PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

IMPLEMENTACIÓN DE LA ITERACIÓN LANCZOS EN ARQUITECTURA CUDA

Tesis para optar el Título de Ingeniero Electrónico, que presenta el bachiller:

Erick Daniel Rosales Jara

ASESOR: Paul Antonio Rodriguez Valderrama

Lima, 2015

Resumen

Los autovalores y autovectores son elementos muy utilizados en diversos problemas como análisis de estructuras, reconocimiento de imágenes, compresión de datos, solución de problemas electrodinámicos, entre otros. Exitsen muchos algoritmos para calcular y tratar con autovalores y autovectores mediante el uso de computadoras, sin embargo, cuando sólo se requiere uno o unos pocos autovalores (los más significativos) y autovectores, se puede optar por *Power Method* o la Iteración Lanczos. Por otro lado, factores como la cantidad de información a procesar o la precisión deseada pueden significar tiempos de ejecución no aceptables para ciertas aplicaciones, surgiendo la alternativa de realizar implementaciones paralelas, siendo la arquitectura CUDA una de la mejores opciones actualmente.

En la presente tesis se propone diseñar e implementar un algoritmo paralelo para la iteración Lanczos en arquitectura CUDA, el cual es un método para el cálculo del mayor autovalor y su correspondiente autovector. La propuesta está dividia en tres bloques principales. El primer bloque realiza la tridiagonalización parcial de una matriz cuadrada simétrica. El segundo bloque calcula la descomposición de Schur de la matriz tridiagonal obteniendo los autovectores y autovalores de esta. El tercer bloque calcula el mayor autovalor y su correspondiente autovector de la matriz inicial a partir de lo obtenido en etapas anteriores y determinará si es necesario seguir realizando cálculos. Los bloques trabajan iterativamente hasta encontrar resultados que se ajusten a la precisión deseada.

Además de la implementación paralela en CUDA, se realizaron implementaciones en el entorno de simulación MATLAB y en lenguaje C secuencial, con el propósito de comparar y verificar una correcta y eficiente implementación paralela. Los resultados computacionales evaluados para una matriz de 4000×4000 elementos reflejan un rendimiento de $13{,}4$ y $5{,}8$ al compararse la implementación en CUDA con MATLAB y C secuencial respectivamente. Estos rendimientos tienden a crecer mientras mayor sea el tamaño de la matriz.

La organización de la tesis es: en el primer capítulo se describe la problemática del tema. En el segundo capítulo se explica la teoría correspondiente a *Power Method* y Lanczos, así como los algoritmos necesarios. En el capítulo tres se exponen conceptos fundamentales sobre arquitectura CUDA. El diseño del algoritmo paralelo se desarrolla en el capítulo cuatro. Finalmente, en el capítulo cinco, se muestran y analizan los resultados computacionales, seguidos de las conclusiones, recomendaciones y bibliografía.

FACULTAD DE
CIENCIAS E INGENIERÍA

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

TEMA DE TESIS PARA OPTAR EL TÍTULO DE INGENIERO ELECTRÓNICO

Título	:	Implementación de la Iteración Lanczos en Arquitectura CUDA.
Área	:	Procesamiento Digital de Señales # 1156
Asesor	:	Paul Antonio Rodriguez Valderrama
Alumno	:	Erick Daniel Rosales Jara
Código	:	20095636
Fecha	:	03/10/13

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
Facultad de Ciencias e Ingeniería
13 NOV. 2013
DANIEL TORREALVA DAVILA
DECANO

Descripción y Objetivos

En ingeniería es de gran importancia lograr diseñar y construir estructuras que sean resistentes al viento o a fuertes temblores. Asimismo, en la época actual donde la sociedad e industria utilizan grandes cantidades de petróleo y sus derivados es bastante necesario tener métodos adecuados para la búsqueda de fuentes de este mineral. Además, en ciudades de todo el mundo, el tráfico vehicular siempre ha sido un problema y continuamente se buscan sistemas inteligentes capacez de controlarlo basándose en el flujo de autos. Todos los ejemplos descritos anteriormente y otros como predicción del clima o electrodinámica pertenecen a distintos campos de estudio como ingeniería civil, mecánica, petrolera, electrónica, estadística, seguridad, etc. pero tienen en común que todos los problemas pueden ser solucionados mediante el uso de autovalores y autovectores.

Los autovalores y autovectores permiten modelar sistemas y verificar su estabilidad además de ser una ayuda para el control de estos. Dicho de una forma matemática, permiten solucionar ecuaciones lineales, pero no solo eso, sino que también es posible realizar aproximaciones matriciales que pueden ser utilizadas para la solución de otros problemas. Cabe señalar que en el campo de matemática aplicada se han desarrollado gran cantidad de algoritmos especialmente para calcular y tratar con autovalores y autovectores mediante el uso de computadoras.

En la presente tesis se propone diseñar e implementar un algoritmo paralelo para la iteración Lanczos en arquitectura CUDA, el cual es un método para el cálculo del mayor autovalor y su correspondiente autovector. Además se realizarán implementaciones en el entorno de simulación MATLAB y en lenguaje C de forma serial, con el propósito de comparar y verificar una correcta implementación paralela.

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
SECCIÓN ELECTRÓNICA
Dr. Ing. BENJAMIN CASTAÑEDA APHAN
Coordinador de la Especialidad de Ingeniería Electrónica

MAXIMO 50 PÁGINAS

FACULTAD DE
CIENCIAS E INGENIERÍA

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

TEMA DE TESIS PARA OPTAR EL TÍTULO DE INGENIERO ELECTRÓNICO

Título : Implementación de la Iteración Lanczos en Arquitectura CUDA

Índice

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
SECCIÓN ELECTRICIDAD Y ELECTRÓNICA

Dr. Ing. BENJAMÍN CASTAÑEDA APHAN
Coordinador de la Especialidad de Ingeniería Electrónica

MÁXIMO 50 PÁGINAS

Índice general

Introducción

En la actualidad la cantidad de información a procesar en una computadora suele ser bastante grande, como es el caso de videos HD de larga duración, aún más si tienen fines científicos, es por eso que en ocasiones se suelen realizarse aproximaciones para transformar la información en una representación compacta. Una de las formas de hacer esto es mediante *Low Rank Approximation* [1].

La forma más utilizada para calcular esta aproximación es *Singular Value Decomposition* (SVD). Sin embargo, este método es considerado lento, pues tiene un costo computacional de $O(nm^2 + n^3)$, donde $m \times n$ son las dimensiones de la matriz inicial. Como alternativa, se puede emplear la iteración Lanczos, la cual entregará el mismo resultado que SVD siempre y cuando la matriz original cumpla ciertos requisitos [2].

La iteración Lanczos es un método utilizado para calcular el mayor autovalor y su autovector correspondiente en pocas iteraciones con gran precisión [3]. Además de aplicaciones como aproximaciones para procesamiento de señales o imágenes, es empleada para análisis de estructuras [4], electromagnetismo [5], entre otros, debido a la relación que guardan los autovalores y autovectores con los sistemas de control y ecuaciones lineales.

Sin embargo, cuando se requiera realizar múltiples pruebas, obtener más de un autovalor y autovector, entre otras opciones, es recomendable desarrollar implementaciones en algún lenguaje de programación en paralelo como OpenMP, Pthreads, MPI [6], SIMD [7], CUDA [8], etc. con la finalidad de reducir el tiempo de ejecución.

La arquitectura CUDA permitirá programar Graphic Processing Units (GPU) y así poder ejecutar códigos de forma paralela causando que el tiempo de ejecución total sea mucho menor respecto a una implementación serial [9]. Para esto se tendrá que diseñar un algoritmo paralelo seleccionando adecuadamente las etapas a paralelizar puesto que algunas no presentarán ventaja alguna, o serán dependiente entre ellas.

Capítulo 1

Problemática

1.1. Descripción y formulación del problema

El cálculo de autovalores y autovectores de una matriz es un problema de algebra lineal bastante sencillo que puede resolverse utilizando *Power Iterations*, *QR-methods*, *Krylov methods* [10], entre otros. En ocasiones no se requerirá calcular todos los autovalores, sino los n mayores (los más significativos) con sus respectivos autovectores, pudiendo recurrirse a procedimientos especializados como *power method* [3] o la iteración Lanczos [3], ahorrando tiempo al sólo calcular los autovalores y autovectores deseados.

Power method es una opción que converge lentamente; mientras la iteración Lanczos es una adaptación de este método para acelerar la convergencia, utilizando además, algoritmos posteriores a *power method*, como el algoritmo QR, el cual fue introducido en 1961 [11]. Debido a que *power method* no es el método más adecuado para el cálculo de autovalores no se ha usado mucho para investigaciones, entre otras para electromagnetismo [5] y filtrado de ruido [12]. Sin embargo se puede decir que forma parte de otros algoritmos más eficientes como en la misma iteración Lanczos [10] y es actualmente usado como base de un sistema por una empresa reconocida a nivel mundial como Google [13].

La iteración Lanczos es uno de los métodos más utilizados para el cálculo de autovalores y autovectores, realizándose investigaciones en diversos campos basadas en ella, como [14] y [15] en electromagnetismo computacional, como una parte de la implementación de *Finite-Difference Time-Domain Method*, *Finite-Element Frecuency-Domain Method* y *Finite-Element Time-Domain Method*. También se han realizado investigaciones orientadas a navegación robótica como [16] donde utilizan Lanczos para implementar filtros que actúan sobre el ruido de sensores de posición. Existen muchas otras áreas de aplicación, por lo que se debe destacar que es bastante utilizada en reducción de la dimensionalidad de datos y

compresión de datos, por ejemplo en [17], [18] y [19] , lo cual puede involucrar ser parte de otros problemas matemáticos como *Principal Component Analysis*.

Actualmente, la información a analizar suele ser bastante grande, lo que implica que la matriz que la representa será también de dimensiones grandes. Esto representa un problema ya que Lanczos demorará más tiempo en el cálculo, disminuyendo la efectividad de las aplicaciones que se le dan a la iteración o incluso haciendo inservibles algunas implementaciones de Lanczos para determinadas aplicaciones. Con el fin de evitar eso, se busca diseñar e implementar algoritmos paralelos de Lanczos.

1.2. Objetivo principal

- Implementar la iteración Lanczos en arquitectura CUDA de forma computacionalmente eficiente.

1.3. Objetivos secundarios

- Implementar la iteración Lanczos serial en el entorno de simulación MATLAB de forma computacionalmente eficiente.
- Implementar la iteración Lanczos serial en lenguaje C de forma computacionalmente eficiente.

1.4. Importancia y justificación del estudio

Los autovalores y autovectores son importantes debido a que representan patrones característicos de una matriz que no variarán frente a posibles transformaciones que puede sufrir permitiendo al analizarlos determinar soluciones a distintos problemas en los que participa la matriz. Si se tiene en cuenta que una matriz puede representar cualquier sistema de control, transformación geométrica, estructura, imagen, señal, entre otros, se podría analizar vibraciones en estructuras [4], implementar filtros para señales [20] [21], analizar sistemas electromagnéticos [22], comprimir información [17], entre otras aplicaciones.

Esta tesis busca implementar una forma de resolver el *eigenproblem*, uno de los más grandes campos de estudio en algebra lineal numérica [3]. Esto lo hará calculando el mayor autovalor y su respectivo autovector de una matriz inicial dada, mediante la implementación de la iteración Lanczos de forma computacionalmente eficiente. Se seleccionó este método debido a que es el más indicado cuando se requiere calcular sólo algunos de los autovalores y

autovectores, siendo generalmente utilizada para problemas que incluyen o están relacionados a aproximación de matrices, algunos de los cuales se mencionan a continuación:

- *Low Rank Matrix Approximation* [1]: El objetivo en este tipo de problemas es aproximar una matriz inicial a otra de menor rango, el cual será elegido [23]. Una forma de hacer esto es mediante Single Value Decomposition (SVD) [3], lo cual permitirá encontrar las matrices de valores y vectores singulares y modificar la matriz de valores singulares para disminuir el rango. SVD dará como resultado lo mismo que la iteración Lanczos siempre y cuando la matriz de análisis sea una matriz normal [2]. Esta aproximación puede ser usada para compresión de información [17], sistemas de información, entre otros, además de aplicaciones relacionadas a procesamiento de imágenes tales como denoising [24]. Además esta aproximación es usada como una parte de la solución de otros problemas matemáticos ya que disminuye el cálculo a realizar y permite un resultado suficientemente exacto.
- *Principal Component Analysis (PCA)*: Es una forma de análisis de información, generalmente está orientada a buscar patrones en información y así poder resaltar las diferencias, las cuales llama componentes principales, o mostrar las semejanzas entre la información [25]. Algunos usos que se le suele dar son el reconocimiento de rostros, video vigilancia, Latent Semantic Indexing [26] y reducción de la dimensionesionalidad de datos [17]. En la gran mayoría de formas de desarrollo de PCA, suele utilizarse Low-Rank Approximation, por ejemplo en [27].

Capítulo 2

Autovalores: concepto y métodos de estimación

2.1. Antecedentes

En la literatura actual, no se han encontrado publicaciones en las cuales se utilicen tarjetas de video Nvidia de última generación, las cuales cuentan con nuevo *hardware* que permite obtener mejores resultados computacionales que sus predecesoras.

Por otro lado, debido a la necesidad de resolver problemas como los explicados anteriormente, existen diversas librerías (*software*) que permiten el cálculo de autovalores y autovectores de matrices utilizando la iteración Lanczos. Estas librerías se encuentran escritas principalmente en lenguaje C (SLEPc [28], PRIMME [29], NAG [30], etc); una librería que utiliza la capacidad de paralelización de las tarjetas Nvidia basándose en lenguaje CUDA C es CULA [31].

Lamentablemente, la librería CULA no es *OpenSource* por lo que en este caso se optó por realizar las verificaciones del rendimiento de la implementación utilizando la implementación BLAS (*Basic Linear Algebra Sub-routines*) de Nvidia, la cual permite acercarse al rendimiento obtenido al utilizar CULA.

2.2. Autovalores y Autovectores

Dada una matriz cuadrada A se considerará un autovalor (λ) y autovector (v) de A, a todo escalar (real o imaginario) y vector (distinto de cero) respectivamente, que cumplan la siguiente relación:

$$A \times v = \lambda \times v \tag{2.1}$$

Considerando que la matriz es de $n \times n$ dimensiones, esta tendrá n autovalores con un autovector único por cada autovalor (asumiendo que todos los autovalores son distintos). Por definición, el cálculo de autovalores se realiza resolviendo (2.2), donde I es un matriz identidad; y posteriormente, los autovalores pueden ser calculados vía (2.3).

$$| \lambda \times I - A | = 0 \tag{2.2}$$

$$(\lambda \times I - A) \times v = 0 \tag{2.3}$$

2.3. *Power Method*

Este método calcula el mayor autovalor de una matriz así como su respectivo autovector [26]. Consiste en una serie de iteraciones basadas en multiplicaciones matriz-vector, las cuales harán converger un vector inicial aleatorio al mayor autovector de la matriz, lo cual a su vez permitirá obtener el mayor autovalor.

Algoritmo 1: Power Method

Entradas : A: matriz entrada

q_0: vector unitario aleatorio

Salidas : λ: autovalor

q: autovector

1: **for** $k = 1, 2, ...$ **do**

2: $\quad z_k = A \times q_{k-1}$

3: $\quad q_k = z_k / \parallel z_k \parallel$

4: $\quad \lambda_k = [q_k]^H \times A \times q_k$

Dado que el conjunto de autovectores es ortogonal, forman una base que permite expresar q_0 como una combinación lineal de ellos:

$$q_0 = a_1 \times v_1 + a_2 \times v_2 + \ldots + a_n \times v_n \tag{2.4}$$

Teniendo en cuenta (2.4), donde $a_{1...n}$ son constantes y $v_{1...n}$ son autovectores de A, se aprecia que la línea 2 del algoritmo se encarga de calcular (2.5), deduciéndose que a mayor valor de k se puede aproximar a una expresión en términos de sólo el mayor autovalor y su respectivo autovector, dado que $\left(\frac{\lambda_j}{\lambda_1}\right)^k$ tendrá un valor mucho menor a 1 pudiendo despreciarse.

$$
\begin{aligned}
A^k \times q_0 &= a_1 \times \lambda_1^k \times v_1 + a_2 \times \lambda_2^k \times v_2 + \ldots + a_n \times \lambda_n^k \times v_n \\
&= a_1 \times \lambda_1^k \times \left(v_1 + \sum_{j=2}^{n} \frac{a_j}{a_1} \times \left(\frac{\lambda_j}{\lambda_1} \right)^k \times v_j \right) \\
&\approx a_1 \times \lambda_1^k \times v_1
\end{aligned} \tag{2.5}
$$

La línea 3 del algoritmo se encarga de normalizar el vector calculado previamente, eliminando componentes como la enésima potencia del mayor autovalor, que se generan del paso anterior, obteniéndose el autovector como se aprecia en (2.6). La línea 4 deriva del reordenamiento de la expresión 2.1.

$$
z_k = \frac{A^k \times q_0}{\| A^k \times q_0 \|} \approx \frac{a_1 \times \lambda_1^k \times v_1}{\| a_1 \times \lambda_1^k \times v_1 \|} \approx \frac{a_1 \times \lambda_1^k \times v_1}{a_1 \times \lambda_1^k \times \| v_1 \|} \approx \frac{v_1}{\| v_1 \|} \tag{2.6}
$$

El gran problema de este método es que presenta convergencia lenta. Dado que la operación principal es la multiplicación matriz-vector, un gran número de iteraciones hará que el costo computacional del método sea grande, el cual crecerá mientras mayor sea el tamaño de la matriz.

2.4. Iteración Lanczos

Es un método que en su forma clásica permite calcular el mayor autovalor y autovector de una matriz cuadrada simétrica, sin embargo para otros casos existen variaciones de este método. La iteración consiste en reducir la matriz inicial mediante transformaciones ortogonales a una ”forma condensada”, en este caso será una forma tridiagonal simétrica. Esto con el fin de poder utilizar métodos especializados para encontrar la solución de la forma tridigonal, es decir, se calcula los autovalores y autovectores de la matriz tridiagonal, pues estos guardarán relación con los de la matriz original. Uno de los métodos por defecto a usar es la Descomposición Schur. Por último, se transformará los resultados de evaluar la matriz tridiagonal a la solución de la matriz original usando la matriz Lanczos.

La gran ventaja de Lanczos es que entrega información sobre el mayor autovalor y autovector mucho antes de terminar la tridigonalización. Lo anterior indica la correcta forma de operación de Lanczos: en la primera iteración se tridiagonaliza una parte de la matriz inicial y se le aplica descomposición Schur, si se lograse obtener el mayor autovalor y autovector, se detendrán las iteraciones, caso contrario, se tridiagonalizará un poco más y nuevamente se aplicará descomposición de Schur y así sucesivamente hasta encontrar el mayor autovalor y autovector. Por lo general, suelen necesitarse muy pocas iteraciones.

2.4.1. Tridiagonalización

Consiste en una transformación ortogonal aplicada a la matriz inicial, con el fin de poder calcular rápidamente el mayor autovalor y autovector de la forma tridiagonalizada, los cuales guardan relación con el par de la matriz inicial. La ecuación representativa es la siguiente:

$$T = Q^T \times A \times Q \tag{2.7}$$

Donde A es la matriz inicial, Q es una matriz ortonormal denominada matriz Lanczos y T es la matriz tridiagonal. Como se aprecia en (2.7), T y A son matrices similares, de esta forma sus autovalores serán iguales, así como existirá una realción entre sus autovectores:

$$v_A = Q \times v_T \tag{2.8}$$

Existen muchas formas de tridiagonalizar una matriz, como el uso de rotaciones de Givens [32], transformación de Householder [33], entre otros, sin embargo, estos resultarían poco útiles en el caso de matrices *sparse*. Es por eso que se plantea la tridiagonalización partiendo de:

$$Q \times T = A \times Q \tag{2.9}$$

Asumiendo que Q puede ser representada por vectores columnas:

$$Q = [q_1, q_2, q_3, ..., q_n] \tag{2.10}$$

y que T presenta la siguiente forma:

$$T = \begin{pmatrix} \alpha_1 & \beta_1 & & \cdots & 0 \\ \beta_1 & \alpha_2 & \ddots & & \vdots \\ & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \beta_{n-1} \\ 0 & \cdots & & \beta_{n-1} & \alpha_n \end{pmatrix}$$

Figura 2.1: Matriz Tridiagonal Simétrica

Se obtendrá el algoritmo base de la tridiagonalización [3], en el cual se observa cierta similitud con $PowerMethod$, debido a que Lanczos es una adaptación de ese método para aprovechar los vectores que se van generando en las iteraciones (vectores Lanczos) y lograr una convergencia más rápida y precisa.

Algoritmo 2: Tridiagonalización. [3]

Entradas : A: matriz entrada

q_1: vector unitario aleatorio

Salidas : $\alpha_{1,2,...}$: elementos de la diagonal

$\beta_{1,2,...}$: elementos de subdiagonal superior e inferior

$q_{1,2,...}$: vectores Lanczos

Requisitos: $r_0 = q_1, \beta_0 = 1, q_0 = 0, k = 0$

1: **while** $\beta_k \neq 0$ **do**

2: $q_{k+1} = r_k/\beta_k$

3: $k = k + 1$

4: $\alpha_k = q_k^T \times A \times q_k$

5: $r_k = (A - \alpha_k \times I) \times q_k - \beta_{k-1} \times q_{k-1}$

6: $\beta_k = \|r_k\|_2$

Analizándose (2.9) con la estructura indicada anteriormente se obtiene (2.11), de lo cual se puede establecer un patrón cuando $k = 1, 2, \ldots, n-1$ mostrado en (2.12). El lado derecho de la expresión muestra la línea 5 del algoritmo; mientras que, con el lado izquierdo se establece la relación $r_k = \beta_k \times q_{k+1}$ la cual al reordenarse se convierte en la línea 2.

$$[Aq_1, Aq_2, \ldots, Aq_n] = [\alpha_1 q_1 + \beta_1 q_2, \beta_1 q_1 + \alpha_2 q_2 + \beta_2 q_3, \ldots, \beta_{n-1} q_{n-1} + \alpha_n q_n] \quad (2.11)$$

$$\beta_k \times q_{k+1} = A \times q_k - \alpha_k \times q_k - \beta_{k-1} \times q_{k-1} \quad (2.12)$$

A partir de (2.12) se pueden deducir deducir las demás líneas del algoritmo, aprovechando las propiedades de ortogonalidad de los vectores q. Considerando que $r_k = \beta_k \times q_{k+1}$, se tiene que $\| r_k \|_2 = \beta_k$ (línea 6). Al multiplicar (2.12) por q_k^T se obtiene la línea 4 del algoritmo:

$$\begin{aligned} q_k^T \times \beta_k \times q_{k+1} + q_k^T \times \alpha_k \times q_k &= q_k^T \times A \times q_k - q_k^T \times \beta_{k-1} \times q_{k-1} \\ \alpha_k &= q_k^T \times A \times q_k \end{aligned} \quad (2.13)$$

2.5. Descomposición Schur

La descomposición Schur indica que toda matriz cuadrada T puede ser representada de la siguiente forma:

$$T = Z \times D \times Z^T \quad (2.14)$$

Donde D es una matriz triangular superior (ver figura 2.2) y Z es una matriz ortonormal. La principal característica de esta descomposición es que los componentes de la diagonal de D son

autovalores de T. Además, si T fuese una matriz normal, D sería una matriz diagonal, así como Z contendría los autovectores de T [34]. Por último, si T es una matriz positiva-definida o una matriz normal, la descomposición Schur de esta, sería equivalente a su Descomposición en Valores Singulares (SVD)[2].

$$\begin{pmatrix} \lambda_1 & e_{12} & e_{13} & \cdots & e_{1n} \\ 0 & \lambda_2 & e_{23} & \cdots & e_{2n} \\ 0 & 0 & \lambda_3 & \cdots & e_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_n \end{pmatrix} \quad \begin{pmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ 0 & 0 & \lambda_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

(a) (b)

Figura 2.2: Resultdo de la descomposición Schur donde: (a) Matriz E estándar , (b) Matriz E cuando A es una matriz normal.

2.5.1. QR implícito con Wilkinson Shift

El algoritmo 3 permite obtener una aproximación de la descomposición de Schur de matrices simétricas, esta variación se enfoca en matrices tridiagonales. Consiste en 3 bloques:

- El primer bloque (entre la líneas 2 y 8) calcula y aplica el *Wilkinson shift* (s), el cual permite lograr una convergencia más rápida.
- El segundo bloque (desde la línea 9 hasta la 26), se encarga de calcular la descomposición QR implícita, operando solo sobre la regíon que aún es tridiagonal. Esta es realizada mediante rotaciones de Givens [32] aplicadas simétricamente a la matriz tridiagonal, con el objetivo de minimizar los elementos de las subdiagonales en cada iteración. Entre las lineas 10 y 13, se calcula la matriz de rotación. Entre las líneas 14 y 25 se aplica la rotación simétrica, mediante unas cuantas operaciones escalares debido a que para el caso de matrices tridiagonales sólo se ven afectados pocos elementos de la matriz; por el contrario, en un matriz densa se verían afectados todos los elementos de dos filas y dos columnas de la matriz a transformar por iteración. Además se calculan los elementos que permitirán calcular la siguiente matriz de rotación (x e y) de la siguiente iteración. Finalmente, en la línea 26, se actualiza la matriz de transformación acumulada (Z).
- Por último, el tercer bloque (líneas 27 y 28) discrimina los componentes de la tridiagonal (α y β) teniendo en cuenta una tolerancia con el objetivo de no seguir operando sobre las regiones que ya han sido diagonalizadas.

Algoritmo 3: QR implícito con Wilkinson Shift. [3]

Entradas : α: elementos de la diagonal

β: elementos de las subdiagonales

Salidas : α: Diagonal de Autovalores

Z: Matriz de Transformación de Schur

Requisitos: $m = Num.\ de\ elementos\ de\ \alpha$

while $m > 1$ **do**
- $d = (\alpha_{m-1} - \alpha_m)/2$
- **if** $d = 0$ **then**
  - $s = \alpha_m - \mid \beta_{m-1} \mid$
- **else**
  - $s = \alpha_m - \beta_{m-1}^2/(d + sign(d) \times \sqrt{d^2 + \beta_{m-1}^2})$
- $x = \alpha_1 - s$
- $y = \beta_1$
- **for** $k = 1$ **to** $m - 1$ **do**
  - **if** $m > 2$ **then**
    - $[c, s] = givens(x, y)$
  - **else**
    - $Determinar\ [c, s]\ tal\ que \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \times \begin{bmatrix} \alpha_1 & \beta_1 \\ \beta_1 & \alpha_2 \end{bmatrix} \times \begin{bmatrix} c & s \\ -s & c \end{bmatrix}\ diagonal$
  - $w = c \times x - s \times y$
  - $d = \alpha_k - \alpha_{k+1}$
  - $z = (2 \times c \times \beta_k + d \times s) \times s$
  - $\alpha_k = \alpha_k - z$
  - $\alpha_{k+1} = \alpha_{k+1} + z$
  - $\beta_k = d \times c \times s + (c^2 - s^2) \times \beta_k$
  - $x = \beta_k$
  - **if** $k > 1$ **then**
    - $\beta_{k-1} = w$
  - **if** $k < m - 1$ **then**
    - $y = -s \times \beta_{k+2}$
    - $\beta_{k+1} = c \times \beta_{k+1}$
  - $Z_{1:N,k:k+1} = Z_{1:N,k:k+1} \times \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$
- **if** $\mid \beta_{m-1} \mid < \epsilon \times (\mid \alpha_{m-1} \mid + \mid \alpha_m \mid)$ **then**
  - $m = m - 1$

2.6. Condición de Parada de la Iteración Lanczos

Ya que se obtinene el mayor autovalor y autovector de la matriz inicial mucho antes de terminar la tridiagonalización, se requerirá una forma de evaluar si los resultados de la iteración de turno son lo sufientemente precisos. Mediante el cálculo del error relativo de la expresión en (2.1) se podrá determinar si se necesita o no seguir iterando. Cabe resaltar, que para calcular el error es necesario calcular previamente el mayor autovalor y autovector corresponiente de los resultados obtenidos en la iteración de turno, como se aprecia en Algoritmo 4.

Por semjanza, A y T tienen igual autovalores, contenidos en D; mientras que sus autovalores guardan la relación descrita en (2.8). Debido a esto, en la línea 1 del algoritmo calcula el mayor autovalor de A y el índice del orden en el que se encuentra. En la línea 2 se calcula el autovector correspondiente al autovalor seleccionado anteriormente. Finalmente, en la última línea se calcula el error relativo, verificando si son valores adecuados.

Algoritmo 4: Cálculo de Error.

Entradas : A: Matriz entrada
Q: Matriz Lanczos
D: Diagonal de Schur
Z: Matriz de Transformación Schur

Salidas : err: Error de la Iteración Actual
$eigVa$: Mayor Autovalor
$eigVe$: Autovector correspondiente a $eigVa$

1: $[eigVa, indice] = max(D)$

2: $eigVe = Q \times Z_{(:,indice)}$

3: $err = \dfrac{\|A \times eigVe - eigVa \times eigVe\|}{\|eigVa \times eigVe\|}$

2.7. Variaciones de Lanczos

Existen variaciones del método tradicional de Lanczos, los cuales son utilizados para ajustarse a diversos tipos de problemas o para brindar alguna mejora como solucionar el problema de perdida de ortogonalidad. Algunas de las variaciones más conocidas son:

- Lanczos no-simétrico [3]: Utilizado para matrices de entrada no simétricas. Fue propuesto por Cornelius Lanczos junto al método tradicional, sin embargo, no es muy utilizado actualmente debido a mejores métodos.

- Golub-Kahan-Lanczos [35]: Se diferencia en que utiliza bidiagonalización en lugar de tridiagonalización.Utilizado comunmente para implementar la Descomposición en Valores Singulares (SVD).
- Lanczos en aritmética exacta [36]: Utilizado para solucionar algunos problemas de ortogonalidad en el método tradicional.

Capítulo 3

Arquitectura CUDA

Desde que se creó la computadora se han buscado diversos medios para lograr que sean mas rápidas. La primera idea que se tuvo fue aumentar la frecuencia del procesador [31]; sin embargo, esto involucraba calentamiento de las tarjetas, mayores dimensiones para ventiladores, etc. haciendo poco viable esta opción a partir de cierto punto. Es por eso que surge la idea de agregar más núcleos a los procesadores, repartiendo el trabajo entre todos, así no se necesitaría incrementar tanto la frecuencia de trabajo. En la figura 3.1 se aprecia una comparación de velocidades entre modelos de GPU y CPU a lo largo de los últimos años.

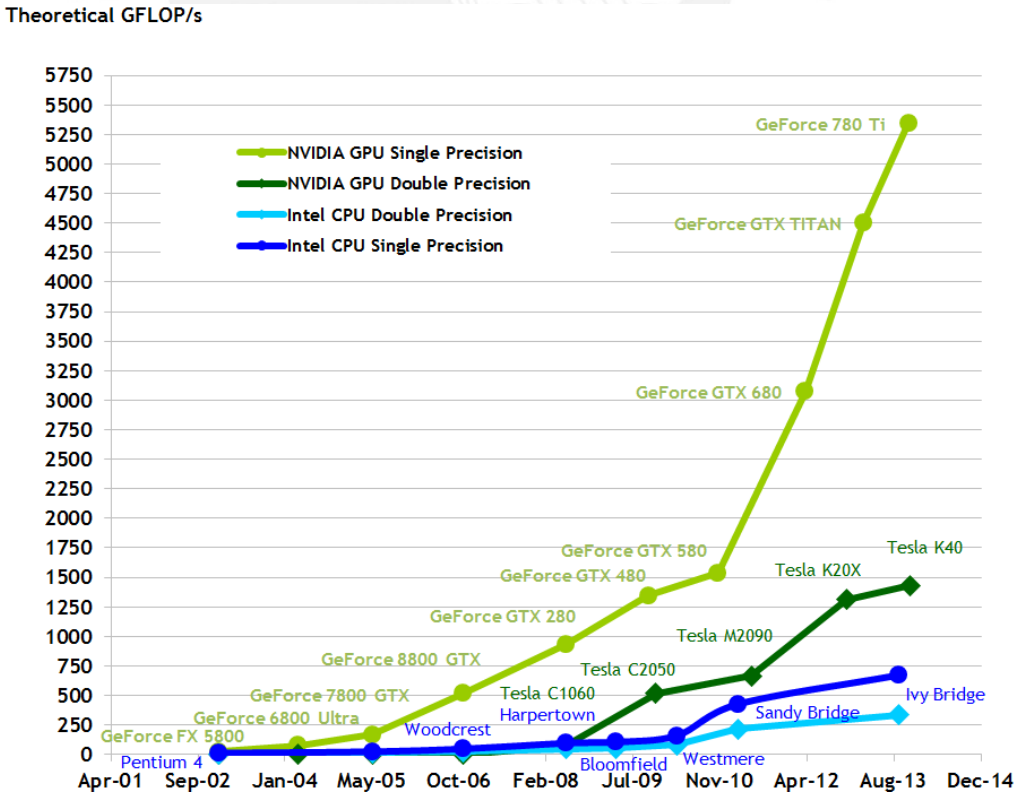


Figura 3.1: Comparación de GFLOPS/s en los últimos años [37].

3.1. Descripción

CUDA o Compute Unified Device Architecture permite utilizar un GPU (especializado en imágenes) como si fuera un procesador más, es decir, sería un GPGPU (General Purpose Graphics Processing Unit). La gran ventaja es que las tarjetas CUDA tienen multiprocesadores que cuentan con 32, 64 o más núcleos dependiendo del modelo, permitiendo procesar datos en un tiempo muy corto. En la figura 3.2 se observa la diferencia en cantidad de núcleos entre un CPU y un GPU.

Las diferencias entre diversas tarjetas CUDA suelen ser el número de núcleos, *streaming processors*, capacidad de memorias, etc. Sin embargo, cada tarjeta posee una capacidad de computo (*compute capability*) la cual indica características que puede soportar el hardware del GPU, tales como utilizar memoria unificada, operaciones atómicas a diferente nivel de memorias, paralelismo dinámico, entre otros. A la fecha se puede encontrar capacidad de computo $1.x$ (micro-arquitectura Tesla), capacidad de computo $2.x$ (micro-arquitectura Fermi), capacidad de computo $3.x$ (micro-arquitectura Kepler) y capacidad de computo $5.x$ (micro-arquitectura Maxwell) y se ha anunciado la siguiente generación (micro-arquitectura Volta) [37].

En cuanto a la forma de operación en esta arquitectura se debe conocer que una sola instrucción es realizada por todos los threads, de los cuales se hablará en la siguiente sección, esto corresponde con lo que Nvidia llama modelo SIMT (*Single Instruction Multiple Threads*). Es así que la forma de programación convencional (serial) varía, ya que requiere diseñar los algoritmos de forma paralela para que se adapten a la modelo SIMT. Otra ventaja de CUDA es que brinda las herramientas para programar en lenguaje C entre otros.

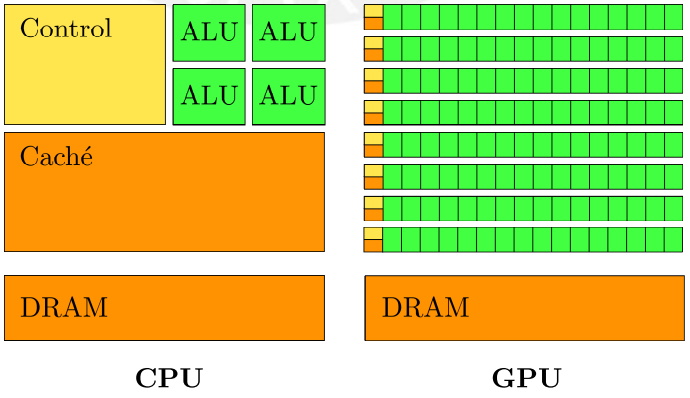


Figura 3.2: Comparación entre CPU y GPU [37].

Por último cabe señalar que la ejecución de programas se realiza tanto en el CPU (*host*) como en el GPU (*device*), pues hay partes de algoritmos que no son posible paralelizar y la

invocación de ejecuciones paralelas se realiza desde el CPU [37]. Lo anterior, indica que información es pasada desde el CPU al GPU, se procesa y luego retorna del GPU al CPU hasta que se requiera alguna otra operación paralela.

3.2. Kernel y Jerarquía de threads

Es importante conocer la forma en que los threads se agrupan pues es necesario al momento de invocar un kernel, el cual es simplemente una función que se ejecuta en el GPU. En la figura 3.3 se observa gráficamente las tres unidades de la jerarquía.

- *Grid*: Es un conjunto de bloques, cuyo número se especifica cuando se llama un kernel. La grilla suele ser de dos dimensiones, sin embargo puede ser de mas teniendo en cuenta que existan los suficientes threads en el dispositivo. Se referencia a los bloques según el número de dimensiones de la grilla.
- *Block*: Es una agrupación de threads. En arquitectura CUDA, al momento de llamar un kernel se especifica las dimensiones del bloque, es decir la cantidad de hilos que tendrá; sin embargo, generalmente el límite es 512 o 1024 threads por bloque [4]. Cada bloque puede tener información distinta a procesar, pero siempre ejecutarán el mismo procedimiento.
- *Thread*: Es la unidad de procesamiento paralelo. En arquitectura CUDA, existen miles de threads, el número exacto depende del modelo de la tarjeta nVidia. Se caracterizan por ejecutar la misma instrucción a la vez (modelo SIMT) y agruparse en dos dimensiones.
- *Warp*: Es el conjunto de 32 threads. Los multiprocesadores crean, gestionan, planifican y ejecutan warps.

3.3. Tipos de Memoria en Device

Un dispositivo CUDA posee diferentes tipos de memoria [37] de las cuales se tienen que escoger las más convenientes para el trabajo que se busca realizar, pues cada una tiene diferentes características. Una adecuada selección permitirá realizar implementaciones computacionalmente eficientes. En la figura 3.4 se muestra la distribución de memorias en el dispositivo.

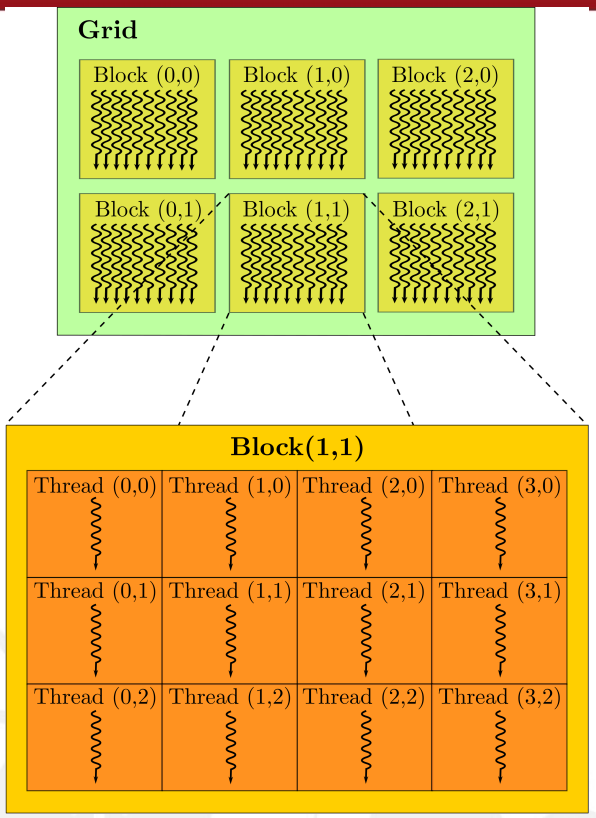


Figura 3.3: Jerarquía de Memorias [37].

- Memoria Local: Espacio de memoria personal para cada thread. Se encuentra en la memoria del dispositivo (no en el mismo chip GPU) por lo que no se tiene un rápido acceso a ella.

- Memoria Compartida: Espacio de memoria compartido por todos los threads de un mismo bloque. Debido a que se encuentra en el chip (GPU) el acceso a esta memoria tendrá muy baja latencia (aprox. 11 ciclos de reloj para *compute capability* $3.x$ y 22 ciclos de reloj para *compute capability* $1.x/2.x$) y gran ancho de banda. La mayor desventaja es que no se puede almacenar tanta información como en otras memorias, por lo que para el uso de esta suele segmentarse la información a procesar y trabajar por partes. Es una especie de memoria cache L1 [9] a la cual tienen acceso todos los threads de un mismo bloque.

- Memoria Global: Es la memoria más grande en el dispositivo y a la vez, presenta un tiempo de acceso mayor a las demás (entre $200 - 400$ ciclos de reloj para *compute capability* $3.x$ y $400 - 800$ ciclos de reloj para *compute capability* $1.x/2.x$). Suele utilizarse junto a memoria compartida para compensar mutuamente las desventajas de cada una. Todos los threads tienen acceso a ella.

- Memoria Constante: Es una memoria de sólo lectura, pues no puede ser modificada durante la ejecución. Una de sus limitaciones es el tamaño, que suele estar limitado a 64KB. Al estar ubicada en el dispositivo no se tiene un rápido acceso a ella. El usar esta memoria en lugar de la global puede reducir el ancho de banda de memoria requerido, esto es debido a que es cacheada en el chip.

- Memoria Textura: Es una memoria de sólo lectura localizada en el dispositivo. Se considera una memoria especializada pues tiene un patrón de acceso a datos distinto a la memoria constante y global, siendo la mejor opción para trabajar con información 2D. El motivo por el que se tiene un mejor desempeño respeto de la memoria global, es que al igual que la memoria constante es cacheada en el chip [8]. La latencia de accesos que presenta es similar a la de memoria compartida.

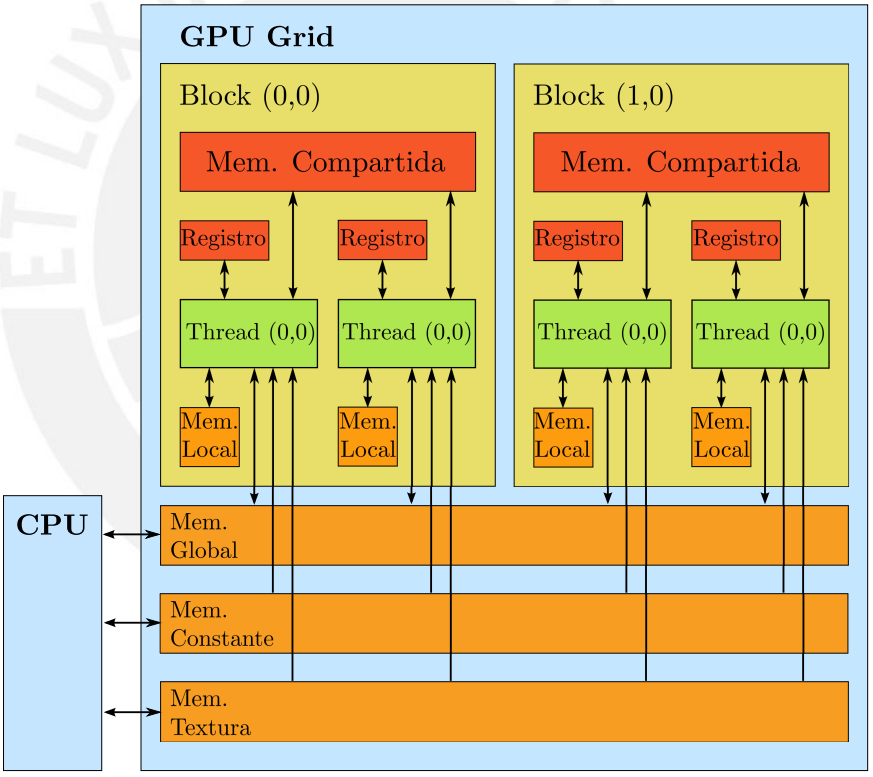


Figura 3.4: Distribución de Memorias en Device [37].

3.4. Tipos de Memoria en Host

En sus inicios, la forma de almacenar data en el host era equivalente a realizarla como si se tratase de una implementación puramente serial (memoria paginable). Sin embargo, debido a la continua evolución de la tecnología CUDA, actualmente es posible almacenar data en el host de formas distintas, siendo las principales:

- Memoria no-paginable: Conocida también como *Pinned Memory*. En algunos dispositivos, permite realizar copias entre la memoria host y la memoria del dispositivo a la vez que se ejecuta kernels, para sacar mayor provecho de esto, suelen usarse transferencia asincronas. Por defecto, la memoria no-paginable es asignada a las memorias caché de host, sin embargo, presenta una opción (*Write-Combining Memory*) que permite dejar libre las caché y a su vez permite realizar transferencias a través del bus *PCI Express* sin interrupciones, pudiendo llegar a ser un $40\,\%$ más rápidas.

- Memoria Unificada: Esta memoria permite maximizar la velocidad de acceso a datos realizando migraciones transparentes hacia el procesador que va a utilizarlos. Esto involucra que el sistema trataá de colocar la data donde pueda ser accesada de forma más eficiente. A su vez, esto eliminará la necesidad de realizar transferencias explícitas de datos. En la Figura 3.5 se observa como cambia la perspectiva del desarrollador en cuanto al almacenamiento de datos al utilizar esta memoria.

  Esta memoria migra data automaticamente hacia el procesador que la va a usar. El sistema tratará de colocar la data donde pueda ser accesada de forma más eficiente sin violar la coherencia. Esto conlleva que no sea necesario realizar transferencias explícitas de datos.

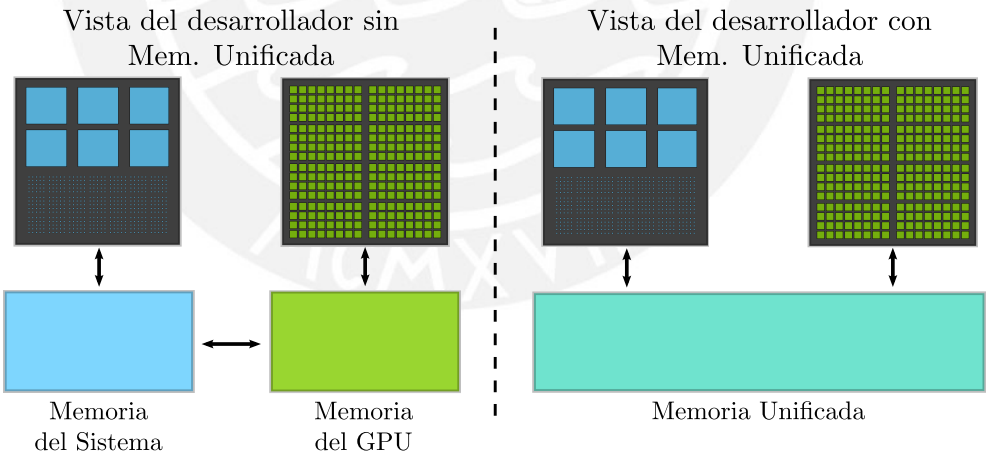


Figura 3.5: Interacción entre CPU y GPU con y sin Memoria Unificada [38].

Capítulo 4

Diseño del algoritmo propuesto

4.1. Consideraciones de diseño

Las matrices a utilizarse para la forma clásica de Lanczos deben ser cuadradas simétricas. Esto permitirá que sean diagonalizables por una matriz unitaria (Descomposición de Schur) y se obtenga un resultado igual al de SVD. Además, serán positivas definidas con el fin de obtener autovalores positivos reales.

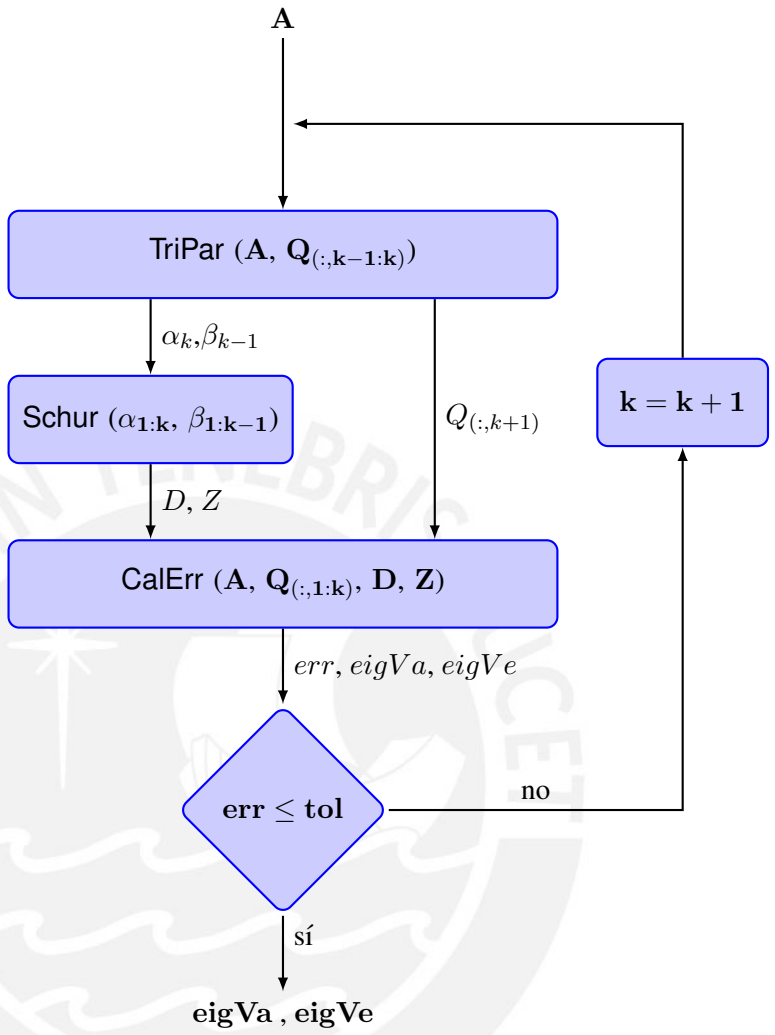
4.2. Diagrama de Flujo

En la figura 4.1 se presenta el diagrama de flujo de la iteración Lanczos. La Matriz de entrada A entra a un bucle con tres procedimientos principales:

- El primer procedimiento tridiagonalizará parcialmente la matriz A. Para esto, en cada iteración (k) este bloque generará dos escalares (α_k y β_{k-1}), los cuales al agruparse con los escalares calculados por este mismo procedimiento en iteraciones anteriores, conformarán la diagonal y subdiagonales (superior e inferior) de la matriz tridiagonal simétrica parcial. Durante este procedimiento también se calculará el siguiente vector que pasará a formar parte de la matriz Lanczos (Q).

- El segundo procedimiento calculará la descomposición Schur de la matriz tridiagonal calculada hasta el momento para obtener la matriz diagonalizada (D) y la matriz de transformaciín Schur (Z).

- El último procedimiento calculará el error de la iteración así como los posibles autovalor ($eigVa$) y autovector ($eigVe$). Luego, evaluándose el error se decidirá si es necesario seguir iterando o si el autovalor y autovector llegaron a converger a los valores adecuados.



Variables

A	: Matriz de Entrada
Q	: Matriz Lanczos
k	: Iteración Actual
α	: elementos de diagonal
β	: elementos subdiagonales
D	: Diagonal de Schur
Z	: Matriz de Transformación Schur
err	: Error de Iteración actual
tol	: Tolerancia de Error
eigVa	: Mayor Autovalor de A
eigVe	: Autovector correspondiente a $eigVa$

procedimientos

TriPar	: Tridiagonalización Parcial
Schur	: Descomposición Schur
CalErr	: Cálculo de Error

Figura 4.1: Diagrama de Flujo de la Iteración Lanczos.

4.3. Identificación de Operaciones a Paralelizar

La iteración Lanczos consta de tres procesos principales: la tridiagonalización parcial, la descomposición Schur y el cálculo del error.

4.3.1. Análisis de la Tridiagonalización Parcial

Analizando el Algoritmo 2 se aprecia que es un proceso secuencial pues cada línea depende de la anterior, haciendo que no se pueda reordenar el algoritmo para paralelizarlo. Sin embargo, presenta operaciones vectoriales y matriciales las cuales sí pueden paralelizarse. Reordenando ligeramente la línea 5 como $r_k = A \times q_k - \alpha_k \times q_k - \beta_{k-1} \times q_{k-1}$, se tendrá que las operaciones a realizarse por cada iteración son:

- 1 multiplicación matriz-vector (línea 4). El resultado calculado en la línea 4 puede ser reutilizado para la línea 5, luego de reordenarla.
- 2 producto punto (líneas 4 y 6). La norma de un vector se puede calcular como la raíz cuadrada del producto punto de dos vectores idénticos a él.
- 2 resta de vectores (línea 5), luego del reordenamiento de la línea 5.

4.3.2. Análisis de la Descomposición Schur

Analizando el Algoritmo 3 se encuentra que todos los pasos a seguir son operaciones que involucran solamente a escalares, a excepción de la multiplicación matriz-matriz en la línea 26. Sin embargo, se debe notar 2 características de esta operación:

- En la iteración Lanczos la cantidad de iteraciones que le toma al mayor autovalor y su correspondiente autovector para converger suele ser alrededor de 10, el cual sería el valor máximo que tomaría N (requisito del algoritmo).
- La parte de matriz Z a utilizar tiene dimensiones $N \times 2$; mientras que la matriz de rotación Givens es de 2×2.

Tomando en cuenta lo anterior se sostiene que no es conveniente paralelizar esta operación pues las dimensiones de las matrices son demasiado pequeñas no aprovechándose las capacidades de un GPU. Por lo tanto, se opta por tener una implementación serial en lenguaje C para la Descomposición Schur.

4.3.3. Análisis del Cálculo de Error

Analizando el Algoritmo 4 se puede reconocer que sólo la línea 3 contiene operaciones que pueden ser paralelizadas reduciendo su tiempo de ejecución respecto a una implementación serial. Considerando que el número de iteraciones necesarias para la convergencia esta representada por k (alrededor de 10), se puede describir lo siguiente:

- En la línea 1 no se necesita paralelizar algún algoritmo de ordenamiento para calcular el máximo valor, pues D es un vector de máximo k elementos y con dimensiones pequeñas es recomendable una implementación serial.
- En la línea 2 se encuentra una multiplicación matriz-vector. Sin embargo, las dimensiones de la matriz Q ($N \times k$, donde N es la cantidad de filas de A) y las del vector,hacen que no resulte lo suficientemente beneficioso paralelizarla.
- En la línea 3 se encuentran: 1 multiplicación matriz-vector, 1 resta de vectores y 2 producto punto (norma euclidiana). La cantidad de datos es lo suficientemente grande para obtener mejoras en cuanto a tiempo de ejecución.

4.4. Multiplicación Matriz-Vector

La multiplicacón matriz-vector se basa en [39]. En cualquier implementación típica de multiplicación matriz-vector, como la que se muestra en Listing 4.1, los accesos a memoria serán de la siguiente manera:

- Por cada elemento elemento de la matriz A se requerirá 1 acceso a memoria, es decir $m \times n$ accesos según Listing 4.1.
- Por cada elemento elemento del vector x se requerirá m accesos a memoria, es decir $m \times n$ accesos según Listing 4.1.

Al implementar esta operación en arquitectura CUDA, se buscaría que el acceso a datos reusables (el caso de x) tenga una latencia cercana a la de los registros (coste cero), utilizándose para ello, memoria compartida. Esto permite acelerar la operación ya que como se mencionó en el capítulo anterior, la latencia de acceso a memoria global puede ser mayor a 20 veces la latencia de acceso a memoria compartida.

Sin embargo, aún si x puede ser almacenado enteramente en memoria compartida. Se debe considerar que esta tiene una capacidad limitada. Es por eso que se utilizará la técnica *tiling*, la

cual consiste en dividir la data para procesarse en varios bloques a la vez, asegurándose que la cantidad de información a guardarse en memoria compartida de cada bloque sea la adecuada.

```
1  // y = Ax
2  // A : matriz de 'm' x 'n' elementos, x : vector de 'n' elementos
3  // y : vector de m elementos
4
5  void mv(float *y, float *A, float *x, int m, int n) {
6      for (int i = 0; i < m; i++) {
7          y[i] = 0;
8          for (int j = 0; j < n; j++)
9              y[i] += A[i * n + j] * x[j];
10         }
11 }
```

Listing 4.1: Multiplicación Matriz-Vector en lenguaje C

En la figura 4.2 se muestra gráficamente la forma de trabajo del método de Fujimoto. El método consiste en dividir la matriz inicial en bloques que serán operados por partes dentro de un bucle que permite desplazarse horizontalmente en la matriz (y en el vector). Esto es debido a que si se operasen todos los bloques de forma paralela (sin necesidad del bucle), por características de la arquitectura CUDA (los threads solo pueden acceder a la memoria compartida de su bloque), no se podrían reunir los resultados parciales, sin recurrir a alguna alternativa que ocupe demasiada memoria o conlleve mayor tiempo de ejecución.

Inicialmente se guardará en memoria compartida n elementos del vector x (cada memoria compartida tendrá la misma información) los cuales serán operados junto a las primeras n columnas de la matriz A de forma parlalela en N bloques de dimensiones $m \times n$. La operación que se llevará a cabo en cada bloque consiste en multiplicar elemento a elemento cada fila de las sub-matrices de A en los bloques por el fragmento del vector x en memoria compartida obteniéndose una sub-matriz de resultados parciales por bloque. Las multiplicaciones dentro de un bloque serán realizadas en paralelo dado que en cada bloque habrá $m \times n$ *threads*. El proceso descrito anteriormente se repetirá hasta haber operado con todos los elementos de A y x, debiendo notarse lo siguiente:

- En cada nueva iteración los siguientes n elementos del vector x serán guardados en memoria compartida.
- En cada nueva iteración las siguientes n columnas de la matriz A serán operadas en los bloques.

- La matriz de resultados parciales obtenida en cada iteración será sumada con la matriz de resultados acumulados, la cual será el resultado del bucle.

Al terminar el bucle se tendrá una sub-matriz de resultados finales en la memoria compartida de cada bloque. Se aplicarán reducciones por filas de forma paralela en cada bloque, logrando obtener la suma de los elementos de cada fila. El vector resultante será el resultado final de la operación matriz-vector.

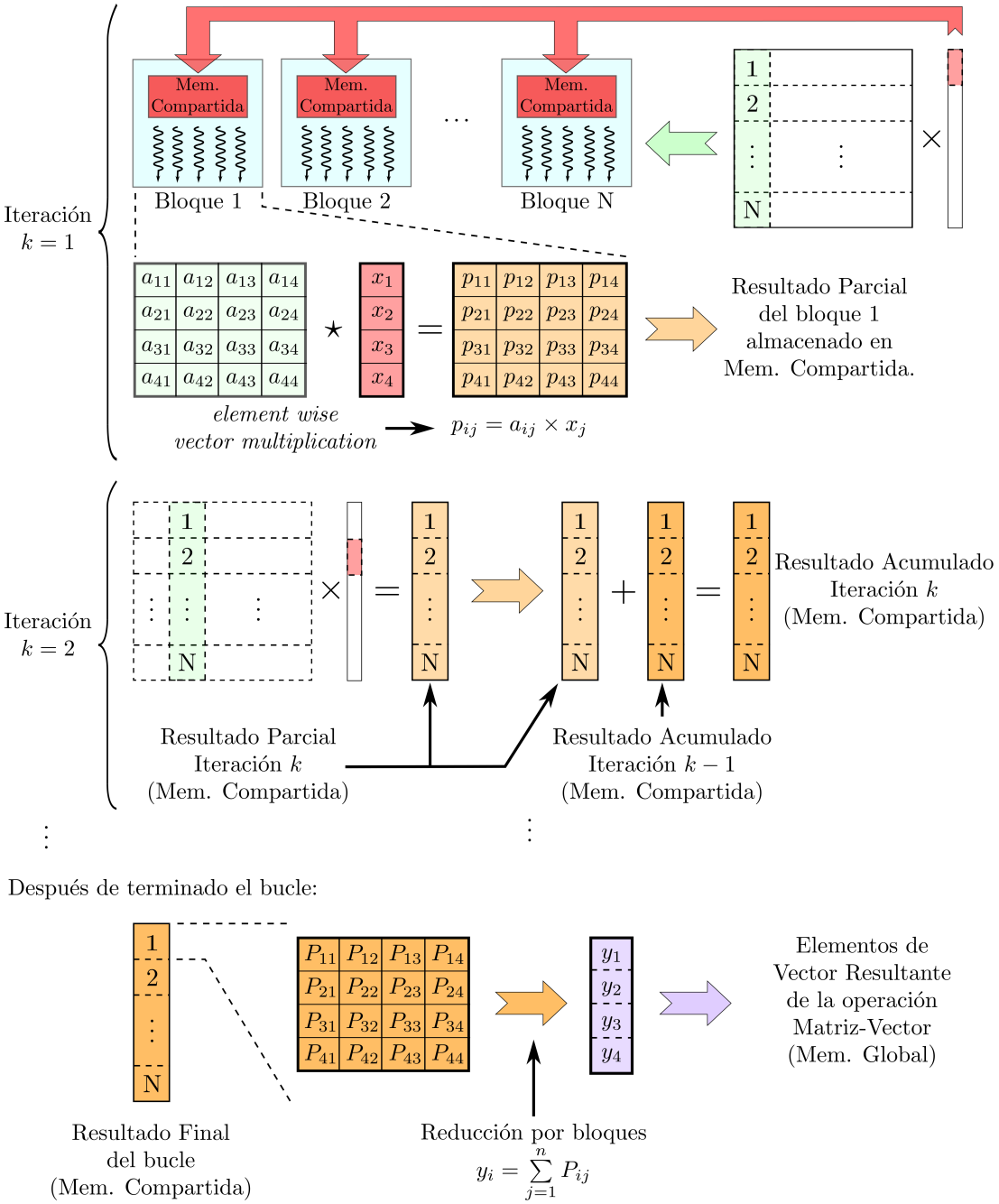


Figura 4.2: Descripción gráfica del método de Fujimoto para bloques de 4×4

4.5. Resta de Vectores

Este tipo de operación no representa gran dificultad en CUDA; sin embargo, cabe esperarse que se necesite cierto tamaño de vectores, para que sea recomendable utilizar la operación paralela, esto se analizará en el siguiente capítulo. En la figura 4.3 se observa que para este tipo de operación se requiere dividir los elementos de los vectores a restar entre los bloques de la grilla. En los bloques, que operarán en forma paralela, se realizará la resta de los sub-vectores para obtener una parte del vector final. Cada thread de cada bloque calculará un elemento del vector final.

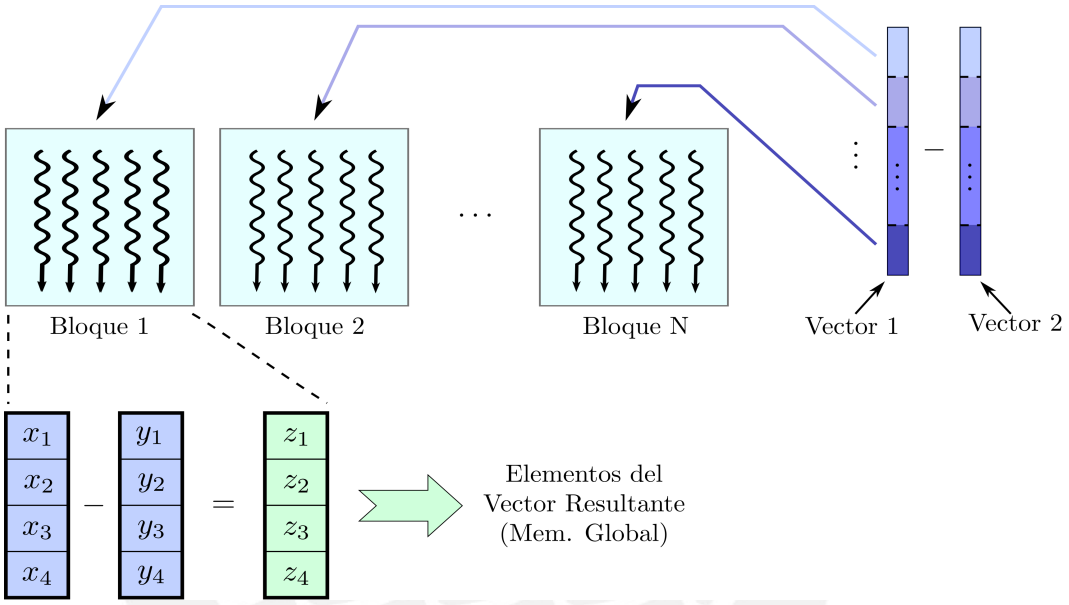


Figura 4.3: Descripción gráfica de la operación resta de vectores en CUDA

4.6. Producto Punto de Vectores / Norma Euclidiana de un vector

Dado que la norma euclidiana de un vector puede ser calculada como la raíz cuadrada del producto punto entre el vector y otro idéntico a él, ambas operaciones pueden ser tratadas de la misma forma en CUDA, ya que la raíz cuadrada es un proceso que debe realizarse en el host.

Por años la mejor forma de implementar producto punto, fue mediante reducciones en memoria compartida. Sin embargo, a partir de la micro-arquitectura Kepler (*compute capability* $3.x$), se cuenta con la instrucción *shuffle* y algunas variaciones [37], que permiten compartir información directamente entre threads de un mismo warp. Entonces es posible implementar reducciones basadas en esta instrucción, las cuales presentarán ventajas frente al anterior método, tales como:

- No se utilizará memoria compartida, pudiendo utilizarla para otra data.

- Sólo requerirá una intrucción frente a tres que utiliza memoria compartida (escribir, sincronizar y leer).
- Ya que la instrucción opera a nivel de warps, no se necesitará sincronizar.

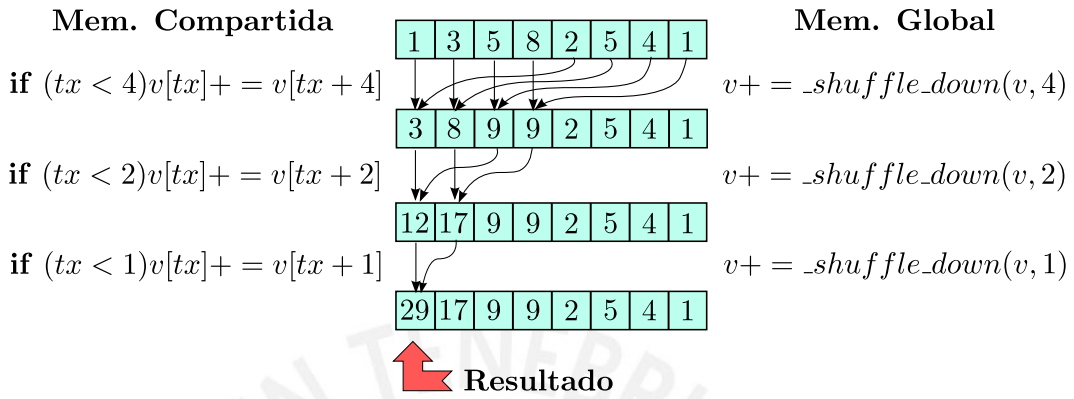


Figura 4.4: Operación de reducciones en memoria compartida y reducciones con shuffle.

En la Figura 4.4 se muestra la forma de operación de ambos métodos. Una vez calculados los resultados parciales con la instrucción $shuffle$, se puede optar por llevarlos al Host donde se realizará la última sumatoria o calcularla en Device utilizando funciones atómicas (para el caso *AtomicAdd*) [37]. Si bien este tipo de funciones siempre fue costosa porque involucran una serialización en *device*, en Kepler son lo suficientemente rápidas para equipararse a la opción de llevar la data a Host.

Por otro lado, se ha considerado limitar la cantidad de bloques máxima y que sea múltiplo de la cantidad de multiprocesadores, de esta forma se reutilizarán threads para los cálculos. Para esto se utiliza un bucle que en la primera iteración abarcará una cantidad k de elementos equivalente a la cantidad de threads por bloque multiplicada por el número de bloques máximo; en la segunda iteración, abarcará los siguientes k elementos, y así sucesivamente, hasta haber abarcado todos. Esto a su vez permitirá que se utilizen menos operaciones atómicas.

Capítulo 5

Implementación y resultados computacionales

El código de la iteración Lanczos implementado en C (serial y CUDA) y las funciones en Matlab para la correspondiente comprobación se encuentra en el CD adjuntado al documento.

5.1. Consideraciones de implementación

5.1.1. Características de los dispositivos utilizados

- **CPU:** Intel i7-2600k @ 3.4 GHz, con 8 MB de cache, 8 GB de RAM.
- **GPU:** Tesla K10.G2 con 3072 núcleos CUDA (2 GPUs,2 $\times$ 1576 núcleos), 4 GB de Memoria y una ancho de banda de memoria de 320 GB/s. Utiliza CUDA Toolkit/SDK version 6.0.

5.1.2. Librerías utilizadas

Se utilizó una librería de desarrollo CUDA [40]. Asimismo, la implementación BLAS (*Basic Linear Algebra Sub-routines*) de nVidia, sirvió para realizar comparaciones con las implementaciones.

5.2. Descripción de la implementación

La implementación consta de 3 funciones: la función tridiagonalización parcial, la función descomposición Schur y la función cálculo de error. En el caso de la primera y la última, estas utilizan operaciones que han sido paralelizadas e implementadas en CUDA: multiplicación matriz-vector, producto punto, resta de vectores.

5.2.1. Implemenación de Tridiagonalización Parcial

Esta función recibe como entradas la matriz inicial, la matriz Lanczos, y los elementos de la Tridiagonal que ya han sido calculados. Devuelve como salida el siguiente vector Lanczos, un elemento más para la diagonal y las subdiagonales que conforman la matriz tridiagonal. Esta función esta implementada tanto en Matlab como en lenguaje C, con y sin operaciones paralelas.

5.2.2. Implementación de Descomposición Schur

La función Descomposición de Schur recibe como entrada los elementos de la tridiagonal y devuelve a la salida los elementos de la matriz diagonalizada (autovalores de la tridiagonal) y la matriz de transformación ortonormal utilizada en la descomposición Schur. Esta función esta implementada tanto en Matlab como en lenguaje C (serial).

5.2.3. Implementación de Cálculo de Error

La función de calculo de error recibe como entrada la matriz inicial, la matriz Lanczos, los elementos de la diagonal obtenidos en la Descomposición Schur y la matriz de Transformación Schur. En la salida se encuentra el mayor autovalor y autovector correspondiente de la matriz inicial estimados en la iteración actual, los cuales permitirán calcular a su vez el error relativo de la iteración actual, que servirá para detener o continuar la iteración. Esta función está implementada tanto en Matlab como en lenguaje C, con y sin operaciones paralelas.

5.2.4. Implementación de Multiplicación Matriz-Vector Paralelizado

Para esta operación se utilizó el método propuesto en [39]. En cuanto a las características CUDA, se utilizarán bloques bidimensionales de 16×16 threads y la grilla será unidimensional compuesta por una cantidad de bloques equivalente a la cantidad de columnas de la matriz inicial dividido por 16, considerándose un bloque más si hay residuo. Se realizó implementaciones para esta operación con 3 tipos de memoria host (Memoria paginable, memoria no-paginable y memoria unificada) mientras que para memoria *device* se consideró utilizar: sólo memoria global, memoria global junto a memoria compartida, y memoria textura (para cargar la matriz inicial en *device*) junto a memoria global y compartida.

5.2.5. Implementación de Producto Punto Paralelizado

Para esta operación se implementó dos versiones de reducciones, una que utiliza reducciones en memoria compartida y otra que utiliza la intrucción shuffle. Se utilizó bloques y grilla unidimensionales. Para determinar que configuración de grilla le conviene más se realizaron pruebas donde la cantidad de threads a usar, y el número máximo de bloques son variados, teniendose 49 combinaciones:

$$threadIdx.x = [16, 32, 64, 128, 256, 512, 1024]$$

$$MaxBlocks = [32, 64, 128, 256, 512, 1024, 2048]$$

Comparando los diversos resultados se determinó que se obtienen los mejores resultados con 128 threads y 64 bloques máximo para la implementación con reducciones en memoria compartida y 128 threads y 128 bloques máximo para la implementación con la instrucción shuffle.

5.2.6. Implementación de Resta de Vectores Paralelizado

Para esta implementación se usaron bloques y grillas unidimensionales y con el propósito de determinar que configuración de grilla le conviene más se realizaron pruebas donde la cantidad de threads varía:

$$threadIdx.x = [16, 32, 64, 128, 256, 512, 1024]$$

Comparando los diversos resultados se determinó que se obtienen los mejores resultados con 128 threads.

5.3. Resultados computacionales

Para obtener una cantidad de ticks (ciclos de reloj) fiables, se ejecutó 100 veces y se calculó la media para cada una de las operaciones descritas a continuación, por cada tipo de implementación y cada tamaño de matriz o vector a analizar.

5.3.1. Resultados de Multiplicación Matriz-Vector

Luego de evaluarse que los distintos kernels y función serial calculen la multiplicación matriz-vector con un error relativo menor a 10^{-6} respecto a la implementación en Matlab, se

procedió a evaluar la cantidad de ciclos de reloj que toma la ejecución de cada una de las implementaciones.

En las siguientes tablas se tiene los ticks registrados para esta operación implementada con diversos tipos de memorias *host* y *device*, asi como una implementación serial. Se aprecia que las implementaciones con menor cantidad de ticks son aquellas que utilizan memoria unificada.

Tabla 5.1: Ticks ($\times 10^8$) de multiplicación matriz-vector en C estándar (serial).

Serial	**Tamaños de Matriz Simétrica**							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
	0.001	0.005	0.019	0.086	0.346	0.782	1.390	3.148

Tabla 5.2: Ticks ($\times 10^8$) de multiplicación matriz-vector con Memoria *Host* Paginable.

Memoria Paginable	**Tamaños de Matriz Simétrica**							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
Mem. Global	0.011	0.036	0.137	0.432	1.396	3.036	4.888	12.534
Mem. Shared	0.004	0.011	0.035	0.191	0.756	1.698	3.009	6.771
Mem. Textura	0.004	0.013	0.043	0.221	0.859	1.918	3.418	7.639
Cublas	0.013	0.027	0.051	0.203	0.768	1.706	3.003	6.745

Tabla 5.3: Ticks ($\times 10^8$) de multiplicación matriz-vector con Memoria *Host* No Paginable.

Memoria No Paginable	**Tamaños de Matriz Simétrica**							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
Mem. Global	0.009	0.032	0.126	0.335	1.018	2.183	3.369	9.122
Mem. Shared	0.002	0.007	0.024	0.093	0.370	0.837	1.477	3.341
Mem. Textura	0.003	0.009	0.028	0.096	0.368	0.821	1.455	3.263
Cublas	0.011	0.019	0.036	0.097	0.371	0.831	1.466	3.285

Tabla 5.4: Ticks ($\times 10^8$) de multiplicación matriz-vector con Memoria Unificada.

Memoria Unificada	**Tamaños de Matriz Simétrica**							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
Mem. Global	0.007	0.026	0.105	0.250	0.672	1.418	2.010	6.048
Mem. Shared	0.0006	0.001	0.003	0.008	0.029	0.070	0.116	0.279
Mem. Textura	0.002	0.005	0.014	0.041	0.152	0.354	0.585	1.441
Cublas	0.016	0.017	0.018	0.022	0.042	0.069	0.108	0.226

En la figura 5.1 se muestra la comparación entre las implementaciones utilizando distintas memorias *device*, C estándar (serial) y utilizando la librería CUBLAS, todas ellas utilizando memoria unificada para el traslado de datos. Se observa que si bien utilizar la librería CUBLAS es la opción más rápida, una de las implementaciones (con memoria compartida) es casi tan buena como ella, a su vez, estas dos son mucho mejor que la implementación serial.

Un detalle a señalar es que las dimensiones máximas de la matriz que puede ser utilizada es aprox. 22000×22000, esto se debe a que considerando que se trabaja con datos tipo *float*, se estaría ocupando aproximadamente 1.8 GB (sólo con la matriz) de los 2 GB disponibles del dispositivo. Si se deseara trabajar con matrices más grandes, se debera dividir la operación en

bloques, p.e. si la matriz inicial fuese de dimensiones 44000×44000, se deberían realizar 4 multiplicaciones matriz-vector con submatrices de 22000×22000.

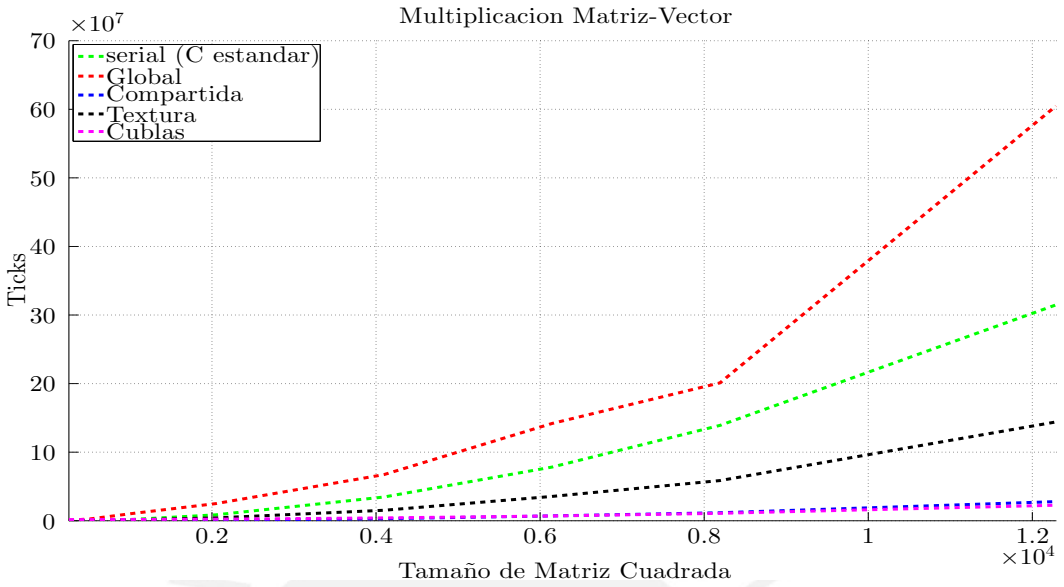


Figura 5.1: Comparación entre implementaciones de multiplicación matriz-vector.

5.3.2. Resultados de Resta de Vectores

Se verificó que las implementaciones paralela y serial presenten un error relativo menor a 10^{-6} respecto a la implementación en Matlab. En la figura 5.2 se compara implementaciones serial, usando memoria unificada en CUDA y finalmente con CUBLAS. Para este caso se calcula la resta de tres vectores expresada en la línea 5 del Algoritmo 2.

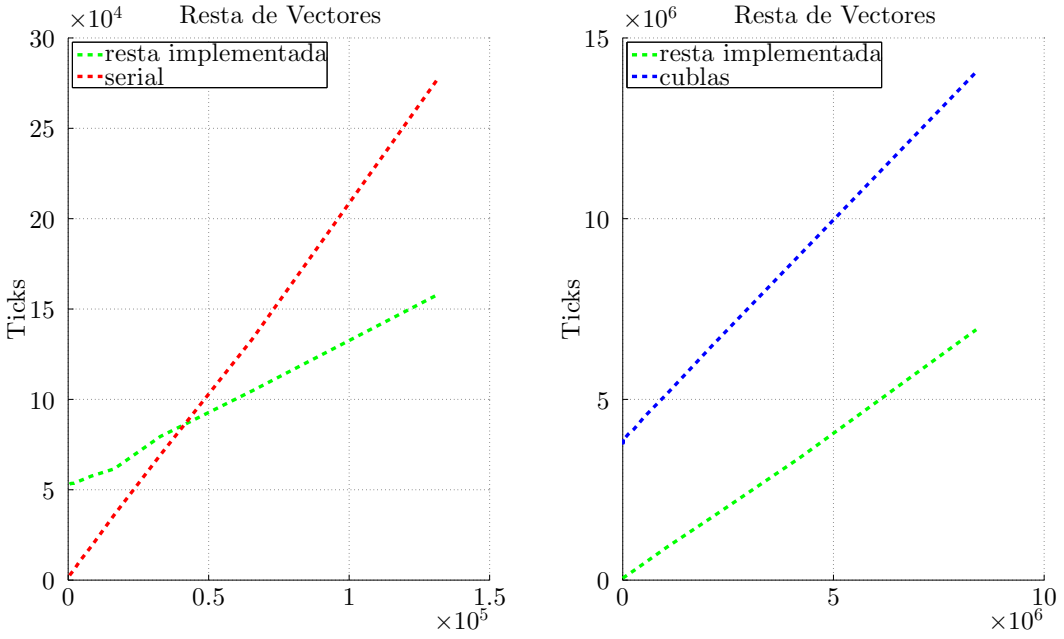


Figura 5.2: Comparación entre implementaciones de resta de vectores.

En la evaluación realizada se determina que no es rentable paralelizar la operación para vectores que no tengan aproximadamente más de 4×10^4 elementos. Además, evaluándose hasta vectores de un tamaño de 8388608 se aprecia que CUBLAS no es la mejor opció paralela.

5.3.3. Resultados de Producto de Vectores

Se verificó que las implementaciones paralelas y serial presenten un error relativo menor a 10^{-6} respecto a la implementación en Matlab. En la figura 5.3 se muestra una comparación entre los métodos de reducción utilizados, observándose que la implementación con la instrucción *shuffle* es ligeramente mejor.

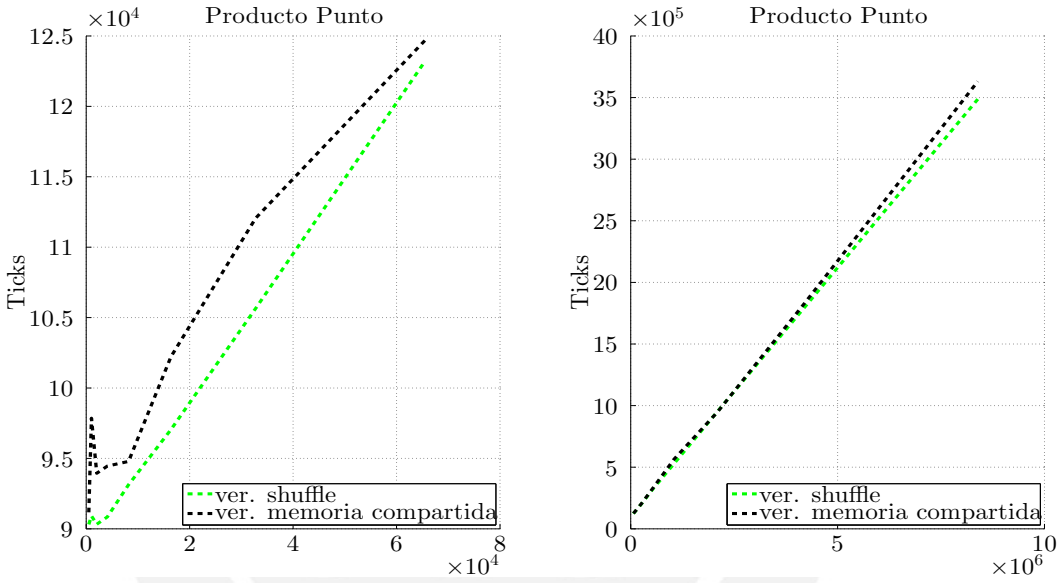


Figura 5.3: Comparación entre reducciones con memoria compartida y la instrucción shuffle.

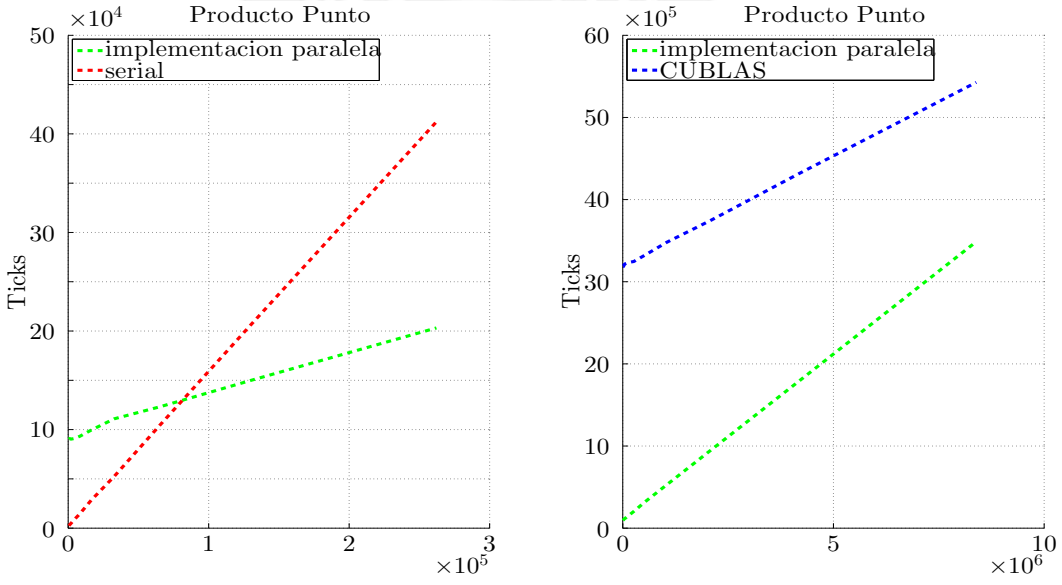


Figura 5.4: Comparación entre operaciones producto punto.

La figura 5.4 muestra una comparación entre utilizar CUBLAS, una implementación serial y la implementación en CUDA con reducciones con la instrucción *shuffle*. Se aprecia que para que sea rentable paralelizar la operación producto punto, se requiere que los vectores a utilizarse deben tener aprox $0{,}75 \times 10^5$ elementos. Además no es recomendable utilizar CUBLAS.

5.3.4. Resultados de la Iteración Lanczos

El autovalor y autovector calculados con cualquiera de las implementaciones permite resolver (2.1) con un error relativo de 10^{-6} respecto a Matlab. En las las tablas 5.7, 5.8 y 5.9 se muestra la cantidad de ticks al utilizar los 3 tipos de memoria *host* y diversos tipos de memorias *device* para la multiplicación matriz-vector. Se aprecia que los mejores resultados pertenecen a la implementación que utiliza memoria unificada debido a la forma eficiente en la que el sistema migra data entre *host* y *device*, resaltando las implementaciones que utilizan memoria compartida y la librería CUBLAS como las que tienen menor tiempo computacional.

Considerando la mejor implementación realizada (memoria unificada y memoria compartida), en la figura 5.5 se compara los tiempos de ejecución de esta implementación, la versión serial en C estándar, Matlab y utilizando CUBLAS (con memoria unificada). Para esta última sólo se considera la multiplicación matriz-vector de CUBLAS, pues se ha probado anteriormente que las operaciones resta de vectores y producto punto de CUBLAS son muy costosas computacionalmente para los tamaños de vectores a evaluarse.

Tabla 5.5: Ticks ($\times 10^8$) de Iteración Lanczos en Matlab.

Matlab	**Tamaños de Matriz Simétrica**							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
	0.022	0.034	0.200	1.001	3.892	8.652	15.511	36.582

Tabla 5.6: Ticks ($\times 10^8$) de Iteración Lanczos en C estándar (serial).

Serial	**Tamaños de Matriz Simétrica**							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
	0.011	0.034	0.133	0.609	1.725	5.475	6.909	21.946

Tabla 5.7: Ticks ($\times 10^8$) de Iteración Lanczos con Memoria *Host* Paginable.

Memoria Paginable	**Tamaños de Matriz Simétrica**							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
Mem. Global	0.060	0.200	0.770	1.939	4.099	8.734	21.005	36.774
Mem. Shared	0.009	0.020	0.056	0.247	0.947	1.996	3.474	7.910
Mem. Textura	0.011	0.023	0.063	0.266	1.051	2.121	3.989	8.413
Cublas	0.027	0.083	0.117	0.243	1.046	1.966	4.045	7.626

Tabla 5.8: Ticks ($\times 10^8$) de Iteración Lanczos con Memoria *Host* No Paginable.

Memoria No Paginable	Tamaños de Matriz Simétrica							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
Mem. Global	0.111	0.297	0.960	2.245	4.332	8.761	12.597	35.726
Mem. Shared	0.060	0.120	0.249	0.555	1.079	2.021	3.125	6.211
Mem. Textura	0.061	0.117	0.243	0.558	1.031	1.880	2.957	5.720
Cublas	0.068	0.179	0.308	0.554	1.365	1.971	3.044	5.906

Tabla 5.9: Ticks ($\times 10^8$) de Iteración Lanczos con Memoria Unificada.

Memoria Unificada	Tamaños de Matriz Simétrica							
	256	**512**	**1024**	**2048**	**4096**	**6144**	**8192**	**12288**
Mem. Global	0.075	0.212	0.765	2.313	6.779	8.743	13.152	41.828
Mem. Shared	0.025	0.040	0.051	0.106	0.298	0.451	0.706	1.580
Mem. Textura	0.027	0.036	0.073	0.130	0.305	0.646	1.054	2.375
Cublas	0.138	0.145	0.162	0.207	0.278	0.443	0.668	1.320

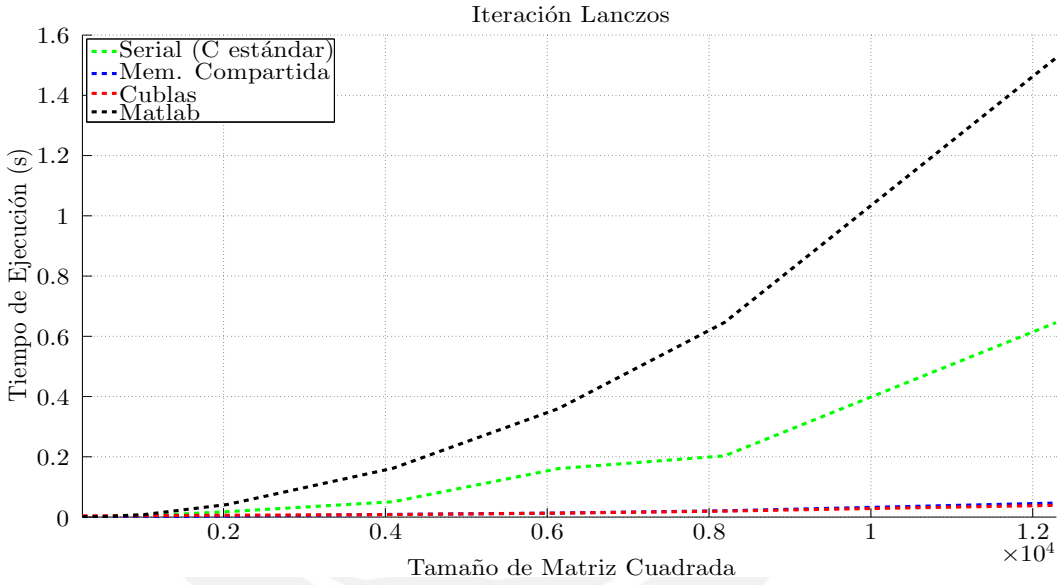


Figura 5.5: Comparación entre implementaciones de la iteración Lanczos.

En la figura 5.6 se muestra las curvas de rendimiento entre la mejor implementación en CUDA (usando memoria unificada y memoria compartida) y Matlab y C estándar. Las curvas de rendimiento, crecen hasta estabilizarse aparentemente en 25 y 15 alrededor de un tamaño de matriz de 12288×12288. Por otro lado, en la figura 5.7 se tiene el rendimiento entre la mejor implementación en CUDA y la implementación utilizando la librería CUBLAS. Se aprecia que la implementación en CUDA es casi tan buena como la que usa la librería CUBLAS al haber un rendimiento cercano a 1. El rendimiento mayor a 1 al inicio de la curva se debe a que el costo de funciones de CUBLAS para tamaños pequeños (en este caso menores a 4000×4000 elementos) de matrices o vectores siempre es alto incluso respecto a una implementación paralela simple. Cabe señalar que la cantidad de iteraciones necesarias para que el autovalor y autovector converjan a los valores deseados varía entre 4 a 6.

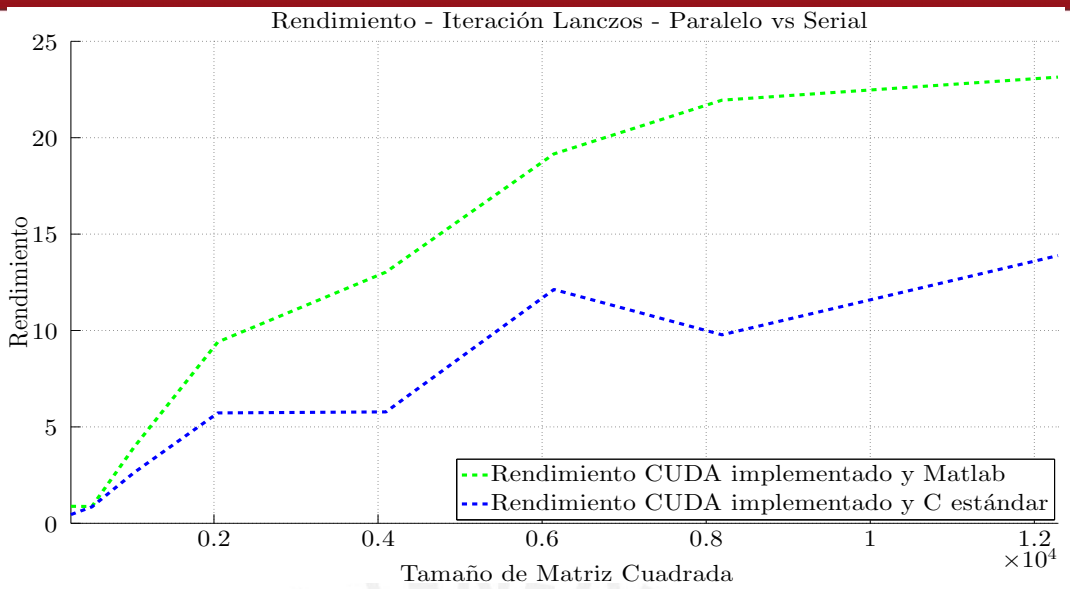


Figura 5.6: Rendimiento entre implementación paralela y seriales de la iteración Lanczos.

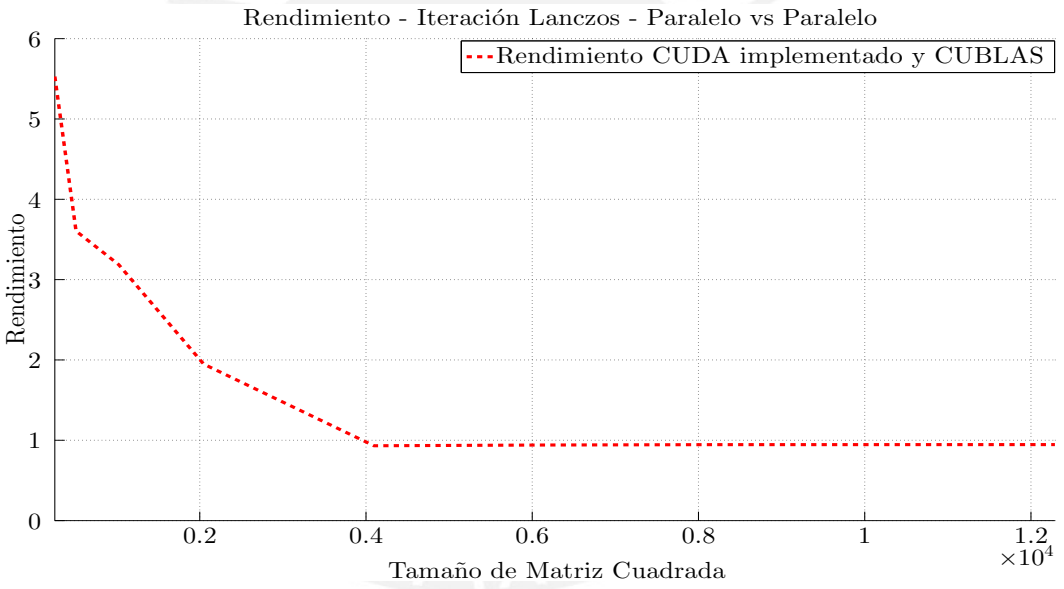


Figura 5.7: Rendimiento entre implementaciones paralelas de la iteración Lanczos.

Analizando todos los ticks y/o tiempos de ejecución mostrados se tiene que se ha logrado implementar versiones de la iteración Lanczos utilizando el último tipo de memoria optimizada de las tarjetas Nvidia (memoria unificada) resultando mucho más ventajosas en cuanto tiempo de ejecución frente a tipos anteriores. En la figura 5.8 se ha tomando como referencia las implementaciones que utilizan memoria compartida, ya que se obtuvieron los mejores resultados en los tres casos de memoria *Host* presentados (mem. paginable, no paginable y unificada). Se aprecia que el rendimiento que se obtiene usando memoria unificada frente a memoria paginable es incremental siendo aproximadamente 3 para tamños de matriz de 4000×4000 hasta aproximadamente 5 para matrices de 12000×12000

elementos, esto debido a que ambas . Mientras que el rendimiento usando memoria unificada frente a memoria no paginable es mayor a 3. Además se observa que se la implementacion con memoria no paginable es mejor que la implementación con memoria paginable para matrices de 6000×6000 elementos.

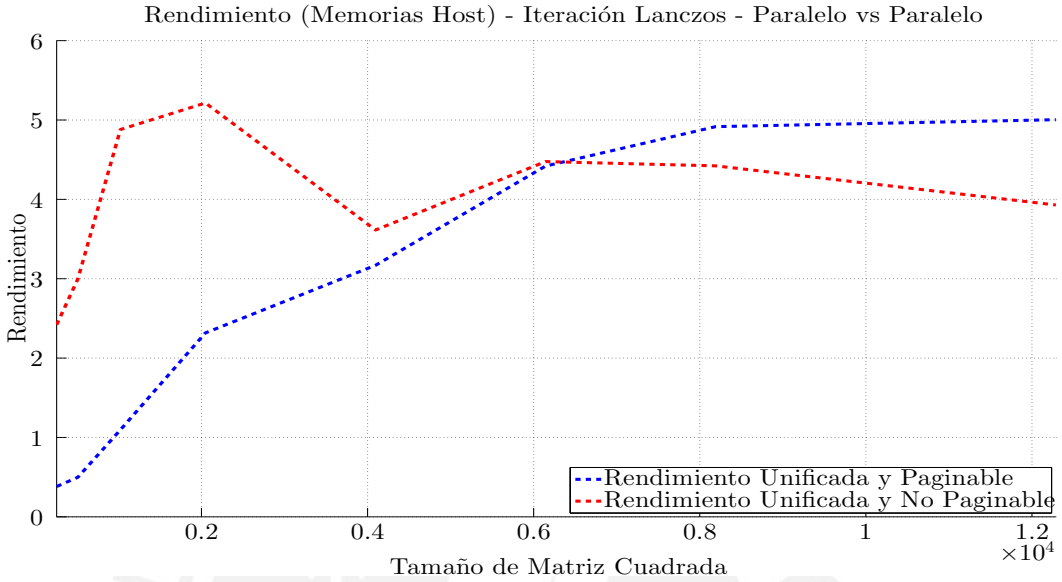


Figura 5.8: Rendimiento entre implementaciones paralelas de la iteración Lanczos.

Conclusiones

- Es posible implementar la iteración Lanczos en la arquitectura CUDA de forma computacionalmente eficiente basándose en una adecuada paralelización de la multiplicación matriz-vector; ya que esta operación es la que cuenta con la mayor carga computacional; operaciones adicionales como suma de vectores y producto punto no influyen significativamente. Esto debido a que para tamaños de matrices mayores a 1024×1024, el tiempo de ejecución de multiplicación matriz-vector es 36 y 18 veces mayor que los de suma y producto punto de vectores respectivamente. Esta diferencia se incrementará drásticamente con mayores tamaños de matriz y vectores.

- Al evaluarse la mejor implementación paralela realizada de la iteración Lanczos, la cual usa memoria unificada y memoria compartida, respecto a la implementación en Matlab y C estándar (secuencial) se obtienen curvas de rendimiento crecientes que llegan a ser de 25 y 15 veces más rápido respectivamente para matrices de 12288×12288. Además, la implementación paralela es casi tan buena como la implementación que utiliza la librería CUBLAS (para la multiplicación matriz-vector) para matrices de más de 4000×4000 elementos, dado que presenta un rendimiento ligeramente menor a 1 al compararlas; pero es claramente superior para matrices de menos elementos. Por último una implementación paralela con memoria unificada para este problema es mínimo 3 veces más rápida que cualquiera de las implementaciones paralelas con cualquier otro tipo de memoria *host*.

- En cuanto a la multiplicación matriz-vector, el tamaño de la matriz será un factor a tomar en cuenta debido a que la tarjeta Nvidia tiene una cantidad de memoria limitada. La tarjeta CUDA utilizada posee 2GB de memoria global, con lo cual se puede evaluar matrices de hasta 22000×22000 elementos.

Recomendaciones

- La mayoría de problemas matemáticos en los que se requiere solucionar el *eigenproblem* se modelan con matrices no simétricas. Para estos casos existen variaciones de la iteración Lanczos [3]. Como trabajo futuro se recomienda implementar alguna de las variaciones en CUDA para lograr resolver el problema de manera general.
- Si se utiliza Lanczos junto a deflacciones [3], se tendría un método adecuado para calcular los primeros k autovalores y autovectores. Sin embargo, esto ocasiona pérdida de ortogonalidad en los vectores Lanczos. Para ello se recomienda utilizar métodos de reortogonalización como en [36].
- También se recomienda realizar una implementación combinando MPI y CUDA [41] ya que esto permitiría distribuir una operación entre múltples GPU obteniéndose un menor tiempo de ejecución. Asimismo la cantidad de data a procesar puede ser mayor ya que se dividiría entre las memorias de los GPU.

Bibliografía

[1] I. Markovsky, *Low Rank Approximation: Algorithms, Implementation, Applications*. Communications and Control Engineering, Springer, 2011.

[2] MobileReference, *Math Formulas and Tables for Smartphones and Mobile Devices - FREE Functions, Equations, and Table of Derivatives in the Trial Version*. Mobi Study Guides, MobileReference, 2007.

[3] G. Golub and C. Van Loan, *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, 1996.

[4] Z.-J. Bai and B. Datta, "Optimization methods for partial quadratic eigenvalue assignment in vibrations," in *Communications, Computing and Control Applications (CCCA), 2011 International Conference on*, pp. 1–6, March 2011.

[5] C.-C. Su, "Fast algorithm for transversely inhomogeneous optical fibres using power method and fast fourier transform," *Optoelectronics, IEE Proceedings J*, vol. 134, pp. 276–280, October 1987.

[6] P. Pacheco, *An Introduction to Parallel Programming*. An Introduction to Parallel Programming, Elsevier Science, 2011.

[7] P. Cockshott and K. Renfrew, *SIMD Programming Manual for Linux and Windows*. Springer Professional Computing, Springer, 2004.

[8] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson Education, 2010.

[9] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series, Elsevier Science, 2012.

[10] G. H. Golub and H. A. van der Vorst, "Eigenvalue Computation in the 20th Century," *Journal of Computational and Applied Mathematics*, vol. 123, pp. 35–65, 2000.

[11] L. Trefethen and D. Bau, *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[12] M. Gupta, S. Angrignon, C. Forrester, S. Simmons, and S. Douglas, "A spatio-temporal power method for time-domain multi-channel speech enhancement," in *Applications of Signal Processing to Audio and Acoustics, 2009. WASPAA '09. IEEE Workshop on*, pp. 137–140, Oct 2009.

[13] K. Bryan and T. Leise, "The $25,000,000,000 eigenvector: the linear algebra behind google," *SIAM Review*, vol. 48, pp. 569–581, 2006.

[14] T.-Y. Huang and R.-B. Wu, "Steady-state response by finite-difference time-domain method and lanczos algorithm," *Microwave Theory and Techniques, IEEE Transactions on*, vol. 54, pp. 3038–3044, July 2006.

[15] A. Cangellaris and L. Zhao, "Rapid FDTD simulation without time stepping," *Microwave and Guided Wave Letters, IEEE*, vol. 9, pp. 4–6, Jan 1999.

[16] L. Yang, "Parallel Lanczos bidiagonalization for total least squares filter in robot navigation," in *Parallel Computing in Electrical Engineering, 2002. PARELEC '02. Proceedings. International Conference on*, pp. 415–418, 2002.

[17] J. Chen and Y. Saad, "Lanczos Vectors versus Singular Vectors for Effective Dimension Reduction," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 8, pp. 1091–1103, 2009.

[18] T. Wittig, I. Munteanu, R. Schuhmann, and T. Weiland, "Two-step Lanczos algorithm for model order reduction," *Magnetics, IEEE Transactions on*, vol. 38, pp. 673–676, Mar 2002.

[19] V. Papakos and I. Jaimoukha, "A deflated implicitly restarted Lanczos algorithm for model reduction," in *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, vol. 3, pp. 2902–2907 Vol.3, Dec 2003.

[20] X. Zhang and H. Iwakura, "Design of IIR digital allpass filters based on eigenvalue problem," *Signal Processing, IEEE Transactions on*, vol. 47, pp. 554–559, Feb 1999.

[21] X. Zhang, K. Suzuki, and T. Yoshikawa, "Complex Chebyshev approximation for IIR digital filters based on eigenvalue problem," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 47, pp. 1429–1436, Dec 2000.

[22] M. Iskander, M. Morrison, W. Datwyler, and M. Hamilton, "A new course on computational methods in electromagnetics," *Education, IEEE Transactions on*, vol. 31, pp. 101–115, May 1988.

[23] C. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[24] V. Patel, Y. Shi, P. Thompson, and A. Toga, "K-SVD for HARDI denoising," in *Biomedical Imaging: From Nano to Macro, 2011 IEEE International Symposium on*, pp. 1805–1808, March 2011.

[25] L. I. Smith, "A tutorial on principal components analysis," tech. rep., Cornell University, USA, February 26 2002.

[26] E. J. Candès, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis?," *J. ACM*, vol. 58, no. 3, pp. 66–102, 2011.

[27] H. Shen and J. Z. Huang, "Sparse principal component analysis via regularized low rank matrix approximation," *Journal of Multivariate Analysis*, vol. 99, no. 6, pp. 1015 – 1034, 2008.

[28] "SLEPc." http://slepc.upv.es/. Accessed: 2014-10-20.

[29] "PRIMME." http://www.cs.wm.edu/ andreas/software/. Accessed: 2014-10-22.

[30] "The NAG Library, The Numerical Algorithms Group (NAG), Oxford, United Kingdom." www.nag.com. Accessed: 2014-10-22.

[31] "CULA." http://www.culatools.com/. Accessed: 2014-10-20.

[32] C. Kelley, *Iterative Methods for Linear and Nonlinear Equations*. Frontiers in Applied Mathematics, Society for Industrial and Applied Mathematics, 1995.

[33] J. Monahan, *Numerical Methods of Statistics*. Cambridge Series in Statistical and Probabilistic Mathematics, Cambridge University Press, 2011.

[34] L. Scott, *Numerical Analysis*. Princeton University Press, 2011.

[35] S. Banerjee and A. Roy, *Linear Algebra and Matrix Analysis for Statistics*. Chapman & Hall/CRC Texts in Statistical Science, Taylor & Francis, 2014.

[36] L. Komzsik, *The Lanczos Method: Evolution and Application*. Software, Enviroments and Tools, Society for Industrial and Applied Mathematics (SIAM), 2003.

[37] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, April 2014. Version 6.0.

[38] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in cuda," 2014.

[39] N. Fujimoto, "Faster matrix-vector multiplication on GeForce 8800GTX," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, April 2008.

[40] P. Rodriguez, *Yupana*. Grupo de Procesamiento Digital de Señales e Imágenes (GPDSI-PUCP), 2014.

[41] "MPI Solutions for GPUs." https://developer.nvidia.com/mpi-solutions-gpus. Accessed: 2014-11-22.