Análisis de Componentes Principales en Paralelo[1]:

G. Figueroa M.[2], E. Carrera R.[3] y A. Jiménez R.[4].
Escuela de Matemática
Instituto Tecnológico de Costa Rica

30 de mayo de 2012

[1]Investigación financiada por la VIE, código: # 5402-1440-3001
[2]email: gfiguero@itcr.ac.cr
[3]email: lecarrera@itcr.ac.cr
[4]email: ajimenez@itcr.ac.cr

Resumen

La computación paralela es una técnica de programación en la que muchas instrucciones se ejecutan simultáneamente. Se basa en el principio de que los problemas grandes se pueden dividir en partes más pequeñas que pueden resolverse de forma paralela. En los últimos años el interés en ella ha aumentado y se ha convertido en el paradigma dominante en la arquitectura de computadores, principalmente en los procesadores multinúcleo.

Por otro lado, el Análisis de Componentes Principales (ACP) es una técnica multivariable utilizada para reducir la dimensionalidad de un conjunto de datos cuantitativos. Su objetivo es extraer la información importante de una tabla de datos y representarla mediante nuevas variables ortogonales, llamadas componentes principales, a fin de hallar la relación entre las variables originales y los individuos en estudio.

En este proyecto se desarrolla una implementación mediante computación paralela para realizar el ACP a tablas de datos de gran tamaño.

Abstract

Parallel computing is a programming technique in which many instructions are executed simultaneously. Is based on the principle that large problems can be divided into smaller parts which can be addressed in parallel. In recent years the interest in it has increased and has become the dominant paradigm in computer architecture, especially in multi-core processors.

On the other hand, Principal Component Analysis (PCA) is a multivariate technique used to reduce the dimensionality of a set of quantitative data. Aims to extract the important information in the table and represent data using new orthogonal variables called principal components, in order to find the relationship between the original variables and the studied specimens.

This project will develop an implementation by computation parallel for the ACP tables large data size.

Palabras claves: análisis de componentes principales, análisis multivariado, programación paralela.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Sobre el proyecto

1.1. Introducción

El análisis de componentes principales (ACP) es un procedimiento matemático que transforma un conjunto de variables correlacionadas de respuesta en un conjunto menor de variables no correlacionadas llamadas componentes principales. Al observar cuidadosamente este nuevo conjunto de variables no correlacionadas, se puede detectar datos atípicos (*outliers*), variables altamente correlacionadas, etc.

El ACP es quizá la técnica más útil para depurar datos multivariados, se acostumbra usar como un primer paso antes de aplicar otras técnicas. Algunas veces un ACP puede ayudar a los investigadores a comprender mejor la estructura de correlación entre las respuestas y, en ocasiones, puede ayudar a generar hipótesis acerca de las variables o de los datos.

El objetivo principal del ACP es descubrir la verdadera dimensionalidad de los datos: aunque se estén midiendo p variables puede suceder que la dimensión real de los datos sea menor que p En este caso se puede usar el ACP para determinar la verdadera dimensionalidad de los datos y así reemplazar las variables originales por un número menor de variables subyacentes, sin que se pierda información; esto permite usar una menor cantidad de variables en los análisis siguientes.

1.2. Problema a investigar

Dada un tabla de datos $\mathbf{X}$ con p variables cuantitativas medidas para n individuos.

	x^1	...	x^p
x_1	x_1^1	...	x_1^p
x_2	x_2^1	...	x_2^p
$\vdots$	$\vdots$		$\vdots$
x_n	x_n^1	...	x_n^p

Se quiere diseñar e implementar el análisis de componentes principales mediante técnicas de computación paralela con el fin de poder aplicarlo a tablas de datos $\mathbf{X}$ de grandes dimensiones, en especial aquellas en las cuales el número de variables p es grande, pues esto incrementa sustancialmente la complejidad del ACP.

1.3. Objetivos

1. **Objetivo general**

   Diseñar e implementar algoritmos en paralelo que realicen el análisis de componentes principales a grandes cantidades de datos, mediante programación en paralelo, tanto de CPU's como de GPU's.

2. **Objetivos específicos**

   a) Diseño e implementación en paralelo del cálculo de la matriz de covarianzas y/o correlaciones.

   b) Diseño e implementación en paralelo del producto de matrices.

   c) Diseño e implementación de cálculo numérico de valores propios y de vectores propios para una matriz simétrica definida positiva.

   d) Diseño e implementación de la técnica de análisis de componentes principales.

   e) Representación gráfica de la solución de análisis de componentes principales.

1.4. Metodología

Se realizó una investigación bibliográfica de cada uno de los temas de investigación: análisis de componentes principales, programación paralela, cálculo de valores propios y representación gráfica de los planos principales y el círculo de correlaciones. Se hicieron reuniones semanales para analizar los resultados obtenidos, definir estrategias y líneas de investigación.

Para cada algoritmos se analizó la pertinencia de programarlo utilizando los GPU's (programación mediante `CUDA`), los CPU's (programación mediante OpenMP) o ambos y se eligió lo mejor en cada caso.

1.5. Resumen de resultados

Se logró implementar por completo la técnica ACP usando computación en paralelo, específicamente se combinaron los paradigmas de programación `CUDA` para hacer uso de los GPU's y `OpenMP` para usar los CPU's. Para la representación gráfica de los resultados se implementó un paquete que construye el plano principal y el círculo de correlaciones, además, de algunos filtros sobre los datos con el fin de facilitar la visualización de resultados.

Capítulo 2

Marco teórico

2.1. Introducción

El Análisis de Componentes Principales (ACP) es una técnica multivariable que analiza una tabla de datos, la cual contiene observaciones cuantitativas realizadas a distintos individuos. El objetivo del ACP es extraer la información más importante de la tabla y representarla, mediante nuevas variables ortogonales, llamadas componentes principales, a fin de hallar la relación entre las variables originales y entre los individuos en estudio. Dicha representación se realiza usualmente en planos, comparando pares de componetes.
Matemáticamente, el ACP se deriva de la diagonalización de una matriz de datos, por medio de sus valores y vectores propios. El problema de determinar los valores propios de una matriz se puede resolver por distintos métodos (Jacobi, QR, etc.) cuando la matriz es simétrica. Considerando el caso en que se desee hacer el análisis a una tabla de datos con muchas observaciones las cuales sean inmanejables para los programas computacionales actuales, se piensa en utilizar la técnica de paralelización en el cálculo de las componentes principales, a fin de realizar una reducción de dimensión que ayude a obtener un problema manejable.

Para el desarrollo del proyecto fué necesario conocer los detalles del ACP, así como el estudio de algoritmos con los que pudiera resolver el problema de determinar los valores y vectores propios de una matriz.

2.2. Análisis de Componentes Principales

El análisis de componentes principales permite analizar una tabla de datos $\mathbf{X}$ que contiene p observaciones cuantitativas realizadas a n individuos con el fin de extraer información importante que permite representarla mediante una cantidad menor de nuevas variables (componentes principales) a fin de hallar alguna relación entre las variables originales y los individuos. Cada componente principal es combinación lineal de las variables originales, tal

que en ella se ve reflejada la información contenida en las variables, al ser las componentes principales no correlacionadas dos a dos se elimina la información redundante y al asegurarse la varianza máxima, las componentes principales contienen la máxima información posible ([1], [16], [18]).

Se dice que un ACP es normado si la tabla de datos $\mathbf{X}$ contiene las variables centradas y estandarizadas (todas tienen media cero y varianza 1). En el ACP normado se tienen las siguientes características:

- La matriz de covarianzas $\mathbf{V}$ y la matriz de correlaciones $\mathbf{R}$ coinciden.
- La inercia de N es igual a la traza de $\mathbf{V}$, la cual es igual a la suma de las varianzas. En este caso, esta suma es exactamente p.

Se desea una pérdida mínima de información al proyectar los n individuos sobre un espacio de dimensión q con $q < p$ tal que la dispersión en el espacio proyectado $\mathbb{R}^q$ debe ser máxima.

Sea $\mathbf{X}$ la tabla de datos definida con p variables cuantitativas tal que $\mathbb{R}^n$ es el espacio de individuos y $\mathbb{R}^p$ es el espacio de variables.

	x^1	...	x^p
x_1	x_1^1	...	x_1^p
x_2	x_2^1	...	x_2^p
$\vdots$	$\vdots$		$\vdots$
x_n	x_n^1	...	x_n^p

Cuadro 2.1: Tabla de datos.

Si $\mathbf{M}$ es la métrica sobre $\mathbb{R}^p$ y $\mathbf{D}_p$ la métrica de pesos sobre $\mathbb{R}^n$, entonces:

$$N = (\mathbf{X}, \mathbf{M}, \mathbf{D}_p)$$

representa la nube de n puntos ponderados del espacio vectorial $\mathbb{R}^p$, junto con las medidas de proximidad y angular definidas por $\mathbf{M}$, y las medidas de tendencia central y de dispersión asociadas a $\mathbf{D}_p$. El concepto de nube de puntos N es geométrico, cuya forma se busca describir y sintetizar mediante métodos estadísticos.

Dada la tabla de datos $\mathbf{X}$, la solución del ACP se obtiene al diagonalizar su matriz de correlaciones $\mathbf{R}$. En el caso general, para una métrica cualquiera, la solución es la diagonalización de una matriz $\mathbf{VM}$, producto de la matriz de covarianzas $\mathbf{V}$ de $\mathbf{X}$ y de la métrica $\mathbf{M}$.

Al diagonalizar $\mathbf{R}$ se obtienen p valores propios denotados $\lambda_1, \lambda_2, ...\lambda_p$ (ordenados en orden decreciente) a partir de los cuales se obtienen los llamados vectores principales $u_1, u_2, ..., u_p$,

donde u_i es un vector propio normado de $\mathbf{R}$ asociado al valor propio λ_i. Siendo $\mathbf{R}$ simétrica y positiva, tiene p valores propios reales. Como es semidefinida positiva, estos valores propios son mayores o iguales que cero, pero su suma es p.

Las componentes principales serán las variables asociadas a estos ejes principales, tales que

$$c_i = \mathbf{X}u_i$$

será llamada la $i-$ésima componente principal. Por definición, las componentes principales son combinación lineal de las variables originales (que son las columnas de $\mathbf{X}$), por lo que su media también es cero. Al vector u_1 asociado al mayor valor propio λ_1 de $\mathbf{R}$ se le llama el primer eje del ACP de la nube N.

En general, los ejes principales del ACP de la nube N son los vectores propios de la matriz de correlaciones $\mathbf{X}$ asociados a los valores propios de ésta, dados en orden decreciente:

$$\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_p \geq 0$$

asociados respectivamente a $u_1, u_2, ..., u_p$. Seleccionando cualquier par de ejes principales, se puede generar un plano principal.

2.2.1. Cálculo de valores y vectores propios

El problema de determinar los valores y vectores propios de una matriz es de gran importancia en el campo del Álgebra Lineal ([10]), así como en otras ramas de la Matemática y la Computación.

Definición 2.1 *Dada una matriz A de dimensión $n \times n$ ($A_{n\times n}$) cuyas entradas son números complejos, se llaman valores propios de A a los n números complejos λ, llamados valores propios, para los cuales existe un vector complejo x (no nulo), llamado vector propio, que satisface:*

$$Ax = \lambda x$$

.

Además, conociendo los siguientes teoremas, se sugiere que calcular los valores propios de una matriz A, puede hacerse reduciéndo A a otra matriz, cuyos valores propios sean mas fáciles de calcular. Es posible transformar una matriz simétrica A en una matriz diagonal D_A, siendo los valores propios de A los elementos diagonales de D_A.

Teorema 2.1 *Los valores propios de una matriz diagonal son sus elementos diagonales.*

Teorema 2.2 *Si A y P son dos matrices complejas de tamaño $n \times n$, con P no singular, entonces λ es un valor propio de A con vector propio x sii λ es un valor propio de $P^{-1}AP$ con vector propio $P^{-1}x$.*

Teorema 2.3 *Una matriz A, de tamaño $n \times n$, tiene n valores propios que son los ceros de su polinomio característico.*

Teorema 2.4 *A y $P^{-1}AP$ tienen el mismo polinomio característico.*

Mediante los teoremas anteriores es posible calcular los valores propios mediante el polinomio característico de A y hallar las raíces de la ecuación $det(A - \lambda I) = 0$. Esto, si A es estructurada y existe un procedimiento eficiente para calcular $det(A - \lambda I)$, pero no así en el caso de matrices densas ya que el costo es grande y se pueden dar muchos errores de redondeo.

El problema de los valores propios se puede resolver por diferentes métodos, entre estos están el algoritmo QR ([20]) y el método de Jacobi ([17], [9],). Algunos métodos consisten en transformar la matriz A en otra matriz, en la que se pueda deducir directamente los valores propios (diagonal, triangular) o en la que sea más fácil realizar el cálculo (matrices tridiagonales o Hessenberg). Normalmente se suele reducir la matriz mediante transformaciones de Householder o de Givens a forma tridiagonal si la matriz es simétrica (o Hessenberg si es no simétrica).Posteriormente se puede diagonalizar (o triangularizar) la matriz tridiagonal por medio de algún método como iteracion QR, bisección o divide y vencerás.

Método de Jacobi

El método de Jacobi se planteó en 1846, sin embargo en los años 60 y 80 se dejó de usar debido a la aparición del algoritmo **QR** que da buenos resultados en máquinas secuenciales. Sin embargo, el método de Jacobi ha adquirido importancia en los ultimos años, por la aparición de los computadores paralelos y a algunos resultados que indican que éste es más estable ([11]).

A principios de los 70 apareció el primer estudio del método de Jacobi en *memoria compartida*[11] y a principios de los 80 para *memoria distribuida* ([12]).

El uso de máquinas paralelas es motivado por la necesidad de resolver problemas cada vez mas grandes o complejos. Los métodos de Jacobi reducen una matriz simétrica a la forma diagonal, con los valores propios los elementos diagonales de esta matriz. Esta reducción se realiza sin operaciones previas de tridiagonalización o paso a otras formas condensadas intermedias.

Un método de Jacobi ([17]) consiste en construir una secuencia de matrices $\{A_l\}$ por medio de la generación:

$$A_{l+1} = Q_l A_l Q_l^t \quad l = 1, 2, ...$$

donde $A_1 = A$, y Q_l es una rotación de Givens en el plano (p, q), con $1 \leq p$ y $q \leq n$.

La sucesión $\{A_l\}$ converge a la matriz diagonal:

$$D = Q_k Q_k - 1 \cdots Q_2 Q_1 A Q_1^t Q_2^t \cdots Q_{k-1}^t Q_k^t$$

cuya diagonal son los valores propios de A y los vectores propios son las columnas de $Q_1^t Q_2^t \cdot \cdot \cdot Q_{k-1}^t Q_k^t$.

Cada producto $Q_l A_l Q_l^t$ anula un par de elementos no diagonales a_{ij} y a_{ji}, de la matriz A_l. La matriz Q_l coincide con la identidad excepto en los elementos $q_{ii} = c$, $q_{ij} = s$, $q_{ji} = -s$, y $q_{jj} = c$, donde $c = \cos\theta$, $s = \text{sen}\theta$, y $\tan(2\theta) = \frac{2a_{ij}}{a_{ii}-a_{jj}}$

En el método clásico de Jacobi se elige en cada iteración, como elemento a anular, el de mayor valor absoluto de entre los no diagonales.

Algorithm 2.1: Método clásico de Jacobi

```
1 while f(A) > cota do
2 |   Calcular a_ij con máximo valor absoluto entre los no diagonales;
3 |   Calcular Rotación Q(i, j, θ) tal que (QAQ^t)_ij = 0;
4 |   A = QAQ^t
```

donde $f(A) = \sqrt{\sum_{j=1, j\neq i}^{n} \sum_{i=1}^{n} a_{ij}^2}$

En el algoritmo 2.1 se elige el elemento mayor en valor absoluto entre los no diagonales. Esto produce que el número de anulaciones para que el método converja sea menor que en otros métodos, pero cada anulación es más costosa al tener que obtener previamente el mayor (en valor absoluto) de $\frac{n(n-1)}{2}$ elementos. El método clásico parece poco apropiado para programación paralela, pues el máximo que se calcula antes de cada anulación habría que calcularlo también de forma distribuida. Existen estudios sobre modificaciones del método clásico que sugieren hacerlo apropiado para multiprocesadores ([11]).

Parece mas apropiado realizar sucesivos barridos de modo que en cada barrido se anule una vez cada elemento no diagonal, realizándose $\frac{n(n-1)}{2}$ anulaciones.

Cuando en un barrido se ha anulado un elemento en el siguiente barrido ese elemento puede volver a ser distinto de cero, con lo que es necesario realizar múltiples barridos para que el método converja.

Métodos cíclicos

Proceden realizando sucesivos barridos, anulando en cada uno (uno a la vez) cada elemento no diagonal (cada análisis consta de $\frac{n(n-1)}{2}$ anulaciones). Para ello se utiliza un orden de anulacion de los elementos.

con $f(A) = \sqrt{\sum_{j=1, j\neq i}^{n} \sum_{i=1}^{n} a_{ij}^2}$

Algorithm 2.2: Método cíclico de Jacobi

```
while existan pares de ordenación do
    Obtener el par (i, j);
    Calcular Rotación Q(i, j, θ) tal que (QAQ^t)_ij = 0;
    A = QAQ^t
```

Para obtener un método de Jacobi cíclico se debe escoger un orden de los pares de índices correspondientes a elementos no diagonales (si se escoge el par (i, j) se anula el elemento a_{ij} y el a_{ji}). Los posibles órdenes que se pueden obtener con los $\frac{n(n-1)}{2}$ índices (no diagonales) se llaman **órdenes de Jacobi**. En su mayoría se agrupan los índices en pares, formando conjuntos de índices que se llaman **conjuntos de Jacobi**. Inicialmente se agrupan los n índices en $\frac{n}{2}$ pares formando un conjunto de pares de índices. Un orden de Jacobi se obtiene pasando de un conjunto de Jacobi al siguiente, por medio de un intercambio de índices, lo que produce un nuevo emparejamiento de índices.

Orden cíclico por filas

Anula los elementos en el orden dado por los pares:

$$(1,2),(1,3),...,(1,n),(2,3),(2,4),...,(2,n),...,(n-1,n)$$

Dado que se anulan de forma consecutiva los elementos en la misma fila, no se considera apropiado para algoritmos paralelos, pues para anular el elemento $a_{i,i+k+1}$, se debió anular previamente el elemento $a_{i,i+k}$ y actualizar la $i-$ésima fila, perdiéndose la posibilidad de anular varios elementos a la vez.

Orden cíclico par-impar

Ejemplo para $n = 8$, de como se obtienen los conjuntos de Jacobi para el orden par-impar.

Cada bloque de datos (columnas), correspondiente a los índices, se almacena en un procesador (representado por cajas), los cambios de índices para pasar de un conjunto de Jacobi al siguiente se muestran por medio de flechas.

Ejemplo 2.1

Se numeran los índices de 0 a $n - 1$.

2.2.2. Nociones sobre paralelización

Un esquema del método utilizando una ordenación cíclica es el algoritmo 2.2.

impar	(1,2)	←	(3,4)	←	(6,5)	←	(8,7)
par	(1,2)	→	(3,4)	→	(5,6)	→	7
impar	(2,4)	←	(1,6)	←	(3,8)	←	(5,7)
par	(2,6)	→	(1,8)	→	(3,7)	→	5
impar	(4,6)	←	(2,8)	←	(1,7)	←	(3,5)
par	(4,8)	→	(2,7)	→	(1,5)	→	3
impar	(6,8)	←	(4,7)	←	(2,5)	←	(1,3)
par	(6,7)	→	(4,5)	→	(2,3)	→	1

Cuadro 2.2: Orden cíclico par-impar.

Para la paralelización del procedimiento de cálculo de los parámetros hay que tener en cuenta que dadas dos rotaciones de Givens Q_1 y Q_2 que anulan elementos a_{ij}, a_{ks}, de índices disjuntos, $\{i,j\} \cap \{k,s\} \neq \emptyset$, el cálculo de sus parámetros se puede hacer en paralelo puesto que dependen de elementos distintos de A. Es posible explotar el paralelismo en los cálculos asociados a las actualizaciones de la matriz A, $A_l + 1 = Q_l A_l Q_l^t$.

Ejemplo 2.2

Considere n = 4,

$$Q_1 = \begin{bmatrix} \hat{Q}_1 & 0 \\ 0 & I \end{bmatrix} \quad , \quad Q_2 = \begin{bmatrix} I & 0 \\ 0 & \hat{Q}_2 \end{bmatrix}$$

donde $\hat{Q}_1$ y $\hat{Q}_2$ representan rotaciones de Givens a nivel 2×2, entonces

$$Q_1 Q_2 = Q_2 Q_1 = \begin{bmatrix} \hat{Q}_1 & 0 \\ 0 & \hat{Q}_2 \end{bmatrix}$$

La actualización de A mediante las correspondientes transformaciones de semejanza viene dada por

$$\begin{bmatrix} \hat{Q}_1 & 0 \\ 0 & \hat{Q}_2 \end{bmatrix} \cdot \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} \hat{Q}_1^t & 0 \\ 0 & \hat{Q}_2^t \end{bmatrix} =$$

$$\begin{bmatrix} \hat{Q}_1 A_{11} \hat{Q}_1^t & \hat{Q}_1 A_{12} \hat{Q}_2^t \\ \hat{Q}_2 A_{21} \hat{Q}_1^t & \hat{Q}_2 A_{22} \hat{Q}_2^t \end{bmatrix}$$

La premultiplicación por $\hat{Q}_i$ sólo afecta a un par de filas y la postmultiplicación por $\hat{Q}_i^t$ sólo a un par de columnas. Estos cálculos pueden realizarse en paralelo si la matriz A se distribuye de forma adecuada.

Una posibilidad es distribuir la matriz de manera que cada procesador contenga un par de columnas. Para la actualización de A hay que difundir los parámetros de las rotaciones pues todos los procesadores necesitaran de ellos para actualizar los elementos que contienen de la matriz.

2.2.3. Rotación de Givens

La matriz Q_l coincide con la identidad excepto en los elementos $q_{ii} = \cos\theta$, $q_{ij} = \text{sen}\theta$, $q_{ji} = -\text{sen}\theta$, y $q_{jj} = \cos\theta$ y $\tan(2\theta) = \frac{2a_{ij}}{a_{ii}-a_{jj}}$

Ejemplo 2.3

$$A = \begin{pmatrix} 2 & 2 & 1 \\ 2 & -1 & -2 \\ 1 & -2 & 2 \end{pmatrix}$$

tomando $a_{21} = 2$ se busca obtener 0 en las posiciones $(2,1)$ y $(1,2)$. Utilizando la matriz Q_1 de rotación de orden 3 de la forma

$$Q_1 = \begin{pmatrix} \cos\theta & -\text{sen}\theta & 0 \\ \text{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

entonces

$$A_2 = Q_1 A_1 Q_1^t =$$

$$\begin{pmatrix} \cos\theta & -\text{sen}\theta & 0 \\ \text{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 2 & 1 \\ 2 & -1 & -2 \\ 1 & -2 & 2 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & \text{sen}\theta & 0 \\ -\text{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

con $\tan(2\theta) = \frac{2a_{ij}}{a_{ii}-a_jj} = \frac{2\cdot 2}{2-(-1)} = \frac{4}{3}$. Mediante este producto se obtiene la matriz

$$A_2 = \begin{pmatrix} 3 & 0 & 0 \\ 0 & -2 & -\sqrt{5} \\ 0 & -\sqrt{5} & 2 \end{pmatrix}$$

Ahora se busca transformar el valor en $(3,2)$, así como en $(2,3)$ en 0. La matriz de rotación en este caso es

$$Q_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\text{sen}\theta \\ 0 & \text{sen}\theta & \cos\theta \end{pmatrix}$$

Por tanto

$$A_3 = Q_2 A_2 Q_2^t =$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\text{sen}\theta \\ 0 & \text{sen}\theta & \cos\theta \end{pmatrix} \cdot \begin{pmatrix} 3 & 0 & 0 \\ 0 & -2 & -\sqrt{5} \\ 0 & -\sqrt{5} & 2 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & \text{sen}\theta & 0 \\ -\text{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

con $\tan(2\theta) = \frac{2a_{ij}}{a_{ii}-a_{jj}} = \frac{2\cdot-\sqrt{5}}{2-(-2)} = \frac{-2\sqrt{5}}{4}$. Con lo cual, se obtiene la matriz

$$A_3 = \begin{pmatrix} 3 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Donde -3 y 3 (con multiplicidad 2) son los valores propios de A y los vectores propios corresponden a las columnas de la matriz

$$Q_1^t Q_2^t = \begin{pmatrix} \cos\theta & \text{sen}\theta & 0 \\ -\text{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos\theta & \text{sen}\theta & 0 \\ -\text{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2.3. Manejo de datos atípicos

Los datos atípicos (*outliers*) ([13], [14], [18]) son aquellas observaciones que pueden ser consideradas no pertenecientes al grupo de datos por su alejamiento al resto de los datos. La importancia en su detección radica en que al considerados en un anális, se afecta notablemente la media y desviaciones de las variables.

Pueden presentarse por varias razones, entre ellas están los errores en la medición, errores en la sustitución de valores ausentes, datos atípicos reales de la población, etc. Si se detecta que el dato proviene de un error, este debe ser eliminado o ser recalculado pero si proviene de datos reales se debe tener cautela en su trato, determinar si es posible su causa y no necesariamente debe ser eliminado. Su trato depende de muchos factores como por ejemplo el tipo de investigación que se realiza, la cantidad de datos atípicos detectados, etc. Una recomendación es que al detectarse uno o varios valores atípicos, se realizen dos análisis de los datos, con y sin los datos atípicos, para poder medir su efecto en los resultados obtenidos e informar acerca del efecto que se tiene en cada caso. En muchas investigaciones, sin embargo, el determinar los datos atípicos puede ser la clave, por ejemplo en el análisis de comportamientos fuera de lo normal como los fraudes.

Un problema que se puede generar al eliminar o no datos atípicos es que al modificarse el conjunto de datos, pueden aparecer nuevamente datos que en apariencia son atípicos pero

en realidad no lo son y caerse en el error de eliminar nuevamente. Por otra parte, la presencia de un solo dato atípico puede estar ocultando la existencia de múltiples datos atípicos.

La detección se puede realizar por medio del análisis de la gráfica (gráficas de puntos, etc.), por medio de pruebas estadísticas o mediante la observación de los datos cuando el grupo es manejable, ya que al ser muy grande se presenta la dificultad del manejo y comparación entre si, ni muy pequeño por la dificultad de discriminar entre ellos.

2.3.1. Pruebas estadísticas.

El detectar un valor atípico implica determinar una medida de su distancia con el resto de datos del grupo. Se recomienda que antes de aplicar algún precodimiento estadístico se realice una observación de los datos, según su valor o gráficamente para respaldar los resultados.

- Prueba de Dixon.

  La prueba de Dixon utiliza las distancias entre los datos. Se calula el valor Q mediante

  $$Q = \frac{\mid x_a - x_p \mid}{\mid x_a - x_d \mid}$$

  donde x_a es el dato que se considera atípico (más lejano del grupo), x_p y x_d son los datos más cercano y distante a x_a respectivamente. El valor del Q debe ser analizado sobre una tabla de datos con valores críticos Q_c, mediante la cual se declara que el dato x_a es atípico si $Q > Q_c$ con cierta probabilidad dada por la tabla.

- Prueba de Grubbs.

  La prueba de Grubbs mide la distancia entre el dato x_a considerado valor atípico y el valor promedio $\bar{x}$, dividida por la desviación estándar s del conjunto de datos. Este estimador T viene dado por

  $$T = \frac{\mid x_a - \bar{x} \mid}{s}.$$

  Si el valor obtendido T es mayor al comparado mediante una tabla, con cierta probabilidad dada, se puede determinar que x_a es un valor atípico.

- Distancia de Mahalanobis.

  La distancia de Mahalanobis es medida al centro de gravedad del grupo de datos, ponderada por la matriz de varianzas y covarianzas. Se espera que en una observación, su distancia no supere el valor de chi-cuadrado para n variables y un nivel de significancia de $0,001$.

- k Rangos intercuartiles.

Considere Q_1 y Q_3 el primer y tercer cuartil, tal que $\Delta Q_3 Q_1$ el intervalo intercuartílico. Se dice que x es un valor extremo si se cumple $x > Q_3 + k(\Delta Q_3 Q_1)$ o bien $x < Q1 - k(\Delta Q_3 Q_1)$. Es usual la asignación $k = 1,5$, sin embargo este valor puede ser modificado según el comportamiento de los datos.

- k Desviaciones estándar.

  Se calcula la distancia entre el valor promedio $\bar{x}$ de los datos y k veces la desviación estándar s, de forma que el

  $$| \bar{x} - ks |$$

  genera una banda, para la cual se consideran atípicos los datos que están fuera de ella. Usualmente se utiliza $k = 2$ ó $k = 3$. Su aplicación depende de la cantidad de datos del conjunto, siendo recomendada su aplicación en conjuntos grandes de datos.

2.4. Representación gráfica

Las componentes principales permiten hacer una representación en pocas dimensiones de los hechos más sobresalientes de una tabla de datos. Las representaciones gráficas pueden ser de dos tipos: mediante planos principales y mediante círculos de correlaciones.

Los planos principales están formados por las coordenadas de los individuos en las componentes principales. En ellos se pueden apreciar las principales agrupaciones y dispersiones de los individuos. El plano definido por las dos primeras componentes es llamado el *primer plano principal* y en general, cualquier plano definido por dos componentes principales es llamado un *plano principal*.

$$\text{Coordenada del individuo } i \text{ en el primero plano es } (c_i^1, c_i^2)$$

Los círculos de correlaciones son obtenidos a partir de las correlaciones entre las variables originales y las componentes principales normalizadas. En ellos se pueden apreciar las agrupaciones de variables y su comportamiento respecto de las componentes principales. Sus construcción se obtiene calculando el coeficiente de correlación lineal entre cada variable x^j y la componente principal c_k correspondiente:

$$\text{Coordenada de la variable } \mathrm{x}^j \text{ en } c^k \text{ es } r(\mathrm{x}^j, c^k)$$

En cualquier interpretación de los gráficos, siempre debe tenerse presente que éstos no son más que simplificaciones de los hechos observados y que ambas gráficas son complementarias.

2.4.1. Representación gráfica mediante R

Se analizaron algunos paquetes libres para realizar la representación gráfica de los resultados obtenidos del ACP entre ellos R, que es un software libre orientado a la graficación y

la estadística ([19]) con grandes facilidades para la manipulación de datos. Sin embargo, se presenta un problema de memoria al trabajar con tablas de datos grandes ya que el programa asigna memoria física para su manipulación, además de que cada objeto en R cuenta con un máximo de tamaño para memoria.

Para solventar este problema existe el paquete *filehash* que permite guardar los objetos en el disco y con esto evita el trabajo en memoria, pero la ejecución se vuelve lenta por las lecturas a disco.

Debido a la limitación que presentan algunos software como R para realizar la representación gráfica de una gran cantidad de datos se decidió implementar un paquete para realizar esta labor. Este paquete construye un archivo de texto en código LaTeXel cual al ser procesado genera un archivo en formato `pdf` con la representación gráfica correspondiente, sea esta el plano principal o el círculo de correlaciones. Por otro lado, para mejorar la visualización de los resultados del ACP se implementó un filtro por variables el cual permite visualizar en el plano principal los individuos que se encuentra en cierto rango de valores, definido por el usuario, para un determinado conjunto de variables.

De forma similar para manejar los datos atípicos se implementó un filtro el cual se aplica antes del ACP para determinar datos *outliers*. Al aplicar el filtro se va etiquetando aquellos individuos que se encuentran fuera del rango, luego de esto el experto toma la decisión de eliminar o no estos datos. El rango para el filtrado de datos se puede construir:

- a partir de la media (μ) y la desviación estándar (σ) de los valores de una variable de la siguiente forma $[\mu - k\sigma, \mu + k\sigma]$, donde $k \in \mathbb{N}$.
- o que el usuario introduzca los valores extremos del rango.

Capítulo 3

Implementación en paralelo

3.1. Introducción

Los avances que se han suscitado en el área de la computación en paralelo han permitido aplicar este paradigma a la solución de problemas muy complejos, en este proyecto se implemente usando esta tecnologia el análisis en componentes principales para trabajar con tablas de datos de gran tamaño, en particular, se aplica el poder de cálculo de los procesadores gráficos GPU's presentes en las tarjetas gráficas para PC ([22]) y en consolas de videojuegos. Este tipo de unidades de proceso gráfico tienen una arquitectura muy sofisticada que explota la idea del paralelismo de una forma innovadora y poco convencional. Usando múltiples procesadores especializados y un extraordinario ancho de banda son capaces de mantener velocidades de cálculo por encima de las CPU's más avanzadas. CUDA (Compute Unified Device Architecture) es un compilador junto con un conjunto de herramientas de desarrollo creadas por NVIDIA que permiten usar una variación del lenguaje de programación C para codificar algoritmos en las unidades de procesamiento gráfico GPU'S.

El modelo de programación CUDA aprovecha el paralelismo que ofrecen los múltiples núcleos de las GPU'S al lanzar un gran número de hilos de forma simultánea. Por ello, si la solución de un problema dado requiere realizar muchas tareas independientes las GPU'S podrían ofrecer un gran rendimiento.

La estructura que se utiliza en este modelo de programación está definido por un grid, dentro del cual hay bloques de hilos (figura 3.1). Esto permite trabajar con matrices bastante "grandes". Aunque el número de bloques en CUDA tiene una restricción de 65 535 bloques en una llamada (por dimensión), así que se tendría la restricción del número de hilos por bloque multiplicado por el número de bloques. Esto no parece importante si se tiene en cuenta que en los dispositivos de computabilidad $2.x$, el máximo número de hilos es de 1024, lo que nos da más de 67 millones de filas. Sin embargo, el problema se presenta con la memoria compartida del dispositivo, y que en algunos casos, para la sincronización de hilos se tiene el límite de 32 hilos, lo cual nos da tan solo poco más de 2 millones de filas ([2]).

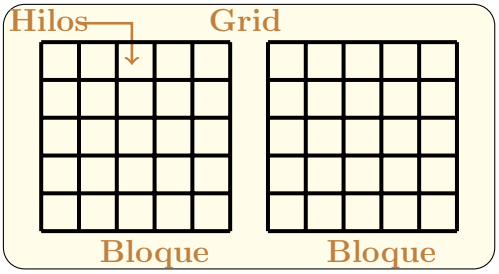

Figura 3.1: Estructura del modelo de programación.

Por otro lado, OpenMP (Open Multiprocessing) es una interfaz de programación de aplicaciones (API), definida conjuntamente por un grupo de programadores y proveedores de software. OpenMP proporciona un modelo portátil y escalable para los desarrolladores de aplicaciones de memoria compartida en paralelo. El API es compatible con C / C ++ y Fortran en una amplia variedad de arquitecturas ([3], [7], [6]).

3.2. Tipos de variables

Se va a trabajar con tipo `float` para los número de punto flotante, pensando en que la precisión no es un factor determinante en el desarrollo de los algoritmos. Utilizar `float` permite compilar código `CUDA` para dispositivos con capacidad de cómputo 1.0 y 1.1, y podría permitir que en dispositivos con mayor capacidad de cómputo el código sea más rápido, a excepción tal vez de código que requiera características de dichos dispositivos para poder ejecutarse.

Además, se va a trabajar con `size_t` para las variables que definen los índices, y en teoría es una referencia para el tamaño de un arreglo, por lo que es necesario además conocer su tamaño máximo, el cual se puede definir mediante:

```
#define MAX_UINT    ((size_t)-1)
```

3.3. Lectura de datos

El primero paso, es leer los datos de un archivo de texto. Se va a leer por columna, puesto que cada columna corresponde a una variable.

El archivo de texto debe comenzar con dos números enteros, que corresponden al número de individuos (filas), y número de variables por individuo (columnas).

```
1 float *leerDatos(char *nombreArchivo) {
2     size_t filas, cols;     // N'umero de filas y columnas.
```

```
3       size_t i, j;            // Contadores.
4       float *valores = NULL;  // Arreglo con los valores.
5       int estadoLectura;      // Estado de lectura.
6
7       // Apertura de archivo.
8       FILE *f = fopen(file, "r");
9       assert(f);
10
11      // Se leen el n'umero de filas y columnas.
12      fscanf(f, "%zu", &filas);
13      fscanf(f, "%zu", &cols);
14
15      // Se verifica que no exceda el m'aximo tama~no posible.
16      assert(MAX_UINT/filas > cols);
17
18      // Se crea el espacio para el arreglo.
19      valores = (float*) malloc (filas * cols * sizeof(float));
20      assert(valores);
21
22      // Se lee el archivo. El indice i se utiliza para la fila actual
23      // y el indice j para la columna actual. En el archivo nos movemos
24      // por fila, pero se guarda por columna.
25      for (i = 0; i < filas; i++) {
26          for (j = 0; j < cols; j++) {
27              estadoLectura = fscanf(f, "%lf", &valores[i + j*filas]);
28              assert(estadoLectura != EOF);
29          };
30      };
31      fclose(f);
32      return valores;
33  };
```

El método recibe como argumento de entrada el nombre de un archivo, y devuelve un puntero a un arreglo de valores de tipo `float`. Se debe recordar liberar el espacio de los valores.

3.4. Programación en CUDA

El método anterior es el único que trabaja directamente con el CPU. El resto de métodos se trabajan en `CUDA`. Para cada uno de ellos se van a trabajar las funciones con un modelo distinto, para comparar tiempos con respecto a los modelos.

Una función en `CUDA` se denomina *kernel*. Un kernel ejecuta N hilos en paralelo, en una estructura de dos niveles. El primer nivel se denomina *Grid*, o rejilla, que puede ser de una o dos dimensiones. El Grid está formado por bloques. Además cada bloque está formado por hilos, que pueden estar configurados en una, dos o tres dimensiones.

3.5. Calcular el promedio μ de cada variable

En lo que sigue, se asume que N es el número de variables (columnas).

3.5.1. Un bloque

Se va a calcular el promedio de cada una de las variables. Para ello, se va a utilizar un hilo de CUDA para cada una de las variables. Primero vamos a trabajar con un solo bloque, pero se debe tener en cuenta que hay un límite de 1024 hilos por bloque.

```
1 __global__ void mu_10(size_t filas, float *valores, float *promedios) {
2     size_t i, icol = (size_t)threadIdx.x, posicion = icol * filas;
3     float suma = 0;
4     for (i = 0; i < filas; i++)
5         suma += valores[i + posicion];
6     promedios[icol] = suma/filas;
7 };
```

La llamada al kernel se hace mediante:

```
mu_10<<<1,N>>>(filas, dev_valores, dev_promedios);
```

donde `filas` es el número de filas, `dev_valores` es el puntero al arreglo de valores en el dispositivo, y `dev_promedios` es el puntero al arreglo de promedios en el dispositivo.

La expresión `<<<1,N>>>` define la estructura de la llamada, donde el primer argumento es la estructura para los bloques, y el segundo la estructura para los hilos. En este caso, la estructura consiste en un bloque con N hilos.

El valor de `threadIdx.x` que se guarda en la variable `icol` nos informa en cuál hilo estamos, por lo que es la columna a la cual se le va a calcular el promedio. La variable `posicion` guarda el valor inicial del índice de la columna con la cual estamos trabajando.

3.5.2. Varios bloques, un hilo por columna

`CUDA` utiliza una arquitectura denominada *SIMT*, que es el acrónimo en inglés por *Instrucción individual, Hilo múltiple* (Single-Instruction, Multiple-Thread). Dicha arquitectura está formada por un arreglo de Multiprocesadores; cada uno crea, gestiona, planifica y ejecuta hilos en paralelo en grupos de 32 hilos. Cada grupo se llama *warp* (urdimbre). Así, vamos a investigar el rendimiento del proceso utilizando diferentes valores para el número de hilos por bloque, en múltiplos de 8 hilos por bloque.

Por ello se va a trabajar con B bloques, cada uno de ellos formado por H hilos, de tal manera que $B \times H \geq N$. Se va a definir entonces H como un múltiplo de 8[1], comenzando en 8 y terminando en 1024, que es el máximo número de hilos por bloque.

[1]El número 8 no es mágico; `CUDA` trabaja con 1 warp, 1/2 warp ó 1/4 de warp.

El número de bloques requerido debe ser tal que $B \times H \geq N$, es decir $B \geq N/H$; como B es un valor entero, entonces $B = \lceil N/H \rceil$.

```
__global__ void mu_11(size_t filas, size_t cols, float *valores, float *promedios) {
    size_t i, icol = (size_t)(threadIdx.x + blockIdx.x * blockDim.x);
    size_t posicion = icol * filas;
    float suma = 0;
    if (icol < cols) {
        for (i = 0 ; i < filas; i++)
            suma += valores[i + posicion];
        promedios[icol] = suma/filas;
    };
};
```

Este código tiene básicamente dos diferencias con respecto al código anterior. La primera de ellas es la forma de calcular la columna a la cual se le calcula el promedio. Cada bloque se encarga de calcular el promedio de H columnas (puesto que cada uno trabaja con H hilos). El valor de H se obtiene mediante `blockDim.x`. El índice del bloque actual se calcula mediante `blockIdx.x`. Como se trabaja 0-indexado, si el bloque tiene índice i, quiere decir que se deben brincar $i \times H$ columnas, y el número de columna actual está dado entonces por `threadIdx.x + blockIdx.x * blockDim.x`.

El segundo cambio, es que $B \times H \geq N$, es decir, podríamos llamar más hilos que el número de columnas; por ello debemos incluir el número de columnas como argumento a la función, y ejecutamos el código solo en caso de que no nos excedamos.

3.5.3. Varios bloques, varios hilos por columna

El trabajo en `CUDA` se justifica si el tamaño de la matriz es grande. En caso contrario, es más eficiente trabajar con los CPU's, porque solamente la primera invocación a `CUDA` puede tomar hasta 5 segundos. Supongamos que tenemos un caso en el cual tenemos pocas variables, pero un gran número de individuos. Así, tal vez sea más eficiente hacer que más de un hilo trabajen en una variable. El siguiente código muestra la implementación en la cual se llaman N bloques, cada bloque con H hilos.

```
__global__ void mu_20(size_t filas, float *valores, float *promedios) {
    size_t hilo = (size_t)threadIdx.x;
    size_t temp = (filas - 1)/blockDim.x + 1;
    size_t i = hilo * temp, fin = i + temp;
    if (fin > filas)
        fin = filas;
    float suma_parcial = 0.0;
    __shared__ float suma_total;
    if (hilo == 0)
        suma_total = 0;
    temp = (size_t)blockIdx.x * filas;
```

```
12      for( ; i < fin; i++)
13          suma_parcial += valores[i + temp];
14      atomicAdd(&suma_total, suma_parcial/filas);
15      __syncthreads();
16      if (hilo == 0)
17          promedios[blockIdx.x] = suma_total;
18  };
```

Este código tiene varias instrucciones nuevas. La instrucción `__shared__ float suma_total` define una variable que se comparte entre los mismos hilos de un bloque, lo cual se instruye mediante el cualificador `__shared__`. Las variables compartidas no se pueden inicializar al momento de definirse; es por ello que se define en la siguiente instrucción, pero de tal manera que sea solamente uno de los hilos el que realice la inicialización.

Como se van a utilizar H hilos para calcular el promedio, lo que hacemos es partir los elementos, de manera que cada hilo calcule una fracción de la suma. En general, cada hilo debería calcular la suma de N/H elementos; pero como se requiere que sea un número entero, entonces el número de elementos $\lceil N/H \rceil$ se guarda en `temp`.

Al igual que en el caso en el que revisábamos no excedernos en el número de la columna, aquí nos aseguramos de que no se exceda el número de filas. Cada hilo guarda la suma parcial en la variable `suma_parcial`. Luego, y para evitar *condiciones de competición* (race conditions), con una instrucción *atómica* se actualiza el valor de la variable compartida `suma_total`; ello funciona siempre y cuando el tipo a utilizar sea `float`, dado que la función `atomicAdd` no está definida para variables de tipo `double`.

Finalmente, hacemos que uno de los hilos guarde el valor, pero debemos asegurarnos primero que *todos* los hilos hayan terminado, para ello sincronizamos todos los hilos con la función `__syncthreads()`. Dicha función es muy costosa, por lo que en la medida de lo posible hay que utilizarla lo menos posible, y hay que estar seguros que *todos* los hilos lleguen a ella (es decir, por ejemplo, que no está incluida dentro de un bloque que no es ejecutado por todos los hilos).

3.5.4. Ancho de banda de la memoria global

Uno de los aspectos más importantes en el rendimiento de la programación en CUDA es el acceso de datos a la memoria global. Las aplicaciones CUDA explotan paralelismo masivo de datos; es decir, procesan una cantidad masiva de datos que son extraidos de la memoria global en un periodo de tiempo muy corto.

En un sistema `CUDA`, la memoria global se implementa utilizando memorias aleatorias de acceso dinámico (DRAMs por sus siglas en inglés). Los bits se almacenan en celdas DRAM, las cuales son capacitores muy débiles, donde la presencia o ausencia de una mínima cantidad de carga eléctrica distingue entre un 0 y un 1. Leer datos de una celda DRAM que contiene un 1 requiere que el debil capacitor comparta su leve cantidad de carga con un sensor y active un mecanismo de detección que determina si una cantidad de carga suficiente está presente en

el capacitor. Debido a que este es un proceso muy lento, los DRAMs utilizan un mecanismo en paralelo para incrementar la tasa de acceso a los datos. Cada vez que una localidad es accesada, muchas localidades consecutivas son accesadas. Si una aplicación puede acceder a múltiples localidades consecutivas de datos, los DRAMs pueden proveer los datos a una tasa mucho mayor que en el caso de datos localizados en secuencias aleatorias. Por ello, se va a modificar levemente el algoritmo para que los hilos obtengan valores que estén contiguos entre sí, en lugar de valores por bloque.

```
1  __global__ void mu_21(size_t filas, float *valores, float *promedios) {
2      size_t i, hilo = (size_t)threadIdx.x, salto = (size_t)blockDim.x;
3      float suma_parcial = 0;
4      __shared__ float suma_total;
5      size_t posicion = (size_t)blockIdx.x * filas;
6      if (hilo == 0)
7          suma_total = 0;
8      for (i = hilo; i < filas; i += salto)
9          suma_parcial += valores[i + posicion];
10     atomicAdd(&suma_total, suma_parcial/filas);
11     __syncthreads();
12     if (hilo == 0)
13         promedios[blockIdx.x] = suma_total;
14 };
```

El valor `blockDim.x` ya se había utilizado en el ejemplo anterior. Se refiere al número de hilos que tiene el bloque; se guarda en la variable `salto`. Así, cada hilo comienza a determinar el valor correspondiente, y luego se brinca el número de hilos para obtener el valor siguiente, hasta que se llegue hasta la última fila.

3.5.5. Memoria de textura

La *textura* es una técnica utilizada en la visualización de objetos por computadora. Las tarjetas nvidia tienen características de hardware que facilitan la textura, en el acceso a la memoria como memoria de textura y de *superficie*. Dicho acceso presupone cierta localidad espacial de los accesos de memoria, en cuyo caso podría ser más eficiente el acceso. Lo que se hace es asignar cierta sección de la memoria global como *memoria de textura*; de dicha memoria se puede leer pero no se puede escribir (al menos no de manera directa, y se asume constante).

Para ello, primero vamos a crear la *referencia* a la memoria de textura. Dicha referencia debe ser una variable global, por lo que se debe definir afuera de cualquier función:

```
texture<float, cudaTextureType1D, cudaReadModeElementType> texRef;
```

El primer argumento es el tipo de variable, que puede ser `[unsigned] int`, `float` o vectores de 1, 2 y 4 componentes predefinidos en CUDA, lo cual se puede leer en la guía de programación, sección B.3.1. El segundo argumento es la dimensión de la memoria, cambiando el 1D

por 2D o 3D según sea. El tercer argumento puede ser también `cudaReadNormalizedFloat` para tipos enteros de 8 o 16 bits. Devuelve un valor de tipo `float` normalizado entre 0.0 y 1.0 si es de tipo `unsigned`, y entre -1.0 y 1.0 en caso contrario.

Luego, dentro del código, relacionamos la referencia, con la variable a la cual referencia:

```
cudaBindTexture(NULL, texRef, dev_valores, filas * cols * sizeof(float));
```

Finalmente, en la función no se requiere pasar ya la variable `valores`, pues es una referencia global:

```
__global__ void mu_22(size_t filas, float *promedios) {
    size_t i, hilo = (size_t)threadIdx.x, salto = (size_t)blockDim.x;
    float suma_parcial = 0;
    __shared__ float suma_total;
    size_t posicion = (size_t)blockIdx.x * filas;
    if (hilo == 0)
        suma_total = 0;
    for (i = hilo; i < filas; i += salto)
        suma_parcial += tex1Dfetch(texRef, i + posicion);
    atomicAdd(&suma_total, suma_parcial/filas);
    __syncthreads();
    if (hilo == 0)
        promedios[blockIdx.x] = suma_total;
};
```

El código es exactamente igual al anterior, con la diferencia de que para acceder a la memoria se utiliza la función `tex1Dfetch`, cuyos parámetros son la referencia a la textura, y el índice respectivo.

En dispositivos de capacidad 2.0, la memoria de textura pareciera no tener un impacto positivo para los casos en una dimensión.

3.6. Calcular la desviación estándar σ de cada variable

Ahora lo que sigue es determinar la desviación estándar de cada variable, que está dada por la fórmula:

$$\sigma = \sqrt{E[(x_i - \overline{x})^2]} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}}.$$

Se utiliza la misma técnica que se utilizó en la Sección 3.5.4, para que el conjunto de hilos lean entre sí datos contiguos. Como el valor de la media es común al bloque, entonces no es necesario que cada hilo obtenga el valor, sino que se guarda en la memoria compartida.

```
__global__ void sigma(size_t filas, float *valores, float *promedios, float *desvStds) {
    size_t i, hilo = (size_t)threadIdx.x, salto = (size_t)blockDim.x;
    float temp, suma_parcial = 0;
```

```
4       size_t posicion = (size_t)blockIdx.x * filas;
5       __shared__ float suma_total;
6       __shared__ float mu;
7       if (hilo == 0) {
8           suma_total = 0;
9           mu = promedios[blockIdx.x];
10      };
11      for (i = hilo; i < filas; i += salto) {
12          temp = valores[i + posicion] - mu;
13          suma_parcial += temp * temp;
14      };
15      atomicAdd(&suma_total, suma_parcial);
16      __syncthreads();
17      if (hilo == 0)
18          desvStds[blockIdx.x] = sqrt(suma_total);
19  };
```

En realidad no se está calculando la desviación estándar, porque no se dividieron los valores entre n, sino lo que se está calculando es $\sigma\sqrt{n}$, que lo vamos a denotar por σ^*.

3.7. Matriz de correlaciones

Se debe construir ahora la matriz de correlaciones $R = [r_{i,j}]$, de tamaño $N \times N$. Dicha matriz es simétrica, y su diagonal está formada por 1's. Así, $r_{i,i} = 1$ y, en general:

$$r_{i,j} = r_{j,i} = \frac{\sum_{k=1}^{M}(x_k^i - \mu_i)(x_k^j - \mu_j)}{\sigma_i^* \sigma_j^*},$$

donde $\sigma_i^* = \sqrt{n}\,\sigma_i$.

```
1   __global__ void correl_1(size_t filas, float *valores, float *promedios,
2           float *desvStds, float *correls) {
3       size_t icol = (size_t)blockIdx.x;
4       size_t hilo = (size_t)threadIdx.x;
5       size_t salto = (size_t)blockDim.x;
6       size_t jposicion, iposicion = icol * filas;
7       float suma_parcial = 0;
8       __shared__ float mu_i;
9       __shared__ float mu_j;
10      __shared__ float ds_i;
11      __shared__ float suma_total;
12      // Inicializaci'on de variables compartidas.
13      if (hilo == 0) {
14          suma_total = 0;
15          mu_i = promedios[icol];
16          ds_i = desvStds[icol];
17          correls[icol * gridDim.x + icol] = (float)1.0;
```

```
18      };
19      __syncthreads();
20      size_t k, j;
21      // Columna $i$-'esima. Se calcula la covarianza r_{i,j} para j > 1.
22      for (j = icol + 1; j < filas; j++) {
23          if (hilo == 0)
24              mu_j = promedios[j];
25          jposicion = j * filas;
26          for (k = hilo, suma_parcial = 0; k < filas; k += salto)
27              suma_parcial +=
28                  (valores[k + iposicion] - mu_i)*(valores[k + jposicion] - mu_j);
29          atomicAdd(&suma_total, suma_parcial);
30          __syncthreads();
31          if (hilo == 0) {
32              suma_parcial = suma_total/(ds_i * desvStds[j]);
33              correls[icol * gridDim.x + j] = suma_parcial;
34              correls[j * gridDim.x + icol] = suma_parcial;
35          };
36      };
37  };
```

En el algoritmo anterior, dado que el trabajo en las columnas no es el mismo, el desbalance podría influir en el tiempo de ejecución. Así, vamos a hacer que cada bloque trabaje con dos columnas, de manera que el trabajo quede balanceado. Debemos por lo tanto conocer el número total de columnas. El código consiste en replicar el código anterior: el primer caso igual, y el segundo la columna "simétrica", de manera que el número total de casos sea siempre el mismo.

```
1   __global__ void correl_2 (size_t filas, size_t cols, float *valores,
2           float *promedios, float *desvStds, float *correls) {
3       size_t icol = (size_t)blockIdx.x;
4       size_t hilo = (size_t)threadIdx.x;
5       size_t salto = (size_t)blockDim.x;
6       size_t jposicion, iposicion = icol * filas;
7       float suma_parcial = 0;
8       __shared__ float mu_i;
9       __shared__ float mu_j;
10      __shared__ float ds_i;
11      __shared__ float suma_total;
12      // Inicializaci'on de variables compartidas.
13      if (hilo == 0) {
14          suma_total = 0;
15          mu_i = promedios[icol];
16          ds_i = desvStds[icol];
17          correls[icol * gridDim.x + icol] = (float)1.0;
18      };
19      __syncthreads();
20      size_t k, j;
21      // Columna $i$-'esima. Se calcula la covarianza r_{i,j} para j > 1.
```

```
22      for (j = icol + 1; j < filas; j++) {
23          if (hilo == 0)
24              mu_j = promedios[j];
25          jposicion = j * filas;
26          for (k = hilo, suma_parcial = 0; k < filas; k += salto)
27              suma_parcial +=
28                  (valores[k + iposicion] - mu_i)*(valores[k + jposicion] - mu_j);
29          atomicAdd(&suma_total, suma_parcial);
30          __syncthreads();
31          if (hilo == 0) {
32              suma_parcial = suma_total/(ds_i * desvStds[j]);
33              correls[icol * gridDim.x + j] = suma_parcial;
34              correls[j * gridDim.x + icol] = suma_parcial;
35          };
36      };
37      // Ahora se trabaja con la segunda columna. Se redefinen las variables.
38      if (cols % 2 == 0) { // El n'umero de columnas es par.
39          icol = cols - icol - 1;
40      } else { // El n'umero de columnas es impar.
41          icol = cols - icol - 2;
42          // En el caso de columnas impares la 'ultima columna
43          // queda por afuera.
44          if (hilo == 0)
45              correls[(cols - 1) * gridDim.x + (cols - 1)] = (float)1.0;
46      };
47      iposicion = icol * filas;
48      // Inicializaci'on de variables compartidas.
49      if (hilo == 0) {
50          suma_total = 0;
51          mu_i = promedios[icol];
52          ds_i = desvStds[icol];
53          correls[icol * gridDim.x + icol] = (float)1.0;
54      };
55      __syncthreads();
56      // Columna $i$-'esima. Se calcula la covarianza r_{i,j} para j > 1.
57      for (j = icol + 1; j < filas; j++) {
58          if (hilo == 0)
59              mu_j = promedios[j];
60          jposicion = j * filas;
61          for (k = hilo, suma_parcial = 0; k < filas; k += salto)
62              suma_parcial +=
63                  (valores[k + iposicion] - mu_i)*(valores[k + jposicion] - mu_j);
64          atomicAdd(&suma_total, suma_parcial);
65          __syncthreads();
66          if (hilo == 0) {
67              suma_parcial = suma_total/(ds_i * desvStds[j]);
68              correls[icol * gridDim.x + j] = suma_parcial;
69              correls[j * gridDim.x + icol] = suma_parcial;
70          };
71      };
```

```
};
```

3.7.1. Matriz triangular

Dado que la matriz es simétrica, no es necesario trabajar con la matriz completa, sino solamente con la mitad de ella. Ello nos ahorraría algunos accesos a la memoria global para modificar los valores de la matriz de correlaciones tanto en la entrada (i, j) como en la entrada (j, i).

¿Cómo se guarda entonces la matriz? Hasta ahora se había guardado la información por columnas para los datos. En el caso de la matriz de correlaciones da igual, porque la matriz es simétrica. En este caso es parecido: hay que decidir cuál de las mitades se guarda y en que forma. Vamos a guardar la matriz triangular superior, y en orden de filas. Es decir, el vector se guarda de la siguiente manera:

$$0,0|0,1|\ldots|0,n-1|1,1|1,2|\ldots|1,n-1|\ldots|n-2,n-2|n-2,n-1|n-1,n-1$$

El tamaño del vector es fácil de determinar, pues la primera fila tiene n elementos, la segunda $n-1$, y así sucesivamente hasta la n-ésima fila que tiene un elemento, por lo que en total son $n(n+1)/2$ elementos. Tal vez la dificultad se debe a determinar la posición del i, j-ésimo elemento de la matriz (0 indexado, con $i \leq j$). Observe que en la i-ésima fila, el i-ésimo elemento está de primero. El primer elemento de la fila 1 tiene índice n; el de la fila 2 tiene índice $n+n-1 = 2n-1$; el de la fila 3 tiene índice $2n-1+n-2 = 3n-3$; el de la fila 4 $3n-3+n-3 = 4n-6$; así el de la fila i-ésima tiene índice $in-i(i-1)/2$. Para acceder a la j-ésima columna, simplemente sumamos $j-i$. Así, la posición del i, j-ésimo elemento está dada por $in-i(i-1)/2+j-i$.

Vamos entonces a definir un par de funciones que nos permitan acceder a los valores, sin preocuparnos de escribir la expresión cada vez.

```
__device__ void setSim(float valor, size_t i, size_t j, float *matriz) {
    if (i <= j)
        matriz[i*n - i*(i-1)/2 + j - i] = valor;
    return;
};

__device__ float getSim(size_t i, size_t j, float *matriz) {
    if (i <= j)
        return matriz[i*n - i*(i-1)/2 + j - i];
    else
        return matriz[j*n - j*(j-1)/2 + i - j];
};
```

Como se puede observar, en lugar de `__global__`, estas funciones se definen con `__device__`. Estas funciones se pueden llamar desde los dispositivos, y pueden tener valores de retorno. Los cambios a `correls` son mínimos, por lo que no vamos a presentarlos acá, pero llamaremos a las funciones `correls_1b` y `correls_2b`.

3.7.2. Reducción de tráfico de la memoria global

Se tiene un compromiso intrínsico en el uso de la memoria de los dispositivos en `CUDA`. La memoria global es grande pero lenta, mientras que la memoria compartida es pequeña pero rápida. Una estrategia común es partir los datos en subconjuntos de datos, de tal manera que pueda ser almacenado en la memoria compartida.

Se quieren calcular los datos de la siguiente matriz:

$$\begin{pmatrix} 1 & r_{0,1} & r_{0,2} & r_{0,3} & \dots & r_{0,n-1} \\ & 1 & r_{1,2} & r_{1,3} & \dots & r_{1,n-1} \\ & & 1 & r_{2,3} & \dots & r_{2,n-1} \\ & & & \ddots & & \vdots \\ & & & & 1 & r_{n-2,n-1} \\ & & & & & 1 \end{pmatrix}$$

donde:

$$r_{i,j} = \frac{\sum_{k=1}^{M}(x_k^i - \mu_i)(x_k^j - \mu_j)}{\sigma_i^* \sigma_j^*},$$

Como se puede observar, en cada fila se comparten los mismos datos, así como también en cada columna. Vamos entonces a hacer lo siguiente: vamos a partir la matriz en una cuadrícula, donde cada cuadro tenga `TAM_CUADRO` filas y `TAM_CUADRO` columnas, donde `TAM_CUADRO` es una constante (definida mediante la directiva del preprocesador `#define`). Eso lo vamos a realizar mediante una partición de bloques en `CUDA`. En cada cuadro, además, van a trabajar `TAM_CUADRO x TAM_CUADRO = NUM_VALORES` hilos, donde la constante `NUM_VALORES` será el número de valores que se cargarán en la memoria compartida. De cada cuadro se necesitarán cargar a lo sumo `2 x TAM_CUADRO = NUM_CASOS`.

Por ejemplo supongamos que se tienen que calcular los siguientes datos con la siguiente partición:

Se requirere entonces la siguiente definición:

```
#define TAM_CUADRO 2 // Dimension del cuadro y el bloque.
#define NUM_CASOS 4 // = 2 x TAM_CUADRO.
#define NUM_VALORES 4 // = TAM_CUADRO x TAM_CUADRO
```

La notación B(x) se refiere al bloque que se va a encargar de la partición. Observe que si el índice del bloque es x, entonces para determinar la fila del bloque se utiliza la ecuación:

$$ibx = \left\lfloor \frac{1 + 2n - \sqrt{(1+2n)^2 - 8x}}{2} \right\rfloor$$

Dicha ecuación se obtiene del hecho de que $ibx \cdot n - ibx(ibx-1)/2 < x$; se resuelve entonces la desigualdad cuadrática respectiva, y se toma el máximo valor antes de que la solución sea negativa.

0,0		0,1	0,2		0,3	0,4		0,5	0,6	
	B(0)			B(1)			B(2)			B(3)
		1,1	1,2		1,3	1,4		1,5	1,6	
			2,2		2,3	2,4		2,5	2,6	
				B(4)			B(5)			B(6)
					3,3	3,4		3,5	3,6	
						4,4		4,5	4,6	
							B(7)			B(8)
								5,5	5,6	
									6,6	
										B(9)

Cuadro 3.1: Ejemplo del uso de la memoria compartida.

Determinado el índice de la fila, el índice de la columna se determina mediante:

$$\begin{aligned} iby &= x + ibx - ibx * n + \frac{ibx(ibx-1)}{2} \\ &= x + ibx\left(1 - n + \frac{ibx-1}{2}\right) \\ &= x + ibx \cdot \frac{2 - 2n + ibx - 1}{2} \\ &= x + ibx \cdot \frac{1 + ibx - 2n}{2} \end{aligned}$$

Además, las filas que analiza son $ibx \times$ `TAM_CUADRO` hasta $(ibx+1) \times$ `TAM_CUADRO` -1, siempre y cuando el índice de fila sea menor que F, donde F es el número de filas (columnas de los datos); de manera similar para las columnas.

3.8. Norma de Frobenius para una matriz

Dada una matriz cualquiera $A = [a_{ij}]$ de tamaño $m \times n$, la norma de Frobenius se define como:

$$F(A) = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2}.$$

Para el algoritmo de Jacobi, el cual trabaja con matrices cuadradas, es necesario implementar una idea similar, en la que no se toma en cuenta la diagonal. En inglés se le llama

off-norm de una matriz, y se calcula mediante:

$$\text{off}(A) = \sqrt{\sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij}^2}.$$

Dado que se va a trabajar con matrices simétricas, entonces puede calcularse mediante:

$$\text{off}(A) = \sqrt{\sum_{i=1}^{n}\sum_{j=i+1}^{n} 2a_{ij}^2}.$$

La pregunta es cómo hacer para que la repartición se realice de manera justa, dado que no requiere calcular toda la fila. Una posibilidad es trabajar con filas *complementarias*, de tal manera que al tomar un par de filas complementarias, el número de cálculos a realizar sea siempre el mismo.

Se asume que la matriz está ordenada por filas (aunque como es simétrica, no hay ninguna diferencia). Sea N el número de filas (y columnas) de la matriz. Si N es par, entonces vamos a emparejar las siguientes parejas de índices: $(0, N-1), (1, N-2), \ldots, (N/2-1, N/2)$ (recordemos que las filas tienen índices $0, 1, \ldots, N-1$). Por otro lado, si N es impar, entonces las filas complementarias van a ser: $(0, N-2), (1, N-3), \ldots, ((N-3)/2, (N-1)/2)$.

Primero se presenta un algoritmo donde se utiliza 1 bloque y $H = \lfloor N/2 \rfloor$ hilos.

```
__global__ void off_1(size_t n, float *matriz, float *off) {
    size_t i, fila = threadIdx.x;
    float temp, suma_parcial = 0, posicion = fila * n;
    __shared__ float suma_total;
    if (fila == 0) suma_total = 0;
    // Sumando los elementos de la primera fila.
    for (i = fila + 1; i < n; i++) {
        temp = matriz[i + posicion];
        suma_parcial += 2 * temp * temp;
    };
    /I Calculando la fila *complementaria*.
    if (n % 2 == 0) fila = n - fila - 1;
    else fila = n - fila - 2;
    posicion = fila * n;
    */
    fila = n - fila - 1 - n&1;
    // Sumando los elementos de la fila complementaria.
    for (i = fila + 1; i < n; i++) {
        temp = matriz[i + posicion];
        suma_parcial += 2 * temp * temp;
    };
    atomicAdd(&suma_total, suma_parcial);
    __syncthreads();
```

```
24      if (threadIdx.x == 0) *off = sqrt(suma_total);
25  };
```

También se podría pensar en utilizar $B = \lfloor N/2 \rfloor$ bloques, y H hilos por bloque.

```
1   __global__ off_2(size_t n, float *matriz, float *off) {
2       __shared__ float suma_bloque;
3       if (threadIdx.x == 0)
4           suma_bloque = 0;
5       float temp, suma_hilo = 0;
6       size_t salto = (size_t)blockDim.x;
7       size_t i, fila = (size_t)blockIdx.x;
8       size_t posicion = fila * n;
9       for( i = (size_t)threadIdx.x + fila + 1; i < n; i += salto) {
10          temp = matriz[i + posicion];
11          suma_hilo += 2 * temp * temp;
12      };
13      fila = n - fila - 1 - n&1;
14      posicion = fila * n;
15      for( i = (size_t)threadIdx.x + fila + 1; i < n; i += salto) {
16          temp = matriz[i + posicion];
17          suma_hilo += 2 * temp * temp;
18      };
19      atomicAdd(&suma_bloque, suma_hilo);
20      __syncthreads();
21      if (threadIdx.x == 0)
22          atomicAdd(off, suma_bloque);
23  };
```

En este caso se está utilizando `atomicAdd` para modificar una variable global. Podría no ser muy eficiente si son muchos bloques (es decir, si el número de hilos por bloque es muy pequeño), puesto que el acceso a una variable global es demasiado lento.

3.9. Valores propios

En muchos campos de la ingeniería y las matemáticas surge el problema de calcular los valores y vectores propios de una matriz, en particular para diagonalizar una matrices es necesario determinar todos sus valores y vectores propios.

3.9.1. Teoría básica

La descomposición en valores propios para una matriz simétrica A de tamaño $n \times n$, consiste en encontrar la matriz diagonal $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$ y la matriz ortonormal U, ambas de tamaño $n \times n$, tales que $A = U\Lambda U^T$, donde U^T es la matriz transpuesta de U. Una matriz ortonormal es aquella tal que $UU^T = U^T U = I_n$, es decir, $U^{-1} = U^T$.

Sea $\overline{u}_{*j}$ la $j-$ésima columna de la matriz U. Entonces se cumple que $\lambda_j \overline{u}_{*j} = A\overline{u}_{*j}$; es decir, λ_j es un valor propio de A y $\overline{u}_{*j}$ es el vector propio respectivo.

La demostración es como sigue. $AU = U\Lambda U^T U = U\Lambda I_n = U\Lambda$, puesto que $U^T U = I_n$. Ahora:

$$(AU)[i,j] = \sum_k A[i,k]U[k,j] = A[i,*]U[*,j]$$
$$\Rightarrow (AU)[*,j] = A[*,*]U[*,j] = AU[*,j] = A\overline{u}_{*j}$$

Ahora calculemos la columna $j-$ésima de la matriz $U\Lambda$:

$$(U\Lambda)[i,j] = \sum_k U[i,k]\Lambda[k,j] = \lambda_j U[i,j]$$
$$\Rightarrow (U\Lambda)[*,j] = \lambda_j U[*,j] = \lambda_j \overline{u}_{*j}$$

Sea ahora U una matriz ortonormal cualquiera. Entonces los valores propios de UAU^T son los mismos de la matriz A. Para ello, dado que $UU^T = I$, entonces $\lambda I = \lambda UU^T = U\lambda IU^T$. Así:

$$|UAU^T - \lambda I| = |UAU^T - U\lambda IU^T| = |U| \cdot |A - \lambda I| \cdot |U^T|,$$

y como $U^T = U^{-1}$, entonces $|U^T| = |U|^{-1}$, por lo que $|UAU^T - \lambda I| = |A - \lambda I|$.

Finalmente, sean U y V dos matrices ortonormales de tamaño $n \times n$. Entonces UV es ortonormal:

$$(UV)(UV)^T = (UV)(V^T U^T) = U(VV^T)U^T = UIU^T = UU^T = I.$$

3.9.2. Rotación de Givens

La rotación de Givens es una matriz de tamaño $n \times n$, tal que para $1 \leq p < q \leq n$, se define $G(p,q,\theta) = [g_{ij}]$ de la siguiente manera:

$$g_{ii} = \begin{cases} c & \text{si } i = p \vee i = q \\ 1 & \text{en caso contrario.} \end{cases}$$
$$g_{ij} = \begin{cases} s & \text{si } i = p \wedge j = q \\ -s & \text{si } i = q \wedge j = p \\ 0 & \text{en caso contrario.} \end{cases}$$

donde $c = \cos\theta$ y $s = \operatorname{sen}\theta$.

Dicha matriz es ortogonal, pues:

$$GG^T[i,j] = \sum_k G[i,k]G^T[k,j] = \sum_k G[i,k]G[j,k] = \sum_k g_{ik}g_{jk}$$

Al analizar varios casos:

$$GG^T[p,p] = \sum_k g_{pk}g_{pk} = g_{pp}^2 + g_{pq}^2 = \cos^2\theta + \operatorname{sen}^2\theta = 1$$
$$GG^T[q,q] = \sum_k g_{qk}g_{qk} = g_{qp}^2 + g_{qq}^2 = (-\operatorname{sen}\theta)^2 + \cos^2\theta = 1$$
$$GG^T[i,i] = \sum_k g_{i,k}g_{ik} = g_{ii}^2 = 1 \qquad i \neq p \wedge i \neq q$$
$$GG^T[p,q] = \sum_k g_{pk}g_{qk} = g_{pp}g_{qp} + g_{pq}g_{qq} = -\cos\theta\operatorname{sen}\theta + \operatorname{sen}\theta\cos\theta = 0$$
$$GG^T[q,p] = \sum_k g_{qk}g_{pk} = g_{qp}g_{pp} + g_{qq}g_{pq} = 0$$
$$GG^T[i,j] = \sum_k g_{ik}g_{jk} = 0 \qquad j \neq i; (i,j) \notin \{(p,q),(q,p)\}$$

Vamos a calcular el producto $A' = AG^T$, donde A es simétrica:

$$a'_{ij} = \sum a_{ik}g^T_{kj} = \sum a_{ik}g_{jk}$$
$$a'_{ip} = \sum a_{ik}g_{pk} = a_{ip}g_{pp} + a_{iq}g_{pq} = ca_{ip} + sa_{iq}$$
$$a'_{iq} = \sum a_{ik}g_{qk} = a_{ip}g_{qp} + a_{iq}g_{qq} = -sa_{ip} + ca_{iq}$$
$$a'_{ij} = \sum a_{ik}g_{jk} = a_{ij}g_{jj} = a_{ij} \qquad j \notin \{p,q\}.$$

Y luego se calcula $A'' = GA' = GAG^T$:

$$a''_{ij} = \sum g_{ik}a'_{kj}$$
$$a''_{pj} = \sum g_{pk}a'_{kj} = g_{pp}a'_{pj} + g_{pq}a'_{qj} = ca'_{pj} + sa'_{qj}$$
$$a''_{qj} = \sum g_{qk}a'_{kj} = g_{qp}a'_{pj} + g_{qq}a'_{qj} = -sa'_{pj} + ca'_{qj}$$
$$a''_{ij} = \sum g_{ik}a'_{kj} = g_{ii}a'_{ij} = a'_{ij} \qquad i \notin \{p,q\}$$
$$a''_{pp} = ca'_{pp} + sa'_{qp} = c(ca_{pp} + sa_{pq}) + s(ca_{qp} + sa_{qq}) = c^2a_{pp} + 2csa_{pq} + s^2a_{qq}$$
$$a''_{qq} = -sa'_{pq} + ca'_{qq} = -s(-sa_{pp} + ca_{pq}) + c(-sa_{qp} + ca_{qq}) = s^2a_{pp} - 2csa_{pq} + c^2a_{qq}$$
$$a''_{pq} = ca'_{pq} + sa'_{qq} = c(-sa_{pp} + ca_{pq}) + s(-sa_{qp} + ca_{qq}) = -csa_{pp} + (c^2 - s^2)a_{pq} + csa_{qq}$$

Se quiere que $a_{pq} = 0$. Para ello se había utilizado la condición que $c^2 + s^2 = 1$. Si se toma $t = s/c$, entonces:

$$\frac{cs}{c^2 - s^2} = \frac{c^2t}{c^2 - c^2t^2} = \frac{t}{1 - t^2}.$$

Luego

$$\begin{aligned}
&\frac{cs(a_{qq}-a_{pp})+(c^2-s^2)a_{pq}}{c^2-s^2}=0\\
&\Rightarrow \frac{t}{1-t^2}(a_{qq}-a_{pp})+a_{pq}=0\\
&\Rightarrow (a_{qq}-a_{pp})t+a_{pq}(1-t^2)=0\\
&\Rightarrow a_{pq}t^2-(a_{qq}-a_{pp})t-a_{pq}=0
\end{aligned}$$

Resolvien la cuadrática, se tiene que:

$$t=\frac{(a_{qq}-a_{pp})\pm\sqrt{(a_{qq}-a_{pp})^2+4a_{pq}^2}}{2a_{pq}}$$

Para asegurarnos que s y c sean positivos, basta con que t sea positivo, así:

$$t=\begin{cases}
\dfrac{(a_{qq}-a_{pp})+\sqrt{(a_{qq}-a_{pp})^2+4a_{pq}^2}}{2a_{pq}}, & \text{si } a_{pq}>0\\
\dfrac{(a_{qq}-a_{pp})-\sqrt{(a_{qq}-a_{pp})^2+4a_{pq}^2}}{2a_{pq}}, & \text{si } a_{pq}<0
\end{cases}$$

Una forma alternativa es la siguiente:

$$\tau=\frac{a_{pp}-a_{qq}}{2a_{pq}} \qquad\qquad t=\frac{\text{signo}(\tau)}{|\tau|+\sqrt{1+\tau^2}}$$

Luego, como $s=ct$ y $c^2+s^2=1$, entonces:

$$c^2+c^2t^2=c^2(1+t^2)=1\Rightarrow c=\sqrt{\frac{1}{1+t^2}}.$$

La matriz GAG^T es simétrica, pues $(GAG^T)^T=(G^T)^TA^TG^T=GAG^T$. Así, si $A''=GAG^T$, entonces:

$$a''_{ij}=a''_{ji}=\begin{cases}
c^2a_{pp}+2csa_{pq}+s^2a_{qq} & \text{si } (i,j)=(p,p)\\
c^2a_{pp}-2csa_{pq}+s^2a_{qq} & \text{si } (i,j)=(q,q)\\
0 & \text{si } (i,j)=(p,q)\\
ca_{ip}+sa_{iq} & \text{si } j=p, i\notin\{p,q\}\\
-sa_{ip}+ca_{iq} & \text{si } j=q, i\notin\{p,q\}\\
a_{ij} & \text{en caso contrario}
\end{cases}$$

3.10. Resultados

El algoritmo diseñado fue implementado combinando los paradigmas `CUDA` y `OpenMP` y con el se pueden realizar las siguientes operaciones sobre la tabla de datos:

1. Operaciones previo a realizar el ACP:

   a) Estadísticas de una variable elegida por el usuario. Imprime la media, la mediana, la desviación estándar, el mínimo, el máximo, el primer cuartil, el tercer cuartil y el rango intercuartil.

   b) Detección de posibles individuos atípicos distantes de la media. Analiza el comportamiento de cada individuo en cada una de las variables y muestra el número de veces que un individuo queda afuera de los rangos establecidos para cada una las variable. Los rangos son definidos de la siguiente forma $[\mu - k\sigma, \mu + k\sigma]$ donde $k \in \mathbb{R}$ es un valor introducido por el usuario, μ es la media y σ es la desviación estándar.

   c) Detección de posibles individuos atípicos por medio de un rango. Muestra los individuos que están fuera de un rango dado por el usuario y para una variable también dada por el usuario.

   d) Eliminar individuo(s) por índice. Dada una lista de índices separados por espacios. elimina todos los individuos cuyos índices están en la lista.

   e) Eliminar individuo(s) fuera de rango. Dado un rango para una variable elimina todos los individuos que se salen de dicho rango.

   f) Eliminar variable(s) por indice. Dada una lista de índices separados por espacios. elimina todos las variables cuyos índices están en la lista.

2. Operaciones para realizar el ACP

   a) Determina la media y la desviación estándar para cada una de las variables. Se programó en CUDA y en OpenMP, y se probó hasta con 5 millones de individuos y 100 variables.

   b) Determina la matriz de correlaciones entre las variables.

   c) Cálculo de los valores y vectores propios. Se trabajó hasta con una matriz de 50 millones de elementos. Con ello, ordenando los valores propios, fue posible determinar la inercia de cada componente.

3. Operaciones posterior a realizar el ACP

   a) Muestra las inercias para cada componente.

   b) Se puede elegir dos componentes para mostrar el plano principal y el círculo de correlaciones correspondiente.

c) Para mejorar la visualización en el plano principal se pueden filtrar variables por rango o por índice.

d) Para mejorar la visualización en el círculo de correlaciones se pueden filtrar variables por índice.

e) Se permite deshacer todos los filtros o el último.

El algoritmo fue ejecutado sobre varios ejemplos de prueba, presentamos los resultados obtenidos para dos de ellos.

3.10.1. Ejemplo 1: Calidad y salud de los suelos bananeros

Corresponde a las observaciones recolectadas en calicatas de 60 cm de profundidad, tomando una calicata por hectárea, esta recolección se realizó en diversas fincas en cinco países: Costa Rica (6 fincas), Panamá (11 fincas), Venezuela (9 fincas) y República Dominicana (8 fincas de producción orgánica y 4 fincas de producción convencional). Corresponden al estudio:

Javier Trejos, Franklin Rosales, Mario Villalobos, Luis Pocasangre, Eduardo Delgado. *Construcción de índices de calidad y salud de suelos bananeros para cuatro países de la Cuenca del Caribe*. Universidad de Costa Rica-Bioversity Intl.

Esta tabla de datos corresponde a 82 individuos sobre los cuales se han medido 29 variables.

	pH	Ca	Mg	K	P
CR1	4,8725	8,2760	3,1775	1,3757	73,3094
CR2	5,0750	10,2567	3,5527	1,3724	83,3680
CR3	4,9000	5,0050	1,5696	1,1310	36,4264
CR4	5,8900	6,6298	2,5033	1,1002	59,7165
CR5	4,7075	7,9537	2,7684	2,9445	213,7983
CR6	4,5250	2,0925	0,9237	0,9428	45,8837
CR7	5,3350	11,9979	2,1373	1,7145	248,6557
CR8	5,1900	14,5571	3,8088	2,2063	160,9424
CR9	5,0075	10,2218	2,7472	2,2301	101,3807
CR10	5,1750	19,0566	8,5455	1,5160	187,8943
CR11	4,4750	18,7554	6,0593	3,6023	273,0157
CR12	4,6700	20,1891	7,6097	2,7294	186,9285
CR13	5,5275	30,0313	3,7947	1,8871	173,8841

Figura 3.2: Muestra de la tabla de datos del ejemplo 1.

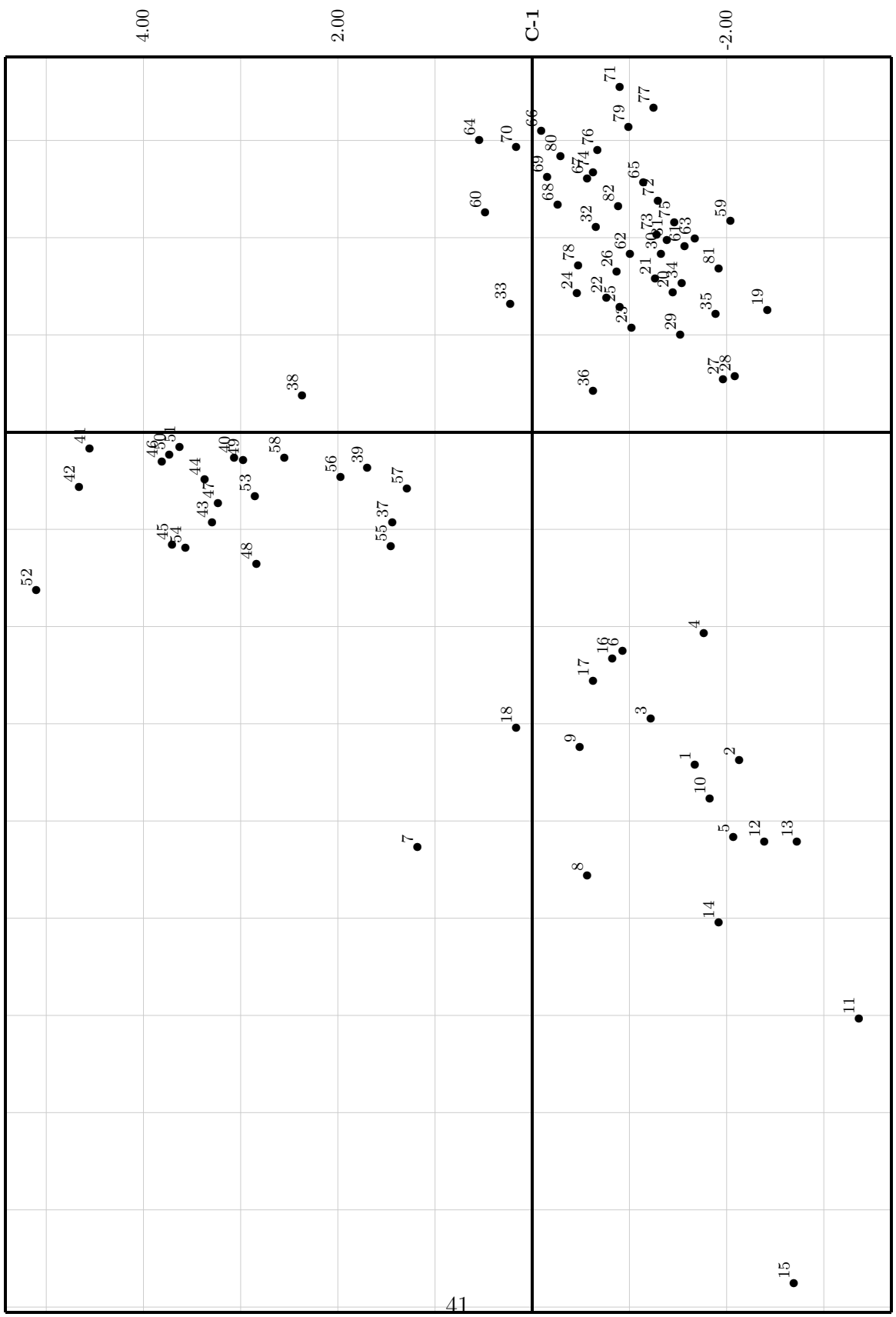

Figura 3.3: Plano principal del ejemplo 1.

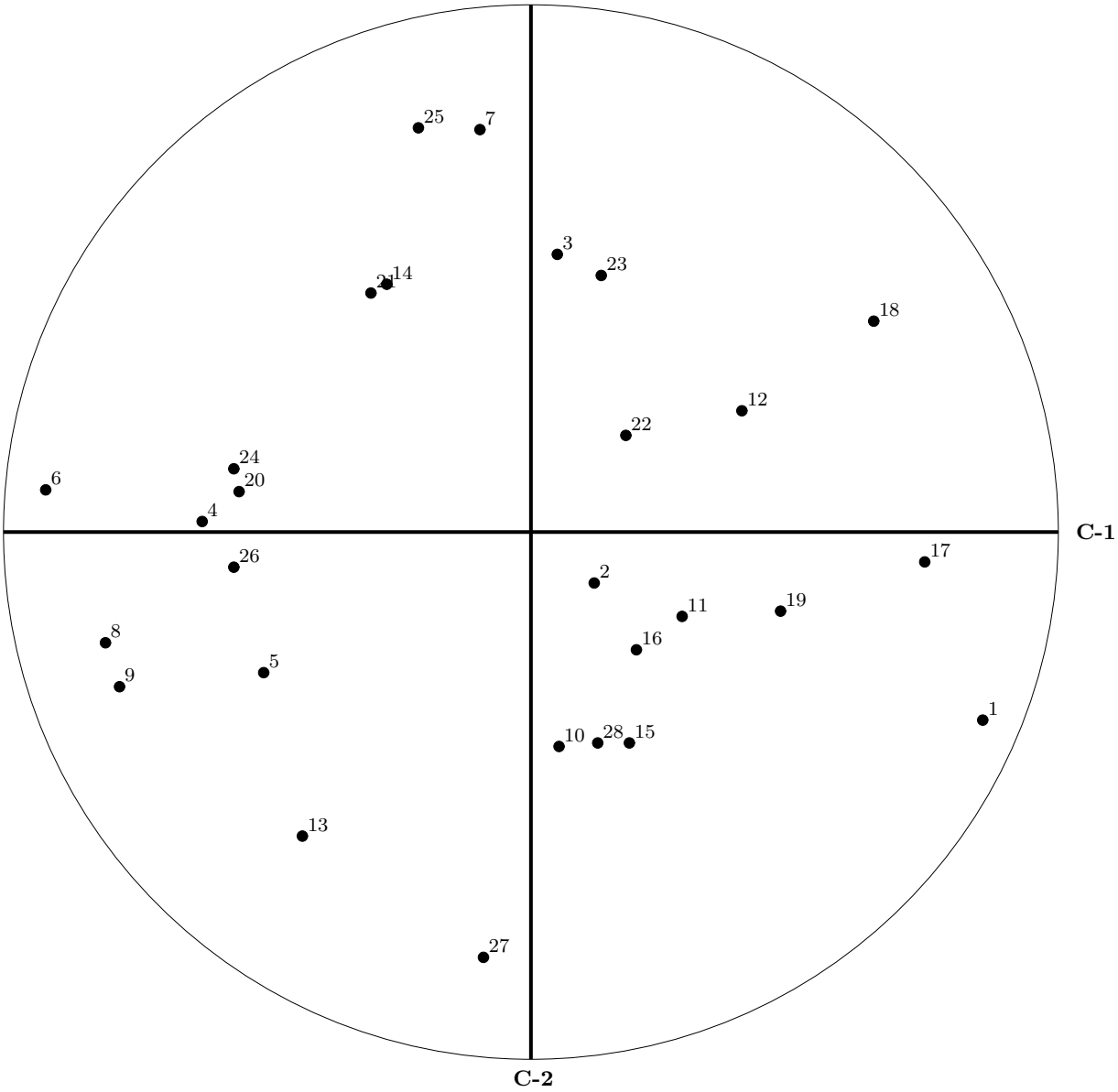

Figura 3.4: Círculo de correlaciones ejemplo 1.

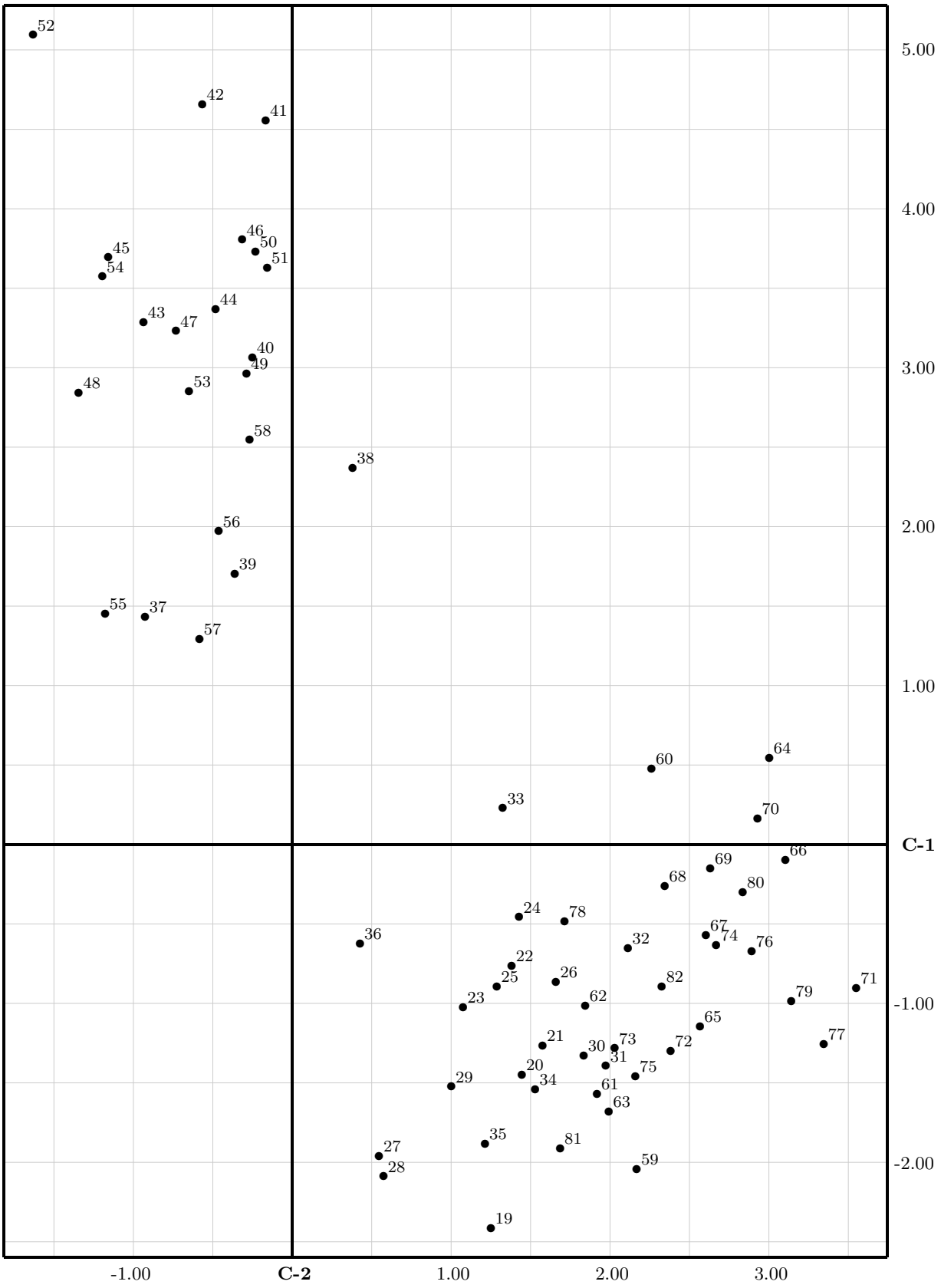

Figura 3.5: Plano principal del ejemplo 1 fitradas las variables de la 1 a la 18.

3.10.2. Ejemplo 2: Perfiles de expresión genética de pacientes que sobreviden a cáncer de pulmón

La tabla de datos en este caso corresponde a 86 individuos sobre los cuales se han medido 7129 variables. Corresponden al estudio:

Davis G. Beer, Sharon L.R. Kardia, Chiang-Ching Huang, Thomas J. Giordano, Albert M. Levin, David E. Misek, Lin Lin, Guoan Chen, Tarek G. Gharib, Dafydd G. Thomas, Michelle L. Lizyness, Rork Kuick, Satoru Hayasaka, Jeremy M.G. Taylor, Mark D. Iannettoni, Mark B. Orringer and Samir Hanash. *Gene-expression profiles predict survival of patients with lung adenocarcinoma*. Departments of Surgery, Epidemiology, Biostatistics, Pathology and Pediatrics, University of Michigan, Ann Arbor, Michigan, USA.

170	69.4	250.7	957.1	25.4	471.2	-52	42.8
59.7	18.1	146.8	186.8	-7.7	309	-99	57.9
80	26	150	340.2	-16.3	225.7	23.5	69.4
92.4	96.9	177.8	515.8	18	296.6	48.5	60.4
104	72.8	228.7	540.8	26	264.1	-10	56.4
88	138.6	115.5	616.6	9	371.9	49.2	37.2
69.7	11.1	177.8	380.5	21	291	-62.5	99
230	176	511.3	523.9	32	664.2	-17.1	295
105	78.1	233.9	602.7	24.3	471.6	20	78.1
53.7	36.7	393.6	160.5	27	407.3	-4.4	94.2
96.9	28.4	339.5	880.5	19	361.8	-0.6	82.6
104	70	331	1766	26	685.9	55.6	185.8
91	91	165.5	442.5	29	410.2	-22.6	91
83	78	922.6	340.4	18.5	544.1	75	112
845.4	126.6	651	769.4	-8	274.2	-43.2	66.2
265.9	17.8	243	458	19.6	249.4	9.8	112.2
343.7	44.1	119.7	155.7	163.8	138.8	32.3	211.1
262	99	866.5	473.8	43.3	281.3	-25.5	83.5
163	76.4	193	330.3	19.8	232	-1.5	57
128.4	54.1	149	312.3	-6.5	277	-49.3	12.5
121.4	80	240.1	168	-35.1	429.4	66.8	110
729.4	137	214.9	1166.4	33.2	406.9	-36.5	322
284.3	236.3	113	405.3	25.5	263.9	-36	144
119	42.6	918.4	703.4	-5.3	410.2	62.3	53.7
191.6	86	342	434.5	19.8	212.4	-43.1	67.6

Figura 3.6: Muestra de la tabla de datos del ejemplo 2.

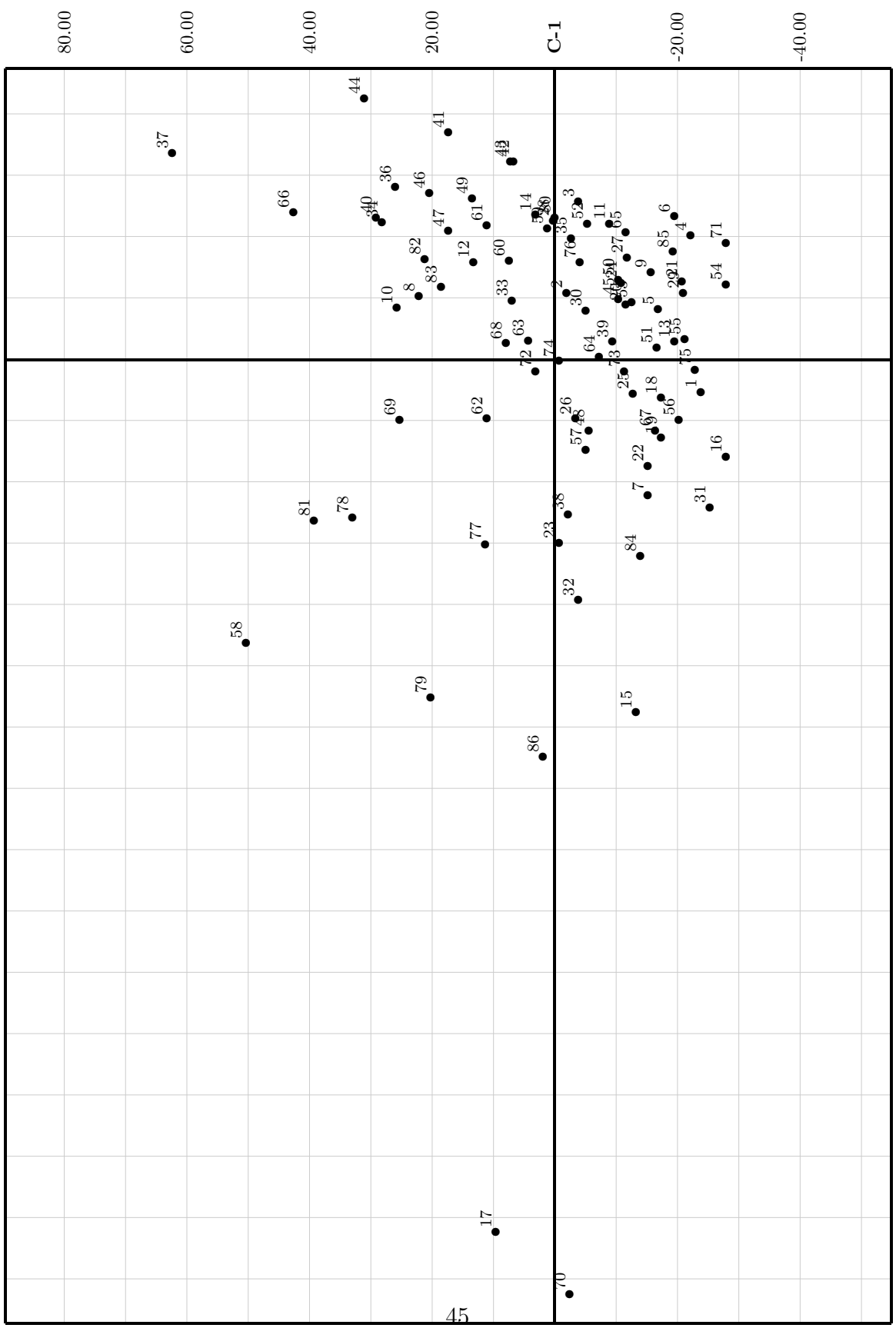

Figura 3.7: Plano principal ejemplo 2.

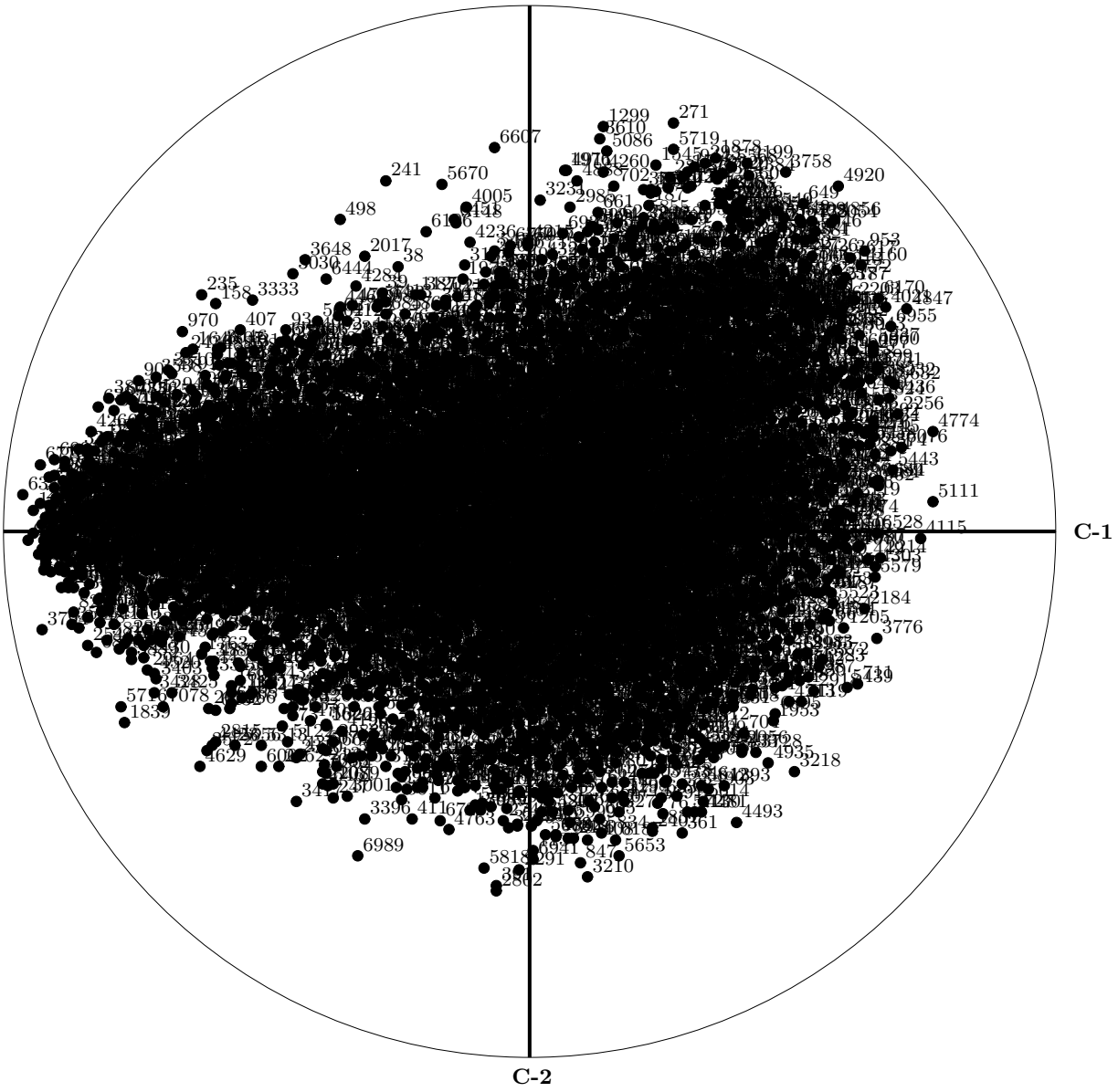

Figura 3.8: Círculo de correlaciones con todas las variables.

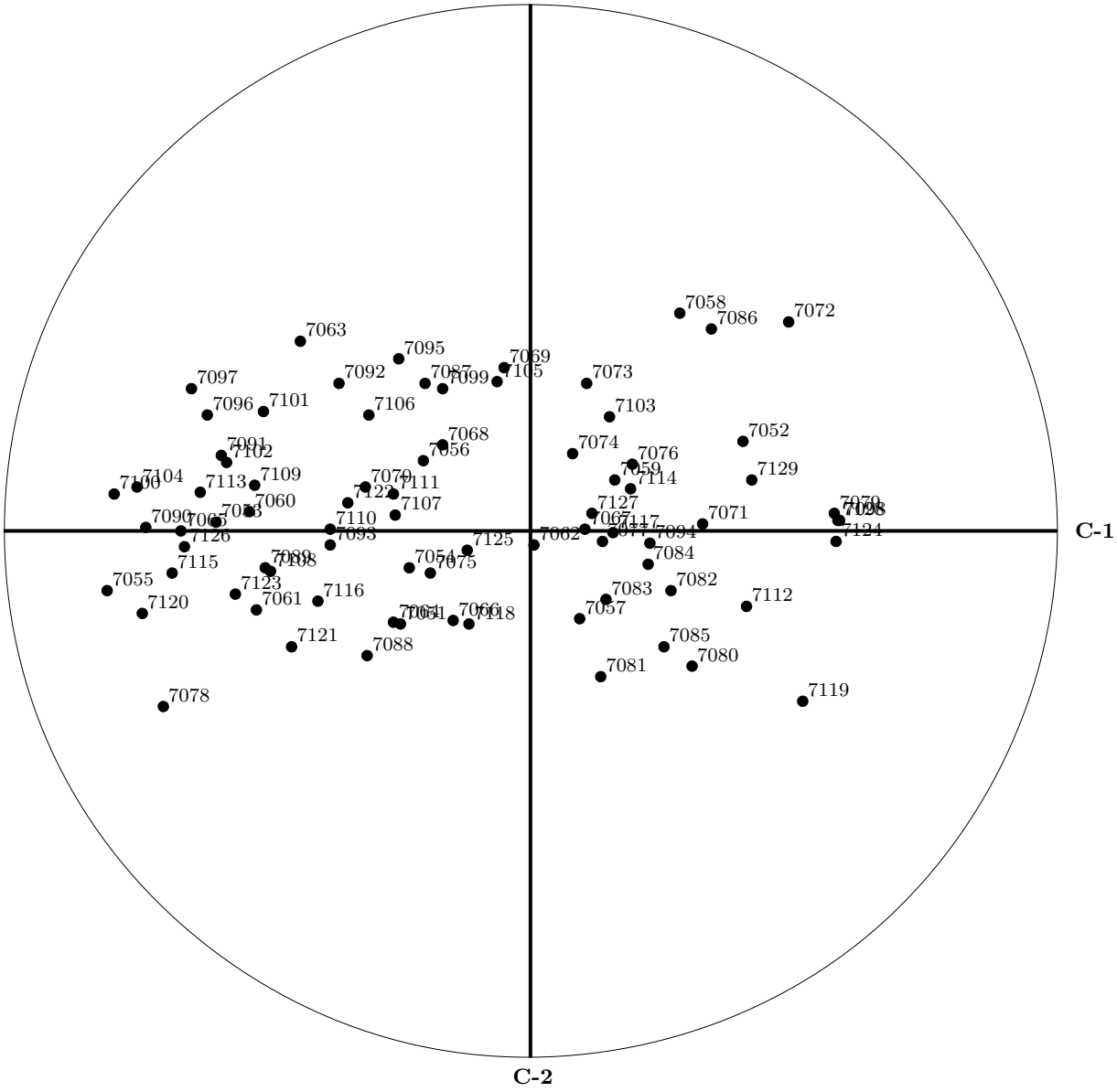

Figura 3.9: Círculo de correlaciones filtrando las variables de la 1 a la 7050.

Bibliografía

[1] Jolliffe, I. T. Principal Component Analysis. 2 edicin, Springer, 2002.

[2] nVidia. nVidia CUDA Programming Guide. Version 3.0. 2/9/2010.

[3] Quinn, Michael. Parallel Programming in C with MPI and OpenMP. 1 edition. McGraw-Hill, 2003.

[4] Kirk, David B.; Hwu, Wen-mei W. Programming Massively Parallel Processors: A Hands-on Approach. 1 edition. Morgan Kaufmann, 2010.

[5] Sanders, Jason; Kandrot, Edward. CUDA by Example: An Introduction to General-Purpose GPU Programming. 1 edition. Addison-Wesley Professioanl, 2010.

[6] Chandra, Rohit et al. Parallel Programming in OpenMP. 1 edition. Morgan Kaufmann, 2000.

[7] Chapman, Barbara et al. Using OpenMP: Portable Shared Memory Parallel Programming. 1 edition. The MIT Press, 2007.

[8] Saad, Yousef. Iterative Methods for Sparse Linear Systems. 2 edition. Society for Industrial and Applied Mathematics, 2003.

[9] Trefethen, Lloyd N.; Bau III, David. Numerical Lineal Algebra. 1 edition. SIAM: Society for Industrial and Applied Mathematics, 1997.

[10] Golub, Gene H.; Van Loan, Charles F. Matrix Computations. 3rd edition. The Johns Hopkins Unversity Press, 1996.

[11] A. H. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer.

[12] Richard P. Brent y Franklin T. Luk. A systolic architecture for almost linear-time solution of the symmetric eigenvalue problem. Technical Report TR-CS-82-10, Department of Computer Science, Australian National University, Camberra, August 1982.

[13] Barnett, V., Lewis T. Outliers in Statistical Data. 3rd edition. J. Wiley and Sons, USA, 1994

[14] Bramer, M. Principles of data mining. Springer-Verlag, USA, 2007.

[15] Spiegel, Murray. Estadística. McGraw-Hill, México, 1991.

[16] Dallas, Johnson. Métodos multivariados aplicados al análisis de datos. International Thomson Editores. México, 2000.

[17] Mathews, John y Kurtis, fink. Métodos numéricos con `Matlab`. Prentice-Hall, España, 2000.

[18] Hair, Joseph; Brack, William; Babin, Barry y Anderson, Rolph. Multivariate Data Analysis. Prentice-Hall, USA, 2010.

[19] `R: http://www.r-project.org/`, consultada el 16 de marzo 2012.

[20] Burden, Richard y Faires, Douglas. Análisis Numérico. International Thomson Editores. México, 2002.

[21] Hogben, Leslie (ed). Handbook of linear algebra. Chapman & Hall, USA, 2007.

[22] `Nvidia: http://www.nvidia.com/object/tesla_computing_solutions.html`, consultada el 15 de febrero 2012.