Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Implementación paralela y heterogénea de la transformación de Householder y sus aplicaciones.

T E S I S

Que presenta

Juan Cipriano Hernández Cortés

Para Obtener el Grado de

Maestro en Ciencias en Computación

Directores de la Tesis:

Dr. Amilcar Meneses Viveros

Dra. Liliana Ibeth Barbosa Santillán

Ciudad de México **Diciembre, 2017**

Agradecimientos

Al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el apoyo económico que me brindo durante esta etapa.

Al Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional(CINVESTAV-IPN) por darme la oportunidad de ser parte de uno de sus posgrados de maestría.

Al Departamento de Computación por las todas las experiencias y el aprendizaje que me dejaron.

A mis directores el Dr. Amilcar Menseses Viveros y la Dra. Liliana Ibeth Barbosa Santillán por aceptarme como su tesista.

A mis sinodales el Dr. Sergio V. Chapa Vergara y el Dr. José Guadalupe Rodríguez García por sus valiosos comentarios.

Resumen

La transformación de Householder es utilizada en distintos métodos de álgebra lineal numérica como lo son: solucionadores de sistemas de ecuaciones, encontrar la forma triangular de matrices, bifactorización de matrices, bidiagonalización de matrices, calculo de la descomposición en valores singulares de una matriz. Problemas como las ecuaciones de Schrödinger, sistemas de Química Cuántica o el problema de muchos cuerpos involucran operadores matriciales de grandes dimensiones. En la actualidad las arquitecturas multi-many core se han consolidado en los distintos tipos de plataformas computacionales desde dispositivos móviles hasta servidores y centros de supercomputación. Este tipo de arquitecturas tienen gran aceleración al realizar operaciones vectoriales, sin embargo, uno de los grandes retos que se presentan al utilizar estas arquitecturas es la limitada memoria con la que cuentan, por ello es necesario realizar implementaciones que sean escalables y utilicen un modelo de memoria distribuida para poder resolver problemas de dimensiones grandes. En el presente trabajo se propone una forma de distribuir el trabajo entre distintos nodos para implementar la transformación de Householder y aplicarla a la bifactorización y bidiagonalización de matrices utilizando MPI como interfaz de comunicación y CUDA para la programación de tarjetas de procesamiento de gráficos, con la finalidad de poder trabajar con matrices de grandes dimensiones.

Abstract

Householder's transformation is widely used in several linear algebra numeric methods instances of this are: equations systems solvers, getting triangular matrices form, matrices bifactorization, bidiagonalization of a matrix, computing singular value decomposition. There are problems such as Schrödinger's equations, Quantum Chemistry or the many bodies problem that use high dimensional matrix operators. Nowadays the multi-many core architectures have been consolidated in all kind of computing platforms from mobile devices to computing servers and supercomputing centers. This kind of architectures has a great speed-up in vectorial operations, nevertheless, their limited memory is a challenge when you have to do computations on large matrices, for this reason is necessary develop scalable applications that could use a memory distributed model. In this work we propose a way to split the Householder's transformation in several computing nodes and apply it to bifactorization and bidiagonalization problems using MPI as communication interface and CUDA as graphic processing units application programming interface.

Índice general

Índice de figuras

Capítulo 1

Introducción

Problemas de ingeniería y de ciencias como la Física y la Química utilizan operadores matriciales, tales matrices suelen ser simétricas y de dimensiones grandes, las tareas más comunes sobre estos operadores son el calculo de valores y vectores propios. Para realizar este tipo de cálculos se utilizan transformaciones matriciales las cuales se usan para obtener matrices similares a las matrices originales, en particular para los problemas de valores y vectores propios una técnica muy utilizada es la bifactorización de matrices y a partir de ésta se aplican técnicas de diagonalización como el método de Cuppen (Cuppen, 1980). Otra transformación muy usada es la descomposición SVD para matrices no cuadradas, esta transformacion se suele usar en problemas de minería de datos, spectral clustering, compresión y en modelos de fenómenos cuánticos. La transformación de Householder (Householder, 1958) es utilizada para obtener la bifactorización y la bidiagonalización de las matrices, a partir de las cuales se pueden obtener los valores propios y la descomposición en valores singulares de las matrices.

En (Tapia-Romero, 2013) se implementó el método de rotaciones de Givens en GPU(*Graphic Processing Unit*, Unidad de Procesamiento de Gráficos), el cual le dio mejores resultados en cuestión de tiempo comparado con la biblioteca MKL de Intel, el método de rotaciones de Givens se utilizo para calcular la factorización QR de una matriz, en esta implementación se hizo con MPI y CUDA.

En el trabajo realizado por Estrella-Cruz (2014) se implementó el algoritmo de Cuppen, el cual calcula los valores y vectores propios de una matriz tridiagonal conocida, esta implementación obtuvo buenos resultados al partir el problema en múltiples nodos en una arquitectura híbrida.

En el curso de Cómputo Científico se realizó una primera implementación del método de Householder para la bifactorización de matrices densas utilizando GPU. Mediante esta implementación se obtuvieron los siguientes resultados:

Tamaño de la matriz	Tiempos(ms)	
	OpenMP	CUDA
32	8	5.353
64	62	19.252
128	977	145.116
256	21423	1871.750
512	347372	60894.199
1024	4282000	748455.125

Como se observa en la tabla anterior, el método de Householder es susceptible a ser paralelizable para mejorar su desempeño, ya que se se obtiene una aceleración hasta de cinco veces en la implementación en GPU.

1.1. Estado del Arte

En (Liu and Seinstra, 2010) se implementa el algoritmo de Householder en una sola GPU, su objetivo principal fue mostrar la importancia que tiene el manejo de la memoria en las GPU, ya que por la forma en como están diseñadas las arquitecturas es posible que múltiples hilos de ejecución accedan a distintas regiones de memoria al mismo tiempo. Sin embargo, en el artículo jamas se menciona el particionamiento de los datos cuando se tienen múltiples tarjetas.

En Cosnuau (2014) también se implementa la bifactorización de Householder para la tridiagonalización de matrices y posteriormente se resuelven problemas de eigenvalores y eigenvectores, sólo que aquí se utilizan matrices pequeñas de 128×128, esto debido a que se tenía la idea que para instancias pequeñas del problema era más eficiente realizar los cálculos en CPU. Los autores resuelven los problemas de valores y vectores propios de 180 matrices pequeñas y encontraron que el rendimiento es mejor en la GPU. En el trabajo a futuro mencionan que se puede reducir el tiempo de procesamiento si se distribuye la carga en múltiples GPU y además podría mejorarse el paso de la diagonalización de las matrices, utilizando el método de Givens.

En Lahabar and Narayanan (2009) implementan un algoritmo para calcular la descomposición SVD de una matriz en GPU. Para realizar la descomposición es necesario aplicar una bifactorización para después diagonalizar la matriz, el paso de la bifactorización se lleva a cabo utilizando el método de las reflexiones de Householder. Su implementación es comparada contra la implementación de MKL. En general la implementación GPU fue mucho mas rápida. En una de las gráficas de resultados los autores dividieron en dos pasos su algoritmo: un paso es la bifactorización y el otro paso es la diagonalización, en la gráfica se aprecia

la aceleración de cada paso y se puede observar como el método de Householder alcanza una aceleración de aproximadamente 16x en matrices densas de tamaños de 4000×4000.

En Kotas and Barhen (2011) se realiza la implementación de la descomposición SVD en paralelo utilizando CPU y GPU, la implementación se realizo de dos formas una utilizando las reflexiones de Householder y la otra utilizando el método de las rotaciones de Jacobi. De estas dos implementaciones encontraron que es mejor utilizar el método de Householder en matrices de grandes dimensiones.

1.2. Planteamiento del Problema

La transformación de Householder se utiliza para expresar matrices simétricas en su forma tridiagonal y en el cálculo de la descomposición SVD. Los problemas que se resuelven son matrices de grandes dimensiones.

Debido a que es posible realizar implementaciones de la transformación de Householder en las nuevas arquitecturas paralelas, surge la necesidad de estudiar estrategias de particionamiento que permitan aplicar estas transformaciones a los problemas anteriormente mencionados.

1.3. Objetivos

General

El objetivo de este trabajo es desarrollar una implementación paralela escalable del método de reflexiones de Householder para matrices densas, la cual pueda ejecutarse en arquitecturas con un modelo de memoria híbrida.

Particulares

1. Estudiar los diferentes tipos de particionamiento de datos en múltiples nodos que pueden ser aplicados a la transformación de Householder para la bifactorización y bidiagonalización de matrices.

2. Estudiar diversas estrategias de particionamiento de datos para la diagonalización de matrices en múltiples nodos.

3. Implementar en una arquitectura híbrida-heterogénea la bifactorización y bidiagonalización de matrices.

4. Implementar en una arquitectura híbrida-heterogénea la descomposición SVD.

1.4. Justificación

En (Evans and Gusev, 2002) se analizan las distintas maneras en que se puede calcular la descomposición en valores singulares de una matriz, en cada una de las distintas formas es necesario reducir la matriz a su forma bidiagonal, para lograr esto la forma más directa es a través de la transformación de Householder.

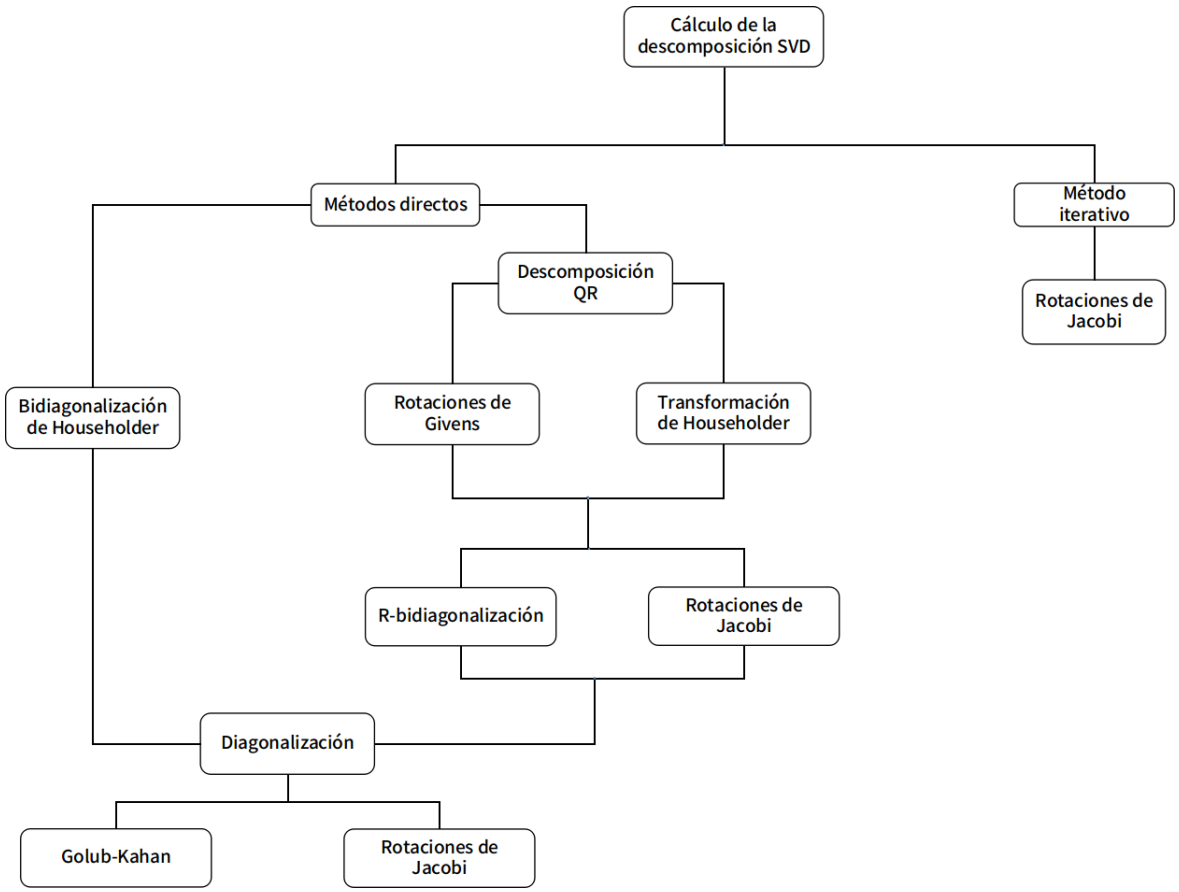


Figura 1.1: Diagrama de (Evans and Gusev, 2002) donde se muestran las distintas formas para calcular la descomposición SVD

En la figura 1.1 se observa que para calcular la descomposición en valores singulares es necesario realizar transformaciones de Householder. Para el paso de diagonalización solo se consideran los métodos de Golub-Kahan (Golub and Reinsch, 1970) (Demmel and Kahan, 1990) y el método de rotaciones de Jacobi. Sin embargo, en (Gu and Eisenstat, 1995), (Hogben, 2006) y (Konda and Nakamura, 2009) se muestra como también puede ser utilizado el método de (Givens, 1958) en la diagonalización de matrices el cual fue implementado en una arquitectura híbrida por (Estrella-Cruz, 2014) obteniendo mejores aceleraciones que las implementaciones en CPU.

En (Acton, 1990) (Press et al., 1992) se presentan métodos para calcular los valores propios de una matriz real y simétrica, en los cuales es necesario obtener la forma tridiagonal de la matriz. También es necesario para calcular las factorizaciones QR y QL de matrices reales y simétricas.

En (Fuller and Millett, 2011) nos muestra como actualmente hay un estancamiento en el rendimiento y mejora de los procesadores, nos plantea la idea de que es necesario utilizar y sacar mejor provecho de las arquitecturas paralelas actuales. También nos explica la importancia de desarrollar nuevas arquitecturas paralelas para propósitos específicos a fin de obtener un mejor rendimiento en la resolución de problemas. Asimismo hace hincapié en la importancia del rendimiento energético, es necesario utilizar arquitecturas que nos brinden un buen balance entre rendimiento y consumo energético.

1.5. Organización de la Tesis

El presente documento se encuentra organizado en seis capítulos, los restantes cinco se describen brevemente a continuación.

- **Capítulo 2. Arquitecturas paralelas.** En este capítulo se explican los conceptos teóricos de la computación paralela: los distintos tipos de paralelismo, los modelos de memoria, las técnicas de programación paralela y la metodología de Ian Foster.
- **Capítulo 3. Transformación de Householder.** Aquí se describe en que consisten las reflexiones de Householder y como se aplican en la bifactorización y en la descomposición de valores singulares. También se describen los algoritmos paralelo y secuencial de estas dos aplicaciones.
- **Capítulo 4. Implementación.** En este capítulo se explica como fueron implementados los algoritmos paralelos de la bifactorización y de la descomposición SVD.
- **Capítulo 5. Pruebas y Resultados.** Se describe en que consistieron las pruebas realizadas así como los resultados que se obtuvieron.
- **Capítulo 6. Conclusiones y Trabajo a Futuro.** Se presentan las conclusiones que se obtuvieron del presente trabajo de tesis, así como también los aspectos que se pueden mejorar y las características que podrían agregarse a este trabajo.

Capítulo 2

Arquitecturas Paralelas

Una computadora paralela es aquella que permite que múltiples procesos se ejecuten simultáneamente, este tipo de computadoras es utilizada para resolver problemas muy grandes en una menor cantidad de tiempo, dividiendo el problema original en pequeños subproblemas y resolviéndolos de manera simultánea. En el presente capítulo se presentan las formas en que son clasificadas las computadoras paralelas, las distintas formas en que se organiza la memoria y el modelo que se debe seguir para desarrollar un algoritmo paralelo.

2.1. Taxonomía de Flynn

Propuesta en 1966 por Michael Flynn, es una forma de clasificar las computadoras paralelas (Flynn, 1972). Esta se basa en los conceptos de de *flujo de datos* y *flujo de instrucciones*. Flynn define estos conceptos de la siguiente manera:

- Flujo de instrucciones: son las instrucciones que viajan de la memoria principal de la computadora hacia la unidad de procesamiento.

- Flujo de datos: es la información sobre la cual van a operar las instrucciones, este flujo viaja de la memoria principal hacia la unidad de procesamiento como datos de entrada y de la unidad de procesamiento hacia la memoria principal como el resultado de la ejecución de las instrucciones.

Los conceptos anteriores se ilustran en la figura 2.1:

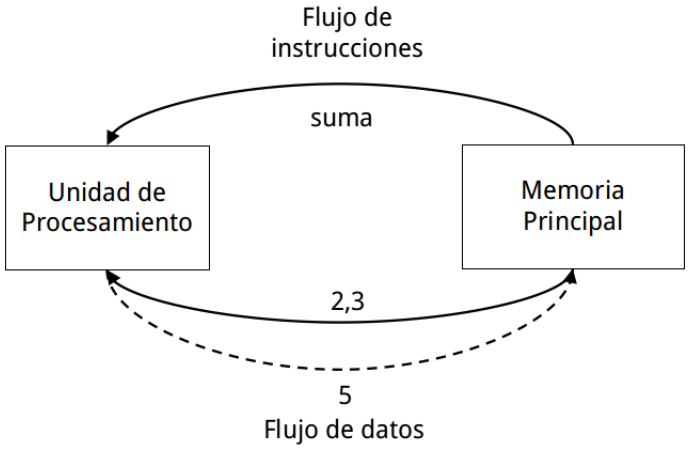


Figura 2.1: Flujos de datos e instrucciones

En la figura 2.1 se ejemplifica como de la memoria principal se envía la instrucción *suma* a la unidad de procesamiento, después son enviados los operandos *2,3* y finalmente el resultado *5* de esta operación es enviado de la unidad de procesamiento hacia la memoria principal.

La taxonomía de Flynn considera cuatro categorías de clasificación:

1. Una instrucción, un dato. *Single instruction, single data(SISD)*

2. Una instrucción, múltiples datos. *Single instruction, multiple data(SIMD)*

3. Múltiples instrucciones, un dato. *Multiple instruction, single data(MISD)*

4. Múltiples instrucciones, múltiples datos. *Multiple instruction, multiple data(MIMD)*

2.1.1. Arquitectura SISD

Este tipo de arquitectura corresponde a las computadoras secuenciales, las cuales no tienen ningún tipo de paralelismo, solo se utiliza un flujo de datos y un flujo de instrucción.

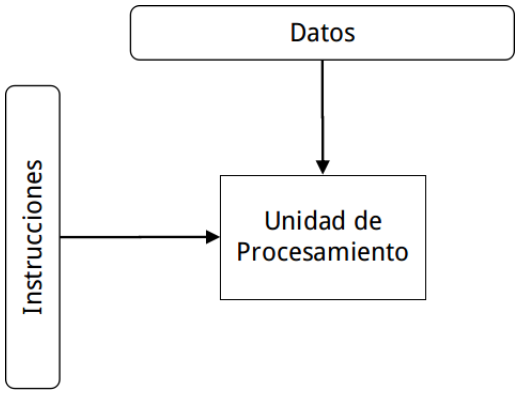


Figura 2.2: Modelo SISD

En la figura 2.2 se puede observa como solo hay un flujo de datos y un solo flujo de instrucciones, en este tipo de computadoras no es posible realizar simultáneamente mas de una operación. Este tipo de arquitectura es la mas antigua las primeras computadoras son de este tipo (Blaise, 2017) (Duncan, 1990).

2.1.2. Arquitectura SIMD

En este modelo es capaz de aplicar una misma instrucción a un conjunto de datos en un solo ciclo de reloj.

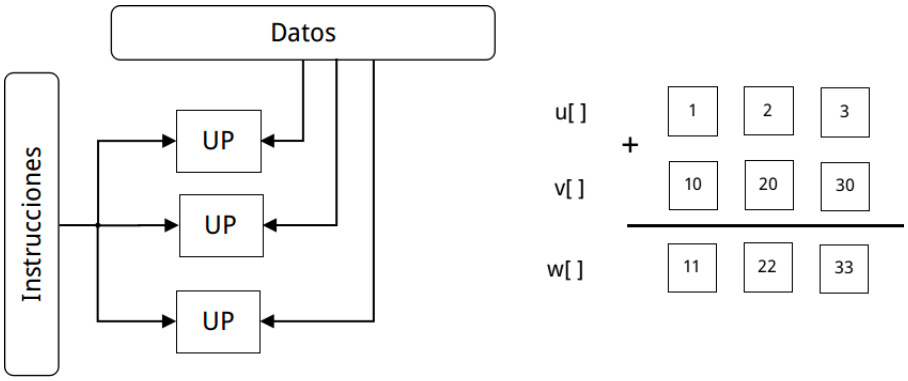


Figura 2.3: Modelo SIMD y ejemplo de una ejecución

Como se puede ver en la figura 2.3, en este tipo de computadoras se realizan operaciones vectoriales, en el ejemplo se suman los vectores **u**, **v** en un solo ciclo de reloj y producen como resultado el vector **w**. En la actualidad un ejemplo de este tipo de arquitecura son las GPUs (Blaise, 2017) (Duncan, 1990).

2.1.3. Arquitectura MISD

Este tipo de paralelismo es el que menos se ha explotado, consiste en aplicar distintas instrucciones a un mismo flujo de datos.

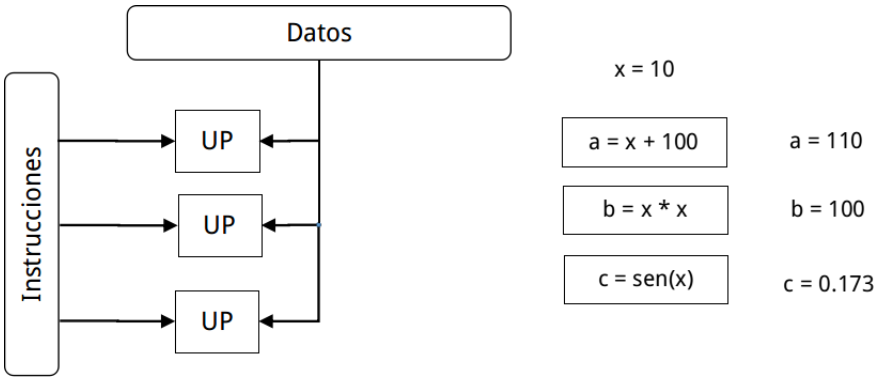


Figura 2.4: Modelo MISD y ejemplo de una ejecución

En la figura 2.4, podemos ver un ejemplo de este modelo, tenemos a la variable **x** a la cual se le aplican distintas operaciones al mismo tiempo y el resultado es almacenado en una variable distinta para cada unidad de procesamiento (Blaise, 2017) (Duncan, 1990).

2.1.4. Arquitectura MIMD

Es el tipo de paralelismo mas común, cada unidad de procesamiento trabaja con un flujo de datos y de instrucciones diferentes.

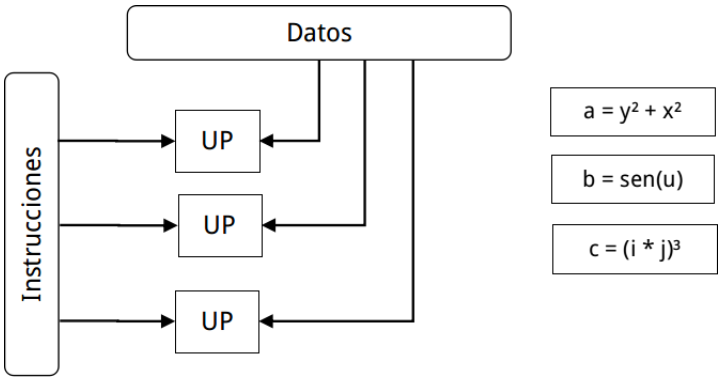


Figura 2.5: Modelo MIMD y ejemplo de una ejecución

En la figura 2.5 se puede apreciar la utilidad de este tipo de computadoras paralelas, ya que con este modelo es con el cual trabajan las computadoras modernas, en el ejemplo se observan tres flujos diferentes de datos

y tres flujos diferentes de operaciones, vemos como cada flujo de instrucciones opera sobre distintos datos. Un flujo de datos envía las variables a, y, x y sobre ellas operan las instrucciones de suma y potencia, otro flujo con la instrucción seno opera sobre las variables u,b y un flujo más que opera sobre las variables i,j,c (Blaise, 2017) (Duncan, 1990).

2.2. Modelos de Memoria en Arquitecturas Paralelas

En las computadoras paralelas la forma en que las unidades de procesamiento acceden a la memoria esta determinada por la forma como está conectada la memoria con las unidades de procesamiento.

2.2.1. Modelo de Memoria Compartida

En este modelo todos las unidades de procesamiento pueden acceder directamente a toda la memoria. Las unidades de procesamiento pueden operar de manera independiente, pero comparten la misma memoria entre todos. Existen dos modelos de memoria compartida:

- Memoria de acceso uniforme. *Uniform memory access*(UMA)
- Memoria de acceso no uniforme. *Non-uniform memory access*(NUMA)

Memoria de Acceso Uniforme

El tiempo de acceso a cualquier región de memoria es el mismo para todas las unidades de procesamiento, si una unidad de procesamiento realiza un cambio en una región de memoria este cambio es visible para todos.

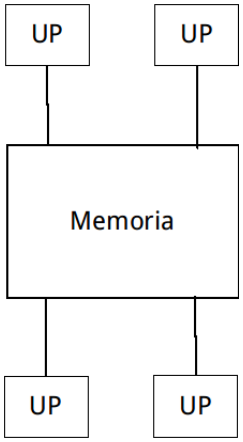


Figura 2.6: Memoria de acceso uniforme

En la figura 2.6 podemos ver que debido a la forma en como están interconectadas las unidades de procesa-

miento se logra que los tiempos de acceso son iguales para todos (Blaise, 2017) (Coulouris et al., 2011).

Memoria de Acceso No Uniforme

En este modelo los tiempos de acceso para las distintas regiones de memoria no son constantes, varían de acuerdo a la unidad de procesamiento y a la región de memoria a la que se quiera acceder.

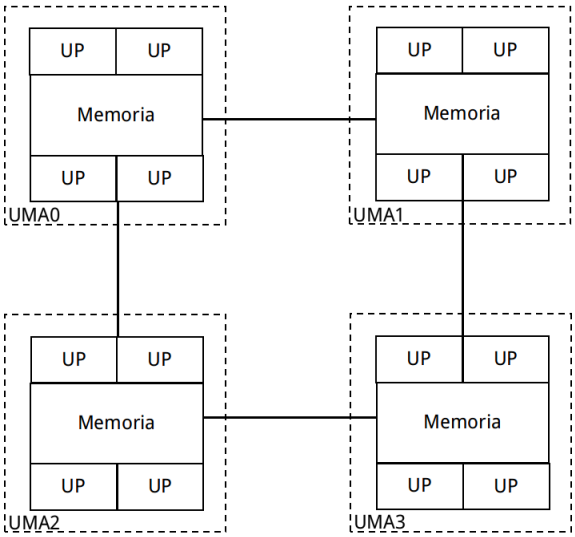


Figura 2.7: Memoria de acceso no uniforme

De la figura 2.7 se puede ver como este modelo es la interconexión de dos o mas conjuntos de unidades de procesamiento con un modelo UMA. Debido a esta interconexión es que los tiempos de acceso son distintos para todos, en la figura 2.7 podemos observar como es más rápido para una UP que está en la UMA1 acceder a una región de la memoria en la UMA1 que acceder a una región en la memoria de la UMA2. Otro detalle que tiene este modelo es que mantener la coherencia en la memoria es complicado, ya que esto implica estar comunicando cada cambio en la memoria a las distintas UMAs (Blaise, 2017) (Coulouris et al., 2011).

2.2.2. Modelo de Memoria Distribuida

Cuando se tiene un modelo de memoria de este tipo cada unidad de procesamiento tiene su propia memoria, pero esta no es visible para las demás unidades de procesamiento. El acceso a los datos entre las diferentes regiones de memoria es posible, pero esto no es tan sencillo como en los modelos de memoria compartida.

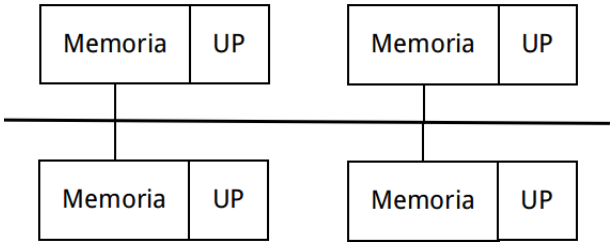


Figura 2.8: Memoria distribuida

La forma de interconexión mas común entre nodos es una conexión tipo bus, para que en caso de requerir acceder a una región de memoria de otro nodo esta sea mas sencilla. Debido a que las unidades de procesamiento tienen su propia región de memoria, las direcciones de memoria de una UP no se mapean a otra, por lo tanto en este tipo de arquitecturas no existe el concepto de coherencia de cache. Si es necesario intercambiar información entre unidades de procesamiento es deber del programador establecer los mecanismos de comunicación entre las distintas unidades (Blaise, 2017) (Coulouris et al., 2011).

2.2.3. Modelo de Memoria Híbrida

Para este modelo cada nodo debe contar con un modelo de memoria compartida, una interconexión con los demás nodos y cada nodo solo puede acceder directamente a su propia región de memoria.

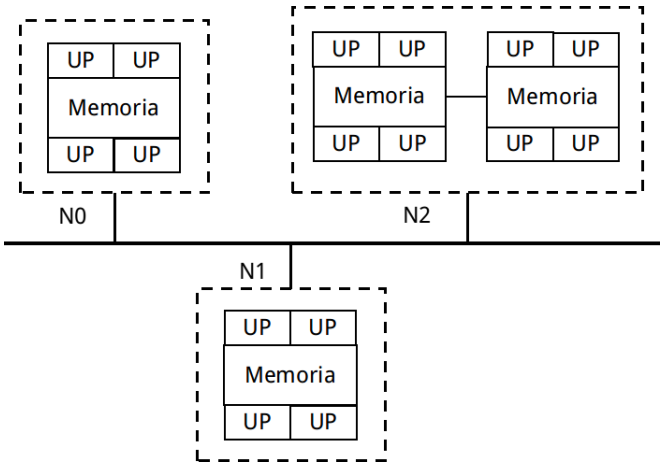


Figura 2.9: Memoria híbrida

En la figura 2.9 podemos observar como N0 y N1 tienen un modelo de memoria compartida tipo UMA, mientras que en N2 se tiene un modelo tipo NUMA. Todos estos nodos están comunicados entre sí, sin embargo, no pueden acceder directamente a la memoria de los otros nodos, si se quisiera compartir lo que

se encuentra en la memoria de un nodo con otro es necesario comunicarlos pasando mensajes entre nodos (Blaise, 2017) (Coulouris et al., 2011).

2.3. Modelos de Programación Paralela

Ya conocemos como se clasifican las computadoras paralelas de acuerdo a como operan sobre los datos y como interactúan con la memoria, esta sección sera acerca de las distintos paradigmas de programación para las computadoras paralelas.

2.3.1. Programación Multiproceso

Este tipo de programación utiliza el modelo de memoria compartida, se crean varios procesos que ejecutan tareas que se comunican entre sí o que sirven para resolver un problema mayor. La mayor complicación de este tipo de programación es la administración de la memoria.

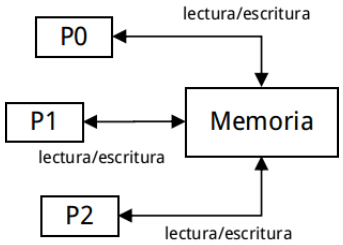


Figura 2.10: Distintos procesos ejecutándose en un mismo nodo

En la figura 2.10 vemos como distintos procesos comparten la misma memoria y realizan operaciones de lectura y escritura, los procesos tienen sus propias regiones de memoria pero es posible tener una región de memoria común a todos ellos para que de esta manera puedan comunicarse entre ellos, pero si se va a contar con este tipo de regiones de memoria es necesario tener mecanismos de control de acceso para mantener la coherencia y el flujo de trabajo (Blaise, 2017) (Kumar et al., 2003).

2.3.2. Programación Multihilo

Al igual que el paradigma anterior es utilizado con memoria compartida en este tipo de programación un proceso crea varios hilos los cuales ejecutan funciones que tienen independencia de datos, al finalizar de ejecutar estas funciones los hilos terminan y el proceso que los creo sigue en ejecución y puede crear mas hilos en caso de ser necesario.

En la figura 2.11 vemos como los hilos que crea un proceso solo tienen acceso a la región de memoria del proceso que los creo, los hilos pueden acceder a las variables globales del proceso ocurriendo lo mismo que

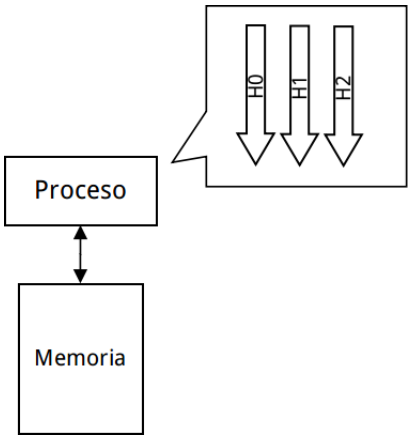


Figura 2.11: Un proceso con distintos hilos en ejecución

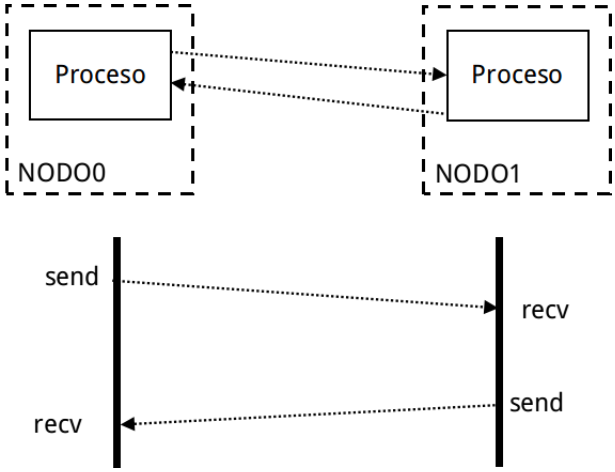


Figura 2.12: Múltiples procesos intercambiando información

en el caso de la memoria común en la programación multiproceso. Una diferencia con respecto a los procesos es que crear un hilo no implica mucho consumo de tiempo y recursos, por esto la vida de un hilo suele ser más corta que la de un proceso (Blaise, 2017) (Kumar et al., 2003).

2.3.3. Programación Basada en Paso de Mensajes

Este tipo de programación se usa con memoria distribuida, se crean procesos en los distintos nodos los cuales usan su memoria local durante la ejecución y al final envían el resultado de su ejecución a un nodo donde se concentrara el resultado de todos los procesos.

De la imagen 2.12 se observa como dos procesos se envían información entre sí, la comunicación además de ser punto a punto puede ser de uno a muchos procesos como se ve en la figura 2.13 o también se puede dar

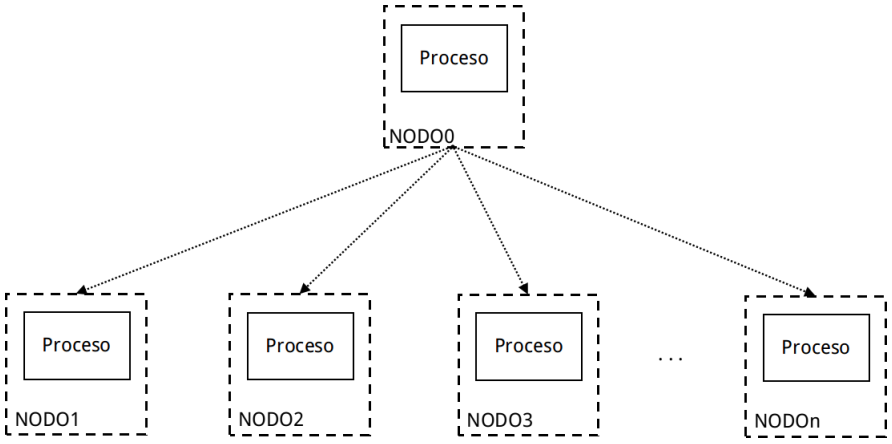


Figura 2.13: Un proceso enviando información a múltiples procesos

entre todos los procesos como se observa en la figura 2.14 (Blaise, 2017) (Kumar et al., 2003).

2.3.4. Programación en Tarjetas de Procesamiento de Gráficos

Las tarjetas de procesamiento de gráficos tienen una arquitectura SIMD lo cual hace que cambie el paradigma de computación. Cuando se programa en CPU se utilizan variables escalares y por lo tanto se realizan operaciones escalares. En una GPU se manipulan datos de tipos vectoriales a los cuales se aplican operaciones vectoriales.

Las aplicaciones vectoriales en un CPU tienen una mayor complejidad que en una GPU, por ejemplo la suma de dos vectores en CPU tiene una complejidad $\mathcal{O}(n)$ mientras que una GPU *idealmente* se tiene una complejidad $\mathcal{O}(1)$ (Cook, 2012) (Kirk and Wen-Mei, 2016).

2.3.5. Programación Híbrida

Combina dos o más modelos de programación paralela, de esta manera se puede contar con dos esquemas de memoria: memoria compartida y memoria distribuida. Con este tipo de programación es posible resolver problemas que demanden cantidades muy grandes de memoria, distribuyendo las cargas de trabajo en los distintos nodos (Foster, 1995) (Blaise, 2017) (Coulouris et al., 2011).

Como apreciamos en 2.15 el conjunto de nodos representa un modelo de memoria distribuida que se comunican entre sí utilizando paso de mensajes, pero cada nodo a su vez tiene un modelo de memoria compartida, como se observa los nodos 1 y 2 utilizan programación multihilo y en el nodo 3 observamos

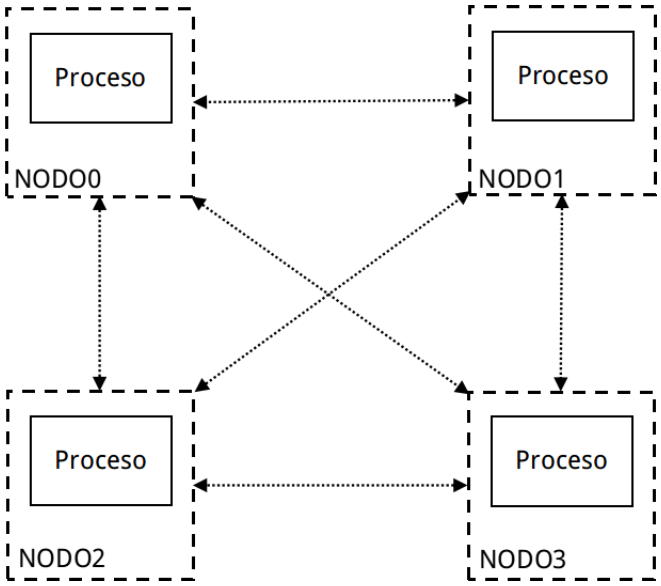


Figura 2.14: Múltiples procesos comunicándose entre sí

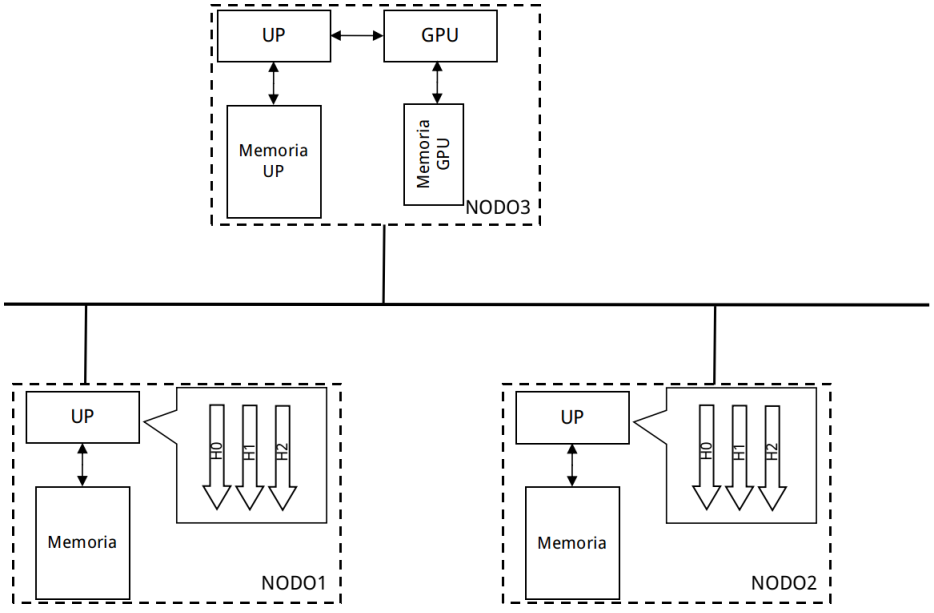


Figura 2.15: Modelo de programación híbrida

como se tiene disponible una GPU.

2.4. Metodología de Ian Foster

Esta metodología es una guía para el diseño de programas paralelos, nos permite analizar aspectos del algoritmo independientes del ambiente de ejecución en las primeras etapas y centrarnos en los aspectos que dependen del entorno de ejecución en las últimas etapas, para mejorar el desempeño de nuestro algoritmo. Las etapas de la metodología son: *particionamiento, comunicación, aglomeración* y *mapeo* (Foster, 1995).

2.4.1. Particionamiento

En esta etapa se identifican las regiones potencialmente paralelizables del algoritmo, el objetivo principal es identificar el mayor número de regiones de modo que se tenga una granularidad muy fina en cada región, la granularidad es el conjunto de operaciones que pueden realizarse en paralelo este conjunto de operaciones también es llamado tarea. El particionamiento se debe realizar a manera de evitar la duplicidad de tareas y datos.

Un buen particionamiento es aquel que divide en pequeñas secciones tanto los datos con los que se va a trabajar como las operaciones que se van a realizar. En el momento de estar realizando el particionamiento se tienen dos alternativas primero dividir los datos y después averiguar la manera en como procesar los datos divididos, este enfoque se denomina *descomposición de dominio*. La segunda alternativa es el proceso inverso, primero dividimos el procesamiento y después encontramos una manera de dividir los datos, a esto se le llama *descomposición funcional* (Foster, 1995).

2.4.1.1. Descomposición de Dominio

Este tipo de enfoque de particionamiento se suele utilizar cuando tenemos un problema que involucra datos de gran dimensión, su objetivo principal es descomponer los datos en piezas muy pequeñas y que todas sean del mismo tamaño. Posteriormente se busca dividir todas las operaciones que serán aplicadas a los datos, al realizar esta división una tarea es asignada a una partición de datos, las tareas también pueden comunicarse entre sí en caso de ser necesario.

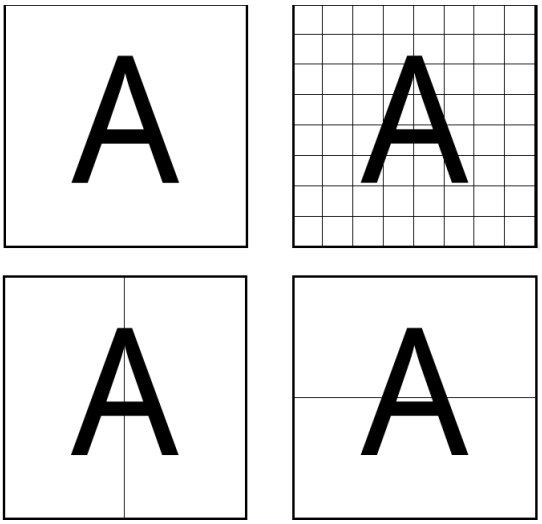


Figura 2.16: Distintas formas de particionamiento para el mismo conjunto de datos

Como podemos ver en la figura 2.16 existen distintas formas de dividir un conjunto de datos, la elección final depende de las características del problema que se quiere resolver (Foster, 1995).

2.4.1.2. Descomposición Funcional

Para realizar esta descomposición nos enfocamos en las operaciones que vamos a realizar a los datos, la descomposición nos debe de dar como resultado tareas que no dependen mutuamente una de otra y posteriormente examinamos los datos que requiere cada tarea para ser ejecutada. Si al dividir los datos podemos dividirlos de manera que no se traslapen con mas de una tarea la descomposición esta completa, en cambio si los datos se traslapan y necesitamos realizar muchas operaciones de comunicación entre tareas generalmente esto nos indica que es mejor considerar una descomposición de dominio.

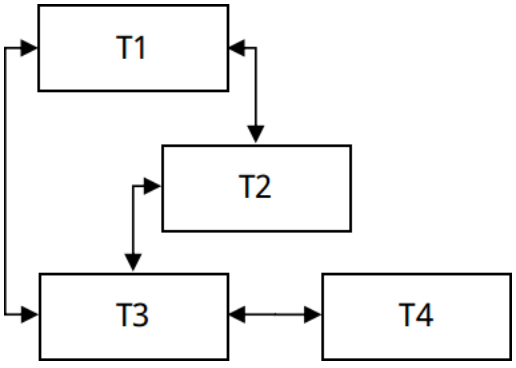


Figura 2.17: Descomposición funcional de un problema

En 2.17 se muestra una descomposición que consta de cuatro tareas, las cuales se comunican entre sí, el resultado de una tarea es la entrada de la otra. Cada tarea debe poder ser paralelizada utilizando un particionamiento de datos 2.17.

2.4.2. Comunicación

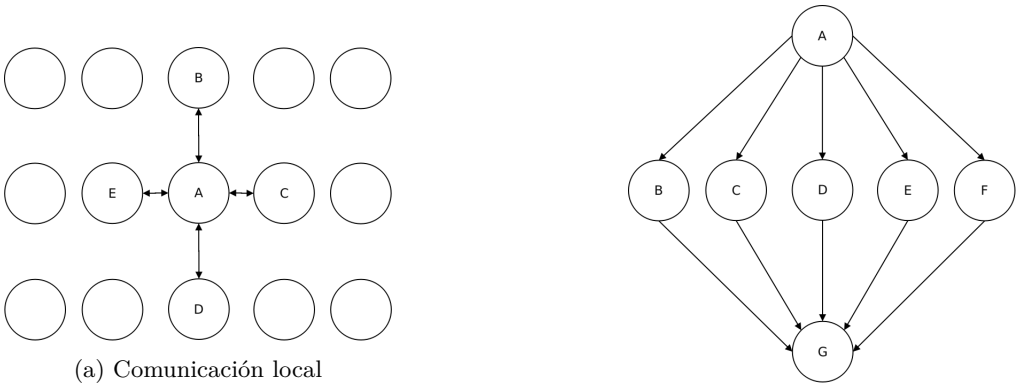
Las tareas identificadas en el particionamiento usualmente suelen requerir un cierto grado de comunicación entre ellas para poder continuar con la ejecución del programa, este flujo de intercambio de datos es diseñado en esta etapa. Para realizar la comunicación entre tareas es necesario establecer un vínculo entre ellas, a este vínculo se le conoce como canal, una vez establecidos los canales es necesario definir los mensajes que van a viajar a través de ellos. En esta etapa del diseño se desea utilizar el mínimo número de canales y operaciones de comunicación.

Cuando se realiza un particionamiento de datos suele ser complicado determinar la manera en que se comunican las tareas. Una vez identificadas todas las interacciones entre las tareas la organización e implementación suele ser complicada, ya que aunque la descomposición sea sencilla suele presentar grandes retos de comunicación. Por el contrario, cuando se realiza una descomposición funcional la identificación de los canales suele ser bastante sencilla ya que suele seguir el flujo del algoritmo.

La comunicación entre las tareas se puede categorizar en los siguientes tipos: *local/global, estructurada/no estructurada, estática/dinámica* y *síncrona/asíncrona.*

Comunicación Local y Global

La comunicación local se da cuando una tarea se comunica con un conjunto pequeño de otras tareas(comúnmente llamado vecindario), en cambio la comunicación global se da con un conjunto más grande de tareas.



(a) Comunicación local

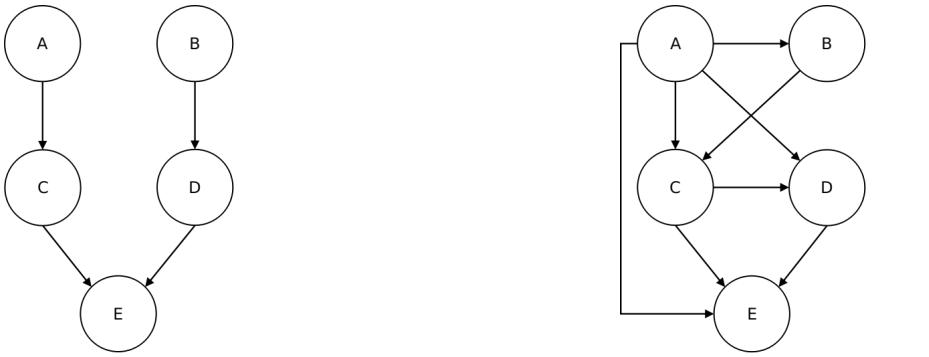
(b) Comunicación global

Figura 2.18: Ejemplos de comunicación global y local

Observamos en la figura 2.18a como el proceso A se comunica únicamente con los procesos más cercanos B, C, D y E, mientras que en la figura 2.18b vemos como el proceso A se comunica con los procesos B, C, D, E y F y a su vez todos éstos últimos se comunican con el proceso G (Foster, 1995).

Comunicación Estructurada y No Estructurada

En el tipo de comunicación estructurada la red de intercomunicación que se forma tiene una estructura regular como una red o un árbol, en cambio en la no estructurada la topología de la red es un grafo arbitrario.



(a) Comunicación estructurada

(b) Comunicación no estructurada

Figura 2.19: Topologías de comunicación

La figura 2.19a representa una estructura regular, se observa un cierto orden en la comunicación entre los procesos. El proceso A solo se comunica con el proceso C, el proceso B solamente se comunica con el proceso D, y los procesos C y D se comunican con el proceso E. Mientras que en la figura 2.19b vemos como no existe un orden en la comunicación entre procesos, aquí prácticamente todos los procesos se comunican entre sí,

sin obedecer a ningún tipo de jerarquía (Foster, 1995).

Comunicación Estática y Dinámica

En la comunicación estática las tareas que se comunican entre sí siempre son las mismas, pero en la comunicación dinámica no es así las tareas que se comunican pueden variar dependiendo del resultado de una tarea.

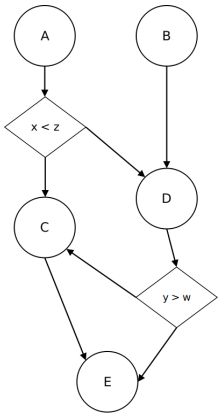
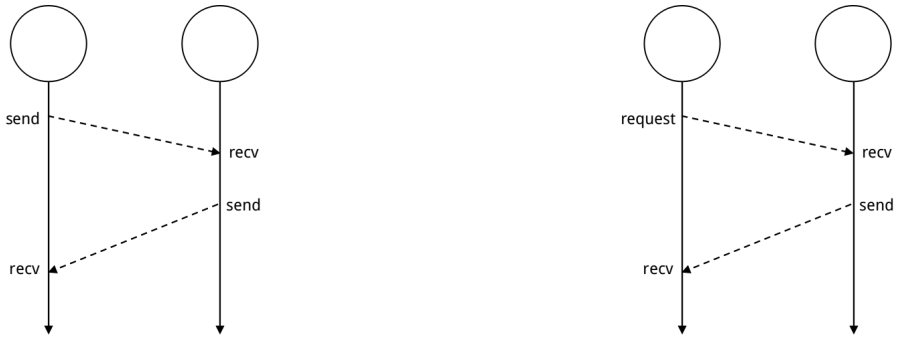


Figura 2.20: Ejemplo de comunicación dinámica

En la figura 2.20 se observa un ejemplo de comunicación dinámica, el proceso A depende de los valores de x y z para decidir si se comunica con el proceso C o con el proceso D, de manera similar pasa con el proceso D que depende de los valores de w e y para decidir si se comunica con el proceso C o con el proceso E. Ejemplos de comunicación dinámica pueden ser cualquiera de los anteriores, en esos casos siempre los procesos se comunican con los mismos procesos (Foster, 1995).

Comunicación Síncrona y Asíncrona

Cuando se tiene un modelo de comunicación síncrona, tanto la tarea que envía como la que recibe los datos están coordinadas mientras que en un modelo asíncrono esto no siempre sucede, puede que la información que necesite cierta tarea este disponible antes de que esta la necesite.



(a) Comunicación síncrona (b) Comunicación asíncrona

Figura 2.21: Ejemplos de comunicación síncrona y asíncrona

En la figura 2.21a se aprecia como una tarea envía la información que necesita la otra sin necesidad de hacer una petición explicita por parte de la tarea que necesita la información, en cambio en 2.21b podemos ver como la tarea que necesita la información tiene que hacer una petición explicita de la información

2.4.3. Aglomeración

En esta etapa el diseño abstracto que se tiene de las etapas previas es llevado a la arquitectura paralela en la cual se va a ejecutar el programa, un aspecto importante de esta etapa es el de optimizar nuestro algoritmo para la arquitectura en la que se va a ejecutar. En este punto las tareas que se identificaron en la etapa de *particionamiento* pueden agruparse para formar tareas de mayor granularidad.

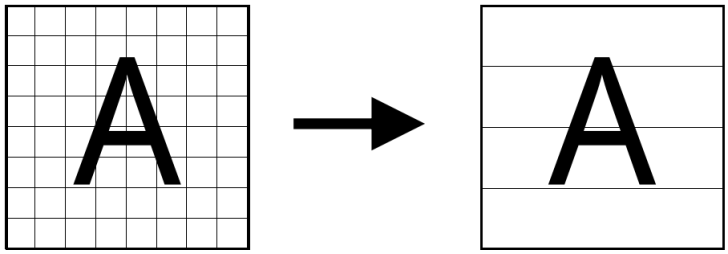


Figura 2.22: Modificación particionamiento en la etapa de aglomeración

Observamos en 2.22 como la descomposición de los datos se vio modificada, esto se debe a que en la etapa de particionamiento realizamos la descomposición "ideal"pero en la etapa de aglomeración los datos se particionan de acuerdo a las características del equipo en el cual vamos a ejecutar nuestro programa. Cuando se realiza la aglomeración es importante tener en cuenta las siguientes consideraciones:

- Las tareas que se pueden ejecutar independientemente las colocamos en procesadores diferentes.

- Las tareas que realizan operaciones de comunicación entre ellas deben ser ejecutadas en el mismo procesador, para disminuir los costos de estas operaciones

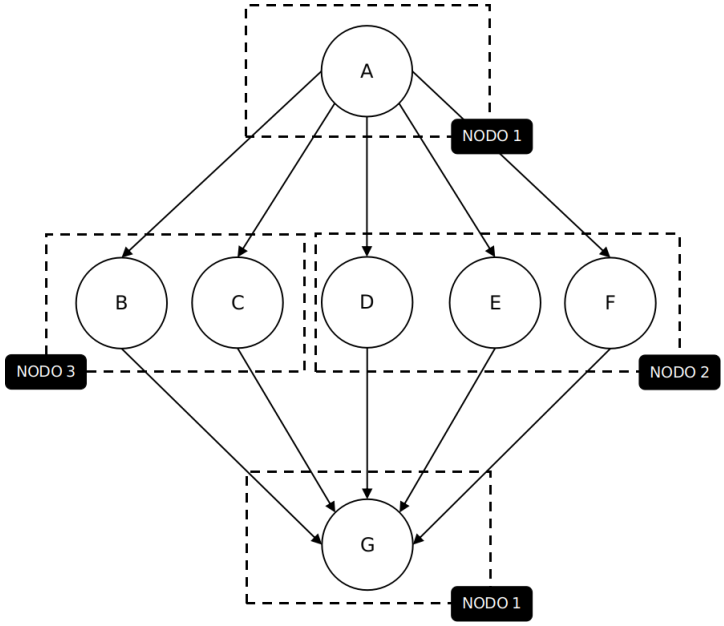


Figura 2.23: Ejemplos de mapeo de diferentes tareas

En la figura 2.23 se muestra un ejemplo de como asignar diferentes tareas en distintos nodos, en este ejemplo las tareas A y G son realizadas en el nodo 1, las tareas B y C en el nodo 3 y las tareas D, E y F en el nodo 2. La asignación de tareas a los nodos se hace dependiendo de las características de las tareas, en el ejemplo de la figura 2.23 puede ser que el nodo 1 no cuente con procesamiento en GPU, y las tareas B, C, D, E y F necesiten realizar operaciones en GPU por lo tanto estas tareas son asignadas a los nodos 2 y 3 los cuales cuentan con GPUs para realizar cálculos.

2.4.4. Mapeo

En esta última parte ya que se tiene el particionamiento de datos, el modelo de comunicación entre tareas y el agrupamiento de tareas, resta por implementar el algoritmo utilizando las tecnologías más convenientes para nuestro diseño. Cuando se realiza el mapeo es importante tener en cuenta las siguientes consideraciones:

- Características del problema
- En que tipo de arquitectura se va a ejecutar nuestro programa
- Las tecnologías que ofrecen mayores ventajas en la implementación

- El soporte técnico de la tecnología

Capítulo 3

Transformación de Householder y sus Aplicaciones

En el presente capítulo se explicara a detalle en que consiste la transformación de Householder y dos de sus aplicaciones la bifactorización de matrices y la bidiagonalización de matrices la cual es utilizada para calcular la descomposición en valores singulares.

En (Householder, 1958) se desarolla una transformación que realiza la reflexión de un punto sobre el plano. La reflexión se define por un vector unitario $\mathbf{v}$ el cual es ortogonal al plano, la reflexión de un punto $\mathbf{x}$ en el plano es la transformación lineal: $\mathbf{x} - 2\langle \mathbf{x}, \mathbf{v} \rangle = \mathbf{x} - 2\mathbf{v}\mathbf{v}^T\mathbf{x}$ o en su forma matricial: $\mathbf{P} = \mathbf{I} - 2\mathbf{v}\mathbf{v}^T$. A esta matriz se le conoce como matriz de Householder, la cual tiene las siguientes propiedades:

- Es simétrica, sea $\mathbf{v} \in \mathbb{R}^n$ el vector unitario que define la matriz $\mathbf{P}$

$$\mathbf{v}\mathbf{v}^T = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{pmatrix} \begin{pmatrix} v_1 & v_2 & v_3 & \cdots & v_n \end{pmatrix}$$

$$\mathbf{v}\mathbf{v}^T = \begin{pmatrix} v_1^2 & v_1v_2 & v_1v_3 & \cdots & v_1v_n \\ v_2v_1 & v_2^2 & v_2v_3 & \cdots & v_2v_n \\ v_3v_1 & v_3v_2 & v_3^2 & \cdots & v_3v_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_nv_1 & v_nv_2 & v_nv_3 & \cdots & v_n^2 \end{pmatrix}$$

de lo anterior observamos que los elementos de la diagonal principal de $\mathbf{v}\mathbf{v}^T$ son los elementos del vector $\mathbf{v}$ al cuadrado y los elementos por encima y por debajo de la diagonal son iguales, para formar la matriz $\mathbf{P}$ hay que multiplicar $\mathbf{v}\mathbf{v}^T$ por una escalar y posteriormente hay que sumarle la matriz identidad se sigue conservando la simetría de la matriz por lo tanto $\mathbf{P} = \mathbf{P}^T$.

- Es unitaria, sea

$$\begin{aligned} \mathbf{P} &= \mathbf{I} - 2\mathbf{v}\mathbf{v}^T \\ \mathbf{P}^T &= \mathbf{I} - 2\mathbf{v}\mathbf{v}^T \\ \mathbf{P}\mathbf{P}^T &= (\mathbf{I} - 2\mathbf{v}\mathbf{v}^T)(\mathbf{I} - 2\mathbf{v}\mathbf{v}^T) \\ \mathbf{P}\mathbf{P}^T &= \mathbf{I} - 2\mathbf{v}\mathbf{v}^T - 2\mathbf{v}\mathbf{v}^T + 4\mathbf{v}(\mathbf{v}^\mathbf{T}\mathbf{v})\mathbf{v}^\mathbf{T} \\ \mathbf{P}\mathbf{P}^T &= \mathbf{I} - 4\mathbf{v}\mathbf{v}^T + 4\mathbf{v}\mathbf{v}^T \\ \mathbf{P}\mathbf{P}^T &= \mathbf{I} \end{aligned}$$

- Se cumple que $\forall \mathbf{v} \in \mathbb{R}^n \setminus \{0\}$, $\mathbf{Pv} = -\mathbf{Pv}$

$$\mathbf{Pv} = (\mathbf{I} - 2\mathbf{v}\mathbf{v}^T)\mathbf{v}$$
$$\mathbf{Pv} = \mathbf{v} - 2\mathbf{v}(\mathbf{v}^T\mathbf{v})$$
$$\mathbf{Pv} = \mathbf{v} - 2\mathbf{v}$$
$$\mathbf{Pv} = -\mathbf{v}$$

- Sus valores propios son ± 1, el valor propio 1 tiene multiplicidad $n-1$, por lo tanto el valor propio -1 tiene multiplicidad de 1
- Su determinante es -1, esto se deduce del punto anterior ya que el determinante de una matriz es igual a multiplicar sus valores propios

3.1. Aplicaciones

La transformación de Householder se utiliza para volver cero todos los elementos de un vector excepto el primer elemento del vector. En este capítulo se presentan dos aplicaciones de esta transformación la primera de ellas es la bifactorización que se usa para encontrar la forma tridiagonal de una matriz simétrica y la segunda es como un paso para calcular la descomposición SVD de una matriz.

3.1.1. Bifactorización

El problema de tridiagonalizar una matriz radica en encontrar la matriz $\mathbf{P}$ que nos da como resultado una matriz semejante a la original y que además es tridiagonal. Las matrices tridiagonales tiene la siguiente forma:

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & 0 & \cdots & 0 & 0 & 0 \\ \alpha_{12} & \alpha_{22} & \alpha_{23} & \cdots & 0 & 0 & 0 \\ 0 & \alpha_{23} & \alpha_{33} & \ddots & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 & 0 \\ 0 & 0 & 0 & \ddots & \alpha_{n-2n-2} & \alpha_{n-2n-1} & 0 \\ 0 & 0 & 0 & \cdots & \alpha_{n-2n-1} & \alpha_{n-1n-1} & \alpha_{n-1n} \\ 0 & 0 & 0 & \cdots & 0 & \alpha_{n-1n} & \alpha_{nn} \end{pmatrix}$$

De lo anterior, observamos que si $A, B \in \mathbb{R}^{n\times n}$, y sea $\mathbf{B}$ una matriz tridiagonal semejante a $\mathbf{A}$, necesita-

remos $\mathbf{n^2}$ espacios de memoria para almacenar la matriz **A**. En cambio, para almacenar la matriz **B** solo se necesitan **n** espacios de memoria para la diagonal principal y para las otras dos diagonales se necesitan $\mathbf{n-1}$ espacios de memoria para cada una. Debido a que la matriz es simétrica podemos almacenar en un solo vector los valores de las diagonales que están por encima y por debajo de la principal. De esta manera se necesitan $\mathbf{2n-1}$ espacios de memoria para almacenar la matriz tridiagonal **B**.

Para transformar la matriz **A** en una matriz tridiagonal, se puede utilizar los métodos que se exponen en (Householder, 1958) o en (Givens, 1958). Los dos métodos presentan propiedades numéricas similares (Golub and Loan, 2012) (Highman, 2002), sin embargo para trabajar con matrices densas se utiliza el método de Householder (Press et al., 1992) (Dongarra et al., 1979), esto debido a que con Householder podemos hacer ceros mas elementos en un solo paso en comparación con el método de Givens el cual solo nos permite hacer cero un elemento a la vez.

El método de Householder consiste en encontrar una matriz **P** de la forma:

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

dónde:

- La matriz **P** es simétrica y ortogonal
- $v \in \mathbb{R}^n \backslash \{0\}$
- $Pv = -v$
- $\forall w \in \mathbf{R}^n, w \perp v \Longrightarrow Pw = w$

Sea $A \in \mathbb{R}^{n \times n}, a_i \in \mathbb{R}^n$ y $A = (a_1|a_2|\cdots|a_n)$, tomamos el vector a_1 y hacemos $v = a_1 \pm \|a_1\|_2 \cdot e_1$ con $e_1 = (1\ 0\ ...\ 0)^T$, ahora calculamos:

$$\begin{aligned} Pa_1 &= \left(\boldsymbol{I} - 2\frac{vv^T}{v^T v}\right) a_1 \\ &= a_1 - 2v\frac{(a_1 \pm \|a_1\|e_1)^T \cdot a_1}{(a_1 \pm \|a_1\|e_1)^T(a_1 \pm \|a_1\|e_1)} \\ &= a_1 - 2v\frac{\|a_1\|^2 \pm \|a_1\|a_{1(1)}}{2\|a_1\|^2 \pm 2\|a_1\|a_{1(1)}} \\ &= a_1 - v = \mp\|a_1\|e_1 \end{aligned}$$

De esto observamos como la matriz $\mathbf{P}$ hace cero todos los elementos de un vector dado, excepto el primer elemento.

Para llevar la matriz $\mathbf{A}$ a su forma tridiagonal, se realiza el proceso de ir encontrando las matrices $\mathbf{P}$ de manera iterativa. Así, para la primera iteración elegimos al vector $\mathbf{x} = (a_{21}\ a_{31}\ a_{41}\ ...\ a_{n1})^T$ para encontrar la primer matriz de Householder, haciendo $v = \mathbf{x} \pm \|\mathbf{x}\|_2 \cdot e_1$, esto es:

$$P_1 \cdot A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & & & & & & \\ 0 & & & \mathbf{P}_1 & & & \\ \vdots & & & & & & \\ 0 & & & & & & \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ k & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}$$

De aquí $k = \pm\|\mathbf{x}\|_2$, para finalizar la primera iteración, realizamos el siguiente producto de matrices para volver cero los elementos de la primera fila que no forman parte de la tridiagonal:

$$A' = P_1^T A P_1 = \begin{pmatrix} a_{11} & k & 0 & \cdots & 0 \\ k & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}$$

En la segunda iteración elegimos al vector $\mathbf{x} = (a_{32}\ a_{42}\ a_{52}\ ...\ a_{n2})^T$ y se procede de la misma manera que en el paso anterior para calcular $\mathbf{P}_2$:

$$P_2 \cdot A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & & & & & \\ \vdots & \vdots & & & \mathbf{P}_2 & & \\ 0 & 0 & & & & & \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{pmatrix}$$

La última iteración será la $n-2$ en la cual la matriz tridiagonal será el producto de todas las matrices $\mathbf{P}$ con la matriz $\mathbf{A}$, esto es:

$$A_{tridiag} = P_{n-2}P_{n-3}\cdots P_2P_1AP_1P_2\cdots P_{n-3}P_{n-2}$$

Algoritmo Secuencial

A continuación se describe el algoritmo para tridiagonalizar una matriz utilizando reflexiones de Householder.

Algoritmo 1 Bifactorización

Entrada: $A \in \mathbb{R}^{n\times n}$ simétrica
Salida: $B \in \mathbb{R}^{n\times n}$ *tridiagonal*

1: $B \leftarrow A$
2: **for** $k = 1, ..., n-2$ **do**
3: $\quad \mathbf{u} = B(k:n, k)$
4: $\quad \mathbf{u}_k = \mathbf{u} + sign(u_1)\|\mathbf{u}\|_2 e_1$
5: $\quad U \leftarrow \mathbf{u}\mathbf{u}^T$
6: $\quad U = I - \frac{2}{\mathbf{u}^T\mathbf{u}}U$
7: $\quad B \leftarrow UBU$
8: **end for**

Algoritmo Paralelo

La versión paralela del método de Householder que se propone en este trabajo, envía a los distintos procesos el vector $\mathbf{u}$ y a partir de éste generar la matriz de reflexión. Recordemos que si $\mathbf{A} \in \mathbb{R}^{n\times n}$ es la siguiente matriz:

$$\mathrm{A} = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2n} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \cdots & \alpha_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{pmatrix} ; \alpha_{ij} = \alpha ji,\ i \neq j$$

El vector **u** se define como:

$$\mathrm{u} = \begin{pmatrix} \alpha_{21} \\ \alpha_{31} \\ \alpha_{41} \\ \vdots \\ \alpha_{n1} \end{pmatrix} = \begin{pmatrix} \mathrm{u}_0 \\ \mathrm{u}_1 \\ \mathrm{u}_2 \\ \vdots \\ \mathrm{u}_\mathrm{n} \end{pmatrix}$$

$$\mathrm{u}_0 = \mathrm{u}_0 + sgn(\mathrm{u}_0) * \|\mathrm{u}\|_2$$

Con este vector se obtiene la matriz simétrica $\mathbf{H} \in \mathbb{R}^{n \times n}$, esta matriz en nuestro algoritmo se divide por filas, cada fila se encuentra en un proceso diferente, como se puede apreciar en la siguiente figura:

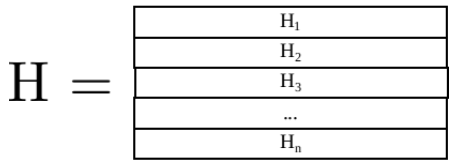


Figura 3.1: Matriz H dividida por filas

El siguiente paso es calcular la escalar β, esta escalar es enviada a cada proceso y cada uno de estos realiza la multiplicación de la fila de matriz **H** por ésta escalar.

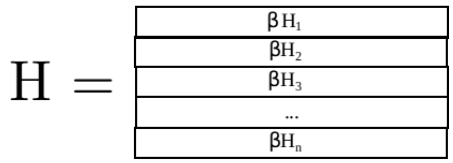


Figura 3.2: Matriz H dividida por filas multiplicada por β

En el paso siguiente es realizar la resta de $\boldsymbol{I} - \boldsymbol{H}$, para hacer esto en cada nodo se multiplica su vector por **-1** y posteriormente se le suma uno a los elementos de la diagonal principal de la matriz.

$$H = \begin{pmatrix} 1-h_{11} & -h_{12} & -h_{13} & \cdots & -h_{1n} \\ -h_{21} & 1-h_{22} & -h_{23} & \cdots & -h_{2n} \\ -h_{31} & -h_{32} & 1-h_{33} & \cdots & -h_3n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -h_{n1} & -h_{n2} & -h_{n3} & \cdots & 1-h_{nn} \end{pmatrix}$$

Figura 3.3: Resta de la matriz identidad menos la matriz H

El último paso es realizar la multiplicación ***HAH***, la matriz ***A*** se encuentra en el proceso principal y la envía por columnas a los demás procesos para que realicen la multiplicación por la fila de la matriz H que cada uno tiene. Esto se puede observar en la figura 3.4

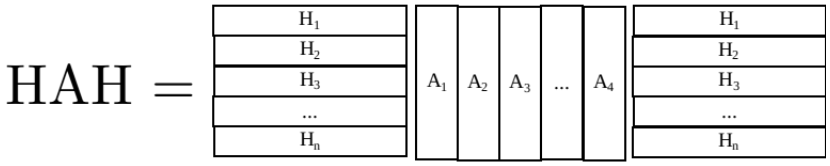


Figura 3.4: Multiplicación de la matriz A y H

Para hacer esta multiplicación primero se realiza la multiplicación por la izquierda ***HA*** y posteriormente se realiza la multiplicación por la derecha.

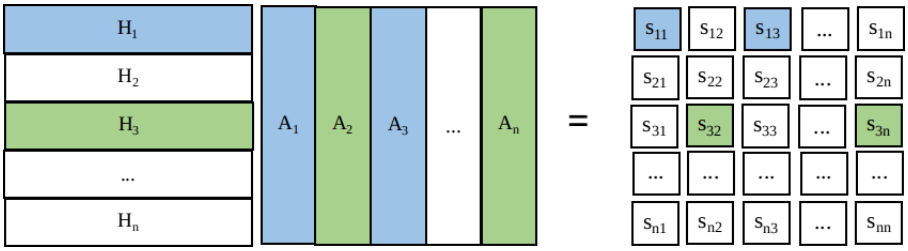


Figura 3.5: Multiplicación HA

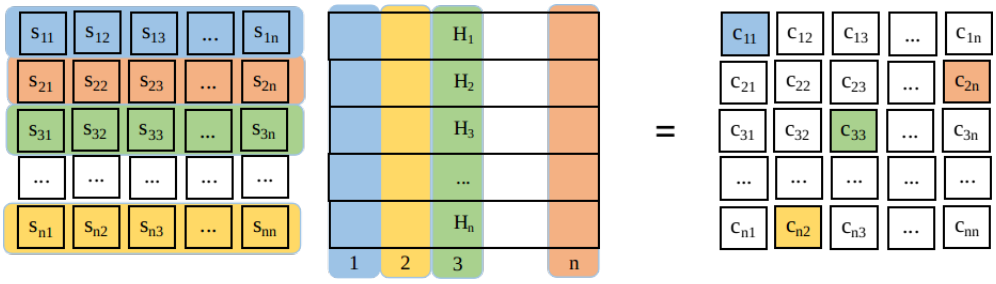


Figura 3.6: Multiplicación SH

En la figura 3.5 vemos como al realizar esta multiplicación en cada nodo, va a dar como resultado una matriz S, la cual se encuentra divida por filas en cada nodo. En la figura 3.6 vemos como para realizar la siguiente multiplicación es necesario recorrer la matriz H por columnas, pero dado que esta es una matriz simétrica podemos realizar la multiplicación haciendo el producto punto entre las filas de las matrices S y H.

Para terminar cada proceso envía la parte que realizó del calculo al proceso principal y éste lo une, dando como resultado la matriz de Householder. Esto se puede apreciar en la figura 3.7.

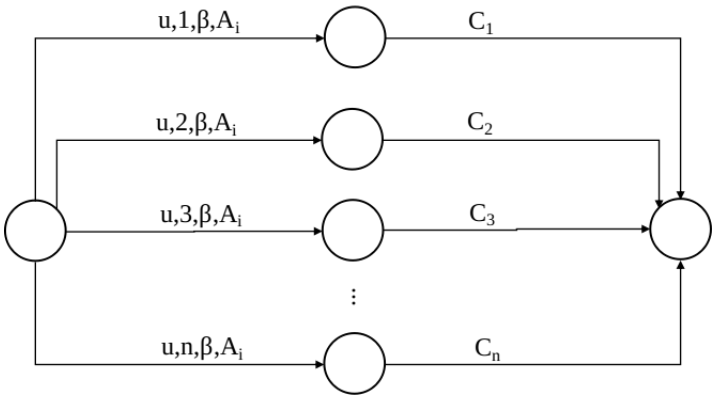


Figura 3.7: Comunicación entre procesos

3.1.2. Descomposición en Valores Singulares

Sean $A \in \mathbb{R}^{m\times n}$, $\sigma \in \mathbb{R}^{+}$, $\mathbf{u} \in \mathbb{R}^{m}$ y $\mathbf{v} \in \mathbb{R}^{n}$; tales que:

$$Av = \sigma u$$
$$A^{T}u = \sigma v$$

se dice que σ es un valor singular de A, $\mathbf{u}$ y $\mathbf{v}$ son vectores singulares(izquierdo y derecho respectivamente) de A. Si para un valor singular existen τ vectores singular por la derecha y τ vectores singulares por la izquierda, se dice que el valor singular tiene multiplicidad τ.

Dada una matriz $A \in \mathbb{R}^{m\times n}$, el producto matricial $\mathbf{U}\Sigma\mathbf{V}^{T}$ es la descomposición en valores singulares de la matriz A si:

- $\mathbf{U}$ y $\mathbf{V}$ tienen columnas ortonormales.

- Σ es una matriz diagonal y sus elementos son no negativos.
- $A = \mathbf{U}\Sigma\mathbf{V}^T$

sean p el numero de filas y q el numero de columnas de Σ, $\mathbf{U} \in \mathbb{R}^{m\times p},\ p \leq m$ y $\mathbf{V} \in \mathbb{R}^{n\times q},\ q \leq n$, hay dos formas de la descomposición en valores singulares:

- Si p = m y q = n. Las dimensiones de Σ son m $\times$ n y los elementos de la submatriz formada por las últimas m-n filas son todos cero.
- Si p = q = *min(*m,n*)*. La matriz Σ es cuadrada.

Los valores singulares de una matriz $B = A^T A$ son las raíces cuadradas de los valores propios, los valores singulares son siempre iguales o mayores a cero, debido a que la matriz $B = A^T A$ es una matriz *positive semidefinite*, esto implica que todos sus valores propios son positivos. El valor singular mayor es igual al valor de la norma espectral de la matriz B (Axler, 1997). Si se tiene una matriz bidiagonal sus valores propios son los elementos en la diagonal principal, pero si deseamos conocer los vectores singulares es necesario aplicar varias transformaciones matriciales para conocerlos (Hogben, 2006) (Golub and Loan, 2012).

La descomposición en valores singulares de una matriz resulta muy útil para realizar aproximaciones de la matriz original utilizando matrices de menor rango. Sea $A \in \mathbb{R}^{m\times n}$ entonces:

$$A = U\Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T \tag{3.1}$$

Dónde r puede tener valores entre $1 \leq r \leq n$, si deseamos hacer una aproximación de rango $r = 1, 2, ..., n$ es necesario tomar los primeros r valores y vectores singulares. De manera geométrica esto se puede ver como si cada punto en A es descompuesto en r componentes en la dirección de v_i, y cada $\sigma_i u_i$ es un vector que corresponde a las proyecciones de las filas de A en v_i, esto se puede ver de la ecuación (3.1)

$$\begin{aligned} A &= \sigma_i u_i v_i^T \\ Av_i &= \sigma_i u_i (v_i^T v_i) \\ Av_i &= \sigma_i u_i \end{aligned}$$

Para cualquier matriz $A \in \mathbb{R}^{m\times n}$ siempre existe la descomposición en valores singulares, si los valores singulares son todos distintos entonces los vectores singulares son únicos, sin embargo, si existen algunos valores

singulares que son iguales entonces los vectores singulares pueden ser cualesquiera vectores ortonormales del espacio generado por el correspondiente vector singular (Blum et al., 2016).

Algoritmo Secuencial

Para el cálculo numérico de la descomposición en valores singulares es necesario, reducir la matriz original a su forma bidiagonal usando la transformación de Householder, el pseudocódigo de este algoritmo corresponde al mostrado en el algoritmo 2.

Algoritmo 2 Bidiagonalización

Entrada: $A \in \mathbb{R}^{m\times n}, m \geq n$
Salida: $B \in \mathbb{R}^{n\times n}$ *bidiagonal superior*, $U \in \mathbb{R}^{m\times n}$ *y* $V \in \mathbb{R}^{n\times n}$, $A = UBV^T$

$B \leftarrow A$
$U = I \in \mathbb{R}^{m\times n}$
$V = I \in \mathbb{R}^{n\times n}$
for $i = 1, ..., n$ **do**
    Determinar la matriz de Householder L_i tal que la multiplicación por la izquierda no cambie el elemento en la posición B_{i-1i} y vuelva ceros los elementos por debajo de la diagonal principal en la columna i

$$L_i \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b_{i-1i} \\ b_i i \\ b_{i+1i} \\ \vdots \\ b_m i \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ b_{i-1i} \\ s \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \; s = \pm\sqrt{\sum_{k=i}^{m} b_{ki}^2}$$

    $B \leftarrow L_i B$
    $U \leftarrow U L_i$
    if $i \leq n-2$ **then**
        Calcular la matriz de Householder R_i tal que la multiplicación por la derecha no cambie el elemento B_{ii} y vuelva ceros los elementos a la derecha de este

$$\begin{pmatrix} 0 & \dots & 0 & b_{ii} & b_{ii+1} & b_{ii+2} & \dots & b_i n \end{pmatrix} R_i = \begin{pmatrix} 0 & \dots & 0 & b_{ii} & s & 0 & \dots & 0 \end{pmatrix}$$

$$s = \pm\sqrt{\sum_{j=i}^{n} b_{ik}^2}$$

        $B \leftarrow B R_i$
        $V \leftarrow R_i V$
    end if
end for

Un paso en el algoritmo de la diagonalización final es el método que se describe en (Golub and Kahan, 1965) el cual se encarga de ir eliminando uno a uno los elementos que no están en la diagonal principal utilizando rotaciones de Givens, el pseudocódigo corresponde al algoritmo 3. El algoritmo completo para calcular la descomposición en valores singulares es pseudocódigo que se muestra en el algoritmo 4.

Algoritmo 3 Golub-Kahan

Entrada: Una matriz $B \in \mathbb{R}^{n\times n}$ que es bidiagonal superior, dos matrices ortogonales L y R tales que $A = LBR^T$

Salida: La matriz B con elementos mas pequeños en su diagonal, las matrices L y R

1: Sea B_22 la sección diagonal de la matriz B con indices $p+1, ..., n-q$
2: Sea C la matriz inferior derecha de tamaño 2×2 del producto $B_{22}^T B$
3: Obtener los valores propios de C, λ_1 y λ_2
4: Hacer μ igual al valor propio mas cercano al valor c_{22}
5:

$$k = p+1$$
$$\alpha = b_{kk}^2 - \mu$$
$$\beta = b_{kk} b_{kk+1}$$

6: **for** $k = p+1, ..., n-q-1$ **do**
7: Calcular $c = \cos\theta$ y $s = \sin\theta$ tal que:

$$\begin{pmatrix}\alpha & \beta\end{pmatrix}\begin{pmatrix}c & s\\ -s & c\end{pmatrix} = \begin{pmatrix}\sqrt{\alpha^2+\beta^2} & 0\end{pmatrix}$$

8: $B \leftarrow BG_{kk+1}(c,s)$, $G_{kk+1}(c,s)$ es la rotación de Givens en las columnas k y $k+1$ de la matriz B
9: $R \leftarrow RG_{kk+1}(c,s)$
10: $\alpha = b_{kk}$ y $\beta = bk+1k$
11: Calcular $c = \cos\theta$ y $s = \sin\theta$ tal que:

$$\begin{pmatrix}c & -s\\ s & c\end{pmatrix}\begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \begin{pmatrix}\sqrt{\alpha^2+\beta^2}\\ 0\end{pmatrix}$$

12: $B \leftarrow G_{kk+1}(c,-s)B$, $G_{kk+1}(c,-s)$ es la rotación de Givens en las filas k y $k+1$ de la matriz B
13: $L \leftarrow G_{kk+1}(c,-s)L$
14: **if** $k \leq n-q-1$ **then**
15:

$$\alpha = b_{kk+1}$$
$$\beta = b_{kk+2}$$

16: **end if**
17: **end for**

Algoritmo 4 Calcular SVD

Entrada: $A \in \mathbb{R}^{m\times n}$
Salida: $\Sigma \in \mathbb{R}^{n\times n}$ matriz diagonal, $U \in \mathbb{R}^{m\times n}$ y $V \in \mathbb{R}^{n\times n}$ tales que $A = U\Sigma V^T$

Aplicar el algoritmo de Householder para bidiagonalizar y obtener B, U, V
repeat
  if $|b_{ii+1}| \leq \varepsilon(|b_{ii}| + |b_{i+1i+1}|)$, $i = 1, ..., n-1$ **then**
    $b_{ii+1} = 0$
  end if
  Sea

$$B = \begin{pmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{pmatrix}$$

  determinar los indices p y q de modo que $p < q$, B_{33} es diagonal y B_{22} es bidiagonal
  if $b_{ii} = 0$, para algún $i = p+1, ..., n-q-1$ **then**
    Aplicar las rotaciones de Givens para que el elemento b_{ii+1} sea cero
  else
    Aplicar el algoritmo de Golub-Kahan a las matrices B, U, V
  end if
until $q = n$
Σ = parte diagonal de B

Algoritmo Divide y Vencerás

Este método fue propuesto por (Cuppen, 1980) para resolver el problema de valores propios de una matriz tridiagonal simétrica, en (Gu and Eisenstat, 1995) muestran que este método puede ser usado también en matrices bidiagonales. La idea de este método es que teniendo una matriz bidiagonal es posible dividirla en varios subproblemas. Sea B la siguiente matriz de dimensiones $(N+1) \times N$:

$$B = \begin{pmatrix} \alpha_1 & 0 & 0 & 0 & \cdots & 0 \\ \beta_1 & \alpha_2 & 0 & 0 & \cdots & 0 \\ 0 & \beta_2 & \alpha_3 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & 0 \\ 0 & 0 & 0 & \beta_{n-2} & \alpha_{n-1} & 0 \\ 0 & 0 & 0 & 0 & \beta_{n-1} & \alpha_n \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

La matriz B puede ser dividida como:

$$B = \begin{pmatrix} B_1 & \alpha_k \mathbf{e}_k & 0 \\ 0 & \beta_k \mathbf{e}_1 & B_2 \end{pmatrix} \tag{3.2}$$

donde $k = \lfloor \frac{n}{2} \rfloor$, B_1 una matriz bidiagonal de dimensiones $k \times (k-1)$ y B_2 una matriz bidiagonal de

dimensiones $(N-k+1)\times(N-k)$.

Sea

$$B_i = \begin{pmatrix} Q_i & q_i \end{pmatrix} \begin{pmatrix} D_i \\ 0 \end{pmatrix} W_i^T \tag{3.3}$$

la descomposición SVD de B_i. Sea l_1^T la última fila de Q_1, λ_1 el último componente de q_1, l_2^T la primer fila de Q_2 y λ_2 el primer componente de q_2. Sustituyendo (3.3) en (3.2) obtenemos:

$$B = \begin{pmatrix} q_1 & Q_1 & 0 & 0 \\ 0 & 0 & Q_2 & q_2 \end{pmatrix} \begin{pmatrix} \alpha_k\lambda_1 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k l_2 & 0 & D_2 \\ \beta_k\lambda_2 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & W_1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & W_2 \end{pmatrix}^T \tag{3.4}$$

Como observamos en (3.4) la matriz del centro solamente tiene un elemento distinto de cero en la última fila, utilizando las rotaciones de Givens para volverlo cero se obtiene:

$$B = \left(\begin{pmatrix} c_0 q_1 & Q_1 & 0 \\ s_0 q_2 & 0 & Q_2 \end{pmatrix} \begin{pmatrix} -s_0 q_1 \\ c_0 q_2 \end{pmatrix} \right) \begin{pmatrix} r_0 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k l_2 & 0 & D_2 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & W_1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & W_2 \end{pmatrix}^T \equiv \begin{pmatrix} Q & q \end{pmatrix} \begin{pmatrix} M \\ 0 \end{pmatrix} W^T \tag{3.5}$$

donde:

$$\begin{aligned} r_0 &= \sqrt{(\alpha_k\lambda_1)^2 + (\beta_k\lambda_2)^2} \\ s_0 &= \tfrac{\beta_k\lambda_2}{r_0} \\ c_0 &= \tfrac{\alpha_k\lambda_1}{r_0} \end{aligned} \tag{3.6}$$

de (3.5) tenemos que la matriz M tiene la siguiente forma:

$$M = \begin{pmatrix} z_1 & 0 & 0 & \cdots & 0 \\ z_2 & d_1 & 0 & \cdots & 0 \\ z_3 & 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ z_n & 0 & 0 & \cdots & d_n \end{pmatrix}$$

ahora es necesario calcular los valores singulares de M, para ello como se muestra en (Gu and Eisenstat, 1995), (Hogben, 2006) se debe resolver la ecuación secular:

$$f(w) = 1 + \sum_{k=1}^{n} \frac{z_k^2}{d_k^2 - w^2} = 0 \tag{3.7}$$

al resolver (3.7) obtendremos los valores singulares aproximados de M, los cuales denotaremos por $\hat{w}_i$, estas aproximaciones pueden llevar a errores de precisión por ello es necesario construir la matriz $\hat{M}$ de la forma:

$$\hat{M} = \begin{pmatrix} \hat{z_1} & 0 & 0 & \cdots & 0 \\ \hat{z_2} & d_1 & 0 & \cdots & 0 \\ \hat{z_3} & 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ \hat{z_n} & 0 & 0 & \cdots & d_n \end{pmatrix}$$

la matriz $\hat{M}$ tiene la característica de que los valores singulares $\hat{w}_i$, son sus valores singulares exactos, para calcular los valores $\hat{z}_i$ utilizamos la siguiente ecuación:

$$\hat{z}_i = \sqrt{(\hat{w}_n^2 - d_i^2) \prod_{k=1}^{i-1} \frac{\hat{w}^2 - d_i^2}{d_k^2 - d_i^2} \prod_{k=1}^{n-1} \frac{\hat{w}^2 - d_i^2}{d_{k+1}^2 - d_i^2}}$$

una vez calculados todos los valores singulares $\hat{w}_i$ y $\hat{z}_i$, calculamos los vectores singulares utilizando las ecuaciones:

$$u_i = \left(\frac{\hat{z}_1}{d_1^2 - \hat{w}_i^2}, \cdots, \frac{\hat{z}_n}{d_n^2 - \hat{w}_i^2} \right)^T \left(\sqrt{\sum_{k=1}^{n} \frac{\hat{z}_k^2}{(d_k^2 - \hat{w}_i^2)^2}} \right)^{-1}$$

$$v_i = \left(-1, \frac{d_2 \hat{z}_1}{d_2^2 - \hat{w}_i^2}, \cdots, \frac{d_n \hat{z}_n}{d_n^2 - \hat{w}_i^2} \right)^T \left(\sqrt{1 + \sum_{k=2}^{n} \frac{d_k^2 \hat{z}_k^2}{(d_k^2 - \hat{w}_i^2)^2}} \right)^{-1}$$

finalmente asignamos $U = (u_1, u_2, ..., u_n)$, $V = (v_1, v_2, ..., v_n)$ y $\Sigma = diag(\hat{w}_1, \hat{w}_2, ..., \hat{w}_n)$.

El pseudocódigo de este método para calcular la descomposición en valores singulares se muestra en el algoritmo 5, en este algoritmo el caso base se resuelve utilizando el algoritmo 4.

Algoritmo 5 SVD Divide y venceras

Entrada: $B \in \mathbb{R}^{(n+1)\times n}$ *bidiagonal inferior*
Salida: $\Sigma \in \mathbb{R}^{n\times n}$ *diagonal*, $U \in \mathbb{R}^{m\times n}$ *y* $V \in \mathbb{R}^{n\times n}$, $A = U\Sigma V^T$

if $n < \eta$ **then**
    Resolver utilizando algoritmo 4
end if
Dividir la matriz B de la siguiente manera:

$$B = \begin{pmatrix} B_1 & \alpha_k \mathbf{e}_k & 0 \\ 0 & \beta_k \mathbf{e}_1 & B_2 \end{pmatrix}, k = \lfloor \frac{n}{2} \rfloor$$

Llamar a este algoritmo con la submatriz B_1
Llamar a este algoritmo con la submatriz B_2
Partir las matrices $U_i = (Qiqi)$
Calcular $l_1 = Q_1^T \mathbf{e}_k, \lambda_1 = q_1^T \mathbf{e}_k$
Calcular $l_2 = Q_2^T \mathbf{e}_1, \lambda_2 = q_2^T \mathbf{e}_1$
Partir B como:

$$B = \left(\begin{pmatrix} c_0 q_1 & Q_1 & 0 \\ s_0 q_2 & 0 & Q_2 \end{pmatrix} \begin{pmatrix} -s_0 q_1 \\ c_0 q_2 \end{pmatrix} \right) \begin{pmatrix} r_0 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k \lambda_2 & 0 & D_2 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & W_1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & W_2 \end{pmatrix}^T \equiv \begin{pmatrix} Q & q \end{pmatrix} \begin{pmatrix} M \\ 0 \end{pmatrix} W^T$$

Calcular

$$r_0 = \sqrt{(\alpha_k \lambda_1)^2 + (\beta_k \lambda_2)^2}$$
$$s_0 = \frac{\beta_k \lambda_2}{r_0}$$
$$c_0 = \frac{\alpha_k \lambda_1}{r_0}$$

Calcular los valores singulares, resolviendo la ecuación secular

$$f(w) = 1 + \sum_{k=1}^{n} \frac{z_k^2}{d_k^2 - w^2} = 0$$

for $i = 1, ...n$ **do**

$$\hat{z}_i = \sqrt{(\hat{w}_n^2 - d_i^2) \prod_{k=1}^{i-1} \frac{\hat{w}^2 - d_i^2}{d_k^2 - d_i^2} \prod_{k=1}^{n-1} \frac{\hat{w}^2 - d_i^2}{d_{k+1}^2 - d_i^2}}$$

end for
for $i = 1, ...n$ **do**

$$u_i = \left(\frac{\hat{z}_1}{d_1^2 - \hat{w}_i^2}, \cdots, \frac{\hat{z}_n}{d_n^2 - \hat{w}_i^2} \right)^T \left(\sqrt{\sum_{k=1}^{n} \frac{\hat{z}_k^2}{(d_k^2 - \hat{w}_i^2)^2}} \right)^{-1}$$
$$v_i = \left(-1, \frac{d_2 \hat{z}_1}{d_2^2 - \hat{w}_i^2}, \cdots, \frac{d_n \hat{z}_n}{d_n^2 - \hat{w}_i^2} \right)^T \left(\sqrt{1 + \sum_{k=2}^{n} \frac{d_k^2 \hat{z}_k^2}{(d_k^2 - \hat{w}_i^2)^2}} \right)^{-1}$$

end for
$U = (u_1, u_2, ..., u_3)$, $V = (v_1, v_2, ..., v_3)$

$$\Sigma = \begin{pmatrix} diag(\hat{w}_1, \hat{w}_2, ..., \hat{w}_n) \\ 0 \end{pmatrix}$$

$U \leftarrow (QU\ q)$
$V \leftarrow WV$ **return** Σ, U, V

Algoritmo Paralelo

El algoritmo paralelo es muy similar, al algoritmo de la bifactorización. Se envían a los distintos nodos el vector **u** y a partir de este generar una porción de la matriz de Householder en cada nodo. Para esto tomemos la matriz $A \in \mathbb{R}^{m \times n}$

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2n} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \cdots & \alpha_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{pmatrix}$$

El vector **u** se define como:

$$\mathrm{u} = \begin{pmatrix} \alpha_{21} \\ \alpha_{31} \\ \alpha_{41} \\ \vdots \\ \alpha_{n1} \end{pmatrix} = \begin{pmatrix} \mathrm{u}_0 \\ \mathrm{u}_1 \\ \mathrm{u}_2 \\ \vdots \\ \mathrm{u}_n \end{pmatrix}$$

$$\mathrm{u}_0 = \mathrm{u}_0 + sgn(\mathrm{u}_0) * \|\mathrm{u}\|_2$$

Al realizar el producto exterior del vector **u**, obtenemos la matriz simétrica **L**, en la versión paralela esta matriz se encuentra dividida por filas en los distintos nodos.

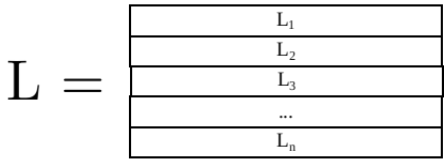


Figura 3.8: Matriz L dividida por filas

El siguiente paso es multiplicar la matriz **L** por la escalar β, esta se calcula en el nodo principal y es enviada al resto para que realizan la multiplicación de la escalar por la porción de la matriz que les corresponde.

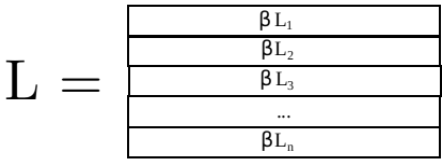


Figura 3.9: Matriz L dividida por filas y multiplicada por β

En el paso siguiente es realizar la resta de $\boldsymbol{I} - \boldsymbol{L}$, para hacer esto en cada nodo se multiplica su vector por **-1** y posteriormente se le suma uno a los elementos de la diagonal principal de la matriz.

$$H = \begin{pmatrix} 1-h_{11} & -h_{12} & -h_{13} & \cdots & -h_{1n} \\ -h_{21} & 1-h_{22} & -h_{23} & \cdots & -h_{2n} \\ -h_{31} & -h_{32} & 1-h_{33} & \cdots & -h_3n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -h_{n1} & -h_{n2} & -h_{n3} & \cdots & 1-h_{nn} \end{pmatrix}$$

Figura 3.10: Resta de la matriz identidad menos la matriz H

Lo siguiente es realizar la multiplicación de **LA**, la matriz A se encuentra en el proceso principal y este la envía por columnas a los demás procesos para que realicen la multiplicación por la fila de la matriz L que cada uno tiene.

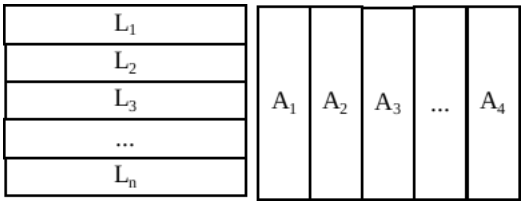


Figura 3.11: Multiplicación **LA**

Para poder realizar esta multiplicación primero hacemos la multiplicación a la izquierda y después por la derecha.

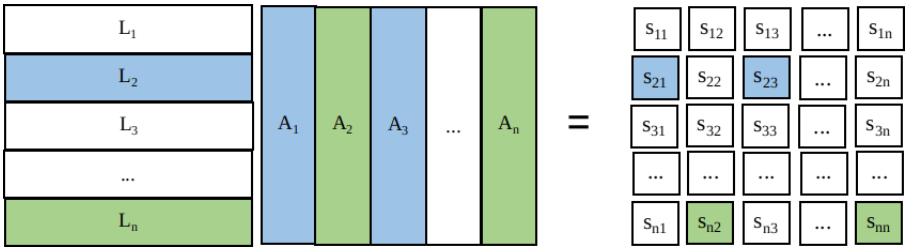


Figura 3.12: Multiplicación por la izquierda **LA**

Esta serie de multiplicaciones nos da como resultado la siguiente matriz

$$A' = \begin{pmatrix} \alpha'_{11} & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1n} \\ 0 & \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2n} \\ 0 & \alpha_{32} & \alpha_{33} & \cdots & \alpha_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{pmatrix}$$

Como observamos todos los elementos de la primer columna son cero excepto el primer elemento de la diagonal principal, ahora es necesario hacer lo mismo pero esta vez con los elementos de la primer fila para ello elegimos el vector $\mathbf{v}$ de la siguiente forma:

$$\mathbf{v} = \begin{pmatrix} \alpha_{12} & \alpha_{13} & \alpha_{14} & \cdots & \alpha_{1n} \end{pmatrix}$$

El procedimiento es igual que para el vector $\mathbf{u}$, se crea una matriz simétrica $\mathbf{R}$ a partir del producto exterior del vector $\mathbf{v}$, la matriz se distribuye de la misma manera en los distintos nodos, al final la multiplicación de la matriz $\mathbf{A'R}$ se realiza de la siguiente manera:

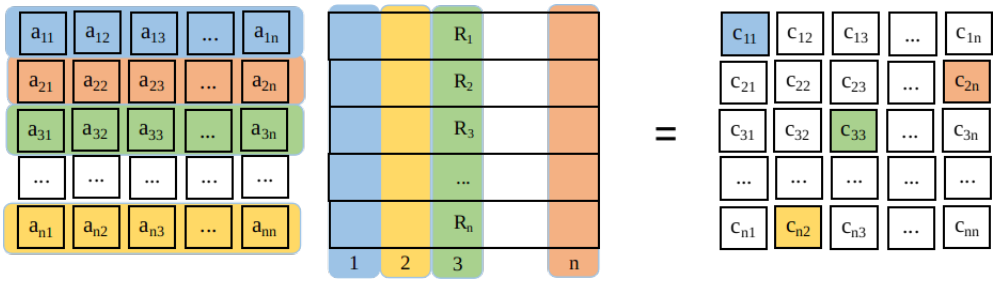


Figura 3.13: Multiplicación por la derecha $\mathbf{AR}$

Al realizar esta multiplicación nos da como resultado la siguiente matriz:

$$A' = \begin{pmatrix} \alpha'_{11} & \alpha'_{12} & 0 & \cdots & 0 \\ 0 & \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2n} \\ 0 & \alpha_{32} & \alpha_{33} & \cdots & \alpha_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{pmatrix}$$

En cada iteración se vuelven ceros los elementos de la i-ésima fila y columna, al realizar n iteraciones se obtiene una matriz bidiagonal de la siguiente forma:

$$B = \begin{pmatrix} \alpha_1 & 0 & 0 & 0 & \cdots & 0 \\ \beta_1 & \alpha_2 & 0 & 0 & \cdots & 0 \\ 0 & \beta_2 & \alpha_3 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots & 0 \\ 0 & 0 & 0 & \beta_{n-2} & \alpha_{n-1} & 0 \\ 0 & 0 & 0 & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

En este punto tenemos una matriz bidiagonal a la cual se le puede aplicar el algoritmo divide y vencerás, para esto primero tenemos que partir el problema original dependiendo del número de procesos disponibles.

a1															
b1	a2														
	b2	a3													
		b3	a4												
			b4	a5											
				b5	a6										
					b6	a7									
						b7	a8								
							b8	a9							
								b9	a10						
									b10	a11					
										b11	a12				
											b12	a13			
												b13	a14		
													b14	a15	
														b15	a16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 3.14: Problema de diagonalización original

El caso mas sencillo es cuando solo tenemos dos procesos como se ilustra en la siguiente figura:

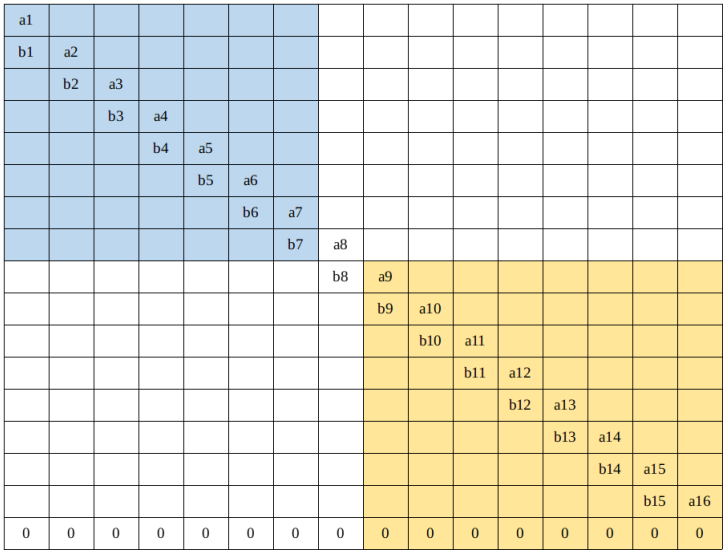


Figura 3.15: Particionamiento con dos procesos

Aquí un proceso se va a encargar de diagonalizar el área de azul y otra el área de amarillo, al final uno de ellos le enviara al otro su resultados para que el otro los una y solucionar el problema original.

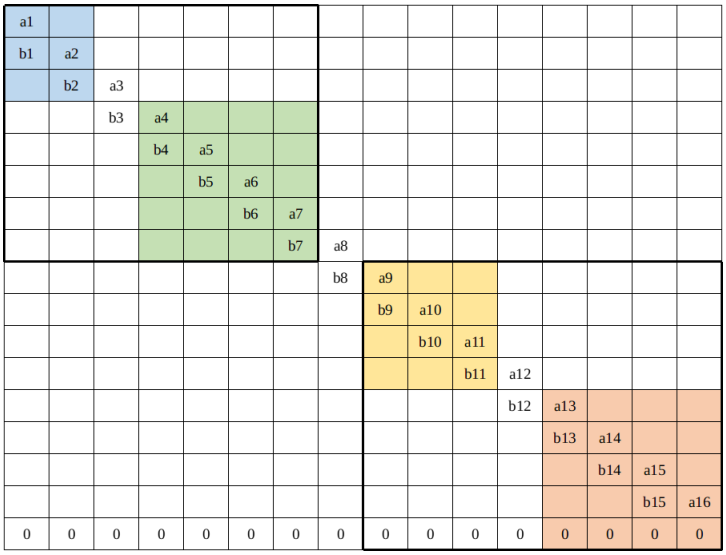


Figura 3.16: Particionamiento con cuatro procesos

En la figura superior vemos como sería el particionamiento con cuatro procesos, cuando los procesos del área azul y verde terminen el proceso de diagonalización, unirán sus resultados y al final los uniran con los resultados de los procesos amarillo y rojo.

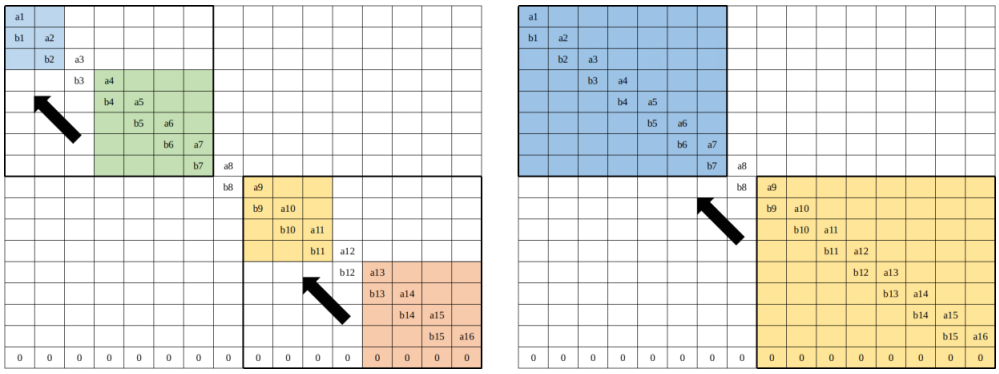


Figura 3.17: Unión de las soluciones parciales del proceso de diagonalización

Capítulo 4

Implementación

En el presente capítulo se describe el desarrollo de la tesis, los problemas que se obtuvieron durante el desarrollo y como se resolvieron. En la primera parte del capítulo se muestran las tecnologías y las caracteristicas de la arquitectura en la que se desarrollo la tesis. Posteriormente se detallan las distintas implementaciones que se desarrollaron de la bifactorización de matrices y de la descomposición SVD.

El presente trabajo se desarrollo en lenguaje C, utilizando las siguientes APIs(*Application Programming Interface*, Interfaz de Programación de Aplicaciones):

- **OpenMP** Es un API que utiliza directivas de compilación y diversas bibliotecas de desarrollo que son utilizadas para especificar paralelismo de alto nivel en programas de Fortran y C/C++. Ofrece un modelo de paralelismo basado en hilos (procesos ligeros) (Chapman et al., 2008).

- **MPI** Es una API para la comunicación de procesos principalmente enfocada en un modelo de memoria distribuida. Utiliza el modelo de programación paralela de paso de mensajes: los datos son enviados de un espacio de direcciones de un proceso a otra de otro proceso a través de operaciones cooperativas en cada proceso. Este tipo de modelos es comúnmente usado en aplicaciones que son ejecutadas en un cluster de computadoras, o una grid. MPI provee distintas funciones para efectuar la comunicación entre procesos. Un programa MPI es un programa escrito en C/C++ o Fortran que se ejecuta sobre un conjunto de procesadores o en todos los procesadores en un cluster. Es labor del programador implementar la distribución de datos y la comunicación entre procesos (Gropp et al., 1999).

- **CUDA** Es una plataforma de cálculo paralelo de NVIDIA que utiliza la arquitectura de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento en el rendimiento del sistema.

CUDA utiliza extensiones del lenguaje C y Fortran como medio de programación. El código de CUDA es ejecutado en la GPU mediante llamadas a funciones que son realizadas en el CPU. El GPU tiene su propio planificador que asigna los kernels (código que se ejecuta en GPU) al hardware del GPU. Un programa consiste de una o mas fases que son ejecutadas en el host(CPU) o en el device(GPU). Las fases que exhiben poco o ningún paralelismo suelen ser implementadas en el código de host. Las fases que son factibles a ser paralelizadas en una arquitectura SIMD son implementadas en el código del device (Cook, 2012) (Kirk and Wen-Mei, 2016) (Sanders and Kandrot, 2010) (Wilt, 2013).

La biblioteca de MPI fue utilizada para realizar la comunicación entre los distintos procesos para tener un modelo de memoria compartida. OpenMP fue usada para realizar algunas operaciones en paralelo usando multihilos y CUDA se utilizó para realizar operaciones en la GPU.

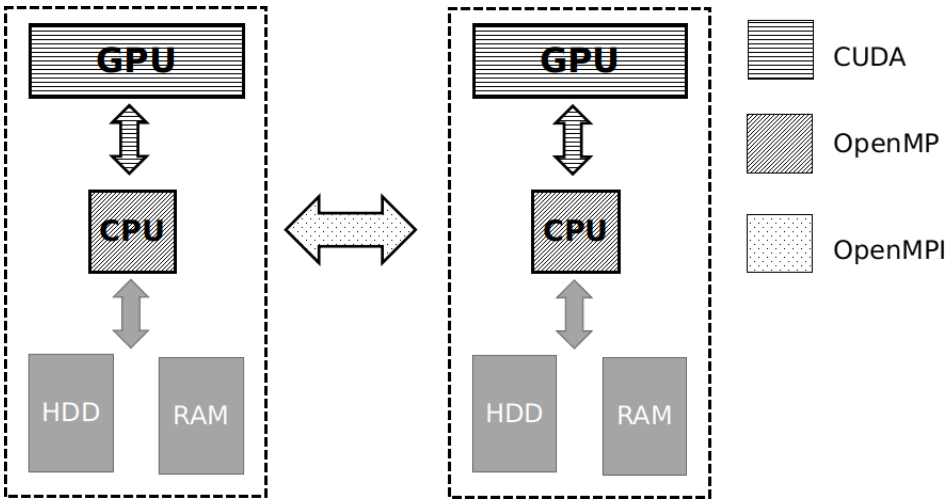


Figura 4.1: En esta imagen se muestra como se utilizaron las bibliotecas de MPI, OpenMP y CUDA en la implementación

En la figura 4.1 observamos como una parte muy importante es la comunicación entre los distintos procesos que se van a estar ejecuntando, para lograr tener una buena implementación es necesario que estos tiempos de comunicación sean lo menor posible. Para lograr esto se necesita contar con una muy buena estrategia de particionamiento de datos que minimice el número de operaciones de comunicación. Otra parte importante es el intercambio de información entre CPU y GPU, ya que los datos sobre los cuales se va a estar operando se encuentran almacenados en la memoria RAM del servidor, uno de las características de nuestra implementación es que hace el menor uso posible de este tipo de instrucciones.

4.1. Bifactorización

Para obtener la forma tridiagonal de una matriz simétrica se utiliza un método iterativo que en cada iteración calcula la matriz de Householder que va a volviendo ceros todos los elementos que se encuentran por debajo de la diagonal inferior de la i-ésima columna y también todos los elementos a la derecha en la i-ésima fila de la matriz. En las siguientes secciones se describen las distintas implementaciones que se realizaron de este método.

4.1.1. Implementación Secuencial

Esta implementación es básicamente la implementación en lenguaje C del algoritmo descrito en 3.1.1, las tareas que se desarrollaron en esta implementación son:

- Cálculo de la norma 2 de un vector
- Producto interno entre vectores
- Producto externo de vectores
- Multiplicación de submatrices
- Suma de submatrices
- Multiplicación de matriz por una escalar

Las estructuras de datos que utilizaron para almacenar las matrices fueron arreglos tipo **double** de tamaño $n \times n$ tanto para la matriz de entrada como para las matrices de Householder y también para la matriz resultante, como en cada iteración el tamaño de la matriz de Householder disminuye en una fila y una columna, las operaciones entre submatrices se implementaron recorriendo los índices del arreglo de tamaño $n \times n$.

4.1.2. Implementación Paralela con OpenMP

Paralelizar un algoritmo utilizando OpenMP es relativamente sencillo y debido a que la transformación de Householder implica operaciones matriciales y vectoriales, como lo son multiplicación de matrices, resta de matrices, productos vectoriales y calculo de normas, estas operaciones son fácilmente paralelizables utilizando las directivas de compilación de OpenMP. Las tareas que se paralelizaron utilizando OpenMP fueron:

- Cálculo de la norma 2 de un vector

- Producto interno entre vectores
- Producto externo de vectores
- Multiplicación de submatrices
- Suma de submatrices
- Multiplicación de matriz por una escalar

Un punto muy importante a tener en cuenta con esta implementación es que los hilos que se crean para ejecutar las tareas en paralelo son administrados por el sistema operativo, por lo cual la paralelización con esta biblioteca puede resultar en operaciones excesivas de creación y destrucción de hilos, lo que puede ser un problema en instancias muy grandes del problema.

4.1.3. Implementación Paralela en GPU

Implementar un algoritmo en una GPU puede representar una gran ventaja, pues este tipo de dispositivos tienen un gran poder de cómputo. Aunque la implementación no es tan directa como en el caso de OpenMP, ya que en estos dispositivos se debe tomar en cuenta que se va a trabajar con una arquitectura paralela de tipo SIMD, esto hace que la implementación sea mas complicada. Las tareas que se implementaron en GPU utilizando CUDA fueron:

- Cálculo de la norma 2 de un vector
- Producto interno entre vectores
- Producto externo de vectores
- Multiplicación de submatrices
- Suma de submatrices
- Multiplicación de matriz por una escalar

Debido a las características de la arquitectura de las GPU, nuevamente se uso como estructura de datos un arreglo unidimensional de tipo **double** de tamaño $n \times n$, un aspecto que es importante a tener en cuenta en esta implementación son las operaciones de intercambio de datos entre CPU y GPU, como la matriz con la que deseamos trabajar se encuentra alojada en la memoria principal del equipo es necesario copiarla a la memoria de la GPU como se ve en la figura 4.2, como este método requiere $n - 1$ iteraciones para terminar, la implementación que se hizo realiza el menor número posible de este tipo de operaciones.

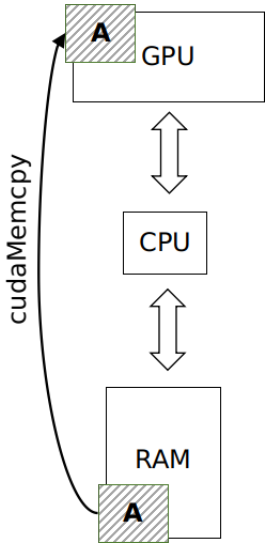


Figura 4.2: Copiado de la matriz A de la memoria principal a la GPU

4.1.4. Implementación en Arquitectura Híbrida-Heterogénea

Como se ha mencionado anteriormente, para lograr bifactorizar una matriz es necesario realizar dos multiplicaciones matriciales en cada iteración del método, además son realizadas operaciones vectoriales como el calcular la norma de un vector, suma de matrices y multiplicación de una escalar por una matriz, todas estas operaciones son efectuadas en la GPU. Las operaciones de envío y recepción de datos se realizan en el CPU.

En la figura 4.3 en la etapa 1 se envía el vector **u** a todos los nodos utilizando una operación de *Broadcast*, el vector **u** es la i-ésima fila de la matriz **A**, los nodos esclavos y maestro calculan una parte de la matriz de Householder(**H**). La parte que se calcula en cada nodo depende del *rango* del nodo y del número total de procesos que se están ejecutando. Es importante hacer notar que a pesar de que MPI balancea las cargas en los nodos, esta implementación asume que todas las GPUs tienen las mismas capacidades.

La etapa 2 que se muestra en la figura 4.3 comienza cuando cada proceso recibe el vector **u**, de acuerdo a lo descrito en el capítulo 3.1.1, lo primero que se hace es calcular la norma 2 del vector, el producto punto y una escalar, todas estas operaciones se realizan en el CPU utilizando OpenMP, ya que al ser operaciones de reducción, es decir, de un vector obtenemos una escalar es mejor realizar este tipo de operaciones en el CPU porque en la GPU se necesitarían demasiadas sincronizaciones entre los hilos. En esa misma etapa ya cuando cada proceso a calculado la parte que le corresponde de la matriz de Householder el proceso maestro envía a cada nodo las columnas de la matriz original para que realicen la multiplicación parcial con la matriz

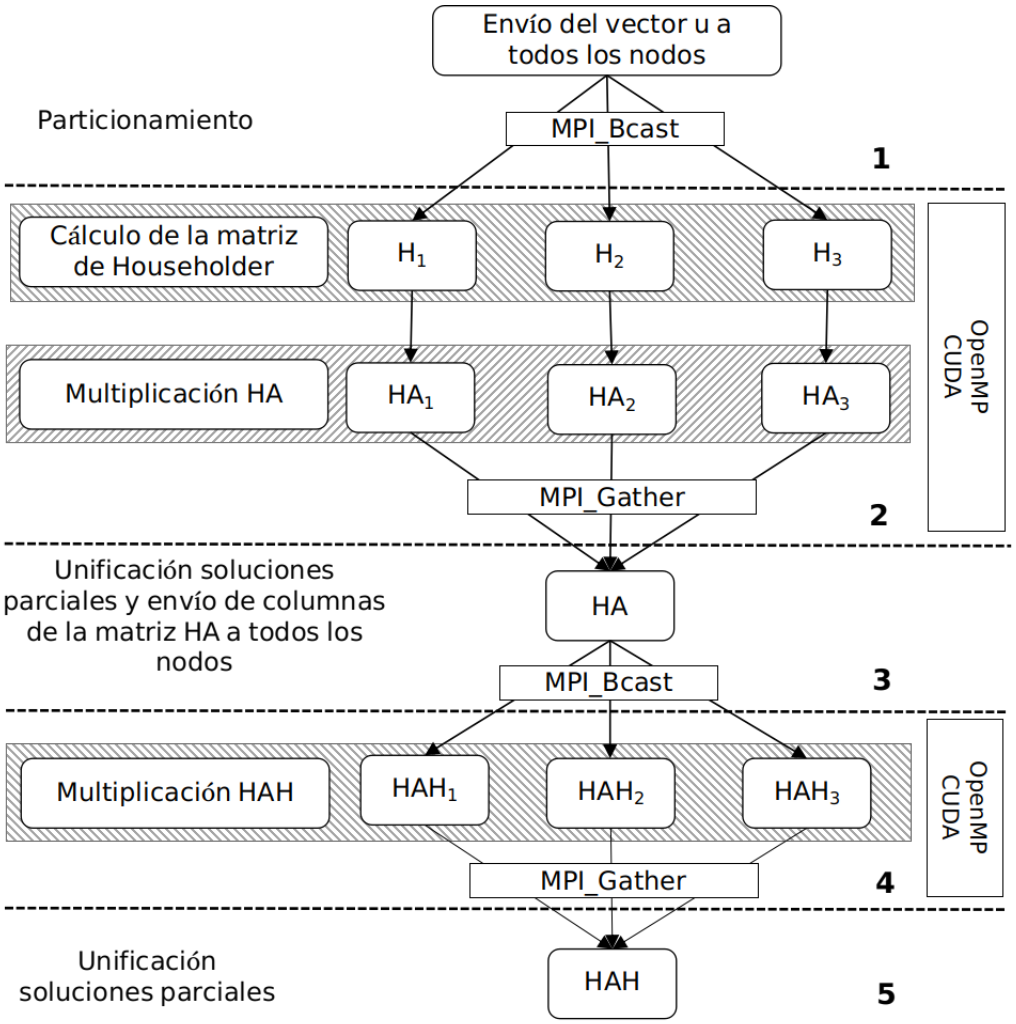


Figura 4.3: Implementación de la bifactorización

de Householder.

En la etapa 3 de la figura 4.3 los procesos esclavos envían las soluciones al proceso maestro y este las unifica para tener la matriz resultante de la multiplicación de la matriz **H** y **A**, ya unificadas las soluciones envía a todos los nodos las filas de la matriz **HA**.

La etapa 4 de la figura 4.3 cada proceso recibe una a una las filas de la matriz **HA** y realiza la multiplicación parcial de las matrices **HA** y **H**, una vez que los procesos tienen la multiplicación parcial la envían al proceso maestro.

En la última etapa, la etapa 5 de la figura 4.3 el proceso maestro unifica las soluciones parciales que le envían los procesos esclavos para formar la matriz **HAH**, si la iteración es menor a $n-1$ se sigue iterando hasta conseguir la matriz tridiagonal.

4.1.5. Consideraciones

Es importante realizar las siguientes aclaraciones respecto a la implementación del algoritmo en lenguaje C.

- Cuando se envía a los nodos las columnas para realizar la multiplicación **HA** como las matrices son simétricas se envían siempre las filas, debido a que las matrices son almacenadas por filas.
- En cada iteración no se añaden unos a la diagonal principal de la matriz de reflexión como se indica en (Householder, 1958), si no que se van recorriendo los indices de las matrices que se van multiplicando.
- Al final de cada iteración se hace un reordenamiento de la mattriz **HAH**, debido a que en esta última multiplicación la matriz queda dividida por columnas en los distintos nodos.

4.2. Descomposición SVD

La implementación de la descomposición SVD esta dividida en dos etapas: la etapa de bidiagonalización y la etapa de diagonalización.

4.2.1. Etapa de Bidiagonalización

Esta etapa es muy parecida a la bifactorización salvo que aquí es necesario calcular dos matrices de Householder, la matriz por la izquierda(**U**) y la matriz por la derecha(**V**) esto por que la bidiagonalización está

definida para cualquier tipo de matrices no solo para matrices simétricas como es el caso de la bifactorización, debido a esto la comunicación es un poco más complicada ya que es necesario realizar mas operaciones de comunicación grupales como se puede ver en la figura 4.4.

4.2.1.1. Implementación Secuencial

Se implementó el algoritmo 2 en lenguaje C, las tareas que se desarrollaron en esta implementación son:

- Multiplicación de submatrices
- Suma de submatrices

Las estructuras de datos que se usaron en la implementación fueron arreglos unidimensionales, de tamaño $m \times n$ para la matriz de entrada, de tamaño $k \times k, k = min(m, n)$ para la matriz de salida, de tamaño $m \times m$ para la matriz de Householder izquierda y de tamaño $n \times n$ para la matriz de Householder derecha. De la implementación de bifactorización se reutilizaron las funciones para el cálculo de la norma 2 de un vector, los productos internos y externos de vectores y multiplicación por una escalar. Se reescribieron las funciones para la multiplicación y suma de submatrices, esto por que las matrices de Householder izquierda y derecha son de distinto tamaño.

4.2.1.2. Implementación Paralela con OpenMP

El método de bidiagonalización es altamente paralelizable debido a las operaciones que utilizan en el proceso. Se realizo una implementación con OpenMP, se utilizaron las directivas que ofrece la biblioteca para paralelizar las tareas que comprende el método. Las tareas que se paralelizaron fueron:

- Multiplicación de submatrices
- Suma de submatrices

4.2.1.3. Implementación Paralela en GPU

Para aprovechar de la mejor manera posible una GPU un punto que es muy importante considerar son los tiempos de intercambio de datos entre el CPU y la GPU, de nueva cuenta esta implementación se realizó con esto en mente, fueron reutilizados de la implementación en GPU de el método de bifactorización las tareas de cálculo de norma 2, productos interno y externo de vectores y multiplicación de una matriz por una escalar. Las tareas que fueron implementadas nuevamente fueron:

- Multiplicación de submatrices

- Suma de submatrices

Las estructuras de datos utilizadas fueron arreglos unidimensionales de tamaño $m \times n$ para la matriz de entrada la cual se encuentra almacenada tanto en la memoria prinicipal como en la GPU, un arreglo de tamaño $k \times k, k = min(m, n)$ para la matriz de salida, un arreglo de tamaño $m \times m$ para la matriz de Householder izquierda y otro de tamaño $n \times n$ para la matriz de Householder derecha.

4.2.1.4. Implementación en Arquitectura Híbrida-Heterogénea

Para lograr bidiagonalizar una matriz se deben calcular dos matrices diferentes en cada iteración del método, si tenemos como entrada una matriz $A \in \mathbb{R}^{m \times n}$ la matrices de Householder tienen dimensiones $m\ timesm$ para la matriz izquierda y $n \times n$ para la matriz derecha.

En la figura 4.4 en la etapa 1 se envían los $m - i + 1$ elementos de la i-ésima columna de la matriz **A**, dónde i corresponde a la iteración actual, con estos elementos se forma el vector **u** y es enviado a todos los procesos utilizando un mensaje tipo *Broadcast*.

La etapa 2 de la figura 4.4 una vez que todos los procesos tienen el vector **u**, calculan una parte de la matriz de Householder izquierda(**U**) de acuerdo a su *rango* y al número de procesos, una vez calculada la matriz el proceso maestro envía una por una las columnas de la matriz A para realizar la multiplicación parcial y se envía al proceso maestro.

En la etapa 3 de la figura 4.4 el proceso maestro recibe las multiplicaciones parciales de todos los procesos y las unifica formando la matriz **UA**, posteriormente envía a cada proceso la primer fila menos el primer elemento de la fila a todos los procesos.

En la etapa 4 de la figura 4.4 se calcula la matriz de Householder derecha(**V**), cada proceso calcula una parte de la matriz de acuerdo a su *rango* y al número de procesos, cuando todos los procesos han terminado de calcular la parte correspondiente de la matriz **V** esperan a que el proceso maestro les envíe una por una las filas de la matriz **UA** como se ve en la etapa 5 de la figura 4.4, para realizar la multiplicación de las matrices **UA** y **V**.

Cada proceso calcula una parte de la matriz **UAV** como se ve en la figura 4.4 en la etapa 6, cuando los procesos terminan de realizar la multiplicación parcial la envían al proceso padre para que este la unifique como se ve en la etapa 7 de la figura 4.4. Este proceso se repite hasta que se haya alcanzado $n-1$ iteraciones.

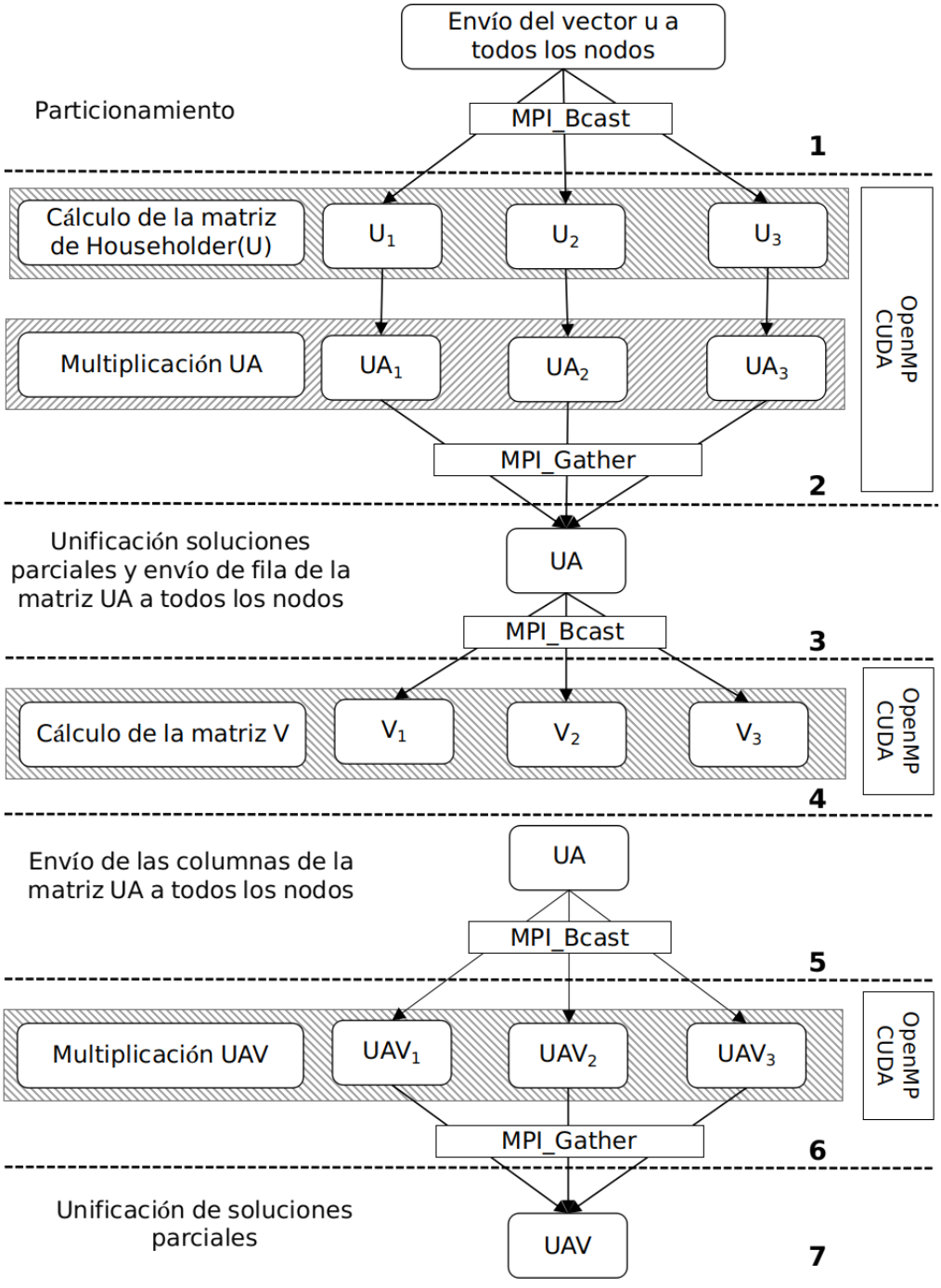


Figura 4.4: Implementación de la bidiagonalización

4.2.2. Etapa de Diagonalización

En este paso se diagonaliza la matriz bidiagonal, resultante de la etapa de bidiagonalización, utilizando el método divide y vencerás implementado en (Estrella-Cruz, 2014). Este algoritmo consiste en ir dividiendo la matriz original en pequeñas submatrices hasta que se tiene una matriz de tamaño muy pequeño, en esta implementación se considero que las matrices para el caso base de la recursión sean de tamaños 2×1, 3×2 y 4×3. Las matrices pequeñas se resuelven utilizando el algoritmo que se presenta en (Press et al., 1992), y posteriormente se van uniendo las soluciones parciales para solucionar el problema principal. Un problema fundamental en el algoritmo divide y vencerás para calcular la descomposición en valores singulares, es encontrar las raíces de la ecuación secular (3.7), para esto se utiliza el método propuesto en (Li, 1993), esta implementación se reutilizó del trabajo de tesis de (Estrella-Cruz, 2014).

4.2.2.1. Implementación Secuencial e Implementaciones Paralelas

Se implementó en lenguaje C el algoritmo 5, se reutilizaron del trabajo de (Estrella-Cruz, 2014) las siguientes funciones:

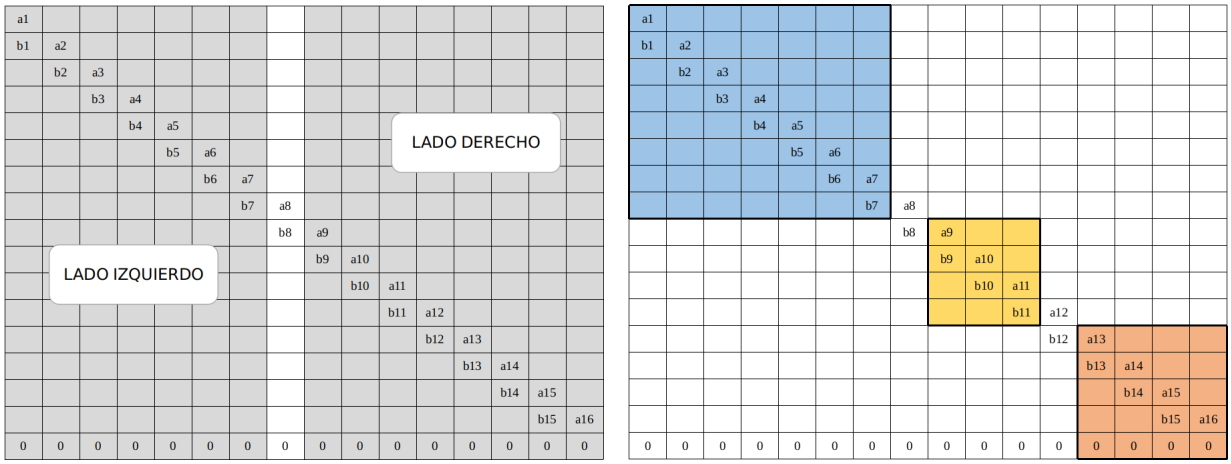
- Función de deflación para la ecuación secular.
- Función para calcular las raíces de la ecuación secular.
- Función para calcular los vectores singulares izquierdos.

Además se implementaron las siguientes funciones:

- Función para calcular la descomposición SVD para las matrices del caso base.
- Función para calcular los vectores singulares derechos.

Las estructuras de datos utilizadas fueron arreglos unidimensionales, para la matriz de entrada se usó un arreglo de tamaño $2n - 1$, para las matrices de salida los arreglos tienen las siguientes dimensiones: para los valores singulares se usó un arreglo de tamaño n y para los vectores singulares izquierdos y derechos se usó un arreglo de tamaño $n \times n$.

Las implementaciones paralelas en OpenMP y en GPU de las funciones para resolver la ecuación secular y calcular los vectores singulares izquierdos fueron tomadas del trabajo de (Estrella-Cruz, 2014), en este trabajo solo se implementó en paralelo tanto en OpenMP como en GPU las funciones para calcular los vectores singulares derechos.



(a) Lado izquierdo y derecho de la matriz

(b) Particionamiento de una matriz con tres procesos

Figura 4.5: Ejemplo de particionamiento de una matriz

4.2.2.2. Implementación en Arquitectura Híbrida-Heterogénea

El método divide y vencerás para diagonalizar matrices es altamente paralelizable ya que se pueden resolver en paralelo los subproblemas del método, la parte complicada de ésta estrategia es realizar las particiones cuando se tiene un número de procesos que no es potencia de dos, ya que el método en cada nivel de la recursión divide en dos los subproblemas, para resolver esto se asigno el número de procesos de acuerdo a la fórmula siguiente:

$$
\begin{aligned}
np &= \text{número total de procesos} \\
np_i &= \text{número de procesos al lado izquierdo de la recursión} \\
np_d &= \text{número de procesos al lado derecho de la recursipon} \\
x &= \lfloor \log_2(np) \rfloor - 1 \\
np_d &= 2^x \\
np_i &= np - np_d
\end{aligned} \tag{4.1}
$$

Un ejemplo de particionamiento de la matriz con tres procesos se puede ver en la figura 4.5, se observa como del lado izquierdo de la matriz solo hay un proceso mientras que el lado derecho se divide en dos procesos, se toma como columna central de la matriz la columna $\lfloor \frac{n}{2} \rfloor$, donde n es el número de columnas de la matriz.

En la figura 4.6 en la etapa 1 se lleva a cabo el particionamiento de la matriz de acuerdo a la fórmula

(4.1) en caso de que el número de procesos no sea potencia de dos, en caso de que los procesos sean potencia de dos se crean tantas particiones como procesos, cada una de tamaño $\frac{n}{np}$, donde n es el número de columnas de la matriz y np el número de procesos. Cuando se han calculado los índices de las submatrices se envían a cada proceso utilizando la operación *Send* de MPI.

En la etapa 2 de la figura 4.6 cada proceso resuelve el subproblema asignado usando el algoritmo 5 una vez que todos los procesos resuelven los subproblemas son enviadas las matrices parciales de vectores singulares izquierdos($\mathbf{U}$), valores singulares($\mathbf{\Sigma}$) y vectores singulares derechos($\mathbf{V}$) al nodo maestro para que este las una.

Cuando el proceso maestro recibe todas las soluciones parciales, envía esas soluciones parciales a los procesos como se puede ver en la etapa 3 de la figura 4.6, de cada lado de la matriz un proceso MPI es descartado, por ejemplo en la figura 4.5b el proceso que esta en amarillo une su solución con la del proceso anaranjado y el proceso anaranjado finaliza, el proceso azul espera a que termine el proceso anaranjado para unir las dos soluciones.

En la etapa 4 los procesos unen las soluciones parciales de acuerdo al algoritmo 5, una vez que las unieron las envían al proceso maestro para que las unifique. El proceso de enviar las soluciones parciales a los nodos esclavos se repite hasta que solo queden dos soluciones parciales y estas son unidas en el proceso padre la cual seria la etapa 5 de la figura 4.6.

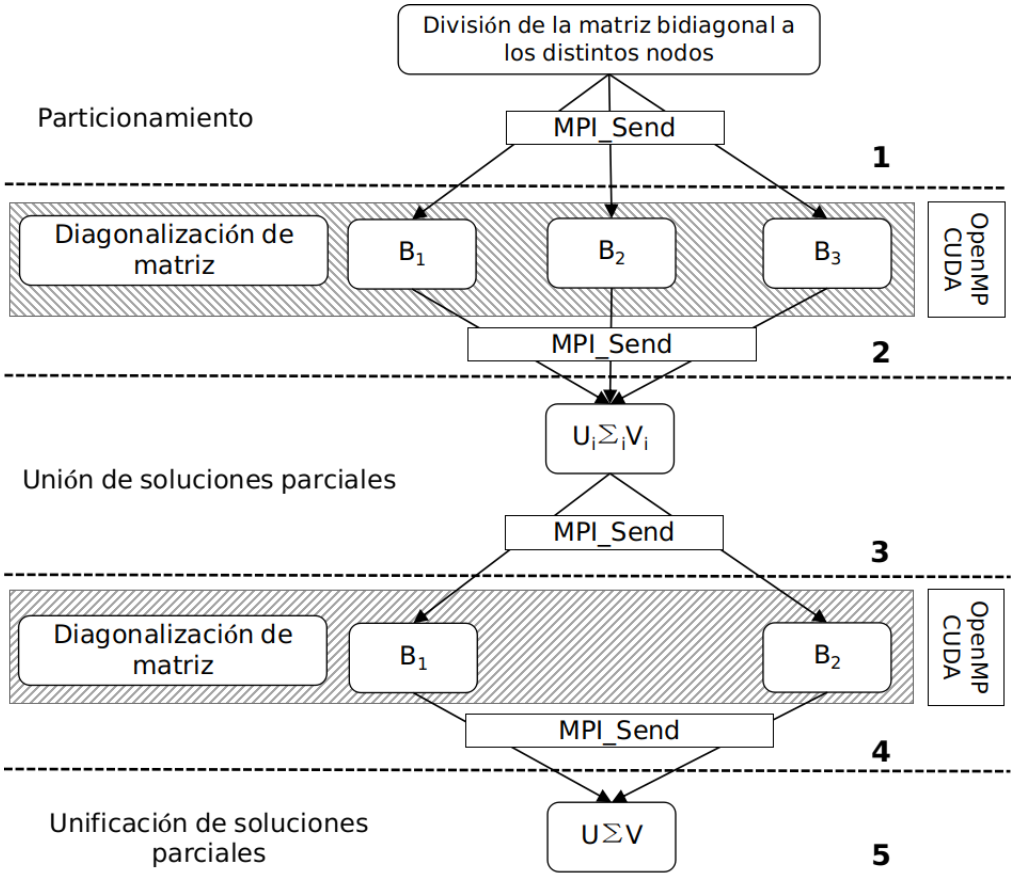


Figura 4.6: Implementación de la diagonalización

Capítulo 5

Pruebas y Resultados

En esta sección se describen las pruebas realizadas y se muestran los resultados obtenidos. Para la ejecución de las pruebas se generaron matrices densas aleatorias de hasta 4000×4000 elementos y se tomaron los tiempos de ejecución, de comunicación entre procesos, de intercambio de información entre CPU y GPU.

5.1. Infraestructura

Para el desarrollo de este trabajo se utilizará un servidor con las siguientes características:

- **CPU**
  - **Modelo:** Intel Xeon X5675
  - **Velocidad:** 3.07GHz
  - **Memoria caché:** 12 MB
  - **Número de cores:** 6 físicos, 6 virtuales
- **Memoria RAM:** 23 GB

El servidor además cuenta con tres GPU, las cuales cuentan con las siguientes características:

GPU	Memoria(MB)	Máx. hilos por bloque	CUDA Cores	Vel. de Memoria(MHz)	Vel. GPU(MHz)
Tesla C2070	5375	1024	448	1494	1147
GeForce GTX 460	1024	1024	336	1850	1530
Quadro K2000	2048	1024	384	2000	954

También se utilizo una laptop con las siguientes características:

- **CPU**

- **Modelo:** Intel i7 6700HQ
- **Velocidad:** 3.5GHz
- **Memoria caché:** 6 MB
- **Número de cores:** 4 físicos, 4 virtuales

- **Memoria RAM:** 8 GB

La laptop cuenta con una GPU con las siguientes características:

GPU	Memoria(MB)	Máx. hilos por bloque	CUDA Cores	Vel. de Memoria(MHz)	Vel. GPU(MHz)
GeForce GTX 960M	4044	1024	640	2505	1176

La programación será realizada utilizando C, en conjunto con las siguientes APIs:

- **CUDA (versión 6.5)**, permite el desarrollo en GPU.
- **OpenMP (versión 4.4.7)**, brinda capacidades de programación multihilo en CPU.
- **OpenMPI (versión 1.4)**, permite realizar la parte de comunicación entre procesos ejecutándose en distintos nodos.

5.2. Bifactorización

En esta sección se encuentran las pruebas realizadas para la bifactorización de matrices utilizando la transformación de Householder.

Se generaron matrices simétricas aleatorias de distintas dimensiones y se ejecutaron implementaciones secuenciales, paralelas en CPU, híbrida utilizando CPU+GPU, y finalmente en una implementación híbrida heterogénea utilizando dos y tres GPUs.

5.2.1. Resultados

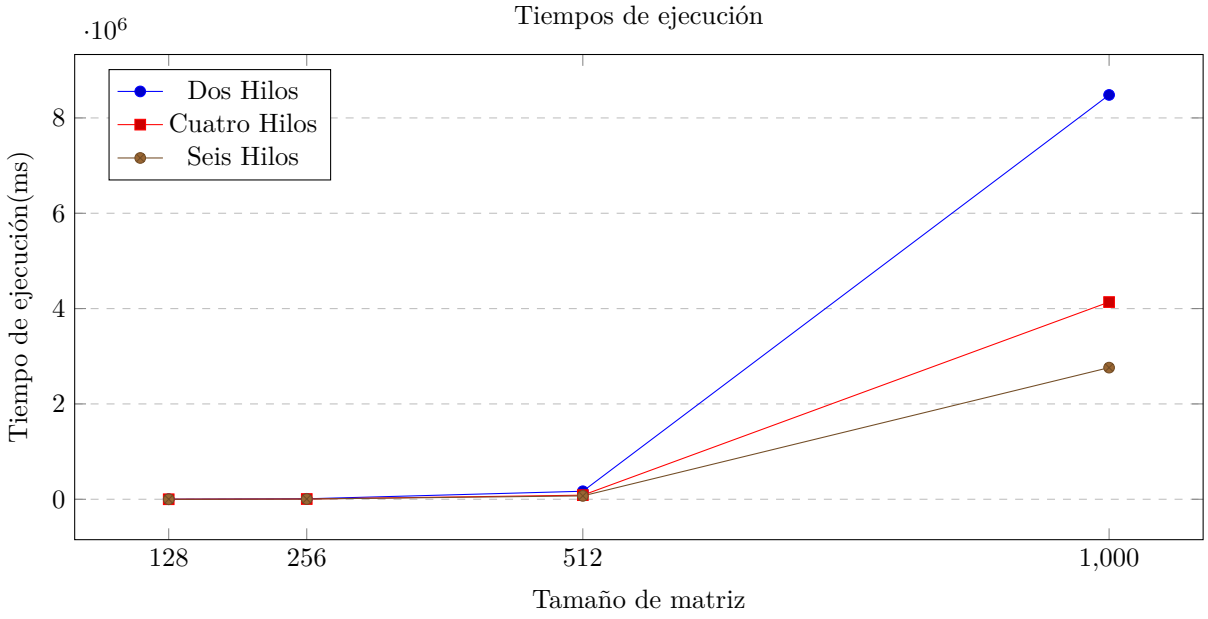
En las siguientes secciones se muestran los resultados de las pruebas realizadas

5.2.1.1. Implementación en CPU

En esta sección se presentan los tiempos de ejecución de la implementación paralela en CPU utilizando OpenMP.

n	Tiempos(ms)			
	Un hilo	Dos hilos	Cuatro hilos	Seis hilos
128	819	413	214	150
256	22 134	9 836	4 997	4 444
512	365 599	168 001	85 826	70 363
1 000	17 046 837	8 482 279	4 137 022	2 761 388

En la siguiente gráfica se pueden comparar mejor los resultados obtenidos:



De la gráfica podemos observar que como a medida que se incrementan el número de hilos mejora el tiempo de ejecución, vemos como los tiempos son muy similares en matrices de tamaño pequeño, pero a partir de matriz de un millón de elementos empieza a hacerse notoria esa diferencia.

5.2.1.2. Implementación Híbrida

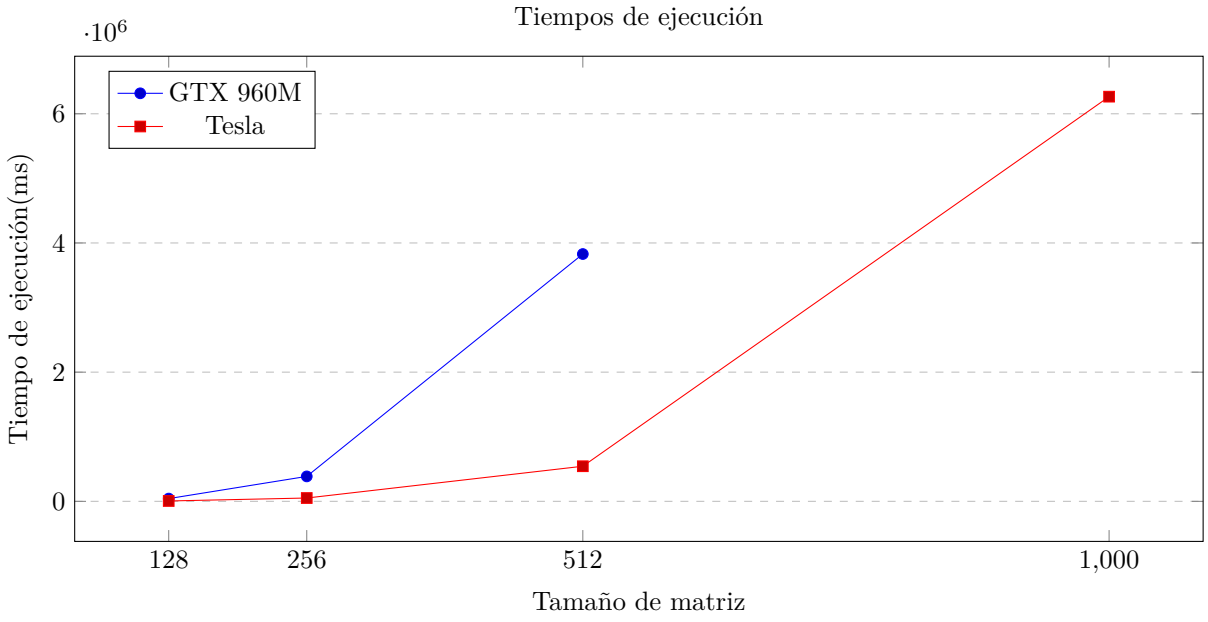
En esta sección se presentan los tiempos de ejecución de la implementación paralela híbrida utilizando GPU+CPU. En la siguiente tabla se muestran los tiempos de ejecución en la tarjeta GTX 960M.

n	Tiempos(ms)		
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Total
128	49.25	42 357.80	42 407.05
256	186.18	385 910.47	386 096.65
512	764.07	3 827 876.50	3 828 640.57

En la siguiente tabla se muestran los tiempos de ejecución en la tarjeta Tesla.

n	Tiempos(ms)		
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Total
128	65.59	5 554.12	5 619.71
256	253.98	51 127.34	51 381.32
512	1 027.45	542 280.13	543 307.58
1 000	3914.59	6 260 953.50	6 264 868.09
2 000	15 671.66	63 241 900.00	63 257 571.66

En la siguiente gráfica observamos como la misma implementación tiene una reducción considerable en el tiempo de ejecución dependiendo de la tarjeta en que se ejecute.



5.2.1.3. Implementación Híbrida Heterogénea

En esta sección se presentan los tiempos de ejecución de la implementación paralela híbrida heterogénea utilizando múltiples CPUs y GPUs.

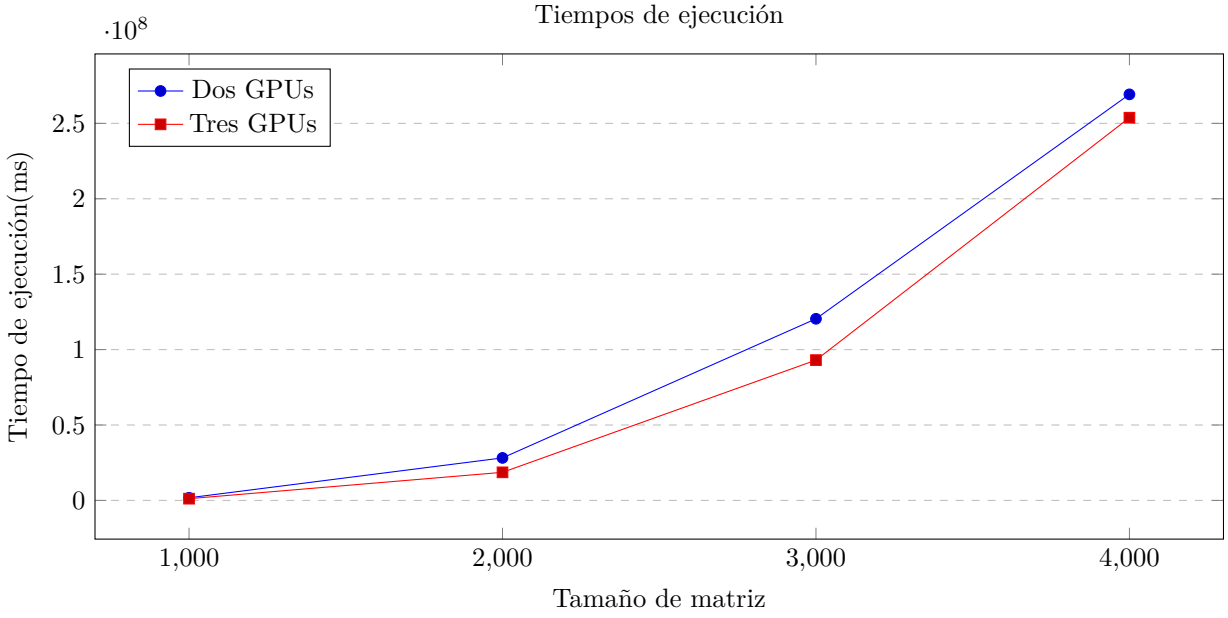
Implementación con dos GPUs

n	Tiempos(ms)			
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Comunicación(MPI)	Total
1 000	13 565.85	1 724 308.75	5 232.46	1 743 107.06
2 000	71 268.21	28 066 792.00	42 426.12	28 180 486.33
3 000	177 776.97	120 106 832.00	135 439.37	120 420 048.34
4 000	348 623.71	268 435 456.00	287 675.46	269 071 755.17

Implementación con tres GPUs

n	Tiempos(ms)			
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Comunicación(MPI)	Total
1 000	15 875.75	1 101 020.00	35 200.49	1 152 096.24
2 000	70 530.17	18 481 052.00	69 298.65	18 620 880.82
3 000	191 979.02	92 628 416.00	206 430.76	93 026 825.78
4 000	354 307.13	253 067 424.00	314 666.45	253 736 397.58

En la siguiente gráfica observamos de mejor manera la disminución en los tiempos de ejecución conforme se incrementan el número de GPUs utilizadas.



5.3. Descomposición SVD

En esta sección se presentan las pruebas y los resultados obtenidos de la descomposición SVD.

Para la descomposición en valores singulares las pruebas se dividieron en dos partes, la primera es la etapa de bidiagonalización de la matriz densa y la segunda es la etapa de diagonalización.

Las pruebas de bidiagonalización consistieron en generar matrices densas aleatorias y se posteriormente se midió el tiempo de ejecución y los tiempos de intercambio de datos. Las pruebas de la etapa de bidiagonalización se realizaron utilizando como entrada una matriz bidiagonal y se midió el tiempo de ejecución y los tiempos de intercambio de datos.

5.3.1. Bidiagonalización

En esta sección se presentan los resultados obtenidos de las pruebas de la parte de bidiagonalización.

5.3.1.1. Implementación en CPU

En esta sección se presentan los tiempos de ejecución obtenidos usando una implementación paralela con OpenMP con distintos números de hilos.

n	Tiempos(ms)			
	Un hilo	Dos hilos	Cuatro hilos	Seis hilos
128	901	517	246	172
256	23 194	10 761	5 281	4 815
512	367 324	169 594	87 376	71 673
1 000	18 639 103	9 736 324	5 985 197	4 271 817

5.3.1.2. Implementación Híbrida

A continuación se muestran los tiempos de ejecución y de intercambio de datos obtenidos de la implementación en CPU+GPU, utilizando la tarjeta Tesla.

n	Tiempos(ms)		
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Total
128	10.13	836.55	846.68
256	26.61	11 925.35	11 951.96
512	85.87	208 298.81	208 384.68
1 000	417.37	3 143 962.75	3 144 380.12
2 000	2 805.07	28 984 152.00	28 986 957.07

5.3.1.3. Implementación Híbrida Heterogénea

En esta sección se muestran los resultados obtenidos de las pruebas de la implementación híbrida heterogénea. usando dos y tres GPUs.

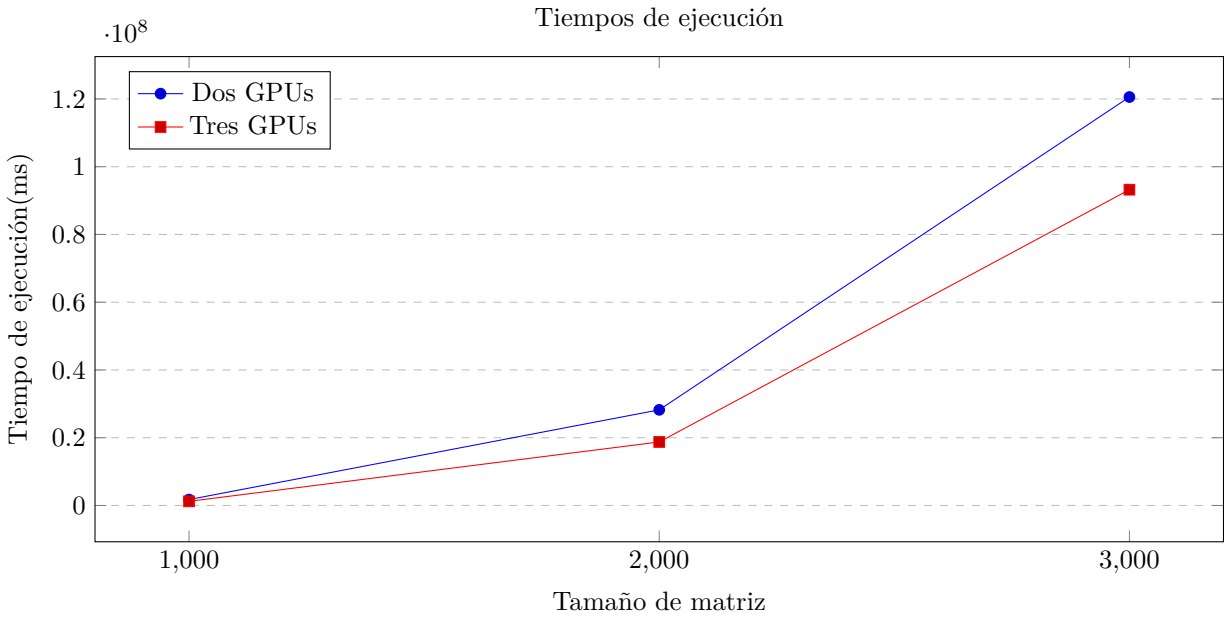
Implementación con dos GPUs

n	Tiempos(ms)			
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Comunicación(MPI)	Total
1 000	13 433.50	1 729 053.00	8 148.91	1 750 635.41
2 000	69 527.45	28 105 076.00	56 087.15	28 230 690.6
3 000	199 250.73	120 191 512.00	176 489.33	120 567 252.06
4 000	415 340.16	268 435 456.00	401 763.37	269 252 559.53

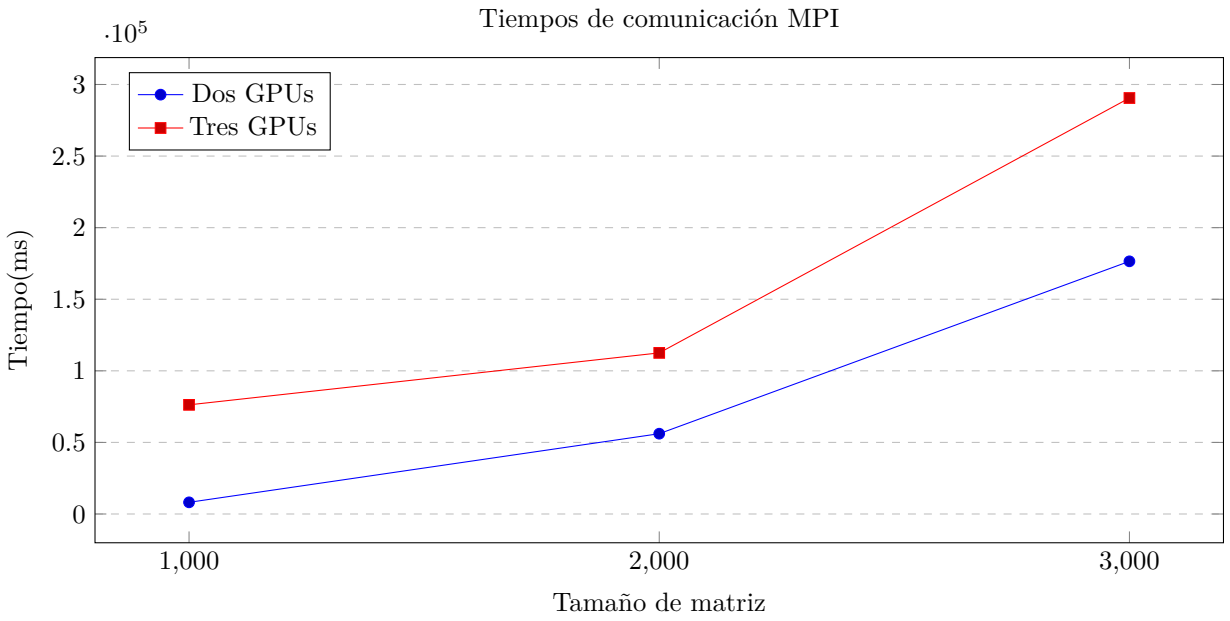
Implementación con tres GPUs

n	Tiempos(ms)			
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Comunicación(MPI)	Total
1 000	15 169.23	1 133 287.75	76 241.72	1 224 698.70
2 000	76 970.31	18 559 042.00	112 525.16	18 748 537.47
3 000	192 834.67	92 728 904.00	290 547.63	93 212 286.30
4 000	525 863.69	253 304 816.00	1 702 132.51	255 532 812.20

En la siguiente gráfica se observa como disminuyen los tiempos de ejecución cuando se utilizan tres GPUs y conforme se incrementa el tamaño de la matriz.



De la misma manera que en el caso de la bifactorización, hay un incremento en el tiempo de intercambio de datos entre los procesos pero este es poco en comparación con la disminución que observamos en los tiempos de ejecución.



5.3.2. Diagonalización

En esta sección se presentan los resultados obtenidos de las diferentes implementaciones de la etapa de la diagonalización.

5.3.2.1. Implementación en CPU

En la siguiente tabla se muestran los tiempos de ejecución obtenidos de la implementación con OpenMP usando distinto número de hilos.

n	Tiempos(ms)			
	Un hilo	Dos hilos	Cuatro hilos	Seis hilos
128	20	11	18	15
256	187	104	72	59
512	1 625	871	483	493
1 000	14 427	6 991	3 674	3 483
2 000	154 686	72 085	41 507	35 280
3 000	627 935	282 732	168 829	136 202
4 000	2 040 730	640 241	456 014	333 527

5.3.2.2. Implementación Híbrida

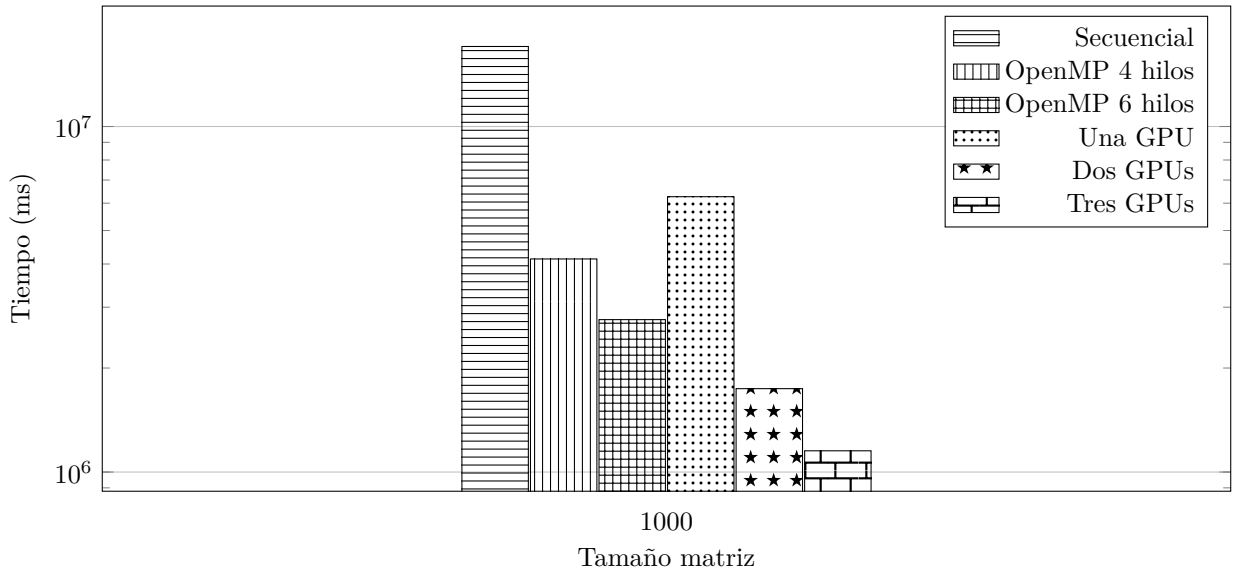
En la siguiente tabla se muestran los tiempos de ejecución obtenidos para la implementación en GPU-CPU.

n	Tiempos(ms)		
	Intercambio Datos(CPU-GPU)	Ejecución Kernel	Total
128	0.23	8.12	8.35
256	1.01	46.76	47.77
512	4.33	366.29	370.62
1 000	16.97	2 905.56	2 922.53
2 000	69.42	15 892.62	15 962.04
3 000	96.87	16 357.87	16 454.74
4 000	173.04	32 312.54	32 485.58

5.4. Discusión

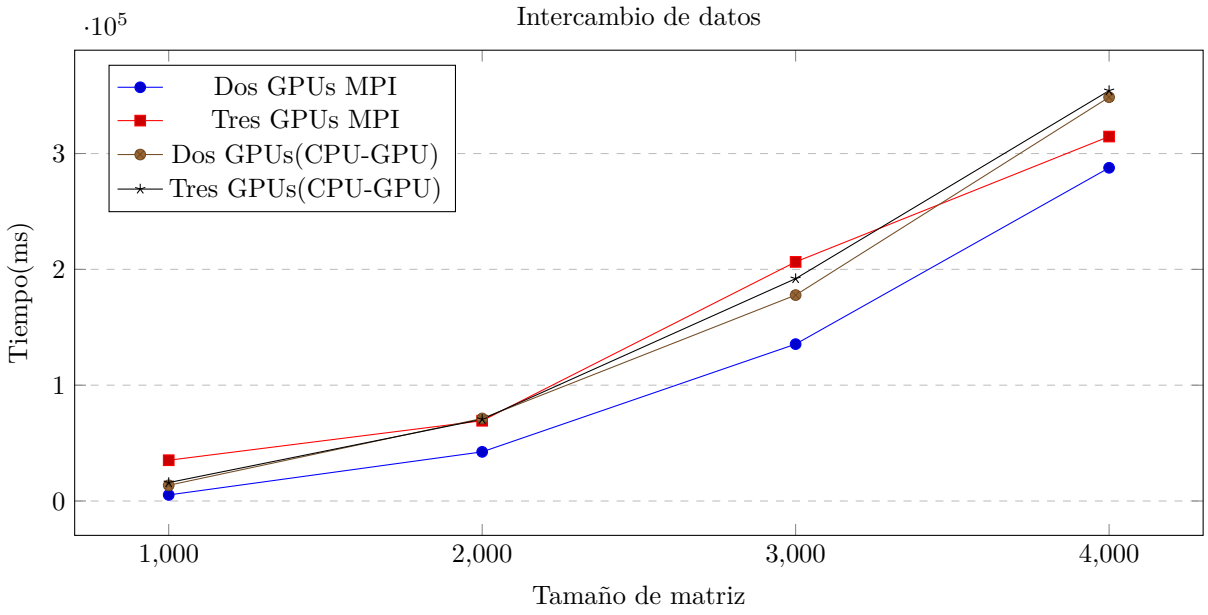
Con los resultado obtenidos de las pruebas se puede observar como el realizar un implementación híbrida-heterogénea de la transformación de Householder trae consigo una disminución en los tiempos de ejecución de la bifactorización y bidiagonalización.

En los resultados obtenidos para el proceso de la bifactorización podemos observar que la aceleración obtenida al incrementar el número de GPUs no se da en la misma proporción que se da en la implementación con OpenMP, esto se debe a que las tarjetas con las que estamos trabajando no tienen las mismas características y afecta el desempeño de la implementación, dicho efecto lo podemos observar en la tabla de la sección 5.2.1.2 en la tabla se aprecia como la misma implementación tiene mejores tiempos de ejecución en la tarjeta Tesla que en la tarjeta GTX. En la siguiente gráfica se hace una comparación entre las aceleraciones obtenidas entre las diferentes implementaciones.

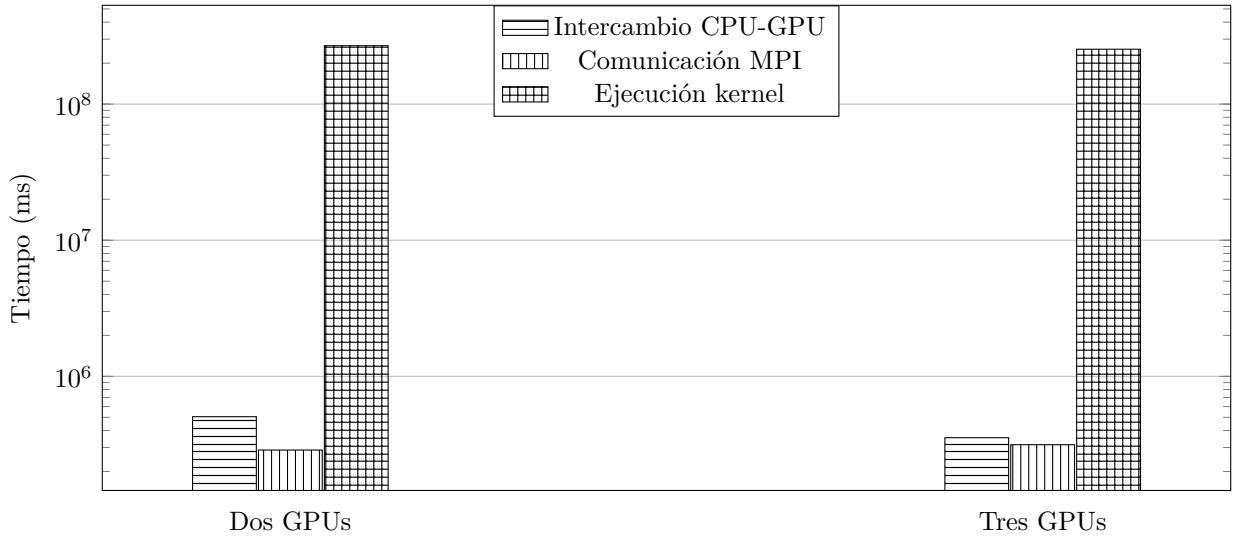


En la gráfica observamos claramente como disminuye notoriamente el tiempo de ejecución conforme se utilizan más GPUs, a pesar de que cuando el programa es ejecutado en una sola tarjeta la implementación en CPU es mas rápida. Es importante señalar que la implementación con OpenMP también fue ejecutada para tamaños de matrices mayores a 1000×1000 pero estas pruebas no terminaron, esto puede deberse a la continua creación y destrucción de hilos de ejecución que deben hacerse, como el método para bifactorizar matrices se ejecuta en $n - 2$ iteraciones, es posible que esto afecte el rendimiento de la implementación con OpenMP.

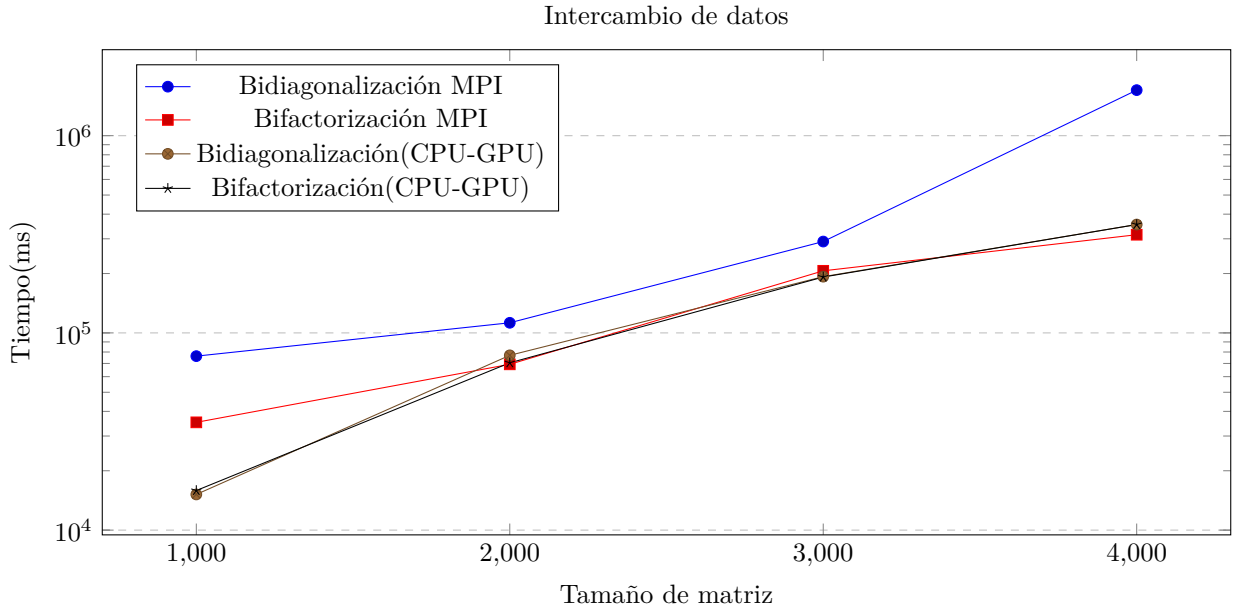
De las tablas de resultados se observa como también al incrementar el número de tarjetas se incrementa el tiempo de intercambio de datos entre los procesos MPI, pero disminuye el tiempo de comunicación entre CPU y GPU.



En la siguiente gráfica se muestran los tiempos de intercambio de datos entre CPU y GPU, los tiempos de comunicación entre procesos y el tiempo de ejecución para la matriz de tamaño 4000×4000, observamos que se incrementan los tiempos de comunicación entre procesos y una disminución en el tiempo de intercambio entre CPU y GPU; y existe una disminución en el tiempo de ejecución del kernel.

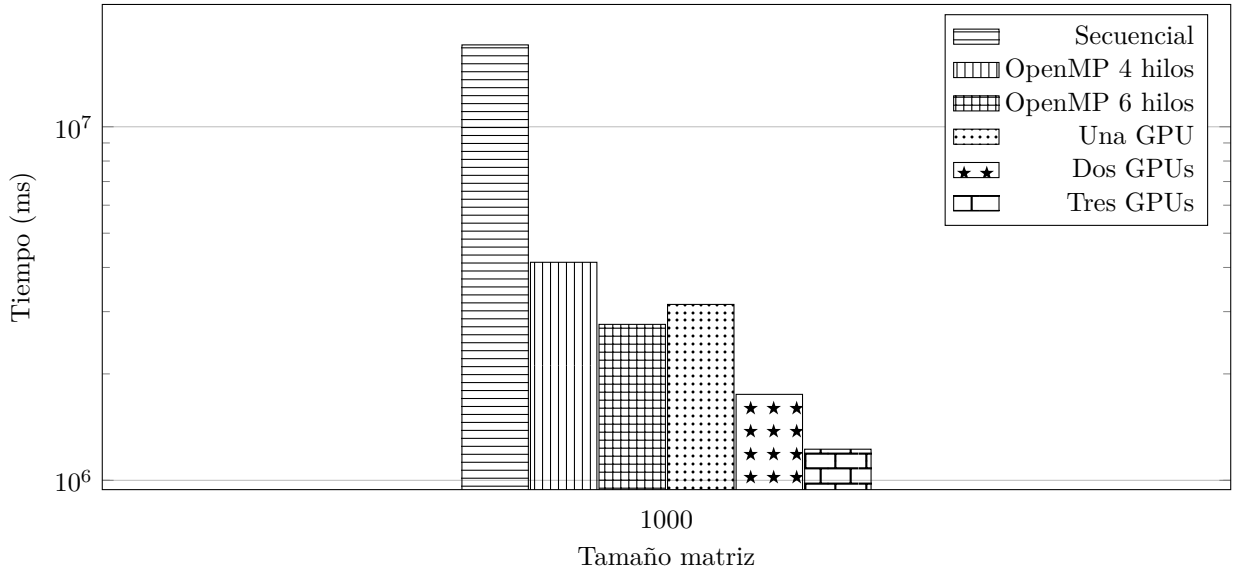


La parte de descomposición en valores singulares la ejecución de las pruebas se dividió en dos etapas, la primer etapa es la fase de bidiagonalización de la matriz, en esta etapa de nueva cuenta se utiliza un método iterativo que se ejecuta en $n-1$ iteraciones, pero a diferencia del método de bifactorización este método calcula dos matrices de Householder diferentes, por esta razón tienden a incrementarse los tiempos de comunicación entre procesos en la implementación híbrida-heterogénea.

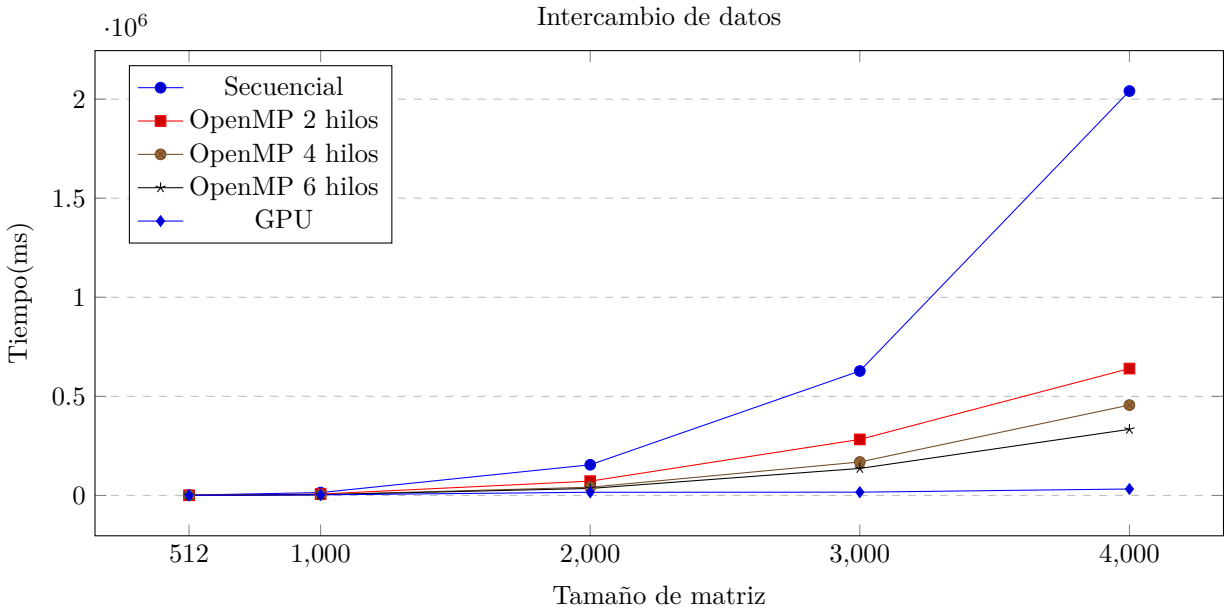


Los tiempos de ejecución de la implementación híbrida-heterogénea fueron mejores que los tiempos de las otras implementaciones, sin embargo, la aceleración no se incremento al mismo ritmo que en la implemen-

tación en OpenMP, esto puede deberse a que las tarjetas presentan características diferentes y la implementación desarrollada supone que todas las tarjetas en dónde se va a ejecutar el algoritmo tienen las mismas características. En la siguiente gráfica podemos ver como la implementación en una GPU resulta un poco mas lenta que la versión con OpenMP y seis hilos, pero es importante señalar que para instancias mayores del problema la implementación en OpenMP no termino de ejecutarse, esto puede deberse a la constante creación y destrucción de hilos ya que en OpenMP los hilos son gestionados por el sistema operativo a diferencia de los hilos en GPU los cuales son gestionados a través de hardware.



Finalmente la parte de la diagonalización fue realizada a partir del trabajo de (Estrella-Cruz, 2014), en su trabajo desarrollo un método para la diagonalización de matrices tridiagonales basándose en el método de (Cuppen, 1980), para el presente trabajo se utilizó el método propuesto por (Gu and Eisenstat, 1995) el cual es una modificación del método de Cuppen. Al trabajo realizado por (Estrella-Cruz, 2014) se le agregó la funcionalidad de escalar a n número de nodos y el cálculo de los eigenvectores derechos. Esta última parte es de suma importancia debido a que en esta se calculan los valores singulares y los vectores singulares izquierdos y derechos de la descomposición SVD. Los valores singulares a diferencia de los valores propios existen para todas las matrices y se utilizan junto con los vectores singulares para hacer aproximaciones a las matrices originales utilizando matrices de menor rango a las originales (Blum et al., 2016). En la siguiente gráfica observamos la aceleración obtenida de la implementación en GPU-CPU con respecto a la implementación en OpenMP.


Intercambio de datos
$\cdot 10^6$
Tiempo(ms)
2
1.5
1
0.5
0
512
1,000
2,000
3,000
4,000
Tamaño de matriz
Secuencial
OpenMP 2 hilos
OpenMP 4 hilos
OpenMP 6 hilos
GPU

Capítulo 6

Conclusiones y Trabajo a Futuro

En este trabajo de tesis se implemento la transformación de Householder en un arquitectura híbrida-heterogénea, como casos de estudio se utilizaron dos aplicaciones de la transformación: la bifactorización y bidiagonalización de matrices. La implementación lograda en este trabajo de tesis es capaz de escalar a múltiples nodos de manera que se pueda aplicar a matrices de mayor tamaño.

Las pruebas realizadas a la implementación mostraron que existe una disminución en el tiempo de ejecución cuando se incrementa el número de tarjetas utilizadas, ademas los tiempos de comunicación entre procesos y los tiempos de intercambio de datos entre CPU y GPU comparados con el tiempo total de ejecución son muy bajos, esto indica que se tiene una buena estrategia de particionamiento de los datos.

Las principales contribuciones de esta tesis son:

- Implementación de la bifactorización de matrices simétricas en una GPU y en múltiples GPUs.
- Implementación de la bidiagonalización de matrices en una GPU y en múltiples GPUs.
- Implementación de la descomposición en valores singulares en una GPU utilizando el método divide y vencerás, propuesto en (Estrella-Cruz, 2014) (Gu and Eisenstat, 1995).
- Se cuenta con implementaciones que no utilizan ninguna biblioteca de terceros para realizar los cálculos numéricos.
- Se muestra la importancia de contar con implementaciones de este tipo para operaciones matriciales.
- Aplicación exitosa en arquitectura híbrida-heterogénea de la transformación de Householder.

La bidiagonalización y bifactorización de matrices son pasos esenciales en problemas que involucran el cálculo de valores propios y valores singulares, de ahí la importancia de haber logrado en esta tesis una implementación que haga uso de las arquitecturas de cómputo modernas, ya que la la tendencia de este tipo de problemas es que cada vez son mas grandes por lo tanto es necesario tener implementaciones paralelas y escalables como la que se logró en este trabajo de tesis.

6.1. Trabajo a Futuro

De los resultados obtenidos en este trabajo queda claro que aún quedaron aspectos que son necesarios estudiar posteriormente, uno de ellos es probar la implementación de este trabajo con múltiples GPUs que tengan las mismas características. También se podría agregar un mecanismo que permita identificar las diferentes capacidades de las distintas tarjetas que se tengan en los diferentes nodos para poder repartir de manera mas balanceada la carga de trabajo.

Se pueden hacer mejoras a la implementación haciendo uso de nuevas tecnologías como CUDA-Aware la cual permite el intercambio de datos entre GPUs de manera mas fácil para el programador y además realiza la comunicación de manera más rápida. Otra tecnología que resulta de bastante interés es el uso de las nuevas arquitecturas de GPUs que cuentan con procesadores tensoriales los cuales de acuerdo a (NVIDIA, 2017) ofrecen mayores aceleraciones en operaciones como la multiplicación matricial, esto podría ser de gran utilidad en la implementación desarrollada ya que esta implementación realiza $4n$ multiplicaciones matriciales para bidiagonalizar una matriz de tamaño $n \times n$.

Un opción a considerar es agregar el trabajo realizado a la biblioteca de NVIDIA cuSOLVER, para esto habría que hacer las modificaciones pertinentes al código para cumplir con los estándares establecidos por NVIDIA para agregar nuevas funcionalidades a sus bibliotecas. También se podría comparar la implementación de este trabajo con una implementación que utilice OpenACC, ya que actualmente es un API que está tomando más fuerza en el desarrollo de aplicaciones para GPUs.

También como trabajo a futuro se pueden realizar implementaciones de aplicaciones donde se puede utilizar la transformación de Householder algunos ejemplos son: resolver sistemas de ecuaciones, el problema de n-cuerpos y en general cualquier problema que involucre el calculo de valores propios.

Bibliografía

Acton, F. S. (1990). *Numerical methods that work*. Mathematical Association of America.

Axler, S. J. (1997). *Linear algebra done right*, volume 2. Springer.

Blaise, B. (2017). Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel_comp/. [Online; último acceso:2017/11/22].

Blum, A., Hopcroft, J., and Kannan, R. (2016). Foundations of data science. *Vorabversion eines Lehrbuchs*.

Chapman, B., Jost, G., and Van Der Pas, R. (2008). *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press.

Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

Cosnuau, A. (2014). Computation on GPU of eigenvalues and eigenvectors of a large number of small hermitian matrices. *Procedia Computer Science*, 29:800–810.

Coulouris, G., Dollimore, J., and Kindberg, T. (2011). *Distributed Systems, Concepts and Design*. Addison Wesley, 5a edition.

Cuppen, J. (1980). A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36(2):177–195.

Demmel, J. and Kahan, W. (1990). Accurate singular values of bidiagonal matrices. *SIAM Journal on Scientific and Statistical Computing*, 11(5):873–912.

Dongarra, J., Moler, C., Bunch, J., and Stewart, G. (1979). *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics.

Duncan, R. (1990). A survey of parallel computer architectures. *Computer*, 23(2).

Estrella-Cruz, A. (2014). Paralelización heterogénea del algoritmo de Cuppen, caso de estudio: problema regular de Sturm-Liouville. Tesis de maestría, CINVESTAV-IPN, Departamento de Computación.

Evans, D. J. and Gusev, M. (2002). Systolic svd and qr decomposition by householder reflections. *International Journal of Computer Mathematics*, 79(4):417–439.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 100(9).

Foster, I. (1995). *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Boston.

Fuller, S. H. and Millett, L. I. (2011). Computing performance: Game over or next level? *Computer*, 44(1):31–38.

Givens, W. (1958). Computation of plain unitary rotations transforming a general matrix to triangular form. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):26–50.

Golub, G. and Kahan, W. (1965). Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224.

Golub, G. H. and Loan, C. F. V. (2012). *Matrix Computations*. Johns Hopkins University Press, 4th edition.

Golub, G. H. and Reinsch, C. (1970). Singular value decomposition and least squares solutions. *Numerische mathematik*, 14(5):403–420.

Gropp, W., Lusk, E., and Skjellum, A. (1999). *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press.

Gu, M. and Eisenstat, S. C. (1995). A divide-and-conquer algorithm for the bidiagonal svd. *SIAM Journal on Matrix Analysis and Applications*, 16(1):79–92.

Highman, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2nd edition.

Hogben, L. (2006). *Handbook of linear algebra*. CRC Press.

Householder, A. S. (1958). Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, 5(4):339–342.

Kirk, D. B. and Wen-Mei, W. H. (2016). *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

Konda, T. and Nakamura, Y. (2009). A new algorithm for singular value decomposition and its parallelization. *Parallel Computing*, 35(6):331–344.

Kotas, C. and Barhen, J. (2011). Singular value decomposition utilizing parallel algorithms on graphical processors. In *OCEANS 2011*, pages 1–7. IEEE.

Kumar, V., Grama, A., Gupta, A., and Karypis, G. (2003). *Introduction to parallel computing: design and analysis of algorithms*. Addison Wesley, 2a edition.

Lahabar, S. and Narayanan, P. (2009). Singular value decomposition on GPU using CUDA. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE.

Li, R.-C. (1993). Solving secular equations stably and efficiently. Technical report, University of California, Berkeley. Department of Electrical Engineering and Computer Sciences.

Liu, F. and Seinstra, F. J. (2010). GPU-based parallel Householder bidiagonalization. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 288–291. ACM.

NVIDIA (2017). Nvidia tesla v100 gpu architecture. Technical report, NVIDIA.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition.

Sanders, J. and Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional.

Tapia-Romero, M. (2013). Diagonalización de matrices a través del método de Givens con múltiples tarjetas de gpus. Tesis de maestría, CINVESTAV-IPN, Departamento de Computación.

Wilt, N. (2013). *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education.