Universidad de la República Oriental del Uruguay
Facultad de Ingeniería

Transformaciones ortogonales de matrices utilizando GPUs

Montevideo, Noviembre de 2015

Tesis de grado presentada por: Agustín Young
Tutores: Pablo Ezzatti, Ernesto Dufrechou

Transformaciones ortogonales de matrices utilizando GPUs

RESUMEN

Las transformaciones ortogonales de matrices son una herramienta ampliamente utilizada en el campo del álgebra lineal numérica. Una de sus principales aplicaciones es el cálculo o la estimación de los valores y vectores propios de una matriz, en donde estas transformaciones representan una parte importante de los requerimientos computacionales de los métodos utilizados. Esta característica de las transformaciones motiva el uso de estrategias de computación de alto desempeño (HPC) para abordarlas.

Por otra parte, en los últimos años las tarjetas gráficas (GPUs) han experimentado una evolución vertiginosa, pasando de ser procesadores dedicados completamente a representar gráficos en pantalla, a transformarse en verdaderos multiprocesadores dotados de gran flexibilidad y capacidad de programación con técnicas de computación paralela. Este hecho ha motivado que muchos científicos busquen la utilización de las GPUs para atacar problemas de propósito general (área conocida como GPGPU). Para tal motivo, NVIDIA (uno de los principales fabricantes de aceleradores gráficos) ofrece el lenguaje de programación CUDA.

El objetivo del proyecto es estudiar y avanzar en el uso de arquitecturas de hardware híbridas (compuestas por CPUs multi-core y GPUs) para acelerar el cálculo de transformaciones ortogonales de matrices en general y, en particular, realizar diseños e implementaciones utilizando CUDA para las técnicas de Householder y de Givens pretendiendo lograr desempeños comparables a los de las bibliotecas de álgebra que representan el estado del arte en la materia.

Palabras claves: Álgebra lineal numérica, Transformaciones ortogonales, Givens, Householder, HPC, Computación paralela, GPU, GPGPU, CUDA

Agradecimientos

Deseo expresar mi más sincero agradecimiento:

A mis tutores Pablo y Ernesto por su confianza y supervisión durante el proyecto.

A mi novia Silvia, familia y amigos que me apoyaron durante todos estos años de estudio.

Índice general

1. Introducción **1**
- 1.1. Objetivos . . . 2
- 1.2. Estructura . . . 3
- 1.3. Plataforma de experimentación . . . 3

2. Conceptos básicos **5**
- 2.1. Introducción . . . 5
- 2.2. Conceptos preliminares . . . 6
  - 2.2.1. Valores y vectores propios . . . 6
  - 2.2.2. Propagación de errores . . . 8
  - 2.2.3. HPC . . . 9
  - 2.2.4. Bibliotecas . . . 14

3. Transformaciones matriciales **17**
- 3.1. Introducción . . . 17
- 3.2. Método de Givens . . . 17
- 3.3. Método de Householder . . . 19
- 3.4. Trabajo relacionado . . . 20
  - 3.4.1. Algoritmos de actualización de la factorización QR sobre GPU . . . 20
  - 3.4.2. Descomposición QR en GPUs . . . 21
  - 3.4.3. Computación en GPU de los valores y vectores propios sobre matrices hermíticas pequeñas con números grandes . . . 22
  - 3.4.4. Implementación de la factorización QR por bloques para GPU y múltiples CPUs . . . 22
  - 3.4.5. Computación de los valores propios con CUDA . . . 23
  - 3.4.6. Una aproximación de alto nivel a la resolución de problemas matriciales con almacenamiento en disco . . . 24

4. Propuestas **25**
- 4.1. Método de Givens . . . 25
  - 4.1.1. Givens en CPU (GC) . . . 25
  - 4.1.2. Givens en GPU con planificación por columnas (GG1) . . . 25
  - 4.1.3. Givens en GPU con planificación máxima (GG2) . . . 27
  - 4.1.4. Givens híbrido a bloques (GH) . . . 28
  - 4.1.5. Evaluación Experimental . . . 29
- 4.2. Método de Householder . . . 32
  - 4.2.1. Householder en CPU (HC) . . . 33

4.2.2. Householder en GPU (HG) . . . 33
4.2.3. Householder híbrido a bloques (HH) . . . 33
4.2.4. Evaluación Experimental . . . 35
4.3. Evaluación comparativa . . . 38
4.3.1. Algoritmos de referencia . . . 38
4.3.2. Casos de estudio . . . 38

5. Conclusiones y trabajo futuro **41**
5.1. Conclusiones . . . 41
5.2. Posibles mejoras y trabajo futuro . . . 42

A. Herramienta desarrollada **45**
A.1. Ejecución en modo debug . . . 47
A.2. Salida del programa . . . 49

Índice de figuras

2.1. Requerimientos computacionales de problemas complejos. Extraído de [3]. 9
2.2. Esquema de arquitecturas de CPU y de GPU. Extraído de [11]. 11
2.3. Comparativa de rendimiento de CPUs vs GPUs en cantidad de operaciones en punto flotante simple/doble precisión. Extraído de [11]. 11
2.4. Comparativa de rendimiento de CPUs vs GPUs en accesos de datos. Extraído de [11]. 12

3.1. Esquema intuitivo de una rotación de Givens. 18
3.2. Factorización QR a bloques. Extraído de [15]. 23

4.1. Esquema propuesto sobre una posible planificación en una iteración del método de Givens (con planificación por columnas). 26
4.2. Esquema propuesto sobre una posible planificación en una iteración del método de Givens (con planificación máxima). 27
4.3. Esquema propuesto para el algoritmo híbrido de Givens a bloques. 29
4.4. Test de hilos para la variante GG2. 30
4.5. Test de bloques para la variante GH. 30
4.6. Test de matrices para Givens. 31
4.7. Test de rendimiento para Givens. 32
4.8. Algoritmo por bloques para calcular la factorización QR mediante transformaciones de Householder. Extraído de [30]. 34
4.9. Algoritmo escalar para calcular la factorización QR mediante transformaciones de Householder. Extraído de [30]. 34
4.10. Test de hilos para la variante HG. 35
4.11. Test de bloques para la variante HH. 36
4.12. Test de matrices para Householder en las diferentes variantes implementadas. 36
4.13. Test de rendimiento para Householder. 37
4.14. Rendimiento de los algoritmos implementados. 40

Índice de tablas

3.1. Trabajos relacionados sobre el uso de GPUs para acelerar trasformaciones matriciales. 20

4.1. Tests desarrollados. 38

A.1. Estructura de archivos del código construido. 45
A.2. Algoritmos desarrollados. 47
A.4. Parámetros de salida del programa. 49

Capítulo 1

Introducción

Las transformaciones ortogonales de matrices son una herramienta ampliamente utilizada en el campo del álgebra lineal numérica. En particular, diversos métodos que implican el cálculo (o estimación) de los valores y vectores propios o el cálculo de la descomposición en valores singulares (SVD, del inglés Singular Value Decomposition) tienen como etapa fundamental, en cuanto a costo computacional, el cálculo de una o varias transformaciones ortogonales (para aritmética real, o transformaciones unitarias en el caso de emplear aritmética con números complejos). En este sentido, dos de las estrategias más difundidas son las transformaciones de Householder y de Givens [25] [41].

Las transformaciones de Householder (o reflexiones de Householder) se basan en reflejar el espacio respecto a un hiperplano determinado. Esta propiedad se puede utilizar para diseñar la matriz (de Householder) de manera que un vector elegido quede con una única componente no nula tras ser transformado (es decir, pre-multiplicando por la matriz de Householder). La otra estrategia, las transformaciones de Givens (o rotaciones de Givens), se basa en aplicar diversas rotaciones, donde cada rotación anula (hace cero) un elemento en la subdiagonal de la matriz a transformar.

Existen diversas implementaciones de estos métodos que permiten aprovechar características de las plataformas de hardware a utilizar o de los patrones de datos de las matrices implicadas. En especial, se distinguen las estrategias dependiendo de si las matrices son densas o dispersas.

Una limitante importante para el uso de este tipo de transformaciones es el elevado costo computacional en el que incurren, ya que en el caso denso estas transformaciones implican un orden cúbico (en la dimensión de la matriz) de operaciones. Esta característica de las transformaciones motiva el uso de estrategias de computación de alto desempeño (HPC, del inglés High Performance Computing) para abordarlas.

Si bien la aplicación de estrategias de HPC tradicionales es una buena opción para abatir las restricciones antes mencionadas, estas técnicas poseen como principal limitante los grandes montos económicos que por lo general implica su utilización. En la última década, se han buscado diversas opciones para alcanzar contextos de HPC de bajo costo económico, siendo uno de los más pujantes el uso de procesadores secundarios y en particular el de los procesadores gráficos (GPU, del inglés Graphics Processing Unit).

En los últimos años las tarjetas gráficas han experimentado una evolución vertiginosa, pasando de ser procesadores únicamente útiles a la hora de representar gráficos en pantalla, a transformarse

en verdaderos multiprocesadores programables mediante lenguajes de alto nivel. Este hecho ha motivado que muchos científicos busquen la utilización de las GPUs para atacar problemas de propósito general (GPGPU del inglés General Purpose Computing on Graphics Processing Units) y no solamente para tareas vinculadas a los gráficos. En gran medida el aumento en la capacidad de cómputo de las GPU se sustenta en su arquitectura. Una gran diferencia entre las GPU y CPU es que las primeras utilizan estrategias de cálculo paralelo al trabajar. En particular, las GPU utilizan una estrategia de paralelismo similar a SPMD (del inglés Single Program Multiple Data), donde las tareas son separadas y ejecutadas simultáneamente en múltiples procesadores con diferentes entradas para obtener los resultados con mayor rapidez.

El desarrollo de GPGPU ha tenido un importante impulso en los últimos años, en especial desde el 2007 cuando la empresa NVIDIA [9] que es uno de los principales fabricantes de GPUs creó CUDA [11] (del inglés Compute Unified Device Architecture). Este es un lenguaje de programación ofrecido por dicha empresa para programar sus GPUs, el cual se encuentra en constante desarrollo y que en la actualidad requiere que el programador entienda, al menos someramente, la arquitectura de las mismas.

1.1. Objetivos

En el contexto antes explicado, y a partir de las posibilidades crecientes que brindan las nuevas arquitecturas de hardware, el área de GPGPU se ha convertido en una alternativa natural y económica para abordar problemas de ALN. Siendo el mencionado campo el foco de estudio del presente proyecto. Por este motivo, y considerando la importancia de las transformaciones ortogonales en la computación científica, el objetivo del proyecto es avanzar en el uso de arquitecturas de hardware híbridas (compuestas por procesadores de propósito general multi-core y GPUs) para acelerar el cálculo de transformaciones matriciales haciendo énfasis en las técnicas de Householder y de Givens.

Para alcanzar el objetivo general del proyecto es necesario el tratamiento de diversos objetivos específicos, entre los más importantes:

- Estudiar y manejar los conceptos matemáticos básicos relacionados con las transformaciones ortogonales matriciales.
- Relevar los avances en el uso de GPUs y en particular del entorno CUDA, contextualizado en las tendencias de HPC para computación científica.
- Estudiar los últimos esfuerzos guiados por objetivos similares a los de este proyecto, qué resultados se han obtenido, cuáles son los aspectos más relevantes, las recomendaciones y conclusiones que aportan.
- Diseñar y desarrollar diferentes variantes de los métodos numéricos implicados.
- Evaluar las diferentes variantes de los métodos desarrollados.
- Identificar líneas de trabajo futuro.

En resumen, el foco del proyecto es estudiar y avanzar en el uso de arquitecturas de hardware híbridas para acelerar el cálculo de transformaciones ortogonales de matrices en general y, en particular, realizar diseños e implementaciones utilizando CUDA para las técnicas de Householder

y de Givens pretendiendo lograr desempeños comparables a los de las bibliotecas de álgebra que representan el estado del arte en la materia.

1.2. Estructura

Lo que resta de la memoria se organiza de la forma que se describe a continuación.

El Capítulo 2 trata sobre los conceptos básicos para el desarrollo del proyecto, repasa temas tales como: valores y vectores propios, propagación de errores y computación paralela.

Luego, se dedica el Capítulo 3 a revisar las técnicas de transformaciones matriciales, profundizando en los métodos de Givens y Householder tanto en su formulación intuitiva como en la formalidad matemática. Se relevan también los trabajos relacionados a las líneas del proyecto.

Una vez presentado el marco teórico y las herramientas mencionadas, en el Capítulo 4 se describen los diseños propuestos y se especifican los algoritmos implementados. También se presenta la evaluación experimental realizada para validar las propuestas.

Finalmente, el Capítulo 5 resume las principales conclusiones alcanzadas durante el trabajo e identifica líneas de trabajo futuro.

1.3. Plataforma de experimentación

Todas las pruebas que se describen en esta memoria fueron realizadas en la misma plataforma de hardware que incluye las siguientes características:

- CPU Intel(R) Core(TM) i3-2100 3.10GHz con 8GB de RAM.
- Tarjeta de video GeForce GTX 480.

En cuanto al software disponible en la plataforma, las principales herramientas son:

- Compilador gcc 64 bits versión 4.8.2.
- BLAS versión 3.5.0 [46].
- Lapack 3.4.2 [37].
- Compilador nvcc con Compute Capability 2.0.
- CUDA 6.0 [11].
- NVIDIA Visual Profiler Toolkit 6.0 [5].

Capítulo 2

Conceptos básicos

2.1. Introducción

En este capítulo se resumen varios temas y conceptos importantes que servirán como base para la comprensión de los posteriores capítulos.

Este trabajo se enmarca en el campo del Álgebra Lineal Numérica (ALN), específicamente se abordan las transformaciones ortogonales de matrices, y con foco en el elevado costo computacional que conlleva su resolución. Directamente relacionado con el objetivo del proyecto se identifica el cálculo de los valores y vectores propios como etapa principal en cuanto a costo computacional en la resolución de una gran clase de problemas. Se detallarán los diversos métodos y estrategias que implican, poniendo especial atención en la propagación de errores en los métodos numéricos.

En el capítulo también se abordará el uso de las técnicas de HPC (computación de alto desempeño), y en este contexto los beneficios y restricciones que requiere su utilización. Además se profundizará en varias sub-áreas de investigación como computación paralela en general y GPGPU (computación de propósito general en unidades de procesamiento gráfico) en particular.

El lenguaje de programación a utilizar en la tarjeta gráfica será CUDA, y el mismo se describe al final del capítulo. Éste fue creado por NVIDIA y está diseñado para el desarrollo de aplicaciones sobre las tarjetas producidas por dicha empresa, por lo cual brinda soporte para manejar la concurrencia y el paralelismo que ofrece este tipo de hardware, complementándose con varias bibliotecas adecuadas para este propósito.

Es importante tener presente para la comprensión de este trabajo los dos objetivos fundamentales en la computación de los métodos numéricos, esto es que tengan bajos tiempos de ejecución y alta precisión en los resultados. Muchas veces en los métodos numéricos estas dos necesidades se presentan como contrapuestas, debido a que para lograr una mayor precisión es necesario aumentar la carga computacional y por consiguiente los tiempos de ejecución. Es por esto que resulta necesario diseñar estrategias que alcancen niveles de compromiso entre ambos objetivos. Generalmente, se deben plantear estas condiciones cuando no existe una solución óptima bajo las restricciones de contexto, tiempo y capacidad de cómputo.

2.2. Conceptos preliminares

2.2.1. Valores y vectores propios

El problema de encontrar los valores y/o vectores propios surge en diversas áreas de la ciencia, entre otras, física, estadística, astronomía, electrónica, etc.

Se definen los valores propios de la matriz cuadrada A, como los escalares λ tales que $det(A - \lambda I) = 0$.

Una vez calculados los valores propios, los vectores propios serán los vectores v no nulos que verifiquen $Av = \lambda v$. Geométricamente, esto se puede interpretar como los vectores que al aplicarles la trasformación lineal A, el resultado es un vector con la misma dirección y multiplicado por un escalar (que es el valor propio asociado).

Para calcularlos, los métodos analíticos no suelen ser útiles en la práctica, por lo que es imprescindible la utilización de métodos numéricos. Sobre el tópico se puede profundizar en [43]. Específicamente, las técnicas numéricas de búsqueda de raíces no resultan convenientes en el cálculo de valores y vectores propios debido a que la localización de las n raíces de un polinomio es muy sensible a las variaciones de los coeficientes del polinomio, sobre todo en los casos de raíces múltiples, por lo tanto los errores de redondeo afectan la estimación de los valores propios. Un ejemplo de estos algoritmos es el de LeVerrier-Faddeev [13]. En caso de solo querer conocer algunas características de los valores propios, por ejemplo conocer una cota superior del máximo valor, o saber si son todos reales, se pueden usar algunas técnicas o propiedades como el esquema de Gerschgorin [2].

Una herramienta habitual para el abordaje de este tipo de problemas en el álgebra lineal, es generar una factorización de la matriz original, esto es descomponer la matriz como el producto de dos o más matrices según una forma canónica. Típicamente para el estudio de sistemas lineales se utilizan las factorizaciones LU, QR, Cholesky (en los casos simétricos), SVD, Jordan, diagonalización, factorización de rango bajo (en el caso de saber que las matrices lo son), Schur, tridiagonalización, entre otras. La elección de la factorización dependerá de las características de la matriz original, de la computadora donde se va a realizar el cálculo y del problema que se desea resolver. Varias de estas factorizaciones, en particular todas las que se revisan en este trabajo se utilizan para calcular los valores y vectores propios.

Una forma canónica que se aplica en este tipo de problemas muy frecuentemente es la matriz de Hessenberg. Esta matriz es casi triangular y tiene todos los elementos por debajo de la primer subdiagonal en cero. Muchos algoritmos del ALN requieren significativamente menos esfuerzo computacional si la matriz es triangular o de Hessenberg. Observar que si la matriz original es simétrica, entonces una semejante de Hessenberg será tridiagonal.

Uno de los primeros algoritmos iterativos para calcular valores y vectores propios fue el método de las potencias, un método básico que luego dio fundamento a varios algoritmos más complejos. El método se puede aplicar bajo ciertas condiciones: la matriz debe ser cuadrada de dimensión $n \times n$, tiene que tener n valores propios distintos asociados a n vectores propios linealmente independientes, y además, un valor propio λ_1 debe ser mayor que todos los demás. El principal objetivo del método es encontrar el valor propio de mayor magnitud (valor propio dominante) y el vector propio asociado. Se puede extender su utilización si se desea hallar el más cercano a un valor determinado, o el valor

propio de menor magnitud, en ese caso se aplica el método a la matriz inversa a la original (los valores propios de la inversa de una matriz son los inversos de los valores propios de la matriz original).

Por otra parte, una vez encontrado el valor propio más grande, para encontrar los demás se pueden usar técnicas de deflación [25]. Este método tiene como desventajas que es efectivo sólo si la matiz posee un valor propio dominante y se elige correctamente un vector inicial (no puede ser perpendicular al vector propio asociado al valor propio dominante), características que generalmente se desconocen antes de empezar. Además, si los dos valores propios principales son parecidos en valor absoluto, la convergencia será lenta. Este último punto se puede optimizar con técnicas de desplazamiento para "separar" los valores propios y ayudar a la convergencia del algoritmo. Específicamente, los cocientes de Rayleigh y Wilkinson suelen usarse como desplazadores para acelerar la convergencia de esta familia de métodos iterativos.

El método de deflación, utiliza un valor propio λ_i y un vector propio v_i de la matriz A ya conocidos, para transformarla en una nueva matriz B. De esta se podrá obtener otro de los valores y vectores propios de mayor módulo. Existen varias técnicas como por ejemplo, semejanza, Hotelling y Wielandt. Por más información acerca de estos métodos ver [43].

Otro método, conocido como la factorización QR, descompone la matriz de trasformación lineal como el producto de una matriz ortogonal por una matriz triangular superior. Es muy utilizada tanto para la solución de mínimos cuadrados como para encontrar los valores y vectores propios por medio del algoritmo QR. La factorización puede calcularse de varias formas como el proceso de ortogonalización de Gram-Schmidt, trasformaciones de Householder o con rotaciones de Givens.

El algoritmo QR es usado para calcular valores y vectores propios de una matriz. Se basa en la descomposición QR y se puede considerar como una versión más sofisticada del método de las potencias. La idea en ambos es multiplicar repetidamente un vector por la matriz de la que se quiere conocer los valores propios, normalizando luego de cada iteración para converger a los valores buscados. En el caso del algoritmo QR, la iteración propuesta es $A_{n+1} = R_n Q_n = Q_k^t A_n Q_n$ por lo tanto A_{n+1} y A_n son semejantes y tienen los mismos valores y vectores propios. La justificación de este algoritmo está relacionada con el teorema de Schur, el cual plantea la igualdad $U^t A U = T$ siendo U unitaria y T triangular con los valores propios de A en su diagonal. El costo general del método es n^3, si la matriz a descomponer ya es de la forma de Hessenberg el algoritmo tendrá costo $4n^2$ y si fuese simétrica tendrá costo $12n$, ver [12]. Es común transformar la matriz original a la forma de Hessenberg antes de aplicar el algoritmo QR.

Otro método muy utilizado para la obtención de valores y vectores propios es el método de Jacobi. Éste sólo se puede emplear para matrices hermíticas (simétricas en caso de trabajar con aritmética real) y tiene, generalmente, un mejor rendimiento que las técnicas de factorización QR (estas no exigen que la matriz sea simétrica). La idea es transformar la matriz en diagonal por semejanza mediante rotaciones, aprovechando que las matrices hermíticas son diagonalizables mediante la aplicación de transformaciones de semejanza ortogonal.

Por último, corresponde mencionar las técnicas de descomposición en valores singulares que son también muy utilizadas aunque no serán abordadas en este proyecto. Toda matriz $A \in M^{m \times n}$ admite esta descomposición, la cual está determinada de forma única. En lugar de trabajar con la matriz A, se trabaja con $A^t A$ la cual tiene mejores características, ya que es simétrica, semidefinida

positiva. Si λ_i es un valor propio de la matriz A^tA, se define el valor singular σ_i de A como $\sigma_i = \sqrt{\lambda_i}$. Los vectores singulares serán los v_i tales que $\|Av_i\| = \sigma_i$. Al igual que los vectores propios, describen direcciones y proporciones que se mantienen al aplicar una transformación lineal.

Los algoritmos de Arnoldi y Lanczos son dos métodos iterativos para obtener los valores singulares, típicamente usados sobre matrices dispersas de grandes dimensiones, ver [1]. Estos métodos se basan en el proceso de Gram-Schmidt o el algoritmo de Householder para reorganizar la matriz en el subespacio de Krylov (busca una base ortogonal en el caso de Arnoldi). Son métodos complementarios ya que Lanczos se aplica sobre matrices hermíticas (la matriz es igual a su transpuesta conjugada) y Arnoldi se aplica sobre matrices no hermíticas. Si bien existe una variante del método de Lanczos para el caso no hermítico no está muy difundido su uso.

2.2.2. Propagación de errores

La utilización de métodos numéricos implica un manejo adecuado de propagación de errores. Recordar que el sistema de punto flotante de simple precisión utiliza 32 bits y el de doble precisión utiliza 64 bits para representar un número real.

En el caso de simple precisión, que es el caso que se usará de ejemplo, se destina un bit para el signo, 23 para la mantisa, y 8 para el exponente. La fórmula para construir la representación en punto flotante normalizada de un número real es: $x = (-1)^S \times 1, M \times B^E$, siendo B una base (en caso de computadoras lo más común es que sea 2). S, M y E son los números enteros que se representan con los bits del signo, mantisa y exponente respectivamente.

Es importante tener en cuenta los cinco errores más comunes de la representación en punto flotante, estos son: *not a number*, *overflow*, *underflow*, *cancelación catastrófica* y *shift out*. Los tres primeros, son claramente identificables, el resultado es siempre incorrecto y la computadora los puede detectar ya que, al conocer los dos números con los que se va a hacer la operación (suma, resta, multiplicación o división) ya se puede definir si el error va a suceder. Los últimos dos errores son más difíciles de detectar, ya que dependen del conjunto global y orden de datos con los que se va a operar, para determinar si el error es significativo. Por ejemplo, *shift out* ocurre cuando a un número grande se le suma uno muy chico. El resultado es solo el número grande ya que el chico no se puede representar en la mantisa del grande. Si al completar las cuentas el número chico era de relevancia para el resultado final, entonces se estará ante un resultado incorrecto. *Cancelación catastrófica* tiene un efecto muy similar a lo explicado anteriormente, solo que la pérdida de precisión sucede al restar dos números grandes cercanos (el resultado tiene un error relativo muy alto), solo se sabe si el error genera un resultado incorrecto en la sucesión global de las cuentas. De hecho la gran mayoría de las ocasiones al operar con punto flotante se incurre en alguna medida en estos dos problemas. En definitiva, no siempre que sucede *shift out* o *cancelación catastrófica*, al final de las cuentas sucede un resultado incorrecto (con una diferencia mayor a una tolerancia), por eso es difícil delimitar el problema para la computadora.

Por último respecto al tema de errores, cabe mencionar la importancia del concepto de ϵ-maquina definido como el número más chico tal que $1 + \epsilon > 1$. Entre otras utilidades se define el número de condición del algoritmo, como la variación del resultado de un algoritmo frente a una pequeña variación en los parámetros de entrada normalizado según el ϵ-maquina.

2.2.3. HPC

La computación de alto rendimiento es el campo de investigación de soluciones a problemas de alto costo computacional como pueden ser simulaciones u otros problemas complejos. Intenta explotar los beneficios de las tecnologías computacionales como supercomputadoras y clusters para resolver dichos problemas mediante técnicas de computación paralela. Fomenta el desarrollo de bibliotecas e interfaces para la programación, facilitando la resolución de los principales problemas de sincronización y acceso a la información que típicamente acarrea la construcción de soluciones bajo técnicas de computación paralela. Por otra parte, esta área se ve fortalecida en los últimos años debido a que los cluster de computadoras de bajo costo son cada vez más comunes y que el avance en las interconexiones puede verse como una fuente potencial de recursos de computación (casi) ilimitados en la medida de que la cantidad de conexiones aumenta y mejora en velocidad.

Existe un ranking mundial de las 500 supercomputadoras más potentes del mundo [28], que se actualiza de forma semestral y es importante destacar que cada vez aumenta la presencia de supercomputadoras con GPUs entre sus componentes, ver por ejemplo [19].

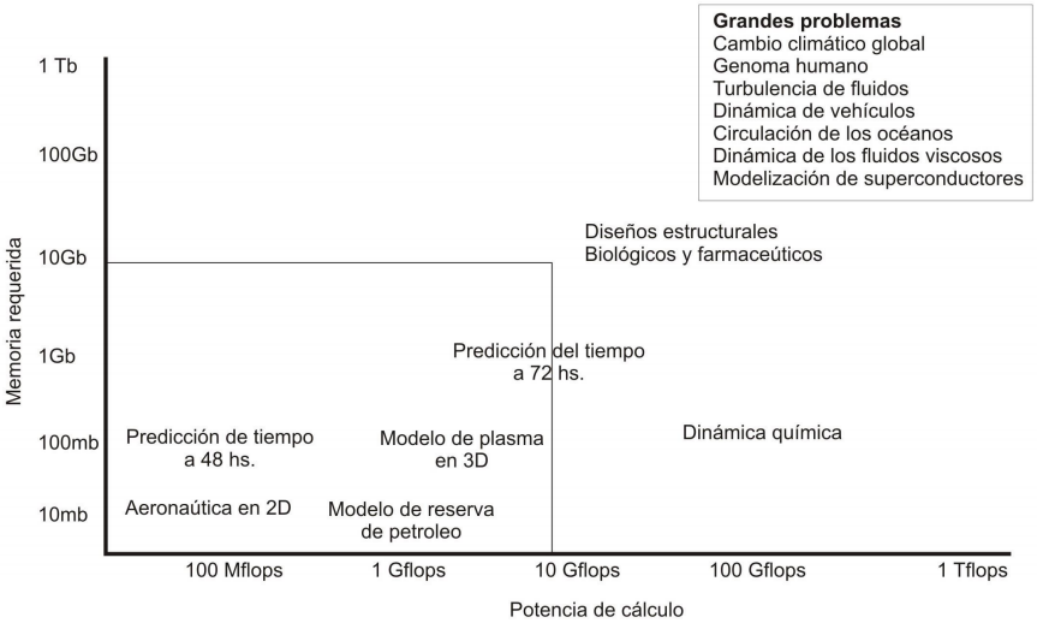

Figura 2.1: Requerimientos computacionales de problemas complejos. Extraído de [3].

Computación paralela

La computación paralela, como su nombre lo indica es una estrategia de cómputo en la que se ejecutan muchas instrucciones simultáneamente con el objetivo de mejorar la velocidad de resolución de problemas computacionalmente demandantes. La idea básica consiste en dividir el problema grande en muchos pequeños para resolverlos concurrentemente. Esto puede acarrear un incremento en las tareas de sincronización (problemas que deberá tener en cuenta el programador) debido a la cohesión de los datos (generalmente alta).

Dentro de la computación paralela se proponen dos grandes paradigmas, por un lado las arquitecturas con memoria compartida (alto acoplamiento) y por otro lado las arquitecturas con memoria distribuida (con bajo acoplamiento).

Cada uno tiene sus ventajas, dependerá de las características del problema y la infraestructura disponible para decidir cuál será más conveniente. Por ejemplo si el problema que requiere mucho cómputo, es muy paralelizable e implica bajo acoplamiento de datos, es decir que se puede partir en problemas pequeños que requieran operar con pocos datos disjuntos, convendrá una arquitectura distribuida, ya que esta ofrece gran escalabilidad. Por otro lado si hay un problema que se puede paralelizar y tiene un volumen de cómputo considerable, pero para computar se requiere acceder a diversos datos entonces convendrá (siempre y cuando la aplicación quepa en memoria) una arquitectura con memoria compartida. También son extendidas las propuestas híbridas que sacan partido de ambos paradigmas en forma conjunta.

El trabajo de este proyecto se enmarca principalmente en el paradigma de memoria compartida trabajando con una CPU multi-core y una GPU. Los fraccionamientos del problema en subproblemas más pequeños que se identificarán son adecuados para construir soluciones con memoria compartida. Particularmente, el problema de ortogonalizar matrices tiene muy alto acoplamiento de datos y esto hace más razonable pensar en soluciones con una jerarquía de memoria rápida y centralizada.

En otro sentido, es importante mencionar que el interés en esta área viene creciendo, como alternativa al problema de las limitaciones físicas que implica el aumento de frecuencia del reloj de los procesadores, en especial las problemáticas referidas al aumento de la temperatura y consumo de energía eléctrica. Esto, cada vez más, está dando lugar al desarrollo de CPUs multi-core y la utilización de aceleradores de hardware (GPUs, Xeon Phi [20], etc.) para la resolución de problemas de propósito general. Un ejemplo de esta tendencia es el caso de las tarjetas NVIDIA, que en el pasaje de la generación FERMI a KEPLER, bajaron la frecuencia del reloj para lograr una disminución de consumo de 244W/h (Fermi GF110) a 195W/h (Kepler GK104) para adaptarse a las nuevas reglamentaciones europeas, más detalles se pueden consultar en [32]. En contrapartida, la cantidad de núcleos se triplicó, entre otros avances tecnológicos, logrando duplicar aproximadamente la capacidad de cómputo (de 1581 a 3090 GFLOPs para las tarjetas mencionadas). La tendencia histórica marca que en general se busca aumentar la cantidad de procesadores y ampliar la capacidad en la jerarquía de memoria, ver [34].

GPU

GPU (del inglés Graphics Processing Unit), es un coprocesador dedicado originalmente al procesamiento de gráficos para aligerar la carga de trabajo al procesador principal en aplicaciones demandantes en cuanto a este tipo de procesamiento, por ejemplo videojuegos o aplicaciones 3D.

La GPU se encarga de lo que se llaman operaciones primitivas para gráficos, como por ejemplo representar figuras geométricas, suavizar bordes (antialiasing) o cálculos de punto flotante. En complemento a la CPU, la GPU ofrece ejecutar una gran cantidad de hilos concurrentemente aunque en general cada uno con menor poder de cómputo y eficientes solo para acciones no tan complejas.

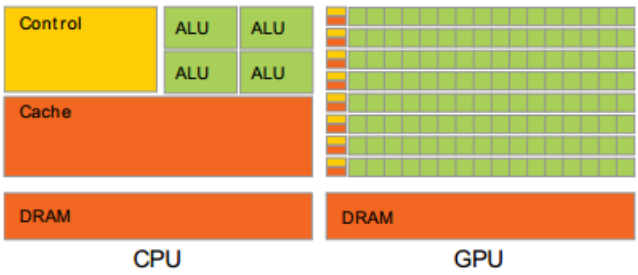

Figura 2.2: Esquema de arquitecturas de CPU y de GPU. Extraído de [11].

En la Figura 2.2 se muestra en forma esquematizada como es la organización de la arquitectura en CPU y en GPU. La CPU cuenta con menos cantidad de unidades funcionales pero con mucha más capacidad individual, mientras que la GPU se puede entender como un arreglo con gran cantidad de pequeños procesadores.

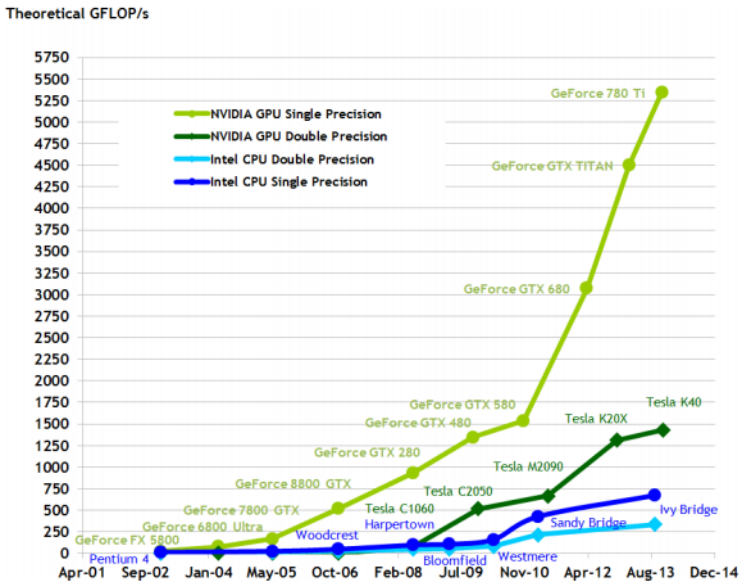

Figura 2.3: Comparativa de rendimiento de CPUs vs GPUs en cantidad de operaciones en punto flotante simple/doble precisión. Extraído de [11].

En la Figura 2.3 se puede ver el gran rendimiento teórico de algunas GPUs con respecto a algunas CPUs (y su evolución en el tiempo) en cuanto a cantidad de operaciones en punto flotante en simple/doble precisión.

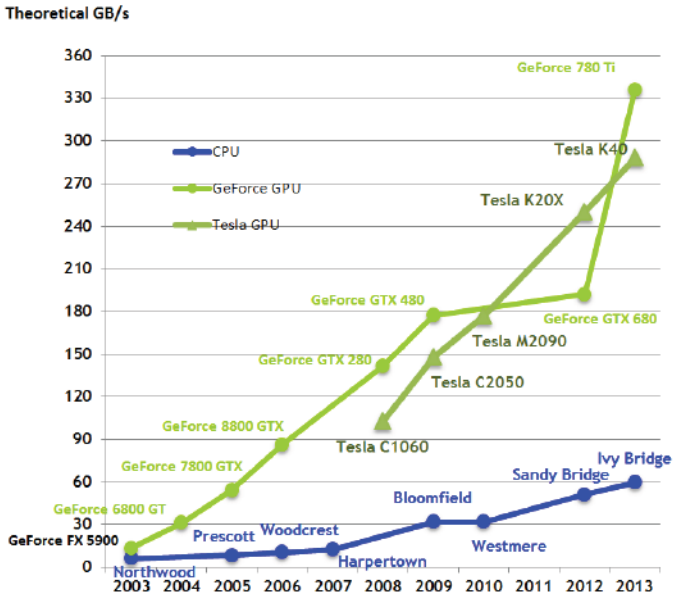

Figura 2.4: Comparativa de rendimiento de CPUs vs GPUs en accesos de datos. Extraído de [11].

En la Figura 2.4 se muestra un comparativo del rendimiento en cuanto al acceso a memoria (ancho de banda teórico) que se dispone en los mismos dispositivos.

La arquitectura de las GPUs de NVIDIA utiliza su propio sistema de microinstrucciones diferente al x86, por lo tanto ningún programa hecho para CPU puede ser ejecutado directamente en una GPU. Si bien esto puede ser visto como una desventaja, NVIDIA lo presenta como un gran acierto ya que es una optimización que se adecúa a esta arquitectura con gran capacidad de paralelismo.

GPGPU

GPGPU se refiere al área de trabajo que busca desarrollar la capacidad de cómputo de las GPUs para resolver problemas de propósito general, es decir, otras áreas que no sean específicamente sobre el procesamiento de gráficos. Ésta ha tenido un importante desarrollo en los últimos años, en especial desde el año 2007 cuando la empresa NVIDIA (uno de los principales fabricantes de tarjetas gráficas) presentó CUDA.

CUDA es un lenguaje de programación. Desde el punto de vista estrictamente de la codificación puede verse como una variación o extensión del lenguaje C, una suite de herramientas y su compilador asociado, creado para poder codificar algoritmos en GPUs de NVIDIA. Actualmente, por medio de *wrappers*, se puede usar también con otra cantidad de lenguajes, entre otros Python [6], Fortran [8] y Java [22].

CUDA permite abstraer la GPU como un conjunto de multiprocesadores compuestos a su vez por un conjunto de núcleos orientados a la ejecución de hilos.

Teniendo en cuenta la taxonomía de Flynn [16] que clasifica las arquitecturas de computadoras

según el uso de una/múltiples instrucciones y el procesamiento de uno/múltiples datos, CUDA propone una arquitectura que a nivel global se puede clasificar como SPMD (del inglés Single Program Multiple Data), ya que en cada multiprocesador ejecuta el mismo programa sobre distintos datos, mientras que a nivel de cada multiprocesador se clasifica como una arquitectura SIMT (del inglés Single Instruction, Multiple Thread), la instrucción es ejecutada en todos los hilos y en caso de divergencia (por ejemplo algunos hilos no entran en un if), la ejecución es serial (algunos hilos esperan), ya que no se puede pasar a la siguiente instrucción hasta que todos los hilos hayan ejecutado o salteado la instrucción.

Se difunde una única instrucción por ciclo a todos los elementos de procesamiento que pueden ejecutar (en función de las divergencias). Estos conceptos deben ser manejados por el programador CUDA para entender correctamente los distintos niveles de sincronización.

Un esquema general del flujo de procesamiento de CUDA, se puede resumir como:

1. Se copian los datos de la memoria principal a la memoria de la GPU.

2. La CPU encarga el proceso a la GPU.

3. La GPU ejecuta, en paralelo, en cada núcleo el algoritmo.

4. Se copia el resultado de la memoria de la GPU a la memoria principal.

Desde el punto de vista conceptual, CUDA organiza los hilos en una grilla de hasta tres dimensiones como un arreglo de bloques, donde cada bloque tendrá una estructura de hasta tres dimensiones con un número limitado de hilos. Existen diferentes comandos para sincronizar dichos hilos, es decir a nivel de hilos de un mismo bloque o en forma global al dispositivo. Mediante esta representación CUDA permite abstraer la GPU como un arreglo de unidades funcionales. Luego, un planificador se encargará de repartir la carga con un bajo overhead.

Otro aspecto muy importante en la GPU es la jerarquía de memoria, las GPUs cuentan con varios niveles de memoria y cachés. A nivel de CUDA (programador), cada hilo puede acceder a memoria privada solo accesible por el mismo, a su vez cada bloque puede tener un espacio de memoria compartida para sus hilos, y finalmente la memoria global es accesible por todos los hilos de todos los bloques. Cuanto más "local" es el acceso, más rápido es, pero la capacidad de la memoria es más limitada.

Una de las últimas funcionalidades que han surgido (recientemente) y que puede ser claves en el futuro, es la funcionalidad de paralelismo dinámico que se incluye en la nueva generación de tarjetas Kepler de NVIDIA, esto es que un kernel (proceso en GPU) tenga la capacidad de lanzar a funcionar otros kernels dinámicamente sin necesidad de la intervención de la CPU. Muchos de los trabajos relacionados que se analizarán en el Apartado 3.4 destacan esta funcionalidad vinculada a posibles optimizaciones a explotar en trabajos futuros.

2.2.4. Bibliotecas

El área de ALN se caracteriza por el uso de varias bibliotecas que son casi estándares. Este tipo de políticas permite que ante un nuevo hardware, si se implementan en forma eficiente las bibliotecas, los cambios en la arquitectura impactan directamente en el desempeño de la aplicación sin tener que modificar su código. En este sentido, por estar ligadas con el objetivo del proyecto, se destacan las bibliotecas BLAS y LAPACK.

BLAS

BLAS [29] (del inglés Basic Linear Algebra Subprograms), es una biblioteca que ofrece un gran conjunto de rutinas básicas para resolver problemas algebraicos. En su organización plantea tres grupos de rutinas, según el orden de operaciones de las mismas:

- BLAS-1: Implementa operaciones entre un vector y otro vector o escalar. Problemas de orden n. Operaciones de punto flotante (flops).
- BLAS-2: Implementa operaciones entre una matriz y un vector. Problemas de orden n^2 flops.
- BLAS-3: Implementa operaciones entre matrices. Problemas de orden n^3 flops.

En general cuanto mayor es el nivel de cómputo, mejores resultados se obtienen con esta biblioteca. En este sentido, BLAS-3 sería la más importante porque resuelve los problemas más costosos. En [46] se ofrece una introducción a la biblioteca.

Es importante destacar que BLAS es una especificación y muchos fabricantes ofrecen sus propias implementaciones ajustados a sus dispositivos. En arquitecturas tradicionales dos de las más eficientes son Intel MKL [21] y OpenBLAS [44].

cuBLAS [7] (del inglés NVIDIA CUDA Basic Linear Algebra Subroutines), es la versión que ofrece NVIDIA de la biblioteca BLAS optimizada para ejecutar en GPU.

LAPACK

LAPACK [37] del inglés Linear Algebra Package, es una especificación de rutinas para resolver problemas estándar del álgebra lineal, como son factorizaciones LU, QR y SVD, así como la resolución de sistemas lineales, en especial diseñado para explotar los beneficios de la biblioteca BLAS. La versión de referencia está implementada en Fortran90 y se encuentra disponible en [17].

ScaLAPACK [18] extiende LAPACK en arquitecturas de HPC distribuidas. ScaLAPACK requiere que se encuentre instalado LAPACK (que requiere BLAS) y el gestor de comunicaciones especializado en álgebra BLACS [45] (Basic Linear Algebra Communication Subprograms).

LAPACK en GPU

Si bien no existe una implementación completa de LAPACK en GPU, se dispone de varias bibliotecas que ofrecen implementaciones parciales. Entre las más importantes:

- Flame GPU [36] es una extensión del framework Flame [42] el cual es una biblioteca de ALN similar a LAPACK pero con un enfoque y metodología diferente, que incluso cuenta con su propio sistema de notación para modelar bucles de algoritmos. Permite obtener importantes aceleraciones sin un conocimiento profundo de CUDA (ver G. Quintana-Ortí y otros [4]).
- CULA [33] es una biblioteca de álgebra que se define en base a dos grandes subconjuntos de funciones:
  - CULA Dense: para trabajar con matrices densas, contiene varias de las implementaciones de LAPACK y BLAS, como por ejemplo rutinas para la resolución de sistemas de ecuaciones lineales, descomposición en valores singulares, y resolución de problemas de valores y vectores propios.
  - CULA Sparse: para trabajar con matrices dispersas, contiene múltiples métodos para operar sobre grandes matrices, soporte para almacenado de matrices dispersas en múltiples formatos y utilización de precondicionadores para la resolución de sistemas lineales.

  Fue desarrollado por la compañía EM Photonics, empresa dedicada a desarrollar computación paralela con GPUs y FPGAs (del inglés Field Programmable Gate Array) con patrocinio de NVIDIA.
- MAGMA [38] (del inglés Matrix Algebra on GPU and Multicore Architectures) pretende ser una biblioteca similar a LAPACK pero para arquitecturas híbridas compuestas por CPUs multi-core y GPUs. Recientemente, la biblioteca ha crecido en funcionalidades para álgebra dispersa.
- PLASMA [40] (del inglés Parallel Linear Algebra for Scalable Multi-core Architectures), es un proyecto similar a MAGMA (ambos son promovidos por los mismos grupos de investigación), que pretende ser un rediseño de LAPACK y SCALAPACK más enfocado a la escalabilidad bajo el paradigma de programación distribuida sobra arquitecturas híbridas de CPU multi-core y GPUs. En [39] se ofrecen más detalles.

Capítulo 3

Transformaciones matriciales

En este capítulo se describen, desde un punto de vista matemático, los diferentes métodos para realizar transformaciones matriciales y también se resumen otros trabajos relacionados con la temática de este proyecto.

3.1. Introducción

En la literatura existen dos métodos principales para realizar transformaciones matriciales, Givens y Householder, ambos consisten en aplicar sucesivas transformaciones lineales a una matriz para obtener una factorización equivalente (con unas de sus matices con forma triangular).

En el caso del método de Givens cada una de estas transformaciones lineales tiene como interpretación geométrica una rotación en el espacio, mientras el método de Householder consiste en transformar la matriz en su reflexión con respecto a un hiperplano determinado (elegido convenientemente). La aplicación de técnicas de Givens y Householder se respalda por un teorema general del álgebra que establece que toda transformación ortogonal (o unitaria) es un producto de simetrías (reflexiones o giros).

Teóricamente, el procedimiento de rotación de Givens es más útil en situaciones donde sólo pocos elementos están fuera de la diagonal y necesitan ser anulados, mientras que una reflexión de Householder tiene mayor costo computacional pero resuelve todos los elementos de la columna y por tanto se espera sea más efectiva en matrices densas.

3.2. Método de Givens

Como ya se introdujo antes, básicamente consiste en aplicar sucesivas rotaciones (transformaciones lineales) en las cuales cada una va anulando (haciendo cero) a un elemento de la subdiagonal inferior. Una transformación solo afecta dos filas de la matriz, como se ve en la Figura 3.1 que muestra un ejemplo de una rotación en el proceso de la escalerización.

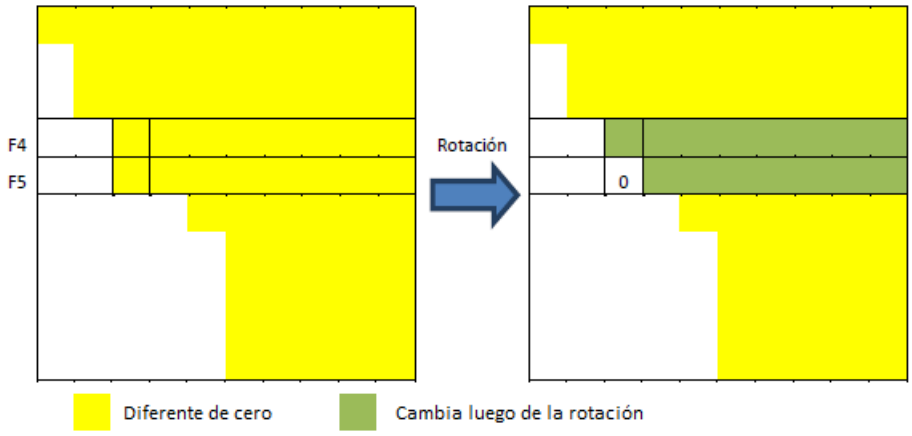

Figura 3.1: Esquema intuitivo de una rotación de Givens.

Las secciones blancas representan posiciones que ya están en cero, las amarillas distintas de cero, y las verdes son las afectadas luego de la rotación. En el ejemplo se toman las filas cuatro y cinco, las cuales tienen su primer elemento distinto de cero en la columna tres. Luego de la rotación se obtiene un cero en la entrada $M_{5,3}$.

Las operaciones para las dos filas involucradas ($f1$ y $f2$) y la columna k que se aplicarán serán las siguientes:

Para $x = 1..n$
$R_{f1,x} = M_{f1,x} \times cos(angulo) + M_{f2,x} \times (-sin(angulo))$;
$R_{f2,x} = M_{f2,x} \times cos(angulo) + M_{f1,x} \times sin(angulo)$;

donde $angulo = \arccos \frac{M_{f1,k}}{\sqrt{(M_{f1,k})^2+(M_{f2,k})^2}}$

Con este algoritmo se pueden paralelizar gran parte de las acciones ya que, como se mostró recién, es posible hacer una rotación para obtener un cero en una posición determinada de tal modo que afecte los valores numéricos solo en dos filas de la matriz. Si la matriz es de dimensión $n \times n$ se podría entonces aplicar $n/2$ rotaciones (para generar $n/2$ ceros) simultáneamente en casos no borde. Se puede ver un ejemplo concreto en el Anexo A.1.

Se define formalmente la matriz de transformación de Givens como la matriz G que cumple las siguientes reglas:

$$G(i,j,k) = \begin{vmatrix} 1 & . & 0 & . & 0 & . & 0 \\ : & \backslash & : & & : & & : \\ 0 & . & c & . & -s & . & 0 \\ : & & : & \backslash & : & & : \\ 0 & . & s & . & c & . & 0 \\ : & & : & & : & \backslash & : \\ 0 & . & 0 & . & 0 & . & 1 \end{vmatrix}$$

donde i y j corresponden a las filas que se desean rotar y k al índice de la celda en la fila j que se

desea hacer cero, $G_{i,i} = c, G_{j,j} = c, G_{j,i} = -s, G_{i,j} = s, G_{m,m} = 1$ para todo $m \neq i$, $m \neq j$, todos los otros lugares en cero, $s = \sin(\alpha)$, $c = \cos(\alpha)$, $\alpha = \arccos \frac{M_{i,k}}{\sqrt{(M_{i,k})^2+(M_{j,k})^2}}$.

Esta transformación genera una rotación de ángulo α en sentido horario en el plano formado por los vectores i y j. Al aplicar esta transformación a una matriz M se obtendrá un cero en la posición (j, k) de la misma. Por más detalles formales del algoritmo ver [25] [41]. En el trabajo [23] los autores muestran que hay un patrón con el que las rotaciones de Givens pueden ser computadas en paralelo.

3.3. Método de Householder

Consiste en construir sucesivas reflexiones tal que en cada una de ellas se resuelve una de las columnas de la matriz.

La matriz de transformación de Householder tiene tres propiedades muy ventajosas en algoritmos matemáticos. La matriz es igual a su matriz inversa (si un algoritmo la requiere, no será necesario calcularla). Es ortogonal, lo cual es muy bueno para trabajar con matrices semejantes. Tiene un único valor propio de multiplicidad n, esto hace que sea muy estable numéricamente (número de condición es uno). Basado en lo anterior, Householder es una de las herramientas más utilizadas para el cálculo de valores y vectores propios, y particularmente utilizado para el cálculo de matrices semejantes con forma de Hessenberg y en descomposición QR.

Una reflexión de Householder es una transformación lineal que refleja el espacio con respecto a un plano determinado. Esta matriz se puede construir de modo que un vector elegido, luego de transformado (reflejado) quede solo con una componente no nula, es decir sobre uno de los ejes de la base cartesiana. Para esto se definen:

- x el vector columna de la matriz A que se quiere diagonalizar,
- $\alpha = \|x\|$ (excepción: si x_1 tiene signo positivo entonces α tendrá signo negativo para evitar perdida de precisión),
- $e = \begin{bmatrix} 1 \\ : \\ 0 \end{bmatrix}$ vector con el primer elemento 1 y los restantes en 0,
- $u = x - \alpha * e$,
- $v = \frac{u}{\|u\|}$,
- $Q = I - 2vv^t$.

Por más detalles formales del algoritmo ver [25] [41].

3.4. Trabajo relacionado

Para realizar este proyecto de grado fue necesario revisar la literatura existente sobre proyectos o investigaciones relativas al uso de GPUs para acelerar transformaciones matriciales, y en especial la factorización QR y el cálculo de los valores y vectores propios. En la Tabla 3.1 se sintetizan los trabajos con mayor aporte entre la bibliografía revisada. Cada fila corresponde a un trabajo, mientras que cada columna indica si el trabajo cubre una temática específica, estas son: [Householder] y [Givens] indicando si la investigación trata sobre estos algoritmos, [Híbrido] refiere a si se trabajo en una arquitectura multi-core más GPU, [Bloques] indica si se investigan estrategias de algoritmos a bloques, [VPs] cálculo de valores y vectores propios, [MC] mínimos cuadrados, [DP] doble precisión, [Sim] si se realizan simulaciones, y [Formal] indica si se analiza el tema matemático estricto, por ejemplo con demostraciones, etc.

Sección	Ref	Householder	Givens	Híbrido	Bloques	VPs	MC	DP	Sim	Formal
3.4.1	[14]	Si	Si				Si			
3.4.2	[35]	Si	Si							Si
3.4.3	[10]	Si				Si		Si	Si	
3.4.4	[15]	Si		Si	Si					
3.4.5	[24]				Si	Si			Si	Si
3.4.6	[30]	Si			Si		Si			Si

Tabla 3.1: Trabajos relacionados sobre el uso de GPUs para acelerar trasformaciones matriciales.

3.4.1. Algoritmos de actualización de la factorización QR sobre GPU

R. Andrew [14] desarrolló una variante de la factorización QR en GPU para sistemas sobredeterminados. Un sistema sobredeterminado es definido como un conjunto de ecuaciones que contienen una o más variables desconocidas, donde la cantidad de ecuaciones es mayor que la cantidad de incógnitas. La técnica de mínimos cuadrados [26] es extremadamente útil para obtener una solución aproximada. Este problema se puede resolver utilizando la factorización QR con diferentes métodos iterativos. Para este objetivo, dos de las técnicas más utilizadas son las transformaciones ortogonales de Givens y Householder. En las sucesivas iteraciones de estos métodos, es muy frecuente que los cambios sean menores (números similares), ya que solo se actualizan ciertas regiones de la matriz y otras regiones permanecen invariantes. Al conocer la diferencia entre los datos nuevos y viejos, se pueden realizar ciertas optimizaciones para que, con pequeños ajustes, se reduzca significativamente el costo computacional de la actualización, y por consiguiente de toda la factorización QR.

El proyecto investiga la viabilidad de implementar algoritmos de actualización de factorización QR en GPU. El autor evalúa el rendimiento frente a algoritmos de actualización análogos en CPU. En especial se estudian cuatro tipos generales de algoritmos de actualización QR:

- Agregando un bloque de columnas. Lo que se puede interpretar como agregar variables al problema.

- Quitando un bloque de columnas. Lo que se puede interpretar como quitar variables al problema.

- Agregando un bloque de filas. Lo que se puede interpretar como agregar ecuaciones al sistema sobredeterminado.

- Quitando un bloque de filas. Lo que se puede interpretar como quitar ecuaciones al sistema sobredeterminado.

En el trabajo se optimiza la actualización QR para Householder y Givens. Se utilizó CULA [33] analizado en el Apartado 2.2.4 para comparaciones de rendimiento. Se trabaja con BLAS-2 y 3 para optimizar las dependencias entre las rotaciones de Givens y así maximizar el nivel de concurrencia. El cuello de botella de la propuesta, es decir donde se presentaron los principales problemas de rendimiento estaban vinculados a la alta frecuencia de invocación del kernel y la sobrecarga que esto implica. A entender del autor, este aspecto se podría mitigar en trabajos futuros explotando el paralelismo dinámico que ofrece la nueva generación de tarjetas Kepler de NVIDIA.

Algunas de las conclusiones destacadas por el autor son que los algoritmos de actualización de la factorización QR sobre GPU son mejores que los existentes para CPU en casi todos los tamaños de problemas abordados. Por otra parte, si la factorización QR se realiza en CPU (y se actualiza en GPU), los resultados son mejores con respecto a los que la realizan completamente en GPU, para ciertos parámetros de entrada (características de la matriz a resolver, como ser, cantidad de filas y columnas, tamaño de los bloques, etc). Por ejemplo utilizando la técnica de Givens y la estrategia de actualización agregando un bloque de columnas, para una matriz de 8000×6000, con tamaño de bloque 200, obtuvo un rendimiento 3,5 veces mejor (speed up) con respecto a una implementación completamente en GPU.

3.4.2. Descomposición QR en GPUs

Kerr et al. [35] también estudian la descomposición QR en GPUs. Este trabajo es similar en objetivos al presente proyecto. En este trabajo se explica cómo hacer una implementación de la factorización QR de alto rendimiento, detallando los cuidados a tener en cuenta. Para esto, los autores hacen una pormenorizada explicación de los algoritmos QR, Givens, Householder y Gram-Shmidt. Se discuten las características arquitectónicas de las GPUs, siempre explicando la necesidad de poseer un conocimiento detallado de la arquitectura para poder obtener buenos rendimientos.

Los resultados muestran un rendimiento muy alto para matrices de valores reales, alcanzando los 143 GFLOP/s en una implementación de Householder enteramente para GPU, en una tarjeta GeForce GTX 280 para matrices de 8192×4096. Estos resultados implican tiempos del orden de 7,63 segundos lo cual era lo más bajo hasta ese momento, por ejemplo 10 veces más rápido que el algoritmo QR de Matlab.

La implementación fue incluida en GPU VSIPL [27] (Vector Signal Image Processing Library), es la versión de la biblioteca para GPUs de NVIDIA.

3.4.3. Computación en GPU de los valores y vectores propios sobre matrices hermíticas pequeñas con números grandes

Otro enfoque es el de A. Cosnuau [10], quien presenta una implementación en GPUs del algoritmo QR con Householder para encontrar valores y vectores propios de matrices hermíticas pequeñas trabajando con aritmética de doble precisión, en tiempos acotados. En general las bibliotecas para GPUs son altamente eficientes para matrices de gran tamaño (por ejemplo a partir del orden de 2000×2000), pero para tamaños pequeños, las CPUs en general son más veloces. Entonces, una solución propuesta para explotar la velocidad de las GPUs consiste en paralelizar las tareas, es decir, realizar el cálculo de los valores y vectores propios de muchas matrices simultáneamente.

El resultado de este trabajo fue la obtención de los valores y vectores propios deseados, en menos de un segundo, con una precisión entre 10^{-6} y 10^{-9} para los valores propios y de entre 10^{-11} y 10^{-13} para los vectores propios. La tarjeta utilizada fue una NVIDIA Tesla M2070 de la arquitectura Fermi. Se obtuvieron resultados para matrices de diferentes dimensiones: $128, 256, 512$ y 1024. Las principales optimizaciones estuvieron en hacer uso masivo de registros y explotar la memoria compartida. Se puso especial atención en disminuir el número de operaciones realizadas, para lograr optimizar el algoritmo.

Se plantea como posible mejora a futuro, distribuir las matrices en varias GPUs y mejorar los pasos de diagonalización mediante el método de Givens.

3.4.4. Implementación de la factorización QR por bloques para GPU y múltiples CPUs

J. Kurzak et al. [15] también abordaron la factorización QR en un sistema con una computadora multi-core y una GPU potente. En particular, este artículo presenta una forma de implementar eficientemente el algoritmo de factorización QR en un sistema con una GPU potente y muchas CPUs multiprocesador.

Los elementos básicos que contribuyen al éxito del algoritmo son el procesamiento de la matriz por bloques de tamaño relativamente pequeño y planificación de operaciones en paralelo de forma dinámica. Las GPUs son tratadas como un conjunto de núcleos, cada uno de los cuales puede manejar eficientemente el trabajo a la misma granularidad que un núcleo de CPU estándar.

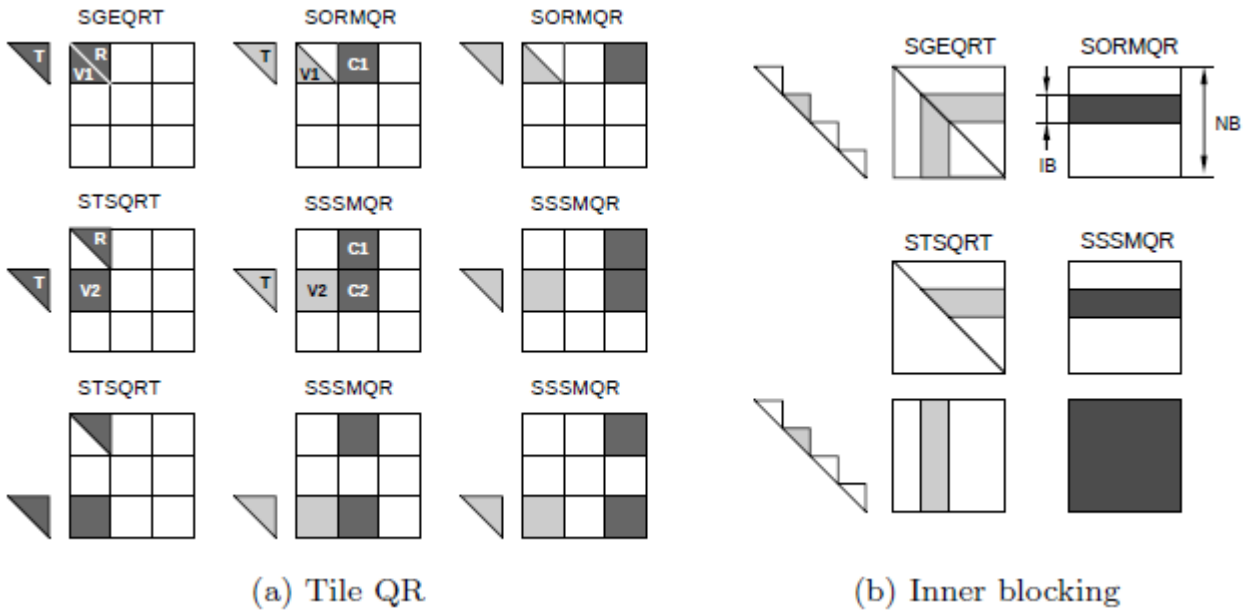

Figura 3.2: Factorización QR a bloques. Extraído de [15].

En la Figura 3.2 se muestra un esquema del algoritmo propuesto, se identifican cuatro procesos, los tres primeros corren en CPU y el último en GPU:

- SGEQRT y STSQRT, se encargan de la factorización de un bloque. Se ejecutan en CPU.
- SORMQR, función de LAPACK para aplicar QR. Se ejecuta en CPU.
- SSSMQR, es el único proceso que corre en GPU. Se encarga de una parte de la actualización y de manejar las columnas adicionales de la matriz.

Los autores dedican buena parte del trabajo a explicar el proceso de planificación y sincronización entre los cuatro tipo de procesos.

El trabajo muestra que un sistema equipado con un gran número de núcleos convencionales y una GPU puede utilizarse eficientemente para resolver matrices densas.

Se propone como objetivos a futuro desarrollar una versión para correr en la generación Fermi de tarjetas de video (que debería al menos duplicar rendimiento), y generalizar el trabajo para múltiples GPUs.

3.4.5. Computación de los valores propios con CUDA

Uno de los primeros trabajos sobre cálculo de valores y vectores propios en GPU es de C. Lessig [24].

Este trabajo se centra en una implementación del método de la bisección para calcular todos los valores y vectores propios con CUDA para matrices simétricas con dimensiones arbitrarias. Con este

objetivo el autor define con buen nivel de detalle varios de los conceptos algebraicos y el método de la bisección.

La implementación propuesta utiliza un método de escaneo *parallel prefix sum scan* ya existente [31]. También utiliza el método de los círculos de Gerschgorin para definir árboles de intervalos (donde buscar los valores propios) y puntos de partida para aplicar las recursiones. Implementa dos algoritmos, uno para matrices pequeñas (dimensiones menores a 512×512) y otra para tamaños variables. El aspecto que más se optimiza es el aprovechamiento de la jerarquía de memoria y acceso unificado (coalesced), además de minimizar la divergencia de código. Por ejemplo, un factor considerado es que las tarjetas con compute capabilities 1.x tienen 16KB de memoria compartida, por lo que matrices mayores a 2048×2048 no pueden ser alojados. En estos casos se almacenará sólo un intervalo.

3.4.6. Una aproximación de alto nivel a la resolución de problemas matriciales con almacenamiento en disco

También relacionado con el proyecto se destaca la tesis de doctorado de M. Marqués [30].

En esta memoria se diseña, desarrolla y evalúa una colección de rutinas para sistemas de ecuaciones lineales y problemas de mínimos cuadrados lineales de dimensiones elevadas haciendo uso de las técnicas OOC (del inglés Out Of Core). Estas pretenden explotar los beneficios de la jerarquía de memoria, en especial haciendo un uso eficiente del disco y combinado con la capacidad de paralelismo de los procesadores actuales como las CPUs y GPUs. Se optimizan tres factorizaciones:

- Cholesky (para matrices simétricas y definidas positivas).
- LU (no presenta estructura particular la matriz).
- QR (la matriz es rectangular con más filas que columnas, sistema sobredeterminado).

Si bien en este trabajo no se experimenta con GPU directamente, se ofrece un análisis detallado de los algoritmos, profundizando especialmente en el uso de bloques para la factorización QR.

Capítulo 4

Propuestas

En el marco del proyecto se desarrollaron diversas variantes de los métodos de Givens y de Householder que permiten sacar provecho de las prestaciones que brindan las GPUs. En forma complementaria se desarrollaron variantes en CPU para usarlas como referencia y en algunos casos se incluyen como parte de las implementaciones híbridas. Cada una de estas variantes será explicada en este capítulo.

4.1. Método de Givens

A continuación se detallan las cuatro implementaciones realizadas sobre el método de Givens, en donde se aplican y combinan algunas de las técnicas relevadas para la resolución del método, la primera es para CPU, luego dos son para GPU y la cuarta es para una arquitectura híbrida con técnicas de procesamiento a bloques. Finalmente, se resume la evaluación experimental realizada. Se recuerda que este método ya fue presentado de forma intuitiva y formal en la Sección 3.2.

4.1.1. Givens en CPU (GC)

La implementación del método de Givens en CPU (GC) plantea la rotación, tal como fue definida en la Sección 3.2, como un conjunto de operaciones que se aplican a cada elemento de las dos filas involucradas en la misma (ver Figura 3.1).

El algoritmo recorre la zona triangular inferior de la matriz por columnas. Para cada entrada, si su valor es diferente a cero, entonces busca otra entrada en la misma columna por debajo de la diagonal para realizar una rotación, y así generar un cero en la primera. Una vez identificadas las dos filas, se define el ángulo y se llama a la función auxiliar $RotacionConVectores$, que aplica las operaciones sobre las dos filas. Dicha función auxiliar requiere un arreglo de largo $2 \times n$ para almacenar, durante los cálculos, los nuevos valores de las dos filas.

Esta implementación utiliza un hilo de ejecución, es decir, las rotaciones se realizan de modo serial. Es esperable que este algoritmo sea adecuado para la resolución de matrices pequeñas.

4.1.2. Givens en GPU con planificación por columnas (GG1)

La implementación del método de Givens en GPU con planificación por columnas (GG1), al igual que GC (4.1.1), puede verse como una implementación simplificada del método ya que no

comienza a reducir una columna antes de haber terminado la anterior. Se distinguen tres etapas:

- Reinicio del contador: se pone en cero el contador de la planificación.
- Planificación: se lanza un hilo por cada fila y cada uno, según la columna que se está reduciendo, en caso de presentar un valor distinto de cero, agrega la coordenada en el plan (arreglo) con ayuda de operaciones atómicas.
- Ejecución del plan: se lanza un bloque por cada rotación a ejecutar (máximo $n/2$). Cada bloque ejecuta una de las rotaciones del plan distribuyendo el problema en todos los hilos del bloque.

Esta estrategia es más simple al momento de planificar con respecto a la planificación máxima (posible), debido a que ya está determinado con cual columna se está trabajando al comienzo de cada iteración. Resulta interesante incluir esta implementación ya que utiliza una planificación más sencilla, implica contar con menos estructuras auxiliares y menos operaciones atómicas respecto al algoritmo GG2 (que se presenta a continuación en el Apartado 4.1.3), pero también tendrá muchos más ciclos por estar limitada a ir resolviendo de a una columna a la vez.

En la Figura 4.1 se muestra un ejemplo gráfico de la planificación propuesta en este algoritmo para dos iteraciones reduciendo la columna tres.

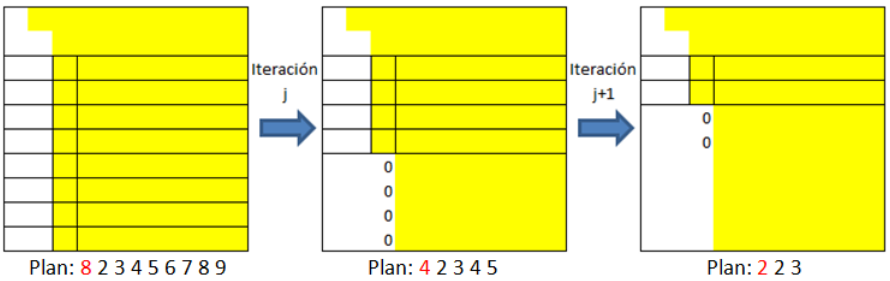

Figura 4.1: Esquema propuesto sobre una posible planificación en una iteración del método de Givens (con planificación por columnas).

Las secciones blancas representan posiciones que ya están en cero. En el plan, el primer número que aparece indica cuantos índices de filas están cargados y luego los índices. En la primera matriz se planifican 8 filas, luego de la iteración, se obtienen 4 casilleros en cero. En la segunda matriz se planifican 4 filas, luego de la iteración, se obtienen 2 casilleros en cero.

Esta implementación requiere un vector auxiliar donde se alojará el plan de rotaciones para cada iteración de tamaño $m \times 2 + 1$, además de las estructuras básicas mínimas ya definidas para GC (ver Apartado 4.1.1).

Teniendo en cuenta el costo adicional que implican las transferencias de datos (de CPU a GPU, y de GPU a CPU), se estima que las implementaciones en GPU sean efectivas para matrices que superen cierto tamaño mínimo, y por lo tanto que el beneficio se obtenga al operar con matrices de mayor tamaño.

4.1.3. Givens en GPU con planificación máxima (GG2)

A continuación se describe la implementación del método de Givens en GPU con planificación máxima (GG2). En cada ciclo del método se realiza la mayor cantidad de rotaciones posible (se pueden generar ceros en diferentes columnas en un mismo ciclo). Como desventaja frente al algoritmo anterior GG1 (4.1.2), éste requiere dos matrices adicionales las cuales se mantienen con operaciones atómicas y por tanto afectan el rendimiento:

- Pivotes: arreglo de largo $m \times 2$ que indica en su primera mitad, para cada fila, donde está la primer ocurrencia distinta de cero. En la segunda mitad se almacena el índice asignado en MPlan.

- MPlan: matriz de dimensiones $(n) \times (m+1)$ que guarda para cada columna los índices de las filas que tienen el pivote en dicha columna, para luego poder planificar agrupando de a pares de filas.

Se distinguen tres etapas en los ciclos de ejecución:

- Reinicio de contadores: se pone en cero la planificación y los índices de las matrices auxiliares.

- Planificación: se lanza un hilo por cada fila, recupera la posición guardada en el arreglo de pivotes y avanza por la fila hasta un valor distinto de cero. Agrega la coordenada en la matriz MPlan y posteriormente (cuando MPlan ya fue cargada), para cada fila, en caso de que el índice asignado sea par, se agregan en el vector de planificación los datos de columna (pivote), el índice de la fila actual y el de la fila anterior guardada en la matriz MPlan.

- Ejecución del plan: se lanza un bloque por cada rotación a ejecutar (máximo $n/2$). Cada bloque ejecuta una de las rotaciones del plan distribuyendo el problema en todos los hilos del bloque.

En la Figura 4.2 se puede ver un ejemplo de planificación en un ciclo donde se estaría reduciendo más de una columna. El mismo ejemplo con números se puede ver en el Anexo A.1.

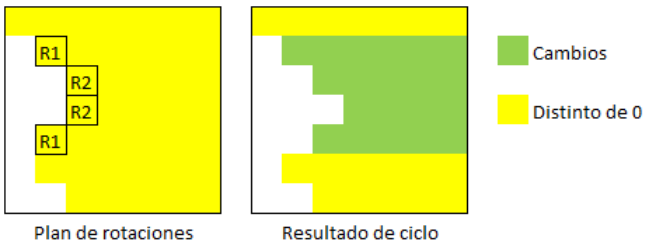

Figura 4.2: Esquema propuesto sobre una posible planificación en una iteración del método de Givens (con planificación máxima).

Se espera que este algoritmo tenga un rendimiento comparable (superior) al GG1 (4.1.2).

4.1.4. Givens híbrido a bloques (GH)

La implementación del método de Givens híbrido (GH), resuelve el problema sacando provecho de ambas plataformas. Las características de este método permiten aplicar técnicas de cómputo por bloques para incrementar la eficiencia.

Esta implementación fue diseñada siguiendo conceptos del método de HH (ver más adelante en el Apartado 4.2.3) y comparte varias de las estrategias de cómputo. La idea general es aplicar el algoritmo GC (4.1.1) para un bloque pequeño (aprovechando la efectividad del algoritmo para matrices de estas dimensiones) y luego actualizar consistentemente con operaciones más sencillas y paralelizables para los otros bloques de la matriz en GPU.

En la Figura 4.3 se describe el procedimiento en detalle, donde se tienen los siguientes pasos:

Paso 1: Localizar el bloque B_1 (de dimensiones pequeñas, definidas previamente) en la zona superior izquierda de la matriz original.

Paso 2: Identificar un bloque B_2 del mismo ancho y en las mismas columnas y por debajo de B_1 donde haya al menos un número diferente de cero. C_1 y C_2 serán los respectivos bloques a la derecha en la matriz original. Si no se pueden identificar más zonas para B_2 mover B_1 por la diagonal de la matriz y comenzar nuevamente el paso 2.

Paso 3: Calcular la factorización QR de $(\frac{B_1}{B_2})$ con $[Q_B, \frac{B_1}{B_2}] = GivensCPU(\frac{B_1}{B_2})$.

Paso 4: Actualizar $(\frac{C_1}{C_2})$ con Q_B en GPU.

Paso 5: Volver al paso 2.

En cuanto a estructuras auxiliares, esta variante requiere 4 matrices pequeñas auxiliares (el tamaño de bloque, *DimBloque*, debe ser chico). Dos matrices de tamaño $DimBloque \times n \times 2$ para los bloques C (una será temporal para los cálculos como pasaba en GC). Una matriz de tamaño $DimBloque \times DimBloque \times 2$ para el bloque B y una de tamaño $DimBloque \times DimBloque \times 4$ para el bloque Q.

Previamente se espera que esta implementación sea comparable con GG1 y GG2 en cuanto a rendimiento y precisión, ya que como ventajas, combina la efectividad de CPU con GC (4.1.1) para matices pequeñas, la efectividad de la GPU para la computación paralela, y las técnicas de procesamiento a bloques para volúmenes grandes de datos (también persigue objetivos de escalabilidad). Como desventajas, requiere el cálculo y transferencia a GPU de la matriz de transformación Q para los bloques B y requiere copiar sub-matrices para el manejo de los bloques B y C.

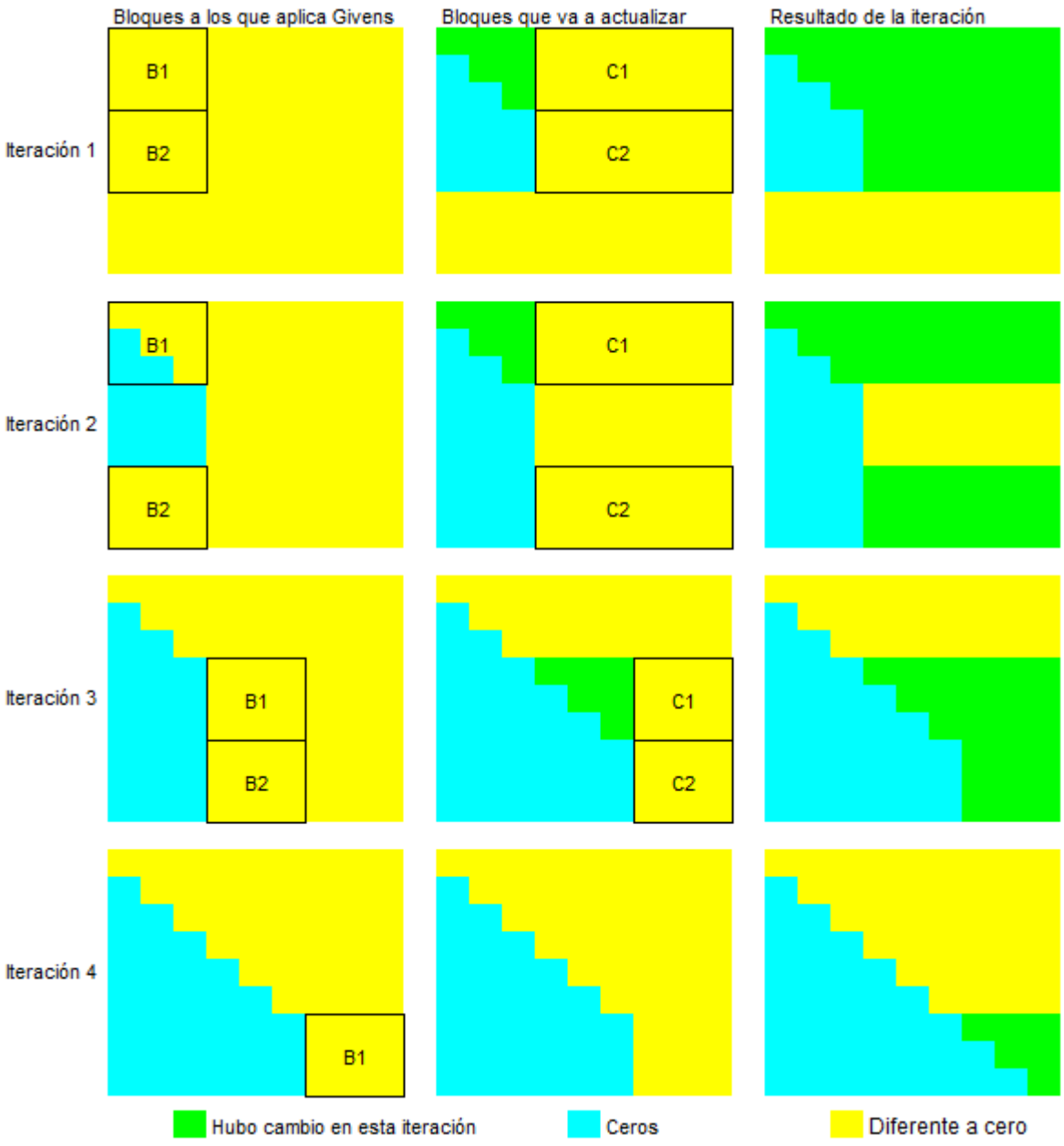

Figura 4.3: Esquema propuesto para el algoritmo híbrido de Givens a bloques.

4.1.5. Evaluación Experimental

La primera prueba que se presenta, busca determinar la cantidad de hilos óptima por bloque para los algoritmos de Givens construidos para GPU. A este tipo de prueba se le denominará Test de hilos, de aquí en adelante. La Figura 4.4 muestra el rendimiento, en tiempo (segundos), del algoritmo GG2 (4.1.3) al variar la cantidad de hilos por bloque entre 8 y 1024 para tres matrices de diferentes dimensiones, 512×512, 1024×1024 y 2084×2048.

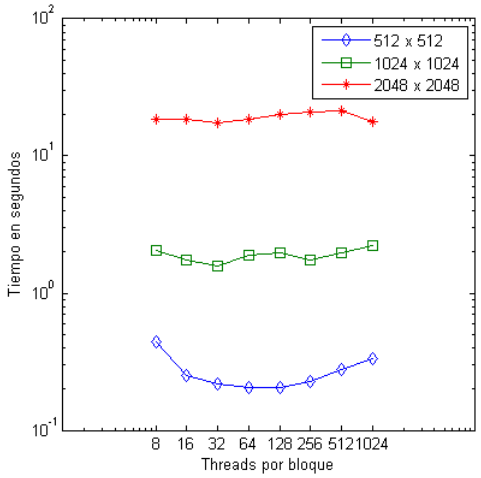

Figura 4.4: Test de hilos para la variante GG2.

Tanto para la matriz de 1024×1024 como para la de 2084×2048 el mejor resultado fue con 32 hilos que será el valor que tomará para el resto de las pruebas. Tal como se ve en la gráfica, para la matriz 512×512 también se obtuvo un resultado relativamente bueno. Si bien no se presentan, los resultados para GG1 (4.1.2) son similares.

Otra de las pruebas preliminares que fue realizada, fue la denominada Test de bloques, que tiene como objetivo establecer el valor óptimo de tamaño de partición de bloques. En la Figura 4.5, en este caso para el algoritmo GH (4.1.4), se observa la variación de rendimiento (en tiempo de ejecución) para los diferentes tamaños de bloques evaluados (2, 4, 8 y 16), y también para tres matrices con la misma cantidad de entradas pero con diferentes dimensiones (cantidad de filas y columnas).

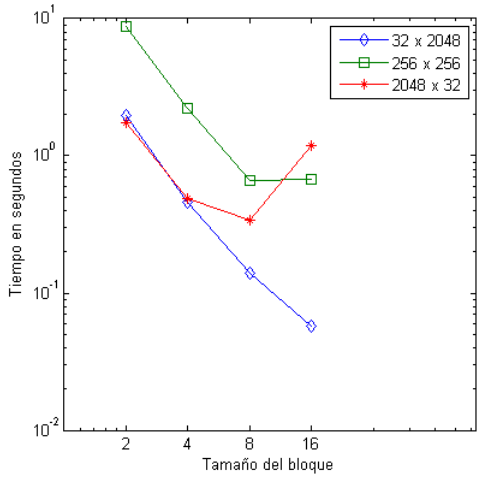

Figura 4.5: Test de bloques para la variante GH.

A partir de la gráfica antes mencionada, se definió que el tamaño de bloque que se utilizará será 8, porque en el promedio obtuvo los mejores rendimientos, aunque no fue el mejor en todos los casos.

En la Figura 4.6 se presenta la prueba que se denominará Test de matrices, ésta tiene como objetivo general, identificar en qué tipo de matrices se desempeña mejor cada algoritmo. Para esto, se evalúa el rendimiento en tiempo de ejecución (segundos) para varias matrices con igual cantidad de entradas pero con diferente largo de fila y columnas con la intención de ver si se desempeña mejor con matrices indeterminadas (128×2048), cuadradas (512×512) o sobredeterminadas (2048×128). Complementariamente, se evalúa el algoritmo con matrices de las mismas dimensiones mencionadas, pero además con la forma canónica de Hessenberg. Para esas matrices el 75 % de los ceros ya están resueltos, por lo que se busca evaluar si el algoritmo aprovecha esta ventajosa propiedad ya explicada en el Apartado 2.2.1.

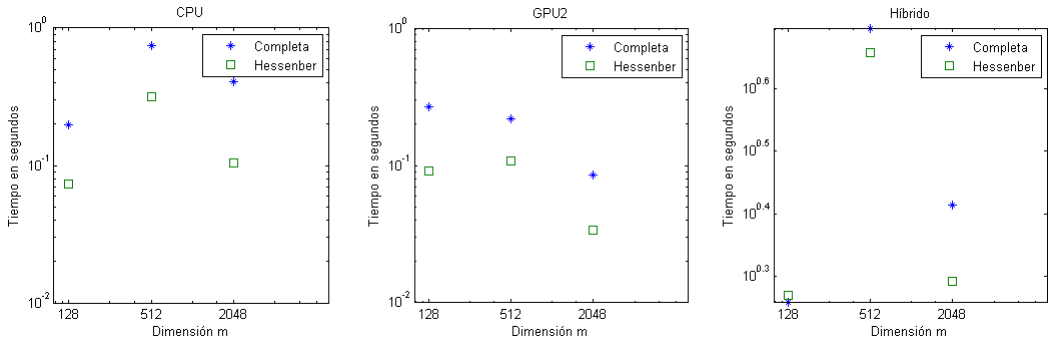

Figura 4.6: Test de matrices para Givens.

En la mayoría de los casos, la forma canónica de Hessenberg generó una mejora significativa en el rendimiento. Por otra parte, si bien cada algoritmo obtuvo mejores tiempos para un tipo u otro de matrices (indeterminada, cuadrada o sobredeterminada), a partir de estas prueba no se puede afirmar que claramente las implementaciones realizadas del método general de Givens sean más adecuadas para uno de los tipos de matrices, teniendo en cuenta además que es difícil establecer una comparativa justa ya que una matriz sobredeterminada requiere generar más ceros que una indeterminada, pero menor cantidad de cálculos de actualización.

En la Figura 4.7 se presentan los resultados en cuanto a rendimiento expresado como tiempos de ejecución para todos los algoritmos de Givens realizados, para matrices cuadradas densas de varias dimensiones (la de mayor tamaño será de 3072×3072). A estas pruebas se les denominará Test de rendimiento.

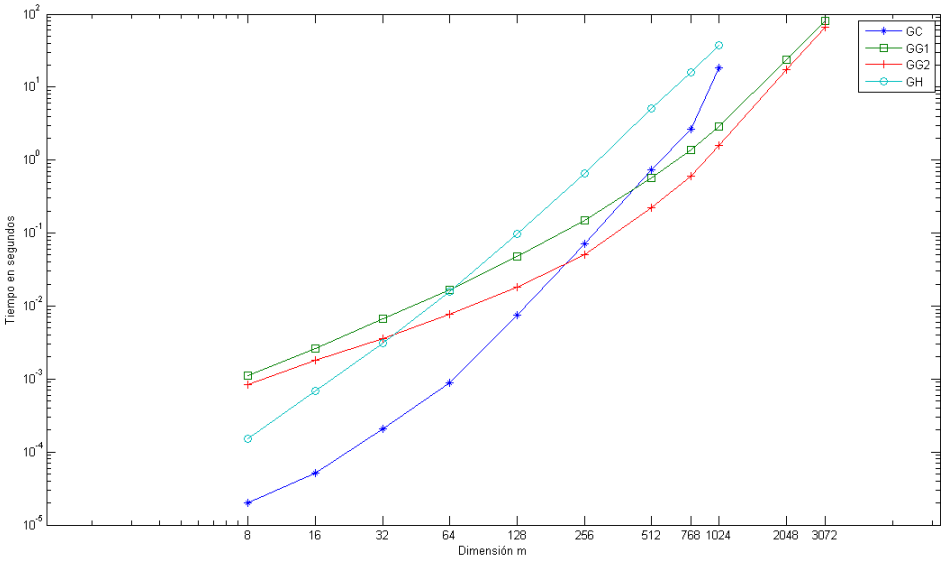

Figura 4.7: Test de rendimiento para Givens.

Lo primero a destacar sobre estos resultados, es que GG1 y GG2 superaron a GC acorde a las predicciones. Otra de las conclusiones alineada con lo previsto, es que que GG1 y GG2 presentan un rendimiento comparable, ya que cada uno tiene sus ventajas, y al variar la plataforma, podría variar la conveniencia de una u otra versión. Con respecto a GH se esperaba un rendimiento más cercano al de GG1 y GG2. Luego de realizar un análisis más detallado, se comprobó que el principal cuello de botella (donde se consume más tiempo) es en las operaciones de copia device to device, por lo que es posible que en otro hardware con mejor jerarquía de memoria tal vez logre un rendimiento más comparable.

En el Apartado 4.3 se pueden ver estos resultados en comparación con las implementaciones de Householder y el algoritmo de Hessenberg para CPU provisto por LAPACK (referencia como estado del arte).

Resumiendo, se ha comprobado experimentalmente la capacidad de paralelizar datos que ofrece el método de Givens para la escalerización de matrices densas en GPU y el buen rendimiento que se obtiene en relación a una implementación en CPU. También se ha verificado que el método como fue abordado puede ser más eficiente si la matriz se asemeja a la forma canónica de Hessenberg. La implementación híbrida construida no arrojó resultados acorde a los esperados. Se estima que la explicación radica en que el método presenta muchas copias device to device.

4.2. Método de Householder

A continuación se detallan las tres implementaciones realizadas sobre el método de Householder, en donde se aplican y combinan algunas de las técnicas relevadas para la resolución del método: la primera para CPU, la segunda para GPU y la tercera para hacer uso de una arquitectura híbrida con técnicas de procesamiento a bloques. Finalmente, se resume la experimentación sobre los diferentes casos evaluados. Se recuerda que este método ya fue presentado de forma intuitiva y formal en la Sección 3.3.

4.2.1. Householder en CPU (HC)

La implementación del método de Householder en CPU (HC) plantea la reflexión tal como fue definida en la Sección 3.3, en la que se le aplica una iteración por cada columna.

Los cálculos requieren al menos tres matrices auxiliares, dos de $m \times n$ y otra de $m \times m$, así como un vector auxiliar de lago m.

La estructura general del algoritmo recorre la matriz por las columnas, para cada una ejecuta la función $Reflexion Householder$, la cual realiza las operaciones, y al retorno de la función mencionada se actualizan las zonas de la matriz original que ya no cambiarán.

Solo cuenta con un hilo de ejecución por lo que los cálculos se realizan de modo serializado. Es esperable que esta variante presente rendimiento bueno pero sólo para matrices pequeñas al igual que GC (4.1.1).

4.2.2. Householder en GPU (HG)

La implementación del método de Householder en GPU (HG), nuevamente, plantea la reflexión tal como fue definida en la Sección 3.3, en la que se le aplica una reflexión por cada columna.

Al analizar el método, se observa que tiene como principal carga computacional operaciones básicas entre matrices como la suma y la multiplicación, además de operaciones con vectores. Para varias de estas operaciones se utilizó la biblioteca cuBLAS, que es la biblioteca que refleja el estado del arte para este tipo de operaciones en GPU.

La estructura del algoritmo es similar a GC, la diferencia es que agrega más operaciones de sincronización y transferencia de información entre CPU y GPU.

Considerando el costo adicional de transferencia de datos, el overhead que implica ejecutar un algoritmo en GPU, y que el algoritmo es básicamente el mismo que HC (4.2.1), se estima que el rendimiento en GPU será efectivo para matrices que superen cierto tamaño mínimo, y por lo tanto que el beneficio se obtenga al operar con matrices de mayor tamaño.

4.2.3. Householder híbrido a bloques (HH)

La implementación del método de Householder híbrido (HH), resuelve el problema en ambas plataformas. Las características de este método permiten aplicar técnicas de cómputo por bloques para incrementar la eficiencia, ver Sección 4.1.4 de [30]. Las Figuras 4.8 y 4.9 que se muestran en esta sección utilizan notación Flame [42] para describir los métodos.

La idea global es similar a GH (4.1.4). La estructura general está definida en el Algoritmo QR_{BLK} (Figura 4.8), que consiste en aplicar en cada iteración el algoritmo QR_{UNB} (Figura 4.9) en CPU para un bloque pequeño y luego actualizar consistentemente con operaciones más sencillas y paralelizables los otros bloques de la matriz en GPU. La diferencia con respecto a GH (4.1.4), es que las actualizaciones deben aplicarse de forma diferente (mayor complejidad) y en toda la matriz. Notar que, en GH (4.1.4) alcanzaba con actualizar las filas involucradas en las rotaciones en cada iteración.

Cabe destacar que se utilizó el algoritmo QR_{UNB} (Figura 4.9) para factorizar los bloques en lugar de utilizar HC (4.2.1), debido a que el primero además calcula vectores de Householder, que luego se utilizaran para construir la matriz S_b.

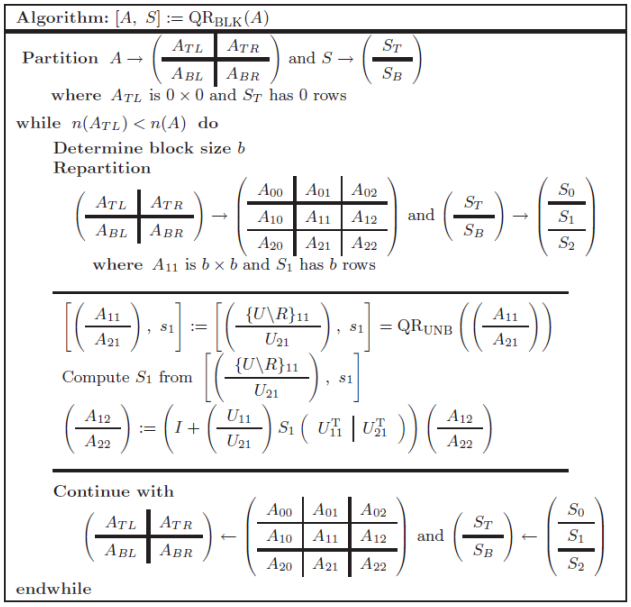

Figura 4.8: Algoritmo por bloques para calcular la factorización QR mediante transformaciones de Householder. Extraído de [30].

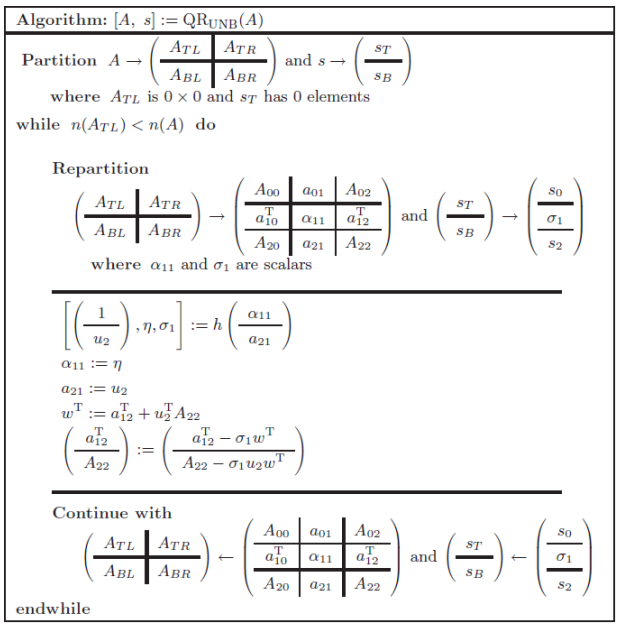

Figura 4.9: Algoritmo escalar para calcular la factorización QR mediante transformaciones de Householder. Extraído de [30].

En la Figura 4.8, cuando dice "compute S_1", se está refiriendo a construir S_b con los vectores de Householder de la siguiente forma:

$Si = -S_1$
$Sb = Si$
Para $i = 2..(A_{11})_n$
$Si = (-S_i * Si' * (A_{11}A_{21})'_{:,1:i-1} * (U_{11}U_{21})_{:,i:i})'$
$Sb_{i:i,1:i-1} = Si$
$Sb_{i,i} = -S_i$
$Si = Sb$

El resultado final de HH, en lugar de contener ceros por debajo de la diagonal contiene los vectores de Householder [1].

Se espera que esta implementación sea la más eficiente para matrices grandes dentro de las implementaciones de Householder, ya que combina el buen desempeño de la CPU para cálculos que que no ofrecen altos niveles de paralelismo, con la efectividad de GPU para la computación masivamente paralela, lograda al aplicar las técnicas de procesamiento a bloques.

4.2.4. Evaluación Experimental

Se comenzará mostrando los resultados del Test de hilos, que busca determinar la cantidad de hilos óptima por bloque en la GPU. La Figura 4.10 muestra el rendimiento en tiempos de ejecución (en segundos) del algoritmo HG (4.2.2) al variar la cantidad de hilos por bloque entre 8 y 1024 para tres matrices, de 512×512, 1024×1024 y 2084×2048.

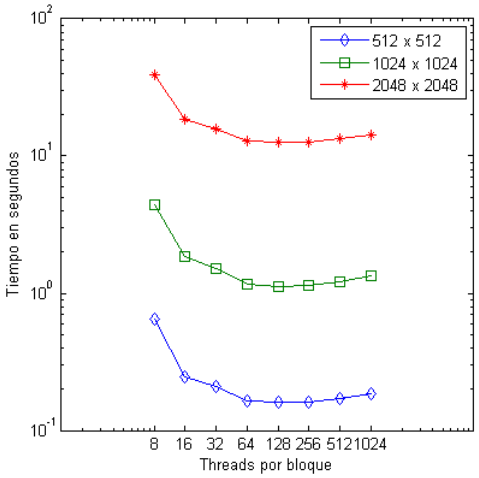

Figura 4.10: Test de hilos para la variante HG.

Para las tres matrices, el mejor resultado obtenido fue con 128 hilos por bloque, y por lo tanto será el valor que se utilizará por defecto.

[1] Solo para este caso, en la función final que verifica si la matriz obtenida es triangular superior, se define la tolerancia hasta el valor 1, ya que las componentes del vector no pueden superar este valor.

La otra prueba preliminar realizada consiste en aplicar el Test de bloques para establecer el valor óptimo de tamaño de partición de bloques. En la Figura 4.11, para el algoritmo HH (4.1.4), se observa la variación de rendimiento (en tiempo de ejecución) para los diferentes tamaños de bloque 2, 4, 8 y 16, y también para tres matrices con la misma cantidad de entradas pero con diferentes dimensiones (cantidad de filas y columnas).

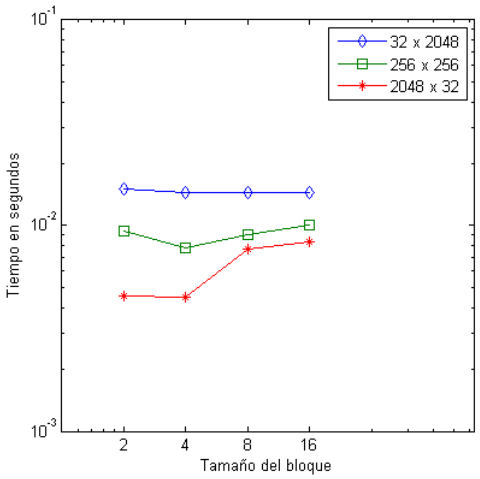

Figura 4.11: Test de bloques para la variante HH.

A partir de esta gráfica, se definió que el tamaño de bloque que se utilizará será 4, ya que en todos los caso dio el mejor rendimiento.

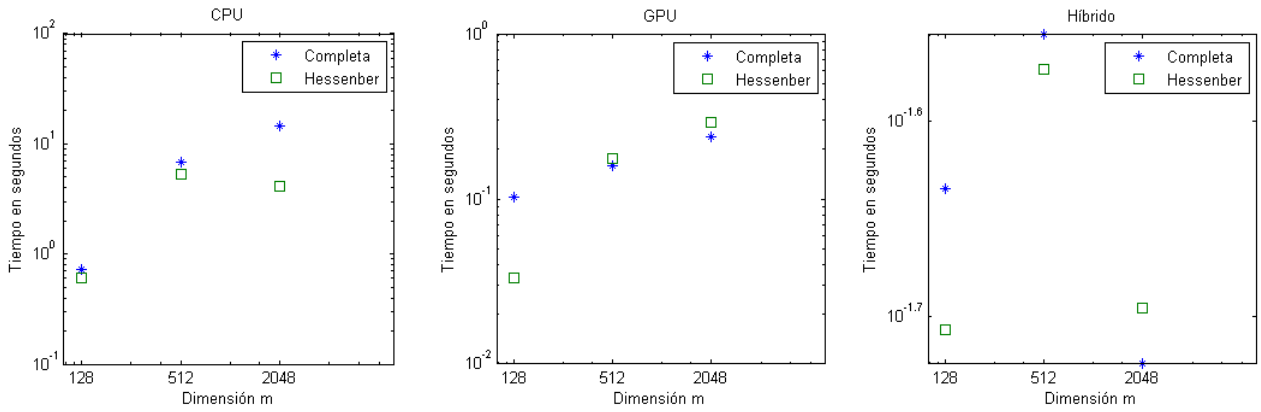

Figura 4.12: Test de matrices para Householder en las diferentes variantes implementadas.

En la Figura 4.12 se presenta el Test de matrices para Householder. Se recuerda que esta prueba tiene como objetivo general identificar en que tipo de matrices se desempeña mejor determinado algoritmo. Para esto, evalúa el rendimiento en tiempo de ejecución (en segundos) para diferentes matrices con igual cantidad de entradas pero con diferentes dimensiones (cantidad de filas y columnas) con la intención de ver si se desempeña mejor con matrices indeterminadas

(128×2048), cuadradas (512×512) o sobredeterminadas (2048×128). Complementariamente, se evalúa el algoritmo con matrices de las mismas dimensiones, pero además con la forma canónica de Hessenberg. Para esas matrices, el 75 % de los ceros ya están resueltos, por lo que se busca evaluar si el algoritmo aprovecha esta ventajosa propiedad ya explicada en el Apartado 2.2.1.

En seis de las nueve pruebas, las matrices con forma canónica de Hessenberg tuvieron mejor rendimiento respecto a las matrices completas, mientas que en los tres casos restantes el rendimiento fue relativamente similar, por lo que se concluye que el método puede ser más eficiente cuando la matriz se asemeja a la forma de Hessenberg. Por otra parte, respecto al tipo de matrices (indeterminada, cuadrada o sobredeterminada), teóricamente era esperable que Householder presentara mejor rendimiento para los casos de matrices sobredeterminadas, si se analiza la gráfica, los resultados no reflejaron una conclusión clara.

En la Figura 4.13 se presentan los resultados del Test de rendimiento para las distintas implementaciones del método de Householder, evaluando el resultado en tiempo de ejecución (en segundos) para matrices cuadradas densas de varias dimensiones (la de mayor tamaño será de 3072×3072).

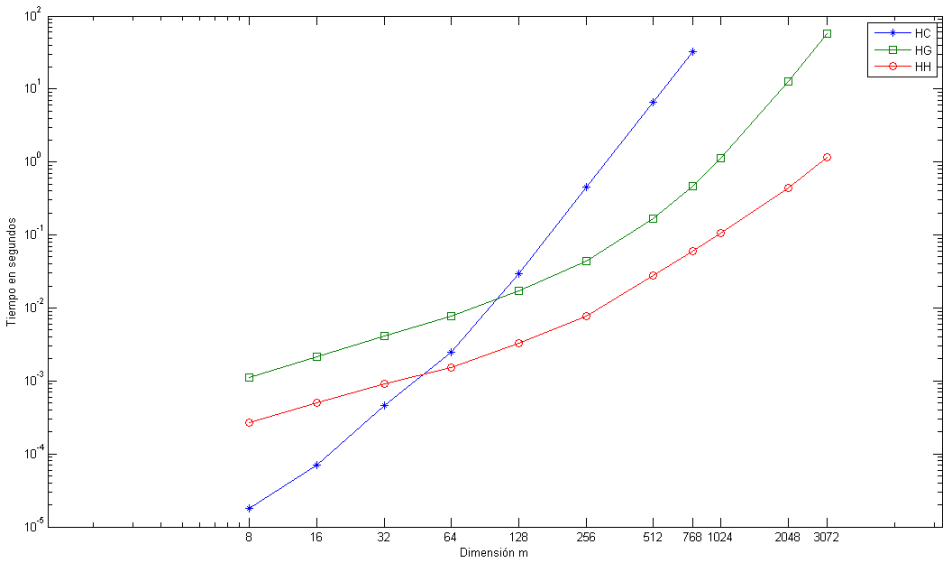

Figura 4.13: Test de rendimiento para Householder.

Lo primero a destacar sobre este resultado es que se cumplieron todas las predicciones: por un lado HG superó a HC en matrices grandes, y por otra parte, la implementación híbrida HH logró aprovechar las ventajas de ambas plataformas, así como también de las técnicas de procesamiento a bloques, logrando mejor rendimiento respecto a HC y HG.

En el Apartado 4.3 se pueden ver estos resultados en comparación con las implementaciones de Givens y el algoritmo de Hessenberg para CPU provistos por LAPACK.

A modo de resumen, se ha comprobado experimentalmente la capacidad de paralelizar datos que ofrece el método de Householder para la escalerización de matrices densas en GPU y el buen

rendimiento que se obtiene en relación a una implementación en CPU. También se ha verificado que el método, como fue abordado, puede ser más eficiente si la matriz se asemeja a la forma canónica de Hessenberg. La implementación híbrida construida logró aprovechar las ventajas de ambas plataformas, así como también de las técnicas de procesamiento a bloques, y por lo tanto se logró el mejor rendimiento.

4.3. Evaluación comparativa

Se definió un conjunto de tests (Tabla 4.1), es decir un conjunto de ejecuciones de los algoritmos para algún conjunto de matrices especiales con el objetivo de caracterizar los algoritmos y poder determinar la conveniencia de selección de éstos al momento de resolver un problema real, y de compararlos con las predicciones teóricas y algoritmos de referencia basados en las bibliotecas estándares. En el Anexo A se describe la herramienta desarrollada donde se detalla cómo probar las variantes implementadas y la organización del código.

Sección	Test	Descripción
4.3.2	Hilos	Encontrar cuál es la cantidad de hilos óptima por bloque en GPU para los algoritmos construidos.
4.3.2	Bloques	Definir para los algoritmos GH y HH el tamaño de bloque más adecuado.
4.3.2	Matices	Evaluar el rendimiento en tiempo de los algoritmos construidos para un grupo variado de matrices con diferentes características.
4.3.2	Rendimiento	Evaluar el rendimiento en tiempo de los algoritmos construidos para un grupo de matrices cuadradas densas.

Tabla 4.1: Tests desarrollados.

4.3.1. Algoritmos de referencia

Se utilizó una implementación del método de Hessenberg en CPU (SC) provistos por las bibliotecas BLAS y LAPACK como implementación de referencia en CPU.

4.3.2. Casos de estudio

Test de hilos

El objetivo de este test es encontrar cual es la cantidad de hilos óptima por bloque en GPU para los algoritmos implementados (para luego realizar los siguientes tests con este valor).

Los resultados ya fueron presentados anteriormente en las Figuras 4.4 y 4.10. Para GG1 y GG2 se determinó que la cantidad de hilos por bloque en la GPU es 32. Para HG se determinó que la cantidad de hilos por bloque en la GPU es 128.

Notar que estos valores son altamente dependientes de la plataforma de ejecución, es decir, si se ejecuta en tarjetas con otras características podrían variar.

Test de bloques

El objetivo es definir para los algoritmos GH y HH el tamaño de bloque más adecuado.

Los resultados ya fueron presentados anteriormente en las Figuras 4.5 y 4.11. Para GH el tamaño de bloque por defecto a utilizar será 8. Para HH el tamaño de bloque por defecto a utilizar será 4.

Test de matrices

Este test tiene como objetivo general, identificar en qué tipo de matrices se desempeña mejor determinado algoritmo. Para esto, evalúa el rendimiento (en tiempos de ejecución) para diferentes matrices con igual cantidad de entradas pero con diferente largo de filas y/o columnas con la intención de ver si se desempeña mejor con matrices indeterminadas (128×2048), cuadradas (512×512) o sobredeterminadas (2048×128). Complementariamente, se evalúa el algoritmo con matrices de las mismas dimensiones descriptas, pero además con la forma canónica de Hessenberg, ya que para esas matrices el 75 % de los ceros ya están resueltos, por lo que se busca evaluar si el algoritmo aprovecha esta ventajosa propiedad ya explicada en el Apartado 2.2.1.

Los resultados ya fueron presentados anteriormente en las Figuras 4.6 y 4.12. En la mayoría de los casos la forma canónica de Hessenberg generó una mejora en el rendimiento significativa, por lo que se puede afirmar que tanto para el método de Givens, como de Householder, si la matriz a la que se le va a aplicar la factorización, se asemeja a esta forma canónica, entonces los algoritmos seguramente logren un mejor desempeño relativo.

Con respecto a la forma de las matrices (indeterminada, cuadrada o sobredeterminada), si bien para cada algoritmo se obtuvieron mejores tiempos para un tipo u otro de matriz, a partir de estas pruebas no se puede afirmar claramente si el método de Givens o el de Householder es el más adecuado para algunos de estos tipos, teniendo en cuenta además que es difícil comparar, ya que una matriz sobredeterminada requiere generar más ceros que una indeterminada, pero menor cantidad de cálculos de actualización (siempre comparando con la misma cantidad total de entradas). Se identifica aquí una posible mejora o trabajo a futuro, que consiste en realizar más pruebas en diferentes contextos para poder caracterizar mejor los contextos favorables de los métodos.

Test de rendimiento

En la Figura 4.14 se presentan los resultados obtenidos, se incluye el algoritmo SC (4.3.1) como referencia de estado del árte.

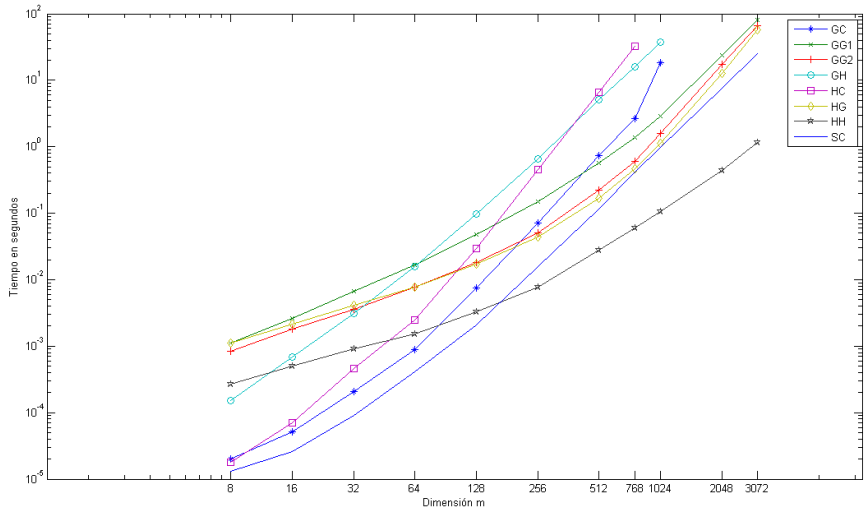

Figura 4.14: Rendimiento de los algoritmos implementados.

El objetivo de este test es evaluar el rendimiento en tiempo de ejecución (en segundos) de los algoritmos construidos para matrices cuadradas densas de varias dimensiones (la de mayor tamaño será de 3072×3072).

Acorde a las predicciones, las implementaciones en GPU superaron a las de CPU para matrices grandes, en el caso de Givens, GG1 y GG2 superaron a GC, y en Householder HG superó a HC. También se cumplió que los algoritmos en CPU fueron más eficientes en tiempos de ejecución para matrices pequeñas.

Otra de las conclusiones acorde a lo previsto, es que que GG1 y GG2 presentan un rendimiento comparable, ya que cada uno tiene sus ventajas, y al variar la plataforma, podría variar la conveniencia de una u otra variante. Con respecto a GH, se esperaba un rendimiento más acorde al de GG1 y GG2. Luego de realizar un análisis más detallado, se comprobó que el principal cuello de botella (donde se consume más tiempo) es en las operaciones de copia de tipo device to device, por lo que es posible que en otro hardware con mejor jerarquía de memoria tal vez logre un rendimiento más adecuado.

La implementación híbrida HH logró aprovechar las ventajas de ambas plataformas, así como también de las técnicas de procesamiento a bloques, logrando el mejor rendimiento, e incluso superó a SC, el cual es el algoritmo de referencia sobre el estado del arte. Esta es una de las conclusiones más importantes del proyecto, pues en esta implementación se han aplicado varias de las técnicas relevadas con resultado satisfactorio.

Los algoritmos en GPU (GG1, GG2 y GH), si bien no lograron superar a SC, presentan tiempos de ejecución comparativos (mismo orden de rendimiento), es decir, que si se ejecutaran en otra plataforma más moderna (la actual fue especificada en la Sección 1.3 y está relativamente desactualizada), el rendimiento comparativo entre los mismos podría cambiar.

Capítulo 5

Conclusiones y trabajo futuro

5.1. Conclusiones

El primer aspecto a destacar es que se alcanzaron los objetivos originalmente propuestos (Sección 1.1) en el proyecto. Entre ellos se encontraban: estudiar y manejar los conceptos matemáticos básicos relacionados con las transformaciones matriciales, relevar los avances de CUDA y diseñar y programar variantes de los métodos de transformaciones matriciales explotando las principales fortalezas de las GPUs de NVIDIA.

Se estudiaron varios de los conceptos matemáticos relacionados con las transformaciones ortogonales de matrices, las cuales tienen como etapa fundamental el cálculo de valores y vectores propios (Sección 2.2.1). En este sentido fue necesario revisar varios de los métodos existentes para calcularlos, evaluando la conveniencia y contexto de su utilización, teniendo en cuenta además la propagación de errores (Sección 2.2.2) debido al uso de la representación de punto flotante. También se relevaron los avances en CUDA y su impacto para el desarrollo del área GPGPU (Sección 2.2.3).

Por último, se diseñaron e implementaron diversas variantes de los métodos de transformaciones matriciales capaces de explotar el poder de cómputo que ofrecen las tarjetas gráficas modernas. En cuanto a los desarrollos, los principales esfuerzos son:

- Método de Givens:
  - Se implementó el método en CPU (GC). Plantea la rotación como una operación escalar que se aplica a cada elemento de las dos filas involucradas en la misma.
  - Se implementaron dos variantes para GPU. La primera (GG1) es una implementación simple del método, ya que no comienza a reducir una columna antes de haber terminado la anterior. Una de sus ventajas es que requiere menos estructuras auxiliares. La segunda implementación (GG2), realiza la mayor cantidad de rotaciones posibles para cada ciclo del método.
  - Se diseñó e implementó el método para hardware híbrido con una estrategia a bloques (GH), en el que se distribuyeron los cálculos dependiendo de la localidad y complejidad de las diferentes etapas. En CPU se hicieron los de mayor localidad y complejidad, mientras que en GPU se realizan los referidos a la actualización (principalmente productos de matrices).

- Método de Householder:
  - Se implementó el método en CPU (HC), que resuelve la factorización tal como fue definida en la Sección 3.3.
  - Se implementó el método para GPU (HG). Una de las principales optimizaciones, fue que a partir del análisis de las operaciones, buscando identificar el mayor consumo computacional, se detectó que los cuellos de botella computacionales se encuentran en operaciones básicas entre matrices como suma y multiplicación, además de operaciones con vectores. Para varias de estas operaciones se utilizó la biblioteca cuBLAS.
  - Se diseño e implementó el método para hardware híbrido con estrategia a bloques (HH), en el que se distribuyeron los cálculos dependiendo de la localidad y complejidad de las diferentes etapas. En CPU se hicieron los de mayor localidad y complejidad, mientras que en GPU se realizan los referidos a la actualización (principalmente productos de matrices).

En la evaluación experimental de las propuestas se destacan los siguientes resultados:

- Se logró verificar experimentalmente que, tanto para el método de Givens como de Householder, si la matriz a la que se le va a aplicar la factorización se asemeja a la forma canónica de Hessenberg, los algoritmos seguramente logren un mejor desempeño relativo.
- Las implementaciones en GPU superaron a las de CPU para matrices grandes. En el caso de Givens, GG1 y GG2 superaron a GC, y en Householder HG superó a HC. También se cumplió que los algoritmos en CPU fueron más eficientes para matrices pequeñas.
- Los algoritmos en GPU (GG1, GG2 y GH), si bien no lograron superar a SC, presentan tiempos de ejecución comparativos (mismo orden de rendimiento), es decir, que si se ejecutaran en otra plataforma más moderna (la actual fue especificada en la Sección 1.3 y está relativamente desactualizada), el rendimiento comparativo entre los mismos podría cambiar.
- La implementación híbrida HH logró aprovechar las ventajas de ambas plataformas, también de las técnicas de procesamiento a bloques, logrando el mejor rendimiento, e incluso superó a SC, el cual es el algoritmo que representa el estado del arte. Esta es una de las conclusiones más importantes del proyecto, pues en esta implementación se han aplicado varias de las técnicas relevadas con resultado satisfactorio.

En definitiva, lo que se obtuvo es una propuesta (Capitulo 4) en la que se diseñaron y desarrollaron diferentes variantes de los métodos de Givens y Householder para las plataformas CPU, GPU e híbridas. Posteriormente, se logró identificar posibles mejoras y líneas de trabajo futuro las cuales se describen en la siguiente sección.

5.2. Posibles mejoras y trabajo futuro

Una de las líneas de trabajo futuro es diseñar, codificar y optimizar nuevos algoritmos que complementen y enriquezcan los obtenidos en el marco de este proyecto. Estos pueden ser varios de los que se mencionan en la Sección 2.2.1. Por ejemplo abordar los métodos específicos para:

- Descomposición en valores singulares.
- Matrices simétricas, por ejemplo con el método de Jacobi.
- Matrices dispersas, tema que implica un amplio abanico de opciones, ya que, al cambiar la forma en que se representan las matrices, la forma de procesarlas es también diferente.
- El método de las potencias con desplazamiento, por ejemplo con el cociente de Rayleigh.

Durante la etapa de experimentación, en el test de matrices 4.3.2, se identificó un área de posible mejora o trabajo a futuro, la idea es profundizar el test, que consiste en realizar más pruebas sobre diferentes aspectos para poder caracterizar mejor los contextos favorables de los métodos y algoritmos con respecto a las dimensiones de la matriz (indeterminada, cuadrada o sobredeterminada).

Una posible mejora de los resultados del proyecto sería incrementar los métodos de verificación actuales. La herramienta sólo tiene una función que valida si un resultado es triangular superior. Por ejemplo, sería útil contar con otra función que verifique que la matriz original y la obtenida son equivalentes del punto de vista de la transformación lineal.

Otra manera de enriquecer la evaluación experimental, podría ser continuar trabajando con los conjuntos de pruebas ya existentes, y ejecutarlas en otras plataformas de hardware, por ejemplo en diferentes generaciones de tarjetas NVIDIA, intentando relevar la perspectiva de mejora de rendimiento, aplicado a este problema, que presenta la evolución tecnológica.

Otra línea de trabajo podría ser, partiendo del conocimiento generado sobre algoritmos de Householder y Givens con las técnicas de procesamiento a bloques, trabajar en implementaciones sobre plataformas distribuidas. Esto implica un desafío especial considerando el alto acoplamiento de datos que presenta el problema. Como paso intermedio en la escalabilidad, se podría profundizar en los algoritmos implementados de Givens y Householder orientados a bloques.

Anexo A

Herramienta desarrollada

Uno de los objetivos principales de este proyecto fue generar algoritmos eficientes en GPU, por lo que es de esperar que alguien pueda usar este código con algún objetivo similar, ya sea para utilizar los algoritmos, implementar nuevos, modificar o crear tests para intentar descubrir alguna característica, exportarlo a otras bibliotecas, etc. Por este motivo se incluye la Tabla A.1 a modo de una breve guía inicial del código para que un programador pueda encontrar lo que busca rápidamente.

Códigos	Cabezales	Descripción
Givens.cpp	Givens.h	Algoritmos para CPU de Givens implementados.
Householder.cpp	Householder.h	Algoritmos para CPU de Householder implementados.
Hessenberg.cpp	Hessenberg.h	Wrapper a la función de Hessenberg de LAPACK.
Auxiliares.cpp	Auxiliares.h	Todas las funciones auxiliares utilizadas.
PrjCuda.cu	-	Código principal del programa y funciones programadas en CUDA.

Tabla A.1: Estructura de archivos del código construido.

Al ejecutar el programa sin parámetros en la entrada se desplegará la lista de algoritmos y tests (conjunto de pruebas sobre los algoritmos) igual a la siguiente salida:

Transformaciones ortogonales de matrices utilizando GPUs

Ejecute nuevamente el programa indicando como parámetro la simulación o test desea correr

GC - Simulación de Givens en CPU *
GG1 - Simulación de Givens en GPU versión 1 *
GG2 - Simulación de Givens en GPU versión 2 *
GH - Simulación de Givens Híbrido *
HC - Simulación de Householder en CPU *
HG - Simulación de Householder en GPU *
HH - Simulación de Householder Híbrido *
SC - Simulación de Hessenberg en CPU *
TR - Test de rendimiento
TM - Test de matrices
TH - Test de hilos
TB - Test de bloques

* Para ejecutar las simulaciones se debe indicar:
2do Parámetro: Cantidad de filas (entre 2 y 10000)
3do Parámetro: Cantidad de columnas (entre 2 y 10000)
4to Parámetro opcional: Para generar matriz inicial con forma de Hessenberg indicar alto del triángulo con ceros
5to Parámetro opcional: Indicar una D para modo debug

A tener en cuenta:

- Para mayor comodidad se recomienda ejecutar el programa desde el script PrjCuda.sh ya que si se quiere correr directamente el ejecutable se debe establecer algunas variables de ambiente.

- Todos estos algoritmos generan los resultados tanto en pantalla como en archivos de texto plano con el nombre SalidaX.txt siendo X el código del algoritmo.

- Si se desea ejecutar un algoritmo específico se debe ingresar cantidad de filas y columnas de la matriz, mientras que si se selecciona un test no hay que ingresar otros parámetros.

- Todos los algoritmos se pueden ejecutar en modo Debug para ver paso por paso como es el proceso.

En la Tabla A.2 se resumen las secciones en que fueron descriptos cada uno de los algoritmos desarrollados.

Sección	Algoritmo
4.1.1	Givens en CPU.
4.1.2	Givens en GPU con planificación por columnas.
4.1.3	Givens en GPU con planificación máxima.
4.1.4	Givens híbrido en bloques.
4.2.1	Householder en CPU.
4.2.2	Householder en GPU.
4.2.3	Householder híbrido en bloques.

Tabla A.2: Algoritmos desarrollados.

A.1. Ejecución en modo debug

Ejecución de ejemplo del programa para el algoritmo GG2 (Givens en GPU segunda implementación), para una matriz de 7x7 en modo debug:

```
sh PrjCuda.sh GG2 7 7 D
```

Salida en un paso intermedio de la ejecución del programa:

22.924232	6.729088	12.241065	17.215782	16.148487	14.118609	14.792727
-0.000000	-2.187568	-2.539570	3.014976	-2.917861	-2.124671	-2.058492
-0.000001	0.000000	1.268949	3.237145	1.282065	-0.243063	-3.768930
-0.000000	0.000000	0.969502	1.255097	-6.272617	-4.152138	4.605588
0.000001	3.107887	-6.951385	4.311803	0.755891	3.046731	3.520329
-0.000001	3.544214	-0.040304	-1.334294	2.281762	3.498647	2.773510
-0.000001	-0.000000	-0.167464	2.389427	1.011905	2.811033	2.918376

—plan————
2 2,3,2 1,4,1
————————
————matriz aux planificación———-
1 0
3 1 4 5
3 2 3 6
0
0
0
0
————————
————pivotes———-
0 1 2 2 1 1 2
————————

22.924232	6.729088	12.241065	17.215782	16.148487	14.118609	14.792727
-0.000000	-3.800581	4.222680	-1.790551	-2.297608	-3.714368	-4.063556
-0.000001	0.000000	1.596923	3.334280	-2.789387	-2.713931	-0.198790
-0.000000	0.000000	-0.000000	-0.967964	-5.762702	-3.151811	5.947836
0.000001	-0.000000	-6.077840	4.947288	-1.950970	0.016233	0.342947
-0.000001	3.544214	-0.040304	-1.334294	2.281762	3.498647	2.773510
-0.000001	-0.000000	-0.167464	2.389427	1.011905	2.811033	2.918376

—plan————
2 2,4,2 1,5,1
————————

Explicación de los datos desplegados:

- Plan: El primer número indica la cantidad de rotaciones que se planificaron. Luego, cada plan indica: columna, fila 1, fila 2. En este caso hay:
  - 2 rotaciones planificadas (rojo).
  - la primera es la columna 2 con las filas 3 y 2 (verde).
  - la segunda es la columna 1 con las filas 4 y 1 (amarillo).

  —plan————
  2 2,3,2 1,4,1
  ————————

- Pivotes: Arreglo de largo n que indica para cada fila donde está la primera ocurrencia distinta de cero. Ejemplos:
  - La fila 0 la primer ocurrencia distinta de cero está en la posición 0 (azul).
  - La fila 1 la primer ocurrencia distinta de cero está en la posición 1 (rojo).
  - La fila 2 la primer ocurrencia distinta de cero está en la posición 2 (blanco).

  ———pivotes———-
  0 1 2 2 1 1 2
  ————————

- Matriz auxiliar de planificación: Cada fila está asociada a una columna de la matriz que se está escalerizando. En la primera posición se guarda la cantidad de pivotes que está en esa columna (rojo) y en las siguientes posiciones se guardan los índices de las filas. En este caso:
  - En la columna 0 hay 1 pivote en la fila 0.
  - En la columna 1 hay 3 pivotes en la filas 1, 4 y 5. Se pudo planificar la rotación de las filas 1 y 4 (amarillo).
  - En la columna 2 hay 3 pivotes en la filas 2, 3 y 6. Se pudo planificar la rotación de las filas 2 y 3 (verde).

```
———-matriz aux planificación———
1 0
3 1 4 5
3 2 3 6
0
0
0
0
————————-
```

Observación: quedaron tres pivotes (filas 0, 5 y 6) que no se pudieron planificar porque no había disponible otra fila con el pivote en la misma columna.

A.2. Salida del programa

La salida del programa en todos los casos es una tabla con las columnas de la Tabla A.4˙

Duración	Duración del algoritmo.
Algoritmo	Número de algoritmo, hay cinco disponibles más dos conjuntos de pruebas (Tests).
m	Cantidad de filas de la matriz.
n	Cantidad de columnas de la matriz.
hess	Cantidad de diagonales en cero de la matriz con forma de Hessenberg.
dimBloque	Dimensión de bloque, solo aplica para GH y HH.
threadsBloque	Cantidad de hilos por bloque, solo aplica para los algoritmos que corren utilizan la GPU.
p(0)	Probabilidad de ocurrencias en cero.
verifica	Luego de correr el algoritmo corre una función para verificar que la matriz obtenida es triangular superior.

Tabla A.4: Parámetros de salida del programa.

Al ejecutar el programa, por cada simulación que se ejecuta se genera una línea con los valores de cada parámetro descripto. Por ejemplo, al ejecutar el algoritmo GG2 se genera la siguiente salida:

```
Transformaciones ortogonales de matices utilizando GPUs

Duracion, algoritmo, m, n, hess, dimBloque, threadsBloque, p(0), verifica
0.139479, GG2, 7, 7, 0, 0, 32, 0.000100, SI
Los resultados fueron guardados en el archivo SalidaGG2.txt
```

Bibliografía

[1] P. Arbenz. Numerical methods for solving large scale eigenvalue problems (libro). http://people.inf.ethz.ch/arbenz/ewp/Lnotes/chapter9.pdf, Fecha Último Acceso (FUA) 2015-09-22.

[2] C. García Argos. Apuntes de métodos numéricos. 2o E.T.S.I. http://www.telecos-malaga.com/descargas/apuntes/2Curso/MN/MN-Apuntes.pdf, FUA 2014-07-26.

[3] Facultad de Ingenieria Centro de Cálculo. Computación de Alta Performance. https://www.fing.edu.uy/inco/cursos/hpc/pmwiki/index.php, FUA 2015-08-22.

[4] G. Quintana-Ortí; E.S. Quintana-Ortí; R.A Van De Geijn; F.G Van Zee; E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, July 2009.

[5] NVIDIA Corporation. Nvidia Compute Unified Device Architecture (CUDA) Toolkit. https://developer.nvidia.com/cuda-toolkit, September FUA 2014-08-04.

[6] NVIDIA Corporation. GPU accelerated computing with Python (wrapper para cuda). https://developer.nvidia.com/how-to-cuda-python, FUA 2015-07-30.

[7] NVIDIA Corporation. CUDA CUBLAS Library. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUBLAS_Library.pdf, FUA 2015-08-23.

[8] NVIDIA Corporation. CUDA Fortran (wrapper para cuda). https://developer.nvidia.com/cuda-fortran, FUA 2015-08-30.

[9] NVIDIA Corporation. Página oficial (fabricante de tarjetas aceleradoras). http://www.nvidia.es, FUA 2015-10-10.

[10] A. Cosnuau. Computation on GPU of eigenvalues and eigenvectors of a large number of small hermitian matrices. *Procedia Computer Science*, 29(0):800 – 810, 2014.

[11] CUDA. CUDA C programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, FUA 2014-09-08.

[12] J.L. de la Fuente. Métodos Matemáticos de Especialidad Ingeniería Eléctrica. Valores y vectores propios, valores singulares (diapositivas). http://jldelafuenteoconnor.es/Clase_valvec_2014.pdf, FUA 2015-10-10.

[13] M. Medina Diaz. Álgebra lineal numérica (libro). http://sb.uta.cl/libros/algebra_lineal_numerica.pdf, FUA 2015-04-03.

[14] R. Andrew; N. Dingle. Implementing QR factorization updating algorithms on GPUs. *Parallel Computing*, 40(7):161 – 172, 2014.

[15] J. Kurzak; R. Nath; P. Du; D. Peng; J. Dongarra. An implementation of the tile QR factorization for a GPU and multiple CPUs. In Kristján Jónasson, editor, *Applied Parallel and Scientific Computing*, volume 7134 of *Lecture Notes in Computer Science*, pages 248–257. Springer Berlin Heidelberg, 2012.

[16] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

[17] National Science Foundation. LAPACK (Linear Algebra PACKage). http://www.netlib.org/lapack/, FUA 2015-09-01.

[18] National Science Foundation. ScaLAPACK (Scalable Linear Algebra PACKage). http://www.netlib.org/scalapack, FUA 2015-09-01.

[19] S. Gupta. New top 500 list 4x more GPU supercomputers (artículo). http://blogs.nvidia.com/blog/2012/07/02/new-top500-list-4x-more-gpu-supercomputers/, FUA 2014-11-21.

[20] Intel. Familia de productos Intel Xeon Phi (fabricación de aceleradoras gráficas entre otros dispositivos). http://www.intel.la/content/www/xl/es/processors/xeon/xeon-phi-detail.html, FUA 2015-08-30.

[21] Intel. Math Kernel Library, Intel MKL (biblioteca, página oficial). https://software.intel.com/en-us/intel-mkl, FUA 2015-08-30.

[22] jcuda.org. Java bindings for NVIDIA CUDA and related libraries. http://www.jcuda.or, FUA 2015-10-10.

[23] A.H. Sameh; D.J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25(1):81–91, January 1978.

[24] C. Lessig. Eigenvalue computation with CUDA. https://developer.nvidia.com/sites/default/files/akamai/cuda/files/eigenvalues.pdf, FUA 2014-08-20.

[25] G.H. Golub; C.F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[26] S.A. Cruces Álvarez. El método de mínimos cuadrados. http://personal.us.es/sergio/PDocente/lectura.pdf, FUA 2015-08-30.

[27] D.A. Schwartz; R.R. Judd; W.J. Harrod; D.P. Manley. VSIPL 1.3 API, VSIPL Forum. http://portals.omg.org/hpec/files/vsipl/VSIPL1p3.pdf, FUA 2015-10-10.

[28] E. Strohmaier; J. Dongarra; H. Simon; M. Meuer. Top 500 supercomputing site. http://www.top500.org, FUA 2014-11-21.

[29] University of Tennessee. BLAS (biblioteca, página oficial). http://www.netlib.org/blas/, FUA 2015-10-11.

[30] M. Marqués Andrés; G. Quintana Ortí; E.S. Quintana Ortí. Una aproximación de alto nivel a la resolución de problemas matriciales con almacenamiento en disco (tesis doctoral). http://hdl.handle.net/10234/29515, FUA 2015-10-11.

[31] M. Harris; S. Sengupta; J.D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

[32] Madbox PC. La Unión Europea quiere regular el consumo de las tarjetas gráficas (artículo). http://www.madboxpc.com/la-union-europea-quiere-regular-el-consumo-de-las-tarjetas-graficas/, FUA 2015-07-13.

[33] EM Photonics. CULA (biblioteca, página oficial). http://www.culatools.com/, FUA 2015-08-24.

[34] U. Pirzada. Nvidia Power Efficency Through the Ages – from the 6600GT to the Maxwell 750 ti (artículo). http://wccftech.com/nvidia-power-efficency-ages-6600gt-maxwell-750-ti/, FUA 2015-01-21.

[35] A. Kerr; D. Campbell; M. Richards. QR decomposition on GPUs. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 71–78, New York, NY, USA, 2009. ACM.

[36] P. Richmond; D. Romano. Flame GPU (biblioteca, página oficial). http://www.flamegpu.com/, FUA 2015-09-01.

[37] E. Anderson; Z. Bai; C. Bischof; S. Blackford; J. Demmel; J. Dongarra; J. Du Croz; A. Greenbaum; S. Hammarling; A. McKenney; D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[38] University of Tennessee The Innovative Computing Laboratory. MAGMA (biblioteca, página oficial). http://icl.cs.utk.edu/magma/, FUA 2015-06-09.

[39] University of Tennessee The Innovative Computing Laboratory. Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. http://icl.cs.utk.edu/news_pub/submissions/plasma-scidac09.pdf, FUA 2015-06-09.

[40] University of Tennessee The Innovative Computing Laboratory. PLASMA (biblioteca, página oficial). http://icl.cs.utk.edu/plasma/index.html, FUA 2015-06-09.

[41] D.P. Bertsekas; J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Athena scientific optimization and computation series. Athena Scientific, 1997.

[42] F.G. Van Zee. libflame: The Complete Reference (biblioteca, página oficial). http://www.lulu.com/shop/field-van-zee/libflame-the-complete-reference/paperback/product-17975241.html, FUA 2015-05-23.

[43] L.J. Alvarez Vazquez A.M. Martinez Varela. *Lecciones de Métodos Numéricos, Tema 4: Cálculo numérico de autovalores*. Universidad de Vigo, 2005.

[44] Z. Xianyi; W. Qian; W. Saar; Z. Chothia; C. Shaohu; L. Wen. OpenBLAS an optimized BLAS library (biblioteca, página oficial). http://www.openblas.net/, FUA 2015-10-09.

[45] J. Dongarra; R.C. Whaley. The BLACS Basic Linear Algebra Communication Subprograms (biblioteca, página oficial). http://www.netlib.org/blacs/, FUA 2015-02-27.

[46] L.S. Blackford; J. Demmel; J.J. Dongarra; I.S. Duff; S.Hammarling; G. Henry; M. Heroux; L. Kaufman; A. Lumsdaine; A. Petitet; R. Pozo; K. Remington; R.C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *Transactions of the ACM on Mathematical Software*, 28(2):135–151, 2002.