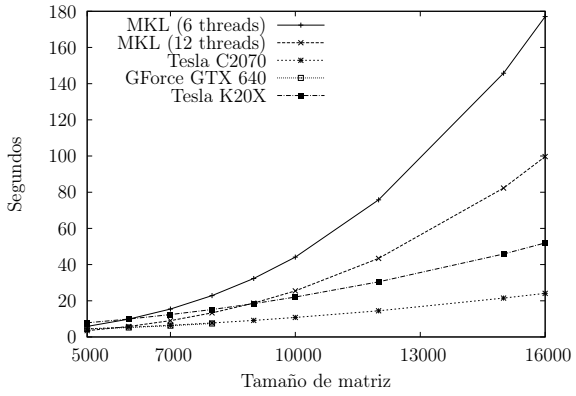

Figura 6.7: Comparación de tiempos de ejecución con matrices tridiagonales de precisión doble.

Tabla 6.5: Tiempos de ejecución con matrices tridiagonales de precisión doble.

	Tiempos de ejecución (seg.)				
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070**	**GTX 640**	**K20X**
5000	5.8173	3.4943	4.5033	4.4015	7.8342
6000	9.8515	5.8032	5.3920	5.2230	9.9040
7000	15.4112	9.0226	6.4559	6.2406	12.3821
8000	22.8717	13.3011	7.7425	7.4096	15.1459
9000	32.3264	18.6843	9.1607	-	18.3827
10000	44.1123	25.4568	10.8074	-	22.0613
12000	75.7752	43.4161	14.4710	-	30.4697
15000	145.7767	82.2894	21.4738	-	45.8728
16000	177.0847	99.7117	24.0774	-	51.9599

Así como en la sección anterior, tal vez en las figuras no podemos observar bien la diferencia entre MKL y las tarjetas de NVIDIA. Por lo tanto se presentaran las tablas con los tiempos para poder estudiarla mas a fondo.

Haremos un análisis de las tablas 6.5, 6.6 y 6.7, para observar la aceleración que las tarjetas de GPUs tienen sobre la biblioteca MKL, utilizando matrices tridiagonales, pentadiagonales y heptadiagonales.

Ya que los tiempos entre los diferentes tipos de matrices es casi igual, se comparan como si fuera un solo tiempo. La tarjeta GTX 640, la que tiene mas restricción de memoria, muestra aceleración con respecto a MKL con 6 hilos, desde un inicio con una aceleración de 1.32x, llegando a una aceleración de 3x en el tamaño máximo soportado por la tarjeta. Continuando la comparación con MKL con 6 hilos, la tarjeta C2070 al igual que la GTX 640 muestra una aceleración de 1.29x desde el inicio, llegando a una aceleración de 7.3x en el tamaño máximo. La tarjeta con menor aceleración, la K20x, comienza a mostrar aceleración en el tamaño de 7000 con 1.24x, llegando a una aceleración de 3.4x.

Con respecto a MKL con 12 hilos, la tarjeta GTX 640 comienza a mostrar una aceleración de 1.1x en el tamaño de 6000 y llega a 1.7x en el tamaño máximo de la

Tabla 6.6: Tiempos de ejecución con matrices pentadiagonales de precisión doble.

	Tiempos de ejecución (seg.)				
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070**	**GTX 640**	**K20X**
5000	5.8158	3.4775	4.4989	4.3981	7.8009
6000	9.8565	5.8132	5.4606	5.2663	9.8370
7000	15.4283	9.0258	6.5283	6.2941	12.5160
8000	22.8503	13.3111	7.8142	7.4928	15.2604
9000	32.3665	18.7568	9.2688	-	18.5411
10000	44.1126	25.4078	10.8738	-	22.1337
12000	75.7890	43.2624	14.7159	-	30.8494
15000	145.9933	82.4304	21.7310	-	46.4527
16000	177.1076	99.4067	24.3618	-	52.3985

Tabla 6.7: Tiempos de ejecución con matrices heptadiagonales de precisión doble.

	Tiempos de ejecución (seg.)				
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070**	**GTX 640**	**K20X**
5000	5.8565	3.4717	4.5329	4.4050	7.8865
6000	9.8636	5.7834	5.5078	5.3335	9.8932
7000	15.4072	9.0010	6.5921	6.3543	12.4012
8000	22.8946	13.3381	7.8820	7.5999	15.2716
9000	32.3599	18.6786	9.4494	-	18.6408
10000	44.1399	25.3754	11.0434	-	22.1242
12000	75.9765	43.2940	14.8612	-	30.5623
15000	146.037	82.3424	22.1180	-	45.9491
16000	177.032	99.5883	24.6834	-	51.7987

tarjeta. La tarjeta C2070 comienza a acelerar en el tamaño de 7000 con 1.3x y llega a 4.14x de aceleración. La K20X comienza su aceleración en el tamaño de 10000 con 1.15x y llega a casi un 2x de aceleración.

6.2.3. Diagonalización de matrices en una sola tarjeta

Los tiempos obtenidos con la implementacion realizada, no fueron nada buenos comparados con MKL, esto ocurrio debido al algoritmo utilizado, el algoritmo QR, ya que le lleva demasiadas iteraciones llegar a una matriz diagonal.

Los tiempos que toma son muy grandes desde matrices pequeñas. La tabla 6.8 muestra el tiempo de ejecucion y el numero de iteraciones que se llevaron para llegar a la matriz diagonal.

Tabla 6.8: Tiempos de ejecución e iteraciones necesarias de la diagonalizacion de matrices.

Tamaño	**Tiempo de ejecución**	**Numero de iteraciones**
50	4.947	4946
100	66.9	48859

6.3. Pruebas de la factorización QR con múltiples tarjetas de GPUs

Las pruebas de la factorización QR que se implemento para múltiples tarjetas, explicada en la sección 5.2 de la página 55, se llevaron a cabo con 2 conjuntos de tarjetas. Una de las pruebas se efectuó utilizando la tarjeta Tesla C2070 y la tarjeta GTX 640, se realizaron pruebas de precisión sencilla y doble. Otra prueba se realizo utilizando 2 tarjetas Tesla K20X y solo se realizaron pruebas de precisión sencilla. Todas las pruebas siempre se realizaron comparando con MKL con 6 y 12 hilos.

6.3.1. Factorización QR con 2 tarjetas Tesla K20X

Como solo se realizaron pruebas con precisión sencilla, solo tenemos 3 graficas diferentes, para matrices tridiagonales, pentadiagonales y heptadiagonales, que se pueden ver en 6.8, 6.9 y 6.10, donde se compara con MKL con 6 y 12 hilos.

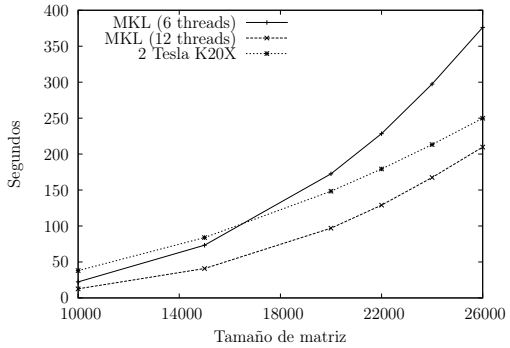

Figura 6.8: Comparación de tiempos de ejecución de la factorización QR con 2 tarjetas K20x con matrices tridiagonales de precisión sencilla.

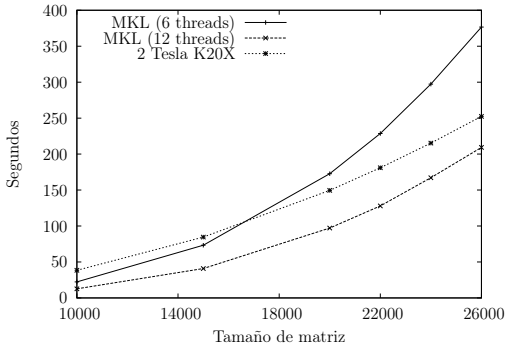

Figura 6.9: Comparación de tiempos de ejecución de la factorización QR con 2 tarjetas K20x con matrices pentadiagonales de precisión sencilla.

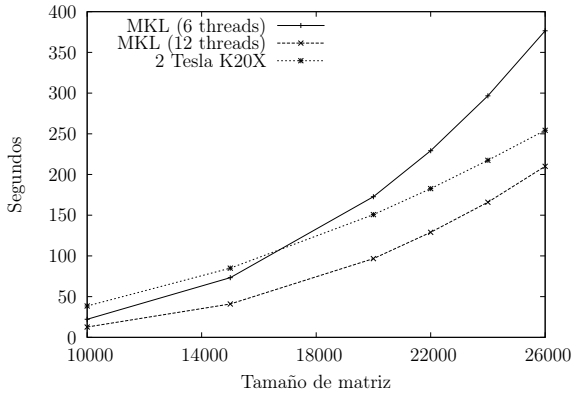

Figura 6.10: Comparación de tiempos de ejecución de la factorización QR con 2 tarjetas K20X con matrices heptadiagonales de precisión sencilla.

Como podemos ver en las figuras, el comportamiento de las 2 tarjetas Tesla K20X esta en un termino medio entre MKL con 6 y 12 hilos. Una ventaja de esta implementación es el uso de mas de una tarjeta de NVIDIA, lo que se traduce, en el caso de esta tesis, en el aumento del tamaño de matriz soportada por una matriz. En el caso de las 2 tarjetas Tesla K20X aumenta a un tamaño de 26000.

Tabla 6.9: Tiempos de ejecución de la factorización QR con 2 tarjetas K20X con matrices tridiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**2 K20X**
10000	22.28536925	12.7146785	38.045183
15000	73.2948375	40.89761575	83.9473015
20000	172.5314735	96.95154125	148.3730815
22000	228.4494833	129.0123748	179.3089645
24000	297.469622	167.4144483	213.186617
26000	376.0774375	209.7476243	249.858848

En las tablas 6.9, 6.10 y 6.11 se muestran los tiempos de manera mas amplia, que obtuvo cada programa. Para poder analizarlos mas a fondo.

En las tablas anteriores podemos observar que las 2 tarjetas K20X no logran una aceleración sobre MKL con 12 hilos. Y con MKL con 6 hilos comienzan a acelerar entre los tamaños 15000 y 20000, empezando con una aceleración de 1.15x y llegando a 1.48x de aceleración el tamaño de 26000.

6.3.2. Factorización QR con una tarjeta Tesla C2070 y una GTX 640

Estas pruebas se realizaron con los 3 tipos de matrices diferentes y con precisión sencilla y doble. En el caso de la precisión sencilla, tomando en cuenta la tarjeta de menor memoria, la GTX 640, el tamaño aumento a 18000. En el caso de la precisión doble, el tamaño aumenta a 12000.

6.3.2.1. Precisión sencilla

Tenemos 3 figuras 6.11, 6.12 y 6.13, cada uno representa a un tipo de matriz.

Tabla 6.10: Tiempos de ejecución de la factorización QR con 2 tarjetas K20X con matrices pentadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**2 K20X**
10000	22.2779915	12.662252	38.3830495
15000	73.26862925	40.84078725	84.603206
20000	172.7621153	97.198688	149.5560425
22000	228.6138805	127.9655578	180.9614535
24000	297.276558	167.0996915	215.250591
26000	376.657193	209.2388725	252.3788475

Tabla 6.11: Tiempos de ejecución de la factorización QR con 2 tarjetas K20X con matrices heptadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**2 K20X**
10000	22.24998125	12.70196825	38.5489915
15000	73.34226025	40.967238	84.951077
20000	172.7288738	96.6412415	150.7447375
22000	229.2751283	129.0128638	182.637326
24000	296.6546478	165.9274763	217.4389965
26000	376.5570043	210.0910878	254.3220125

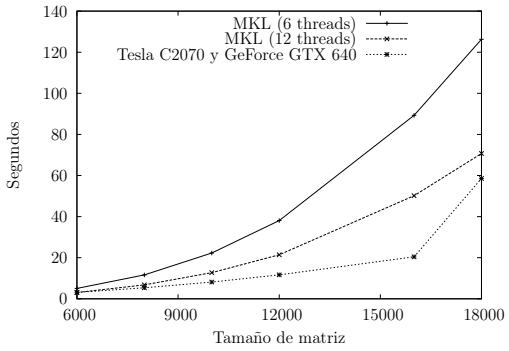

Figura 6.11: Comparación de tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices tridiagonales de precisión sencilla.

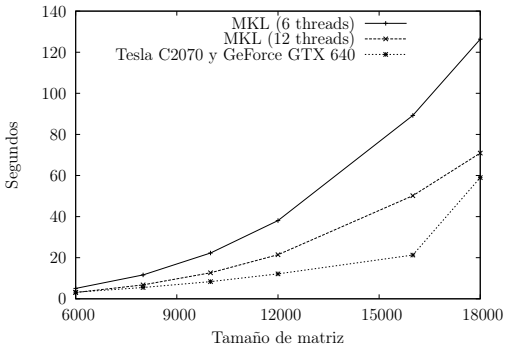

Figura 6.12: Comparación de tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices pentadiagonales de precisión sencilla.

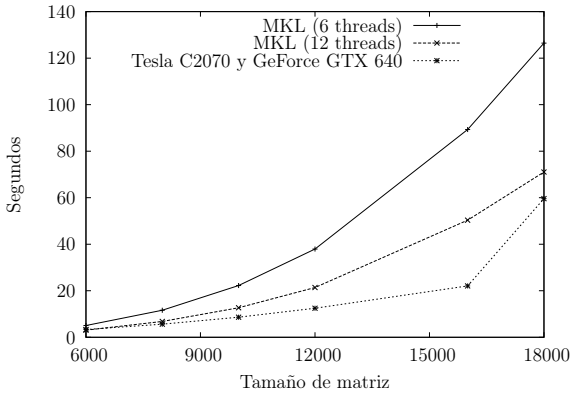

Figura 6.13: Comparación de tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices heptadiagonales de precisión sencilla.

En las tablas 6.12, 6.13 y 6.14, podemos observar que el método que ocupa las tarjetas C2070 y la GTX 640, acelera desde un principio a la biblioteca de MKL con 6 hilos, aunque es poco, pero acelera, y en el tamaño de matriz máximo llega a una aceleración de 2.14x. En cuanto a MKL con 12 hilos, comienzan con tiempos similares y poco a poco va acelerando hasta alcanzar un 1.2x de aceleración. En este caso contrario a las 2 tarjetas K20X, logra obtener mejores resultados que MKL con 6 y 12 hilos.

Tabla 6.12: Tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices tridiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070 y GTX 640**
6000	5.069462	3.0561085	3.133921
8000	11.61772625	6.78740725	5.348618
10000	22.28536925	12.7146785	8.145723
12000	38.054924	21.429109	11.670436
16000	89.2726135	50.158922	20.4117655
18000	126.2861433	70.7047495	58.549526

Tabla 6.13: Tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices pentadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070 y GTX 640**
6000	5.063496	3.05424775	3.182458
8000	11.609	6.78976425	5.4918455
10000	22.2779915	12.662252	8.3566295
12000	38.05622425	21.4619395	12.122634
16000	89.23257225	50.18168125	21.2946535
18000	126.345051	70.90448025	58.9706655

Tabla 6.14: Tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices heptadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070 y GTX 640**
6000	5.05746	3.05409	3.343184
8000	11.62196075	6.79531825	5.625851
10000	22.24998125	12.70196825	8.625267
12000	37.928559	21.37907675	12.4785045
16000	89.31594025	50.3303555	22.0926865
18000	126.449971	71.03795625	59.6142455

6.3.2.2. Precisión Doble

La figuras siguientes nos muestran los resultado obtenidos utilizando las tarjetas C2070 y la GTX 640 con matrices de precisión doble, donde 6.14, 6.15 y 6.16 corresponden a cada tipo de matriz diferente.

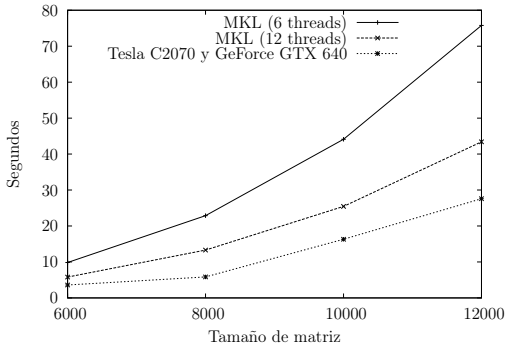

Figura 6.14: Comparación de tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices tridiagonales de precisión doble.

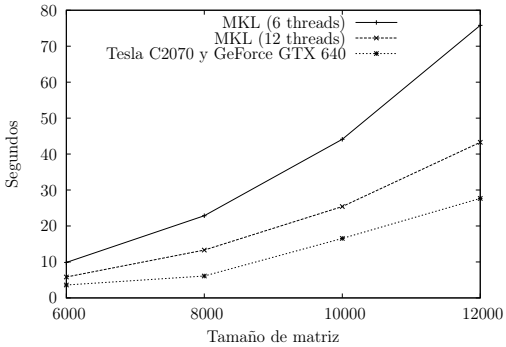

Figura 6.15: Comparación de tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices pentadiagonales de precisión doble.

Tabla 6.15: Tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices tridiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070 y GTX 640**
6000	9.85151275	5.80324625	3.589367
8000	22.8717055	13.301183	5.8313435
10000	44.11235	25.4568875	16.3190995
12000	75.77521025	43.41613875	27.5913185

Tabla 6.16: Tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices pentadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070 y GTX 640**
6000	9.85652425	5.813216	3.603074
8000	22.85033125	13.31117925	6.088403
10000	44.11266525	25.407812	16.53908
12000	75.789095	43.262438	27.6592505

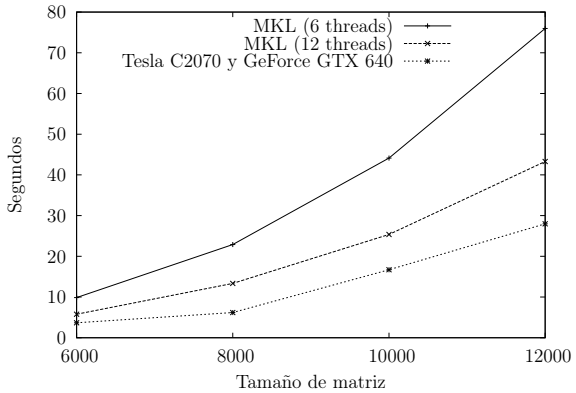

Figura 6.16: Comparación de tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices heptadiagonales de precisión doble.

Tabla 6.17: Tiempos de ejecución de la factorización QR con una tarjeta C2070 y una GTX 640 con matrices heptadiagonales de precisión sencilla.

	Tiempos de ejecución (seg.)		
Tamaño	**MKL 6 hil.**	**MKL 12 hil.**	**C2070 y GTX 640**
6000	9.86363625	5.7834095	3.6662875
8000	22.8946235	13.33819875	6.1803205
10000	44.13997175	25.37548475	16.696302
12000	75.97651125	43.29401675	27.9818585

En las tablas 6.15, 6.16 y 6.17, podemos observar que el método implementado utilizando las tarjetas C2070 y la GTX 640, obtenemos mejores resultados desde un principio, obteniendo 2.7x de aceleración sobre MKL 6 hilos y 1.6x sobre MKL con 12 hilos. En el tamaño máximo se mantienen las aceleraciones del principio un 2.7x sobre MKL con 6 hilos y de 1.6x sobre MKL con 12 hilos.

Capítulo 7

Resultados, Discusión, Conclusiones y Trabajo a Futuro

7.1. Resultados

- Se implemento la factorización QR utilizando el método de Givens para una tarjeta grafica de GPUs.
- Se implemento la diagonalización de matrices utilizando el algoritmo iterativo QR para una tarjeta grafica de GPUs.
- Se implemento la factorización QR utilizando el método de Givens para múltiples tarjetas graficas de GPUs.
- Se hicieron pruebas de tiempo con la biblioteca MKL para la factorización QR y la diagonalización de matrices.
- Se compararon los tiempos de ejecución de MKL y las implementaciones realizadas.
- Se estudiaron las diferentes estrategias de particionamiento de datos para elegir las que mas se adaptaran a esta tesis.
- Se observo que utilizar MPI es suficiente para poder utilizar mas de una tarjeta de GPUs.

7.2. Conclusiones

Es posible la implementación de la diagonalización de matrices en una sola tarjeta, como se observo en las pruebas, en cuestión de tiempos de ejecución, esta muy por debajo de la biblioteca MKL. Aunque aun esta la duda del consumo energético de cada implementación.

En cuanto a la implementación para múltiples tarjetas, no se pudo realizar por dos motivos principales, uno, como se observo en la implementación en una sola tarjeta, no se obtuvieron buenos resultados. El segundo motivo es la multiplicación de matrices, ya que CUBLAS, la biblioteca utilizada para multiplicación de matrices, no trabaja bien con MPI y el particionamiento de datos no es el mismo que utilizamos para la factorización QR.

La implementación en una sola tarjeta de la factorización QR, como vimos en la sección anterior, se comporto de mejor manera que la biblioteca MKL, en cuestión de tiempos de ejecución. Donde la tarjeta con mejor rendimiento fue la GTX 640, esto debido a que cuenta con un *overclocking*, aunque esta tarjeta es la que cuenta con mayor restricción. La tarjeta C2070 es la que sigue en rendimiento, teniendo un rendimiento de uno o dos segundo mas que la GTX 640, aunque la restricción de memoria es mucho mayor a esta. Y por ultimo la K20X es la de menor rendimiento.

Al utilizar matrices de precisión sencilla y doble, mostramos que los resultados como se esperaba, se tardan unos segundos mas en la precisión doble que la sencilla, esto debido a que las tarjetas de GPUs deben utilizar 2 CUDA cores para la precisión doble, ya que solo cuentan con una unidad de precisión flotante cada core.

La tarjeta K20X teóricamente es la que mayor rendimiento puede aportar de las 3 tarjetas probadas, aunque uno de los problemas que se encontró y por el cual fue la que mas se tardo en las pruebas, fue un problema de comunicación entre el *host* y el dispositivo, que en los casos de las pruebas con esta tarjeta tomaba la mitad o en algunos casos mas de la mitad del tiempo. Mientras que en las tarjetas C2070 y GTX 640 la comunicación es menor y por lo tanto los resultados también.

Con la implementación de la factorización QR para múltiples tarjetas, observamos que utilizar MPI es suficiente para escalar a múltiples tarjetas, y que es la mejor opción en casos donde se tengas muchos nodos con tarjetas de GPUs, ya que la tecnología de GPUDirect funcionar bien pero para un numero limitado de tarjetas.

Revisando las pruebas de la implementación para múltiples tarjetas, observamos que el rendimiento de las 2 tarjetas K20X esta en un termino medio con MKL con 6 y 12 hilos, debido a lo que habíamos comentado del tiempo de la comunicación. Mientras que la combinación de la GTX 640 y la C2070 logran un mejor rendimiento que MKL.

Implementaciones de la diagonalización de matrices y la factorización QR para una sola tarjeta ya existen bibliotecas que las contienen, MAGMA y CULA, las implementaciones de la diagonalización están basadas en un método directo. Pero una implementación de una factorización QR para múltiples tarjetas de GPUs en la actualidad no existen, por lo que es una gran innovación para el campo del supercopita.

En conclusión, MKL para la diagonalización es mucho mas rápido que la implementación de una tarjeta, ya que el algoritmo QR requiere de muchas iteraciones para lograr diagonalizar una matriz. En cuestión de la factorización QR, se observo que la implementación de una tarjeta es mas rápida que MKL, mientras que la implementación de múltiples tarjetas es mas rápida con la combinación de la tarjeta GTX 640 y la C2070, sin embargo con las 2 tarjetas K20X esta en un termino medio.

Las contribuciones que se lograron de esta tesis son las siguientes:

- Diagonalización de matrices en una tarjeta de GPUs.
- Factorización QR con el método de Givens para una tarjeta de GPUs.
- Revisar si una matriz es diagonal para una tarjeta de GPUs.
- Factorización QR con el método de Givens para múltiples tarjetas de GPUs.
- Estrategia de particionamiento para la factorización QR con el método de Givens.
- Una política de asignación de tarjetas utilizando MPI.

7.3. Trabajo a Futuro

Implementar la diagonalización de matrices, con un mecanismo efectivo de multiplicación de matrices, con el fin de utilizar la particiones de las matrices, sin la necesidad de descargar los datos al *host* y después volver a subir la información al dispositivo.

Medir el consumo energético de las diferentes implementaciones y comparar con la biblioteca MKL, así como los flops por segundo que obtiene cada implementación. También se podría comparar contra otra biblioteca que trabaje en GPUs como es MAGMA o CULA.

Estudiar e intentar implementar si es posible un método directo para la diagonalización de matrices que utilice múltiples tarjetas de GPUs.

Implementar la factorización QR y la diagonalización de matrices utilizando la tecnología de GPUDirect para utilizar múltiples tarjetas de GPUs, para medir el rendimiento y que tanta escalabilidad se logra con esta tecnología.

Bibliografía

[1] Françoise Tisseur and Karl Meerbergen. The quadratic eigenvalue problem. *Society for Industrial and Applied Mathematics*, 43(2):235–286, 2001.

[2] Multipath Corporation. *Why GPUs?*, 2011. URL: `http://www.fmslib.com/mkt/gpus.html`.

[3] Haicheng Qu, Junping Zhang, Zhouhan Lin, and Hao Chen. Parallel acceleration of SAM algorithm and performance analysis. *IEEE Journal of Selected Topics in Applied Earth Observation and Remote Sensing*, 6(3):1172–1178, 2013.

[4] Jose M. Domínguez, Alejandro J.C. Crespo, and Moncho Gómez-Gesteira. Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method. *Computer Physics Communications*, 184(3):617–627, 2013.

[5] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[6] David S. Watkins. *Fundamentals of matrix computations*. Wiley, 1991.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[8] Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics, 2000.

[9] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipies The Art of Scientific Computing*. Cambridge University Press, third edition, 2007.

[10] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Li-Wen Chang, Nasser Anssari, Geng Liu, Wen mei W. Hwu, and Nady Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *Computer*, 45(8):26–32, 2012.

[11] Marc Hofmann and Erricos John Kontoghiorghes. Pipeline givens sequences for computing the QR decomposition on a EREW PRAM. *Parallel Computing*, 32(3):222–230, 2006.

[12] Cristian Gatu and Erricos J. Kontoghiorghes. Parallel algorithms for computing all possible subset regression models using the QR decomposition. *Parallel Computing*, 29(4):505–521, 2003.

[13] A. H. Sameh and D. J. Kuck. On stable parallel linear system solvers. *Journal of the Association for Computing Machinery*, 25(1):81–91, 1978.

[14] Lawrence Livermore National Laboratory. *OpenMP*. URL: `https://computing.llnl.gov/tutorials/openMP`.

[15] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, 2007. Version 0.8.2. URL: `http://developer.download.nvidia.com`.

[16] George Em Karniadakis and Robert M. Kirby II. *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press, 2003.

[17] Message passing (MPI) and GPU programming. Swiss National Supercomputing Centre And Swiss Federal Institute of Technology Zurich, Octubre 2011. URL: `http://corsi.cineca.it/courses/scuolaAvanzata/MPI+CUDA.pdf` [cited 11 de Mayo del 2013].

[18] NVIDIA Corporation. *Tesla Kepler GPU Accelerators*, 2012. URL: `http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf`.

[19] Intel Corporation. *Intel Xeon Processor X5680*. URL: `http://ark.intel.com/products/47916/`.

[20] NVIDIA Corporation. *Tesla K20X GPU Accelerator*, 2012. URL: `http://www.nvidia.com/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf`.

[21] NVIDIA Corporation. *Tesla C2050 and Tesla C2070 computing processor board*, 2010. URL: `http://www.nvidia.com/docs/IO/43395/BD-04983-001_v04.pdf`.

[22] NVIDIA Corporation. *NVIDIA GeForce GTX 460*. URL: `http://www.nvidia.es/object/product-geforce-gtx-460-es.html`.