Universidad Autónoma de Baja California

Instituto de Ingeniería

Maestría y Doctorado en Ciencias e Ingeniería

Ambiente de Trabajo para GPU Basado en Operaciones Matriciales Explícitas

Tesis que para obtener el grado de:

MAESTRO EN CIENCIAS

Presenta

Luis Roberto Ramírez Hernández

Director de Tesis:

Dr. Félix Fernando González Navarro

Codirector de Tesis:

Dr. Rainier Romero Parra

Mexicali, B. C. Agosto de 2015

Dedicatorias

A mi madre por apoyarme siempre en mis estudios.
A mis hermanos por ayudarme siempre en todos los aspectos.
A mis maestros por enseñarme y apoyarme.
A mis amigos y compañeros por el apoyo en los estudios.

Reconocimientos

Al Consejo Nacional de Ciencia y Tecnología (CONACYT) que me brindó el apoyo de la beca con el cual tuve sustento económico a lo largo de mi periodo de Maestría.

A la Universidad Autónoma de Baja California (UABC) por darme el apoyo y la oportunidad de ingresar con mi proyecto a la Mestría en ciencias e Ingeniería.

Al Instituto de Ingeniería por aceptarme dentro de las instalaciones y brindarme herramientas de trabajo con las cuales realicé mis experimentos de investigación.

Al director de mi Tesis el Doctor Félix Fernando González Navarro y co-director el Doctor Rainier Romero Parra que estuvieron siempre enseñándome, guiándome y ayudándome para la realización de mi trabajo de tesis.

A mi familia, amigos y compañeros que estuvieron siempre apoyándome incondicionalmente en todo momento el tiempo que estuve realizando mi Maestría.

Resumen

El cálculo acelerado en una tarjeta gráfica es el uso de una unidad de procesamiento gráfico en combinación con una unidad de procesamiento central para acelerar aplicaciones utilizadas en el análisis y cálculo científico. El cálculo acelerado ofrece un rendimiento sin precedentes ya que traslada las partes de la aplicación con mayor carga computacional a la tarjeta gráfica y deja el resto del código ejecutándose en la unidad de procesamiento central. Desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido. En esta tesis se presenta una librería para cálculos científicos matriciales, se realizan experimentos en donde las funciones creadas en esta librería optimizan el trabajo de operaciones matriciales básicas en comparación con las unidades de procesamiento central, siendo de gran utilidad para cuando se manejan grandes bases de datos. También se presenta un análisis dentro de la minería de datos en donde se emplea esta librería en tres diferentes clasificadores (vecinos cercanos, análisis discriminante lineal y análisis discriminante cuadrático) obteniéndose comparaciones de tiempos entre la utilización de esta librería que trabaja con tarjetas gráficas y la unidad de procesamiento central.

Abstract

The accelerated calculation in a graphic card is the use of an unit of graphic processing in combination with a central processing unit to accelerate applications used in analysis and scientific computing. The accelerated calculation offers an unprecedented performance since transfer the parts of the application with bigger load computational to the graphic card and leave the rest of the code running in the central processing unit. From the perspective of the user, the application simply execute themselves faster. This thesis shows a library for solve matrix calculations, perform experiments in where the functions created in this library optimize the work of matrix basic operations in comparison with the central processing unit, being very useful when large databases are managed. Also present an analysis within data mining where this library is used in three different classifiers (nearest neighbors, discriminant linear analysis and quadratic discriminant analysis) obtained comparisons of times between this library that work with graphic cards and the central processing unit.

Indice

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

La frecuencia del crecimiento de la unidad de procesamiento central (CPU) esta ahora limitada por materiales físicos y gran consumo de energía. Su rendimiento es generalmente aumentado incrementando el número de núcleos que posee. Hoy en día contienen hasta 4 núcleos, donde cada uno trabaja independientemente de los otros ejecutando varias instrucciones por varios procesos entrando en un sistema denominado MIMD (múltiples instrucciones, múltiples datos).

Por otra parte la unidad de procesamiento gráfico (GPU) a diferencia del CPU cuenta con una mayor cantidad de núcleos, los cuales ejecutan una misma instrucción simultáneamente entrando en un sistema SIMD (simple instrucción, múltiples datos). Este estilo de programación es común para algoritmos gráficos y muchas tareas de computación científica, en donde el procesamiento que se requiere es mucho más complejo (IXBT Labs, 2008).

A la fecha, el interés por la computación general sobre GPU va en aumento. FoldingHome es un proyecto de la Universidad de Stanford que basándose en la computación distribuida permite realizar simulaciones por ordenador de plegamiento de proteínas (NVIDIA, 2008). La investigación actual del proyecto (en colaboración con ATI) se centra en el aprovechamiento de la GPU en cada nodo de computación, para acelerar los cálculos realizados. Según los estudios solo ciertas partes de las que forman la totalidad del cálculo a realizar son implementadas en GPU consiguiéndose en ellas rendimientos entre veinte y cuarenta veces mejor que sus correspondientes implementaciones en CPU.

La empresa ATI creó el procesador de flujo llamado FireStream que utiliza

al máximo el procesamiento en paralelo de la GPU para realizar aplicaciones de cálculo intensivo que utilizan científicos, ingenieros y consumidores. NVIDIA presentó en junio de 2007 una nueva línea de hardware orientado a la computación general de altas prestaciones de nombre Tesla basada en sus productos gráficos de calidad. Tesla creó una arquitectura de cálculo en paralelo llamado CUDA que aprovecha la gran potencia de la GPU para proporcionar un incremento extraordinario del rendimiento del sistema (Jaume, I., 2010).

1.1 Planteamiento del problema

El GPU ofrece gran rendimiento en tareas que requieren gran capacidad computacional, así desde la perspectiva del usuario, las aplicaciones simplemente se ejecutan más rápido. Actualmente, ramas de la computación basan su trabajo algorítmico en operaciones matriciales como es el caso de el aprendizaje máquina o minería de datos. Esto representa una serie de desventajas si se desea implementar algoritmos con CPU en esta rama, ya que se realizan cálculos masivos sobre grandes bases de datos, teniendo que esperar una larga cantidad de tiempo para esperar resultados.

Es por ello que en este trabajo de investigación, se creó un ambiente de trabajo (framework) para GPU, que sirve como capa intermedia entre los lenguajes de programación, donde se llevan a cabo las implementaciones algorítmicas con operaciones matriciales, y el hardware GPU, donde las operaciones son de tipo vectorial. De esta manera, se incrementará la velocidad de procesamiento aprovechando el paralelismo que GPU utiliza. Hoy en día se utilizan herramientas de computo científico como MatLab y R para obtener los resultados en problemas con datos matriciales, sin embargo se necesita de otra herramienta para manipular datos de tipo vectorial como los hace el GPU. Para esto se utilizó el lenguaje de programación OpenCL como motor de implementación, con la cual se interactua en los sistemas GPU y así se realiza la manipulación de datos.

Utilizar el GPU conlleva a grandes ventajas sobre el CPU, entre las mas importantes se destacan las siguientes:

1. Se tiene la facilidad de manejar grandes vectores de datos.
2. Tiene una baja latencia en operaciones de punto flotante.
3. Trabajan con miles de núcleos de procesamiento, lo que ayuda a la aceleración de operaciones de cálculo.
4. Utiliza paralelismo para el procesamiento de datos.

En contraste alguna de sus principales desventajas son las siguientes:

1. Trabajan de forma vectorial, lo que dificulta la manipulación de datos de forma matricial.
2. Tiene restricción de instrucciones de operaciones con enteros, ya que están diseñados para trabajar con operaciones de punto flotante.
3. Su comunicación es limitada (Escolano, S., 2011).

El objetivo principal es trabajar con sistemas GPU, aprovechar su capacidad para manejar grandes vectores de datos que agilizan cálculos de computo complejo y crear una librería eficiente en OpenCl para la manipulación de datos matriciales. El interés de empresas tan importantes como ATI o NVIDIA, así como el creciente número de estudios, demuestran las ventajas del uso del GPU sobre cierto tipo de aplicaciones y hacen de esta disciplina un campo de estudio relativamente nuevo y con un futuro prometedor.

1.2 Objetivos

1.2.1 Objetivo General

Crear una librería para trabajo matricial con la tecnología GPU dentro de OpenCL, comparar tiempos de ejecución de las funciones que esta contiene contra el CPU y comprobar su eficiencia en cálculos matriciales.

1.2.2 Objetivos Específicos

1. Crear una librería en OpenCL que maneje cálculos matriciales por medio del GPU.

2. Comparar tiempos de ejecución de las funciones creadas en la librería contra funciones de CPU.

3. Utilizar la librería en tres algoritmos de clasificación (vecinos cercanos, análisis discriminante lineal y análisis discriminante cuadrático) utilizados en minería de datos y hacer comparaciones de tiempos de ejecución entre GPU y CPU.

1.2.3 Esquema general de la tesis

La tesis está organizada de la siguiente manera: en el capítulo uno se da una introducción del tema a investigar y desarrollar, se ve un planteamiento del problema en donde se identifica el objeto de estudio a tratar y se propone una solución para dicho campo. También se encuentran tanto el objetivo general como los objetivos específicos que se alcanzan en esta tesis. En el capítulo dos se explica la teoría en donde se detalla el funcionamiento de tecnología CPU y GPU, así también las bases de las operaciones matriciales y los algoritmos de minería de datos que se utilizaron. En el capítulo tres se describe el equipo y métodos utilizados para la creación de los experimentos. En el capítulo cuatro se presentan los resultados experimentales obtenidos, y finalmente se dan las conclusiones que se tienen de estos experimentos, así como algunos trabajos que se realizan a futuro sobre este tema.

Capítulo 2

Teoría de unidades de procesamiento, lenguaje OpenCL y operaciones matriciales

En este capítulo, se expondrán las principales definiciones tanto de la tecnología GPU como CPU, se verá la arquitectura de cada uno de ellos y sus diferencias en cuanto al uso de la memoria. Se explicará el lenguje de programación OpenCL que fue el software con el que se trabajó en la GPU en la presente tesis.Tambíen se definen los conceptos de las operaciones matriciales y los algoritmos que se utilizarán en minería de datos (los clasificadores k-vecinos cercanos, discriminante lineal y discriminante cuádratico) de los cuales hará uso esta librería para comprobar su eficiencia con respecto a tiempos de ejecución.

2.1 Unidad de Procesamiento Central

2.1.1 Definición

El CPU (central processing unit) también llamado microprocesador o simplemente procesador, es el componente principal del ordenador y otros dispositivos programables, que interpreta las instrucciones contenidas en los programas y procesa los datos. El CPU proporciona la característica fundamental del ordenador (la programabilidad) y son uno de los componentes necesarios encontrados en los ordenadores de cualquier tiempo, junto con la memoria principal y los dispositivos de entrada/salida. El término en sí mismo y su acrónimo han estado

en uso en la industria de la Informática por lo menos desde el principio de los años 60. La forma, el diseño y la implementación del CPU ha cambiado drásticamente desde los primeros ejemplos, pero su operación fundamental sigue siendo la misma (Martin, 2003).

Las tentativas de alcanzar un desempeño escalar y mejor, han resultado en una variedad de metodologías de diseño que hacen comportarse al CPU menos lineal y más en paralelo. El paralelismo es una forma de computación en la cual varios cálculos se realizan simultáneamente. Basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo. Cuando se refiere al paralelismo en el CPU, generalmente son usados dos términos para clasificar estas técnicas de diseño:

1. El paralelismo a nivel de instrucción, en inglés *Instruction Level Parallelism* (ILP), busca aumentar la tasa en la cual las instrucciones son ejecutadas dentro de un CPU, es decir, aumentar la utilización de los recursos de ejecución en la pastilla (Pai & Ranganathan, 1997).

2. El paralelismo a nivel de hilo de ejecución, en inglés *Thread Level parallelism* (TLP), que se propone incrementar el número de hilos(efectivamente programas individuales) que un CPU pueda ejecutar simultáneamente (Flautner *et al.*, 2000).

Cada metodología se diferencía tanto en las maneras en las que está implementada, como en la efectividad relativa que produce el aumento del desempeño del CPU para una aplicación.
Un menos común pero cada vez más importante paradigma de CPU (y de hecho, de computación en general) son los vectores. Como implica su nombre, los procesadores vectoriales se ocupan de múltiples piezas de datos en el contexto de una instrucción. Esto contrasta con los procesadores escalares, que tratan una pieza de dato por cada instrucción. Estos dos esquemas de ocuparse de los datos son generalmente referidos respectivamente como SISD (Simple Instrucción, Simple Dato) y SIMD (Simple Instrucción, Múltiples Datos) (Flynn, 1972). La gran utilidad en crear procesadores que se ocupen de vectores de datos radica en la optimización de tareas que tienden a requerir la misma operación, por ejemplo, una suma o un producto escalar.

2.1.2 Operación del CPU

La operación fundamental del CPU es ejecutar una secuencia de instrucciones almacenadas llamada programa. El programa es representado por una serie de números que se mantienen en una cierta clase de memoria de ordenador. Hay cuatro pasos que todas las CPU usan en su operación: leer, decodificar, ejecutar y escribir (Godfrey, 1993). El primer paso, leer, implica el recuperar una instrucción de la memoria de programa. Esta es determinada por un contador de programa, que almacena un número que identifica la dirección de la siguiente instrucción que se debe buscar. En el paso de decodificación, la instrucción es dividida en partes que tienen significado para otras unidades de la CPU. La manera en que el valor de la instrucción numérica es interpretado, está definida por la arquitectura del conjunto de instrucciones de la CPU. Posteriormente en la ejecución, varias unidades del CPU son conectadas de tal manera que ellas realizan la operación deseada. Finalmente, en la escritura, simplemente se escriben los resultados del paso de ejecución a una cierta forma de memoria, estos resultados son escritos a algún registro interno del CPU para acceso rápido por subsecuentes instrucciones.

2.1.3 Diseño del CPU

La base de diseño del CPU está basada en la arquitectura Von Neumann que se describe de la siguiente manera:

1. Una arquitectura de diseño para un computador digital electrónico con partes que constan de una unidad de procesamiento que contiene una unidad aritmético lógica y registros del procesador.

2. Una unidad de control que contiene un registro de instrucciones y un contador de programa.

3. Una memoria para almacenar tanto datos como instrucciones, almacenamiento masivo externo y mecanismos de entrada y salida (LASS, 2009).

El diseño de una arquitectura Von Neumann tiene un conjunto dedicado de direcciones y buses de datos para leer datos desde memoria y escribir datos en la misma, y otro conjunto de direcciones y buses de datos para la búsqueda de

instrucciones. Un computador digital de programa almacenado mantiene sus instrucciones de programa, así como sus datos, en memoria de acceso aleatorio (RAM) de lectura-escritura.

Una máquina Von Neumann, al igual que prácticamente todos los computadores modernos de uso general, consta de cuatro componentes principales:

1. El dispositivo de operación que ejecuta instrucciones de un conjunto especificado, llamado sistema de instrucciones, sobre porciones de información almacenada, separada de la memoria del dispositivo operativo, en la que los operandos son almacenados directamente en el proceso de cálculo, en un tiempo relativamente corto.

2. La unidad de control que organiza la implementación consistente de algoritmos de decodificación de instrucciones que provienen de la memoria del dispositivo, responde a situaciones de emergencia y realiza funciones de dirección general de todos los nodos de computación.

3. La memoria del dispositivo es un conjunto de celdas con identificadores únicos (direcciones), que contienen instrucciones y datos.

4. El dispositivo entrada salida que permite la comunicación con el mundo exterior de los computadores, son otros dispositivos que reciben los resultados y que le transmiten la información al computador para su procesamiento (Godfrey, 1993).

2.2 Unidad de Procesamiento Gráfico

2.2.1 Definición

El GPU (graphic processing unit) es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central. Los coprocesadores matemáticos son utilizados en muchos sistemas para acelerar el procesamiento de datos, utilizándose como un segundo procesador, si bien algunas de las diferencias respecto del CPU son claras. Los coprocesadores no tienen acceso a los datos directamente o ejecutan un juego

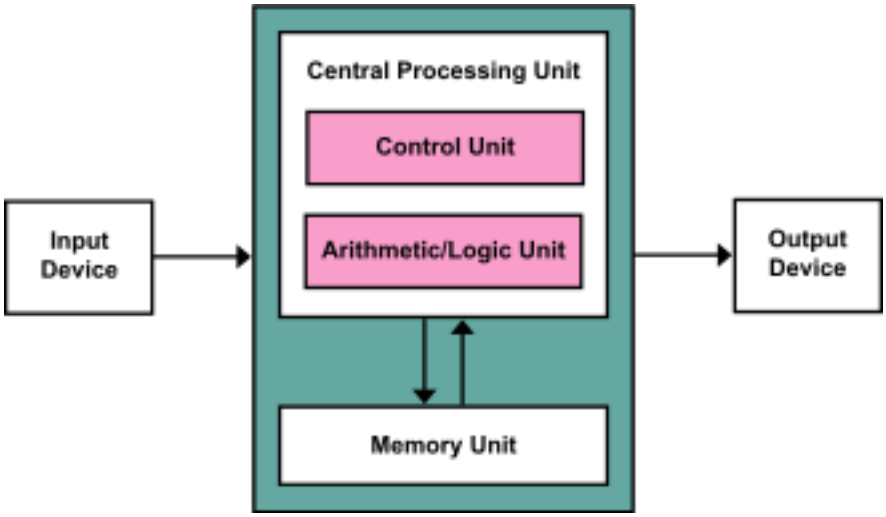

Figura 2.1: Arquitectura Von Neumann

de instrucciones mucho más sencillo pensado para tratar datos en coma flotante (Xataca, 2014).

El GPU es la evolución de los chips gráficos, inicialmente controlaban el mapeado de la memoria y el control de bitmaps, ahora el GPU introduce capacidades de programación independientes. Las operaciones típicas del GPU incluían movimientos desde el core a la memoria para mostrarlos a la pantalla y el manejo de bloques gráficos independientes, también llamados sprites. Mas adelante, añadieron funciones adicionales que facilitaron que las aplicaciones software y las interfaces hicieran un uso mas amplio de los gráficos.

2.2.2 Tipos de Unidades de procesamiento gráfico

Actualmente existen tres grandes tipos de unidades de procesamiento gráfico. más que por la arquitectura, estos difieren entre ellos por el modo en que son implementadas:

1. Las tarjetas dedicadas que son unidades gráficas que proporcionan gran potencia, tienen una serie de especificaciones y están expresamente diseñadas para cumplir con sus tareas específicas, por lo que son mucho más eficientes. Generalmente se suele entender que una tarjeta dedicada es aquella que se integra a la tarjeta madre mediante un puerto aparte.

2. Las tarjetas integradas gráficas que a diferencia de las unidades dedicadas, utilizan la memoria del sistema para realizar sus funciones. Este tipo de tarjetas son las más comunes en los ordenadores modernos, estando hasta el 90% en los equipos de cómputo. Con frecuencia el núcleo central de estas unidades solía estar en la tarjeta madre, pero más recientemente las cosas han cambiado y compañias como Intel y AMD las han puesto en sus procesadores.

3. Las tarjetas híbridas que estan diseñadas para mantener precios relativamente bajos y al mismo tiempo asegurar niveles de potencia adecuados. Las unidades gráficas híbridas también comparten la memoria del sistema, pero para disminuir el tiempo de latencia de esta última, integran una cantidad limitada de memoria propia que se encarga de realizar las labores inmediatas (Torres, J., 2013).

2.3 Cálculo Acelerado de la GPU

El cálculo acelerado de la GPU es el uso de una GPU junto con una CPU para acelerar las aplicaciones científicas y de ingeniería con fines generales. La computación de la GPU se ha convertido rápidamente en un estándar del sector que ha sido aprovechada por millones de usuarios en todo el mundo y la han adoptado prácticamente todos los proveedores de computación. Los Sistemas GPU pueden trabajar bajo un sistema SIMD (simple instrucción, múltiples datos):

En este sistema todos los núcleos ejecutan la misma instrucción al mismo tiempo. Solo se necesita decodificar la instrucción una única vez para todos los núcleos como por ejemplo en la suma o el producto de vectores (Morell, V., 2011).

2.3.1 Arquitectura de sistemas CPU y GPU

El GPU moderno usa la mayoría de sus transistores para el cálculo de gráficos 3D. Se comenzó por la aceleración del mapeado de texturas en memoria y la renderización de polígonos, y mas tarde se añadieron unidades para acelerar los cálculos geométricos y el mapeo de vértices dentro de las distintas coordenadas

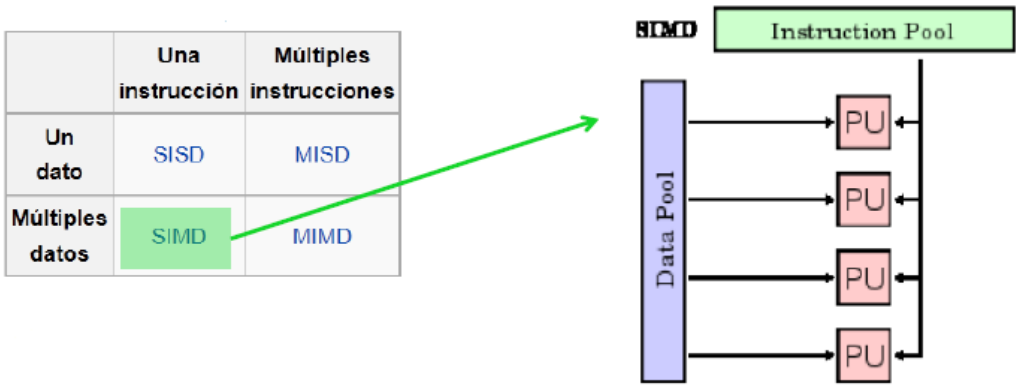

Figura 2.2: Sistema SIMD en una GPU

del sistema. El último GPU desarrollado manipula vértices y texturas con las mismas operaciones soportadas por el CPU (NVIDIA, 2010).

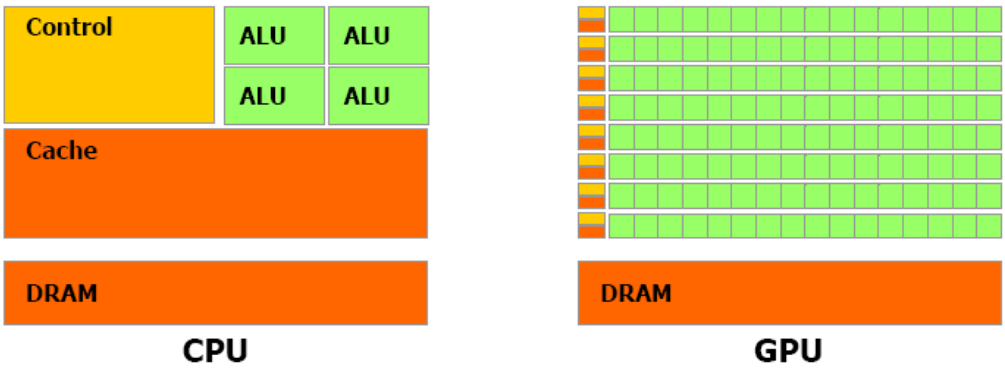

Figura 2.3: Arquitectura de sistemas CPU y GPU

En el GPU se ha sacrificado espacio de control sofisticado y de caché, para aumentar la superficie dedicada a cómputo. Esto desemboca en una arquitectura de propósito específico con caches reducidas, elevado número de elementos de proceso y una optimización para acceso secuencial a los datos (Gutiérrez, A., 2012).

El procesamiento paralelo con GPU se utiliza para resolver todo tipo de problemas de cómputo. A fin de cuentas, al igual que un CPU, un GPU es una unidad

de procesamiento que recibe datos e instrucciones y las procesa. La diferencia es que la arquitectura de un GPU se presta mucho más al procesamiento paralelo que la de un CPU tradicional. Mientras que un CPU está formado por algunos cuantos núcleos con mucha memoria cache que pueden manejar unos cuantos hilos al mismo tiempo, un GPU típicamente está compuesto por cientos de núcleos de procesamiento que manejan miles de hilos de manera simultánea. Gracias a esto, el procesamiento paralelo de datos usando GPU llega a ser mucho más rápido, económico y con menos gasto de energía que si usáramos CPU (Arvizu, H. , 2012).

2.3.2 Procesamiento compartido

Los sistemas informáticos están pasando de realizar el procesamiento central en la CPU a realizar coprocesamiento repartido entre la CPU y la GPU. Una CPU + GPU constituye una potente combinación porque la GPU está formada por varios núcleos optimizados para el procesamiento en serie, mientas que la GPU consta de millares de núcleos más pequeños y eficientes diseñados para el rendimiento en paralelo. Las partes en serie del código se ejecutan en la CPU mientras que las paralelas se ejecutan en la GPU (NVIDIA, 2010).

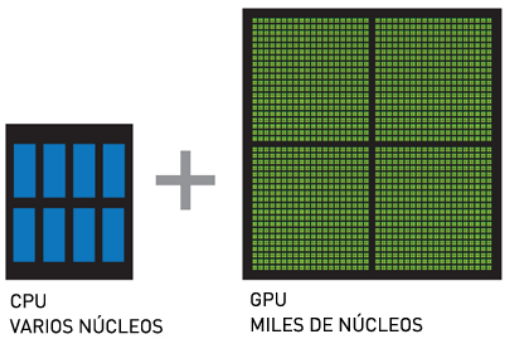

Figura 2.4: núcleos de procesamiento CPU y GPU

Los núcleos de un GPU están agrupados de forma que nos ayuda a estructurar la programación paralela. La tarjeta gráfica, se subdivide en unidades de procesamiento, que a su vez se subdividen en elementos de procesado. Asimismo, cada

una de estas unidades de procesamiento se sincroniza entre ellas para dar lugar a grupos de trabajo, cuya finalidad es proporcionarnos una estructura jerárquica que nos permita un mejor manejo de todas esas unidades de procesamiento.

2.4 Lenguaje de programación OpenCL

Una de las herramientas con las que se puede trabajar con GPU es el lenguaje de programación OpenCL (Open Computing Lenguage) que fue creado en 2009 por el grupo Khronos. Este Grupo se enfoca en la creación de APIs estándares abiertos y libres de regalías que permiten la creación y reproducción multimedia en un amplio abanico de plataformas y dispositivos(Khronos Group, 2008). OpenCL se creó con el objetivo de unificar todas las plataformas con la que se trabaja el cómputo paralelo tales como las que diseñan dispositivos gráficos(como NVIDIA e intel).

2.4.1 Compilación en OpenCL

Los programas de OpenCL se compilan formando código objeto para el CPU, y para el GPU. El código objeto que se ejecuta en el CPU determina los kernels(cantidad mínima de código ejecutable) a ejecutar en cada GPU. Cabe destacar la posibilidad de sincronizar estos kernels a nivel de tarea o de información. Cada kernel será ejecutado por un objeto de trabajo.

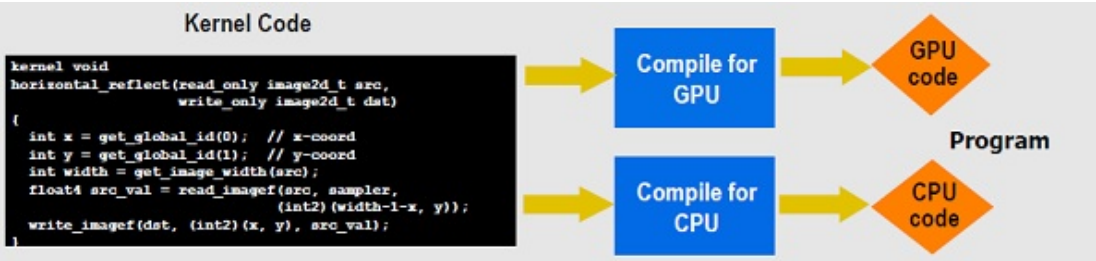

Figura 2.5: Esquema de compilación OpenCL (S.A.B.I.A., 2011)

Una vez que se tiene el código compilado del CPU y el GPU, OpenCL genera un contexto, que se asocia a la unidad que se encarga de ejecutar el programa.

Este contexto se encarga de manejar los programas, los kernels, los objetos de memoria y las colas de comandos (S.A.B.I.A., 2011). La forma en que un programa en OpenCL esta estructurado, se ve en el siguiente esquema:

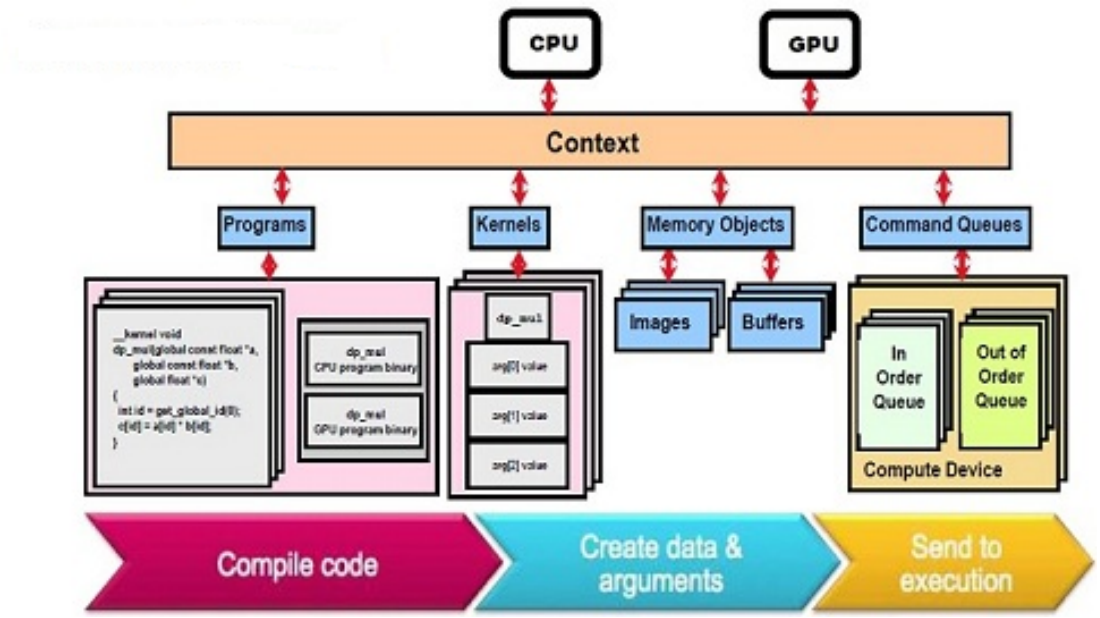

Figura 2.6: Estructura de programa en OpenCL (S.A.B.I.A., 2011)

2.4.2 Manejo de memoria en OpenCL

Al trabajar con programas en OpenCL hay que tener en cuenta el funcionamiento interno de la memoria del hardware gráfico que estamos utilizando. Esta se estructura en una jerarquía que recuerda a las memorias cache de los procesadores, memorias pequeñas y rápidas para cada núcleo, y cada vez mas grandes y lentas a medida que subimos del nivel hasta conectarse con la memoria RAM.

La memoria se divide en:

1. Memoria privada: memoria compartida por un objeto de trabajo.
2. Memoria local: memoria compartida por un grupo de trabajo.

3. Memoria constante: compartida por todos los grupos de trabajo, puede ser leída y escrita por datos de la memoria del host, pero solo es leída por los grupos de trabajo.

4. Memoria global: es leída y escrita por cada una de los grupos de trabajo (Banger & Bhattacharyya, 2013).

"El manejo de la memoria debe hacerse por pasos, para información que entra: memoria del host-memoria global/memoria constante-memoria local-memoria privada, y al revés para devolverla al host" (S.A.B.I.A., 2011).

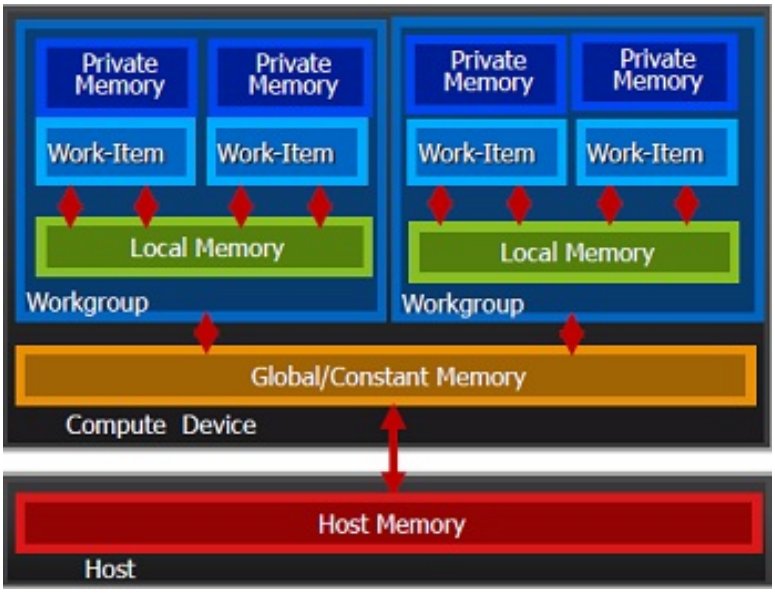

Figura 2.7: Flujo de memoria en programas OpenCL (S.A.B.I.A., 2011)

2.5 Paralelismo de operaciones matriciales

Una matriz de m filas y n columnas sobre un cuerpo IK es una aplicación:

$$\begin{aligned} \mathbf{A}\colon \{1,...,m\}x\{1,...,n\} &\longrightarrow \mathrm{IK} \\ (i,j) &\longmapsto a_{ij} \end{aligned}$$

La matriz **A** suele representarse por:

$$\mathbf{A} = (a_{ij}) = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

con $1 \leq i \leq m$ y $1 \leq j \leq m$, y se dice que $\mathbf{A}$ es de orden $m\ x\ n$ (Meneses, L., 2008).

En esta sección se presentan las definiciones de las operaciones matriciales básicas con las que se trabajo para la creación de esta librería y la manera en donde se llevo a cabo la paralelización de cada una de ellas. Las operaciones matriciales son las siguientes:

1. Suma de matrices.
2. Multiplicación de matrices.
3. Traspuesta de una matriz.
4. Inversa de una matriz.
5. Determinante de una matriz.
6. Covarianza de una matriz.
7. Descomposición QR.

2.5.1 Suma de matrices

Sean A,B $\in \mathbf{M}_{mxn}(IK)$, $\mathbf{A} = (a_{ij})$, $\mathbf{B} = (b_{ij})$ con $1 \leq i \leq m$ y $1 \leq j \leq m$. Se define $\mathbf{A} + \mathbf{B} = \mathbf{C} \in \mathbf{M}_{mxn}(IK)$, con $\mathbf{C} = (c_{ij})$, $1 \leq i \leq m$ y $1 \leq j \leq m$ tal que:

$$\forall i \in \{1, ..., m\}, \forall i \in \{1, ..., n\}, c_{ij} = a_{ij} + b_{ij}, \tag{2.1}$$

El programa en lógica secuencial suma cada uno de los componentes de las matrices de uno por uno, en un programa en paralelo se suma toda una fila con otra al mismo tiempo(Meneses, L., 2008).

Este mismo proceso se aplica para restar, multiplicar y dividir los datos en forma paralela (Fixstars, 2009).

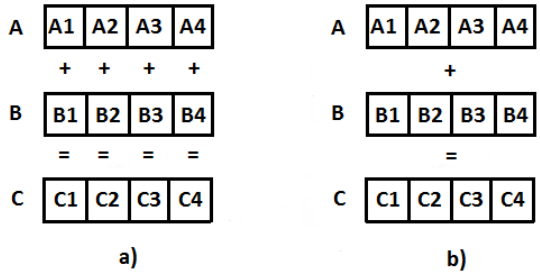

Figura 2.8: Operación suma: a)Suma secuencial, b)Suma en paralelo

2.5.2 Multiplicación de matrices

La multiplicación de matrices es comúnmente usado para demostrar el alto desempeño computacional en la optimización de programa. Consideremos tres matrices, la matriz **A** (con dimensiones $N\ x\ P$), **B** (con dimensiones $P\ x\ M$) y **C** (con dimensiones $N\ x\ M$). La multiplicación de **A** veces **B** con el resultado agregado en **C** esta matemáticamente definido como:

$$C_{ij} = C_{ij} + \sum_{k=0}^{P} A_{ik} * B_{kj}, 1 \leq i \leq N, 1 \leq j \leq M \tag{2.2}$$

Como se muestra en la figura 2.9 los ij elementos de **C** se obtienen de un producto punto de las i filas de **A** con las j columnas de **B**:

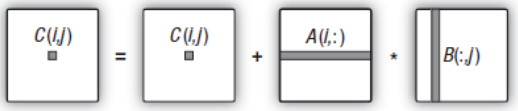

Figura 2.9: Operación de multiplicación en serie

En un proceso en paralelo, se obtiene un elemento de **C** computando la i fila de **A** colocándola en la memoria privada con la j columna de **B** puesta en la memoria local (Munshi *et al.*, 2012):

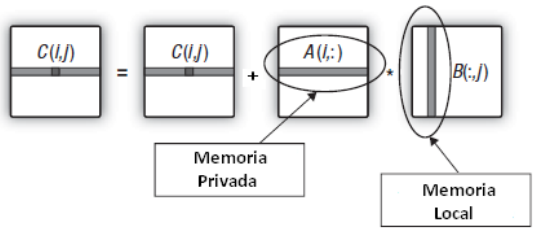

Figura 2.10: Operación de multiplicación en paralelo

2.5.3 Traspuesta de una matriz

Sea **A** una matriz con m filas y n columnas, La matriz transpuesta denotada con A^t esta dada por:

$$(A^t)_{ij} = A_{ji}, 1 \leq i \leq n, 1 \leq j \leq m \tag{2.3}$$

En donde el elemento a_{ji} de la matriz original **A** se convertirá en el elemento a_{ij} de la matriz transpuesta A^t (Merayo, 1995).

Una de las formas para paralelizar este algoritmo es escribir la columna de la matriz **A** en la fila de una matriz **B** de las mismas dimensiones, mientras al mismo tiempo se realiza la escritura de la fila de la matriz **A** en la columna de la matriz **B**, el resultado de la traspuesta quedará en esta ultima.

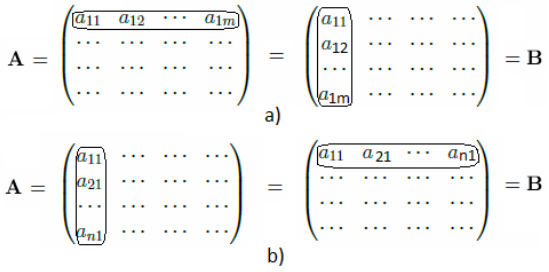

Figura 2.11: Traspuesta en paralelo: a)escribiendo filas de A en columnas de B; b)escribiendo columnas de A en filas de B

Esto se puede implementar como paralelo de tareas en OpenCL, en el que se pueden hacer 2 o mas tareas a la vez (Fixstars, 2009).

2.5.4 Inversa de una matriz

Una matriz cuadrada **A** de orden n se dice que es invertible, no singular o regular si existe otra matriz cuadrada de orden n, llamada matriz inversa de **A** y representada como A^{-1}, tal que:

$$A \cdot A^{-1} = A^{-1} \cdot A = I_n \tag{2.4}$$

Donde I_n es la matriz identidad de orden n y el producto utilizado es el producto de matrices.

El método de Gauss-Jordan para encontrar la inversa de una matriz es uno de los mas conocidos; este método se puede dividir en 2 pasos:

1. Convertir los elementos a_{ii} en uno, por la transformación: $F_i \leftarrow \frac{F_i}{a_{ii}}$. Si a_{ii} es cero, se cambia una fila por otra que no contenga cero antes de realizar esta operación.

2. Reducir los otros elementos de la j^{it} columna a cero seguido de la transformación de todas las filas excepto la numero j^{it} de la siguiente manera: $F_i \leftarrow F_i - F_j$ x a_{ii}.

Después de la transformación de la primera columna, la matriz se reduce a:

$$\mathrm{A'} = \left[\begin{array}{ccccc|ccccc} 1 & a_{12}/a_{11} & a_{13}/a_{11} & \cdots & \cdots & 1/a_{11} & 0 & 0 & \cdots & 0 \\ 0 & a_{22} - a_{21} \times a_{12}/a_{11} & a_{23} - a_{21} \times a_{13}/a_{11} & \cdots & \cdots & -a_{21}/a_{11} & 1 & 0 & \cdots & 0 \\ 0 & a_{32} - a_{31} \times a_{12}/a_{11} & a_{33} - a_{31} \times a_{13}/a_{11} & \cdots & \cdots & -a_{31}/a_{11} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} - a_{n1} \times a_{12}/a_{11} & a_{n3} - a_{n1} \times a_{13}/a_{11} & \cdots & \ddots & -a_{n1}/a_{11} & 0 & 0 & \cdots & 1 \end{array}\right]$$

Figura 2.12: Transformación de la primera columna

El proceso continua con la siguiente fila hasta obtener una matriz diagonal solo de unos, y una matriz triangular inferior de ceros. El resultado de la inversa es la matriz que se encuentra del lado derecho, donde se encontraba la matriz identidad en un principio.

En este algoritmo se paralelizan los dos pasos mencionados. En el primero se procesan n elementos al mismo tiempo:

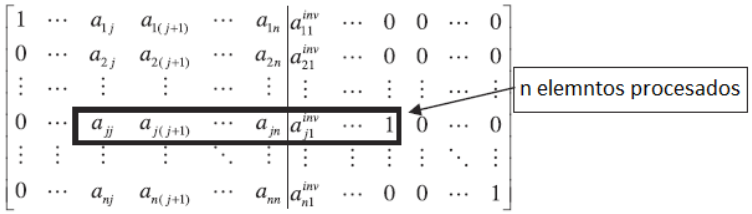

Figura 2.13: Conversión de elementos a_{ii} a uno y cambiando la fila en $F_i \leftarrow \frac{F_i}{a_{ii}}$

En el segundo paso se procesan hasta n^2 elementos al mismo tiempo, este proceso impacta de manera significativa en la velocidad de cálculo de este algoritmo (Sharma *et al.*, 2013):

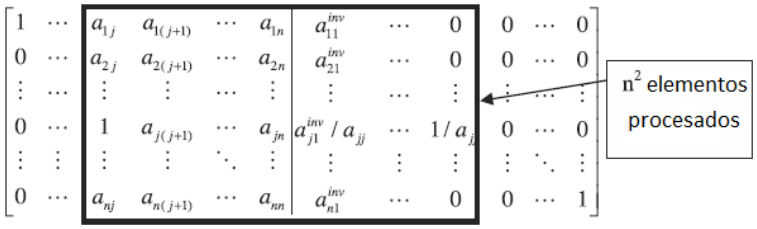

Figura 2.14: Reducción de los elementos de la j columna a ceros, transformando $F_i \leftarrow F_i - F_j$ x a_{ii}

2.5.5 Determinante de una matriz

El determinante de una matriz determina si los sistemas son singulares o mal condicionados, se utiliza para determinar la existencia y la unicidad de los resultados de los sistemas de ecuaciones lineales (Cruz, M., 2010).

Uno de los métodos que se utilizan para el cálculo del determinante es el de Gauss, consiste en transformar la matriz **A** en una matriz de ceros en su triangular inferior:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \longrightarrow \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ 0 & b_{22} & \cdots & b_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & b_{mn} \end{pmatrix}$$

El resultado de la determinante será el producto de los elementos de la diagonal principal $(b_{11} \cdot b_{22} \cdot ... \cdot b_{mn})$. Al emplear el método de Gauss para transformar

la matriz en otra equivalente pero con todos los elementos nulos por debajo de la diagonal principal hay que tener en cuenta varias reglas al igual que para el cálculo de una inversa de una matriz (Calameo, 2011).

El cálculo de la determinante en paralelo se realiza de la misma manera que el presentado para la inversa de una matriz, basta con guardar el valor de los elementos de la diagonal principal y finalmente obtener el producto de estos dando como resultado el valor de la determinante.

2.5.6 Covarianza de una matriz

La matriz de covarianza es una matriz que contiene la covarianza entre los elementos de un vector. Es la generalización natural a dimensiones superiores del concepto de varianza de una variable aleatoria escalar.

Dados n^{it} conjuntos de variables denotando $X_1, ..., X_n$, la matriz de covarianza es definida por:

$$V_{ij} = cov(x_i, x_j) = (x_i - \mu_i)(x_j - \mu_j), \tag{2.5}$$

Donde μ_i es la media (Snecdocter & Cochran, 1980). Para matrices de mayor tamaño se tiene :

$$V_{ij}^{mn} = (x_i - \mu_i)^m (x_j - \mu_j)^n, \tag{2.6}$$

La anterior definición es equivalente a la igualdad matricial :

$$Cov = \frac{(x_{ij} - \mu_i)(x_{ij} - \mu_j)^T}{m - 1}, \tag{2.7}$$

Donde m representa el número de filas que contiene la matriz. La paralelización en este algoritmo se realiza con los métodos descritos anteriormente para paralelizar la multiplicación y la traspuesta de matrices, ya que como se puede apreciar, se realiza una multiplicación entre los valores de la matriz con la media substraida y los valores de la traspuesta de esta.

2.5.7 Descomposición QR

La descomposición o factorización QR de una matriz es una descomposición de la misma como producto de una matriz ortogonal por una triangular superior. La descomposición QR es la base del algoritmo QR utilizado para el cálculo de los vectores y valores propios de una matriz (Scarpino, 2012).

Dada una matriz $\mathbf{A}$(no necesariamente cuadrada), con columnas linealmente independientes, encontraremos matrices $\mathbf{Q}$, $\mathbf{R}$ tales que:

1. $\mathbf{A} = \mathbf{QR}$.
2. Las columnas de $\mathbf{Q}$ son ortonormales.
3. $\mathbf{Q}$ es del mismo tamaño que $\mathbf{A}$.
4. $\mathbf{R}$ es triangular superior invertible.

Para obtener esta descomposición se utiliza el método de ortogonalización de Gram-Schmidt (López, C., 2009).

Dado $\mathbf{A} = (\mathbf{a}_1|\cdots|\mathbf{a}_n)$. Entonces:

$\mathbf{u}_1 = \mathbf{a}_1, \qquad \mathbf{e}_1 = \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|}$

$\mathbf{u}_2 = \mathbf{a}_2 - \text{proj}_{\mathbf{e}_1} \mathbf{a}_2, \qquad \mathbf{e}_2 = \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|}$

$\mathbf{u}_3 = \mathbf{a}_3 - \text{proj}_{\mathbf{e}_1} \mathbf{a}_3 - \text{proj}_{\mathbf{e}_2} \mathbf{a}_3, \qquad \mathbf{e}_3 = \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|}$

$\vdots$

$\mathbf{u}_k = \mathbf{a}_k - \sum_{j=1}^{k-1} \text{proj}_{\mathbf{e}_j} \mathbf{a}_k, \qquad \mathbf{e}_k = \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}$

Naturalmente, utilizamos los a_is de $\mathbf{A}$ para obtener:

$\mathbf{a}_1 = \mathbf{e}_1\|\mathbf{u}_1\|$

$\mathbf{a}_2 = \text{proj}_{\mathbf{e}_1} \mathbf{a}_2 + \mathbf{e}_2\|\mathbf{u}_2\|$

$\mathbf{a}_3 = \text{proj}_{\mathbf{e}_1} \mathbf{a}_3 + \text{proj}_{\mathbf{e}_2} \mathbf{a}_3 + \mathbf{e}_3\|\mathbf{u}_3\|$

$\vdots$

$\mathbf{a}_k = \sum_{j=1}^{k-1} \text{proj}_{\mathbf{e}_j} \mathbf{a}_k + \mathbf{e}_k\|\mathbf{u}_k\|$

Ahora estas ecuaciones se escriben en forma matricial de esta manera:

$$(\mathbf{e}_1 \,|\ldots|\, \mathbf{e}_n) \begin{pmatrix} \|\mathbf{u}_1\| & \langle \mathbf{e}_1, \mathbf{a}_2 \rangle & \langle \mathbf{e}_1, \mathbf{a}_3 \rangle & \ldots \\ 0 & \|\mathbf{u}_2\| & \langle \mathbf{e}_2, \mathbf{a}_3 \rangle & \ldots \\ 0 & 0 & \|\mathbf{u}_3\| & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Alternativamente, la matriz $\mathbf{R}$ puede calcularse de la siguiente manera: Recordemos que: $\mathbf{Q} = (\mathbf{e}_1 \,|\ldots|\, \mathbf{e}_n)$. Entonces, tenemos:

$$R = Q^T A = \begin{pmatrix} \langle \mathbf{e}_1, \mathbf{a}_1 \rangle & \langle \mathbf{e}_1, \mathbf{a}_2 \rangle & \langle \mathbf{e}_1, \mathbf{a}_3 \rangle & \ldots \\ 0 & \langle \mathbf{e}_2, \mathbf{a}_2 \rangle & \langle \mathbf{e}_2, \mathbf{a}_3 \rangle & \ldots \\ 0 & 0 & \langle \mathbf{e}_3, \mathbf{a}_3 \rangle & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{2.8}$$

La paralelización se puede realizar en cada una de las operaciones vectoriales que este método contiene.

2.6 Algoritmos de Clasificación

Los algoritmos de clasificación se utilizan en problemas en los cuales se conoce a priori el número de clases y los representantes de cada clase. Básicamente consisten en que, para clasificar automáticamente una nueva muestra, se tiene en cuenta la información que se extrae de un conjunto de objetos disponibles divididos en clases y la decisión de una regla de clasificación o clasificador.

Los algoritmos dedicados al problema de la clasificación operan usualmente sobre la información suministrada por un conjunto de muestras, patrones, ejemplos o prototipos de entrenamiento que son asumidos como representantes de las clases, y los mismos poseen una etiqueta de clase correcta. A este conjunto de prototipos correctamente etiquetados se le llama conjunto de entrenamiento y es el conocimiento empleado para la clasificación de nuevas muestras.

Estos algoritmos tienen como objetivo determinar cuál es la clase, de las que ya se tiene conocimiento, a la que debe pertenecer una nueva muestra, teniendo en cuenta la información que se extrae del conjunto de entrenamiento. En esta

tesis se presentan tres de estos algoritmos de clasificación: vecinos cercanos, discriminante lineal y discriminante cuadrático (Pang-Ning *et al.*, 2006).

2.6.1 Vecinos Cercanos

Vecinos cercanos (KNN) es un método de clasificación supervisada que sirve para estimar la función de densidad $F(x/C_j)$ de las predictoras x por cada clase C_j. Es usado como método de clasificación de objetos basado en un entrenamiento mediante ejemplos cercanos en el espacio de los elementos (Fix & Hodges, 1989). Los ejemplos de entrenamiento son vectores en un espacio característico multidimensional, cada ejemplo está descrito en términos de p atributos considerando q clases para la clasificación. Los valores de los atributos del i^{it}-esimo ejemplo (donde $1 \leq i \leq n$) se representan por el vector p-dimensional:

$$x_i = (x_{1i}, x_{2i}, ..., x_{pi}) \in X$$

El espacio es particionado en regiones por localizaciones y etiquetas de los ejemplos de entrenamiento. Un punto en el espacio es asignado a la clase C si esta es la clase más frecuente entre los k ejemplos de entrenamiento más cercano. Una de las medidas que se utilizan para encontrar la distancia entre vecinos es la distancia euclidiana. Sea $D(x, y)$ la distancia entre dos puntos, la distancia euclidiana se define como:

$$D(x, y) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2}, \tag{2.9}$$

Donde n es el número de dimensiones y x_k y y_k son respectivamente los atributos (Pang-Ning *et al.*, 2006). La fase de entrenamiento del algoritmo consiste en almacenar los vectores característicos y las etiquetas de las clases de los ejemplos de entrenamiento. En la fase de clasificación, la evaluación del ejemplo es representada por un vector en el espacio característico. Se calcula la distancia entre los vectores almacenados y el nuevo vector, y se seleccionan los k^{it} ejemplos más cercanos. El nuevo ejemplo es clasificado con la clase que más se repite en los vectores seleccionados.

Este método supone que los vecinos más cercanos nos dan la mejor clasificación y esto se hace utilizando todos los atributos. El problema de dicha suposición es que es posible que se tengan muchos atributos irrelevantes que dominen sobre la clasificación.

2.6.2 Análisis discriminante lineal y cuadrático

Supongamos que un conjunto de objetos se clasifica en una serie de grupos, el análisis discriminante equivale a un análisis de regresión donde la variable dependiente es categórica y tiene como categorías la etiqueta de cada uno de los grupos. Las variables independientes son continuas y determinan a que grupos pertenecen los objetos. Se requiere encontrar relaciones lineales entre las variables continuas que mejor discriminen en los grupos dados a los objetos. Además, se trata de definir una regla de decisión que asigne un objeto nuevo, que no sabemos clasificar previamente, a uno de los grupos prefijados.

En el caso de un discriminante lineal (LDA), se busca separar dos clases mediante un hiperplano en dos regiones. Las funciónes LDA para cada clase son dadas por la siguiente expresión:

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T + \log \pi_k \tag{2.10}$$

Donde Σ es la covarianza de los datos, μ_k es el vector de medias y π_k son las probabilidades de clase. De esta manera la regla de clasificaón es asignar la clase al discriminante mas alto.

En un discriminante cuadrático (QDA) el hiperplano generado es una superficie de naturaleza cuadrática (William *et al.*, 1979). Las funciónes QDA para cada clase son dadas por la siguiente expresión:

$$\delta_k(x) = -\frac{1}{2} \log |\Sigma_k| - \frac{1}{2} (x - \mu_k)^T \Sigma_k^{-1} (x - \mu_k) + \log \pi_k \tag{2.11}$$

Donde $|\Sigma_k|$ es el determinante de la covarianza de clase k.

Capítulo 3

Equipo y Métodos

En este capítulo explicaremos la metodología seguida para desarrollar el trabajo de investigación. Tambíen se detalla el hardware y el software utilizados para la creación y experimentación de los algoritmos.

3.1 Metodología

La metodología utilizada para este trabajo es el desarrollo de aplicaciones rápidas (RAD)(James, 1990), la cual favorece la obtención rápida de prototipos con un mínimo de planeación. La metodología RAD consta de cuatro fases o etapas:

1. Planeación de requerimientos: En esta etapa, todo el personal involucrado (usuarios, administradores y programadores) discuten y acuerdan sobre necesidades, alcances, restricciones y requerimientos de sistemas. Los principales usuarios de las librerias serán involucrados, como investigadores y estudiantes. En esta fase se planeo introducir las operaciones matriciales básicas para que trabajen con GPU(suma, multiplicación, traspuesta, inversa, determinante y covarianza de una matiz).

2. Diseño del usuario: Aquí, los usuarios finales interactúan con los analistas de sistemas y se desarrollan los modelos y prototipos de todo el sistema en su conjunto. Este es un proceso interactivo que permite a los usuarios entender, modificar y finalmente aprobar el modelo final que cumpla con los requerimientos. Se detalló la forma en que los algoritmos iban a hacer implementados dentro de OpenCL para la obtención de mejor desempeño.

3. Construcción: Esta etapa consiste propiamente de la construcción o programación del modelo.

4. Corte y cambio: En esta última fase se realizan pruebas finales de los prototipos, se hacen experimentos para comprobar su correcta funcionalidad, empleándose cambios si el prototipo lo requiere. Para comprobar su correcta funcionalidad se compararon resultados de estas operaciones matriciales con el software matemático MATLAB y para comprobar su eficiencia se creó también una librería para CPU que contiene estas mismas funciones que trabajan con el GPU, realizándose comparaciónes de tiempos entre funciones. Por último, se implementó esta librería en 3 algoritmos de clasificación utilizados en minería de datos(k-vecinos cercanos, discriminante lineal y discriminante cuádratico) para ver el comportamiento en cuanto a la velocidad de procesamiento en comparación con una librería de trabajo matricial para CPU. Tambíen se realizó un manual de uso de cada una de las funciones de la librería que se mostrará en los anexos de esta tesis.

3.2 Hardware y software utilizados

Para comprobar la correcta funcionalidad de las diferentes funciones realizadas se utilizó el software matemático MATLAB de la compañia Mathworks en su versión R2013b, corriendo sobre Microsoft Windows 8.1 Professional. Para la creación de los algoritmos y la experimentación de tiempos entre CPU y GPU se utilizó el lenguaje de programación OpenCL creado por la empresa Khronos en su versión 1.2 en un sistema operativo linux sobre un clúster de computación con 24 procesadores Intel (R) CPU x565 @2.679Gz y un sistema de memoria de 94 Gb. Los algoritmos fueron ejecutados de forma paralela gracias a la utilización de la tarjeta GPU tesla C2075 que cuenta con 448 núcleos de procesamiento montada dentro del clúster de computación. También se utilizó la herramienta para documentación de código doxygen en su versión 1.8.9 con lo que se documento la librería.

Capítulo 4

Resultados

En este capítulo se presentarán los resultados experimentales que fueron logrados mediante la aplicación de las funciones creadas en la librería para cálculo matricial utilizando GPU. En la primera sección se mostrarán las funciones creadas y las bases de datos utilizadas. En la segunda sección se muestran las gráficas de comparación de tiempos entre GPU y CPU para cada una de las funciones creadas y por último se muestran gráficas de comparación de tiempos entre GPU y CPU para cada algoritmo de clasificación utlizado en minería de datos.

4.1 Funciones creadas y descripción de los datos

Se creó una librería para OpenCL que cuenta con las funciones descritas a continuación:

1. GPUOpen.- Inicializa plataforma, device (GPU), contexto y cola de comandos.
2. GPUClose.- Cierra contexto y cola de comandos.
3. ShowMat.- Muestra el resultado de una matriz.
4. SaveMat.- Guarda el valor de una matriz en texto.
5. Dimensions.- Obtiene las dimensiones del archivo a analizar.
6. Data.- Obtiene los valores del archivo a analizar.

7. ParSum.- Realiza la suma de una matriz a vector en paralelo
8. ParMatSum.- Realiza la suma de 2 matrices en paralelo
9. ParMatRes.- Realiza la resta de 2 matrices en paralelo
10. ParTranspose.- Realiza la traspuesta de una matriz en paralelo
11. ParMatMult.- Realiza la multiplicación de 2 matrices en paralelo
12. ParInv.- Realiza la inversa de una matriz en paralelo
13. ParCov.- Realiza la covarianza de una matriz en paralelo
14. ParDet.- Realiza la determinante de una matriz en paralelo
15. ParQR.- Realiza la descomposición QR en paralelo.

Para comprobar tanto el correcto funcionamiento como la velocidad de procesamiento de cada uno de los algoritmos implementados con GPU en comparación con la CPU, se crearon bases de datos artificiales de gran dimensión usando un generador web encontrado en (Torres *et al.*, 2014). A continuación se muestran las diversas tablas correspondientes a las bases de datos de cada uno de los experimentos realizados:

Nombre	Muestras	Atributos	Origen	Tipo
Matriz1000x100	1000	100	Artificial	Continuos
Matriz5000x100	5000	100	Artificial	Continuos
Matriz10000x100	10000	100	Artificial	Continuos
Matriz15000x100	15000	100	Artificial	Continuos

Tabla 4.1: Bases de datos utilizadas en algoritmos de clasificación KNN y QDA para pocos atributos.

Nombre	Muestras	Atributos	Origen	Tipo
Matriz1000x500	1000	500	Artificial	Continuos
Matriz1500x1000	1500	1000	Artificial	Continuos
Matriz2200x1500	2200	1500	Artificial	Continuos
Matriz3000x2000	3000	2000	Artificial	Continuos
Matriz3600x2500	3600	2500	Artificial	Continuos
Matriz4500x3000	4500	3000	Artificial	Continuos
Matriz5020x3430	5020	3430	Artificial	Continuos

Tabla 4.2: Bases de datos utilizadas en algoritmos de clasificación KNN y LDA para muchos atributos.

Nombre	Muestras	Atributos	Origen	Tipo
Matriz500x100	500	100	Artificial	Continuos
Matriz1200x300	1200	300	Artificial	Continuos
Matriz2000x500	2000	500	Artificial	Continuos
Matriz2600x700	2600	700	Artificial	Continuos
Matriz3200x1000	3200	1000	Artificial	Continuos

Tabla 4.3: Bases de datos utilizadas en algoritmo de clasificación QDA para muchos atributos.

Nombre	Muestras	Atributos	Origen	Tipo
Matriz500x500	500	500	Artificial	Continuos
Matriz1000x1000	1000	1000	Artificial	Continuos
Matriz1500x1500	1500	1500	Artificial	Continuos
Matriz2000x2000	2000	2000	Artificial	Continuos
Matriz2500x2500	2500	2500	Artificial	Continuos
Matriz3000x3000	3000	3000	Artificial	Continuos
Matriz3500x3500	3500	3500	Artificial	Continuos
Matriz4000x4000	4000	4000	Artificial	Continuos

Tabla 4.4: Bases de datos para funciones utilizadas en la librería.

4.2 Experimentación de tiempos de procesamiento en operaciones matriciales

Se realizó una corrida de experimentos en cada una de las 5 funciones utilizadas dentro de la librería, estas se muestran a continuación :

1. Suma de matrices.
2. Multiplicación de matrices.
3. Inversa de una matriz.
4. Covarianza de una matriz.
5. Determinante de una matriz.

4.2.1 Experimentación en la suma de matrices

La tabla 4.5 muestra los tiempos de procesamiento obtenidos al realizar experimentos de una suma matricial procesados con el CPU y el GPU. La figura 4.1 muestra la gráfica de las bases de datos vs tiempo de duración de cada uno de los procesos, tanto del CPU como el GPU.

Tamaño de la matriz	Tiempo de procesamiento (seg.)	
	Proceso en serie (CPU)	Proceso en paralelo (GPU)
500x500	0.01	0.72
1000x1000	0.02	0.75
1500x1500	0.06	0.8
2000x2000	0.1	0.88
2500x2500	0.18	0.92
3000x3000	0.28	1
3500x3500	0.38	1.1
4000x4000	0.56	1.2

Tabla 4.5: Tiempos de procesamiento para suma de matrices

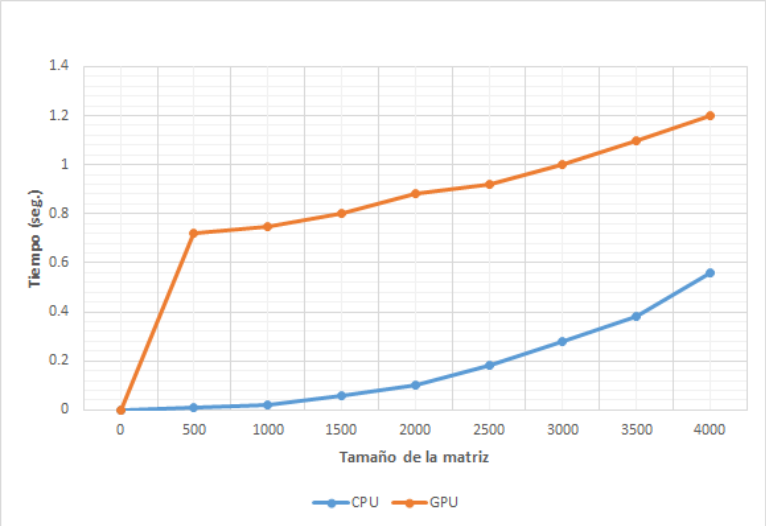

Figura 4.1: Comparación de tiempos para suma de matrices

4.2.2 Experimentación en la multiplicación de matrices

La tabla 4.6 muestra los tiempos de procesamiento obtenidos al realizar experimentos de una multiplicación matricial procesados con el CPU y el GPU. La figura 4.2 muestra la gráfica de las bases de datos vs tiempo de duración de cada uno de los procesos, tanto del CPU como el GPU. El tiempo de los datos se muestra en forma logarítmica ya que la diferencia de tiempos fue considerable.

Tamaño de la matriz	Tiempo de procesamiento, log(seg.)	
	Proceso en serie (CPU)	Proceso en paralelo (GPU)
500x500	0.041392685	-0.096910013
1000x1000	0.929418926	0.113943352
1500x1500	1.600972896	0.278753601
2000x2000	2.037426498	0.477121255
2500x2500	2.358125285	0.51851394
3000x3000	2.593618308	0.770852012
3500x3500	2.830781076	0.857332496
4000x4000	2.983851719	0.995635195

Tabla 4.6: Tiempos de procesamiento para multiplicación de matrices

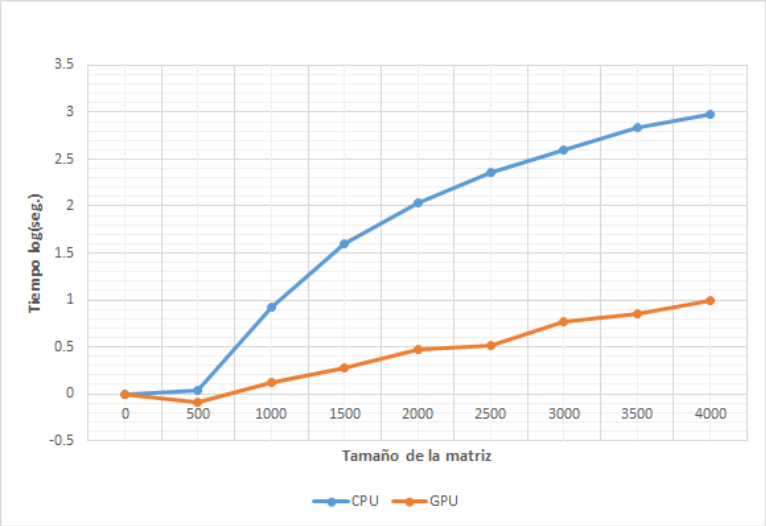

Figura 4.2: Comparación de tiempos para multiplicación de matrices

4.2.3 Experimentación en la inversa de una matriz

La tabla 4.7 muestra los tiempos de procesamiento obtenidos al realizar experimentos de la inversa de una matriz procesados con el CPU y el GPU. La figura 4.3 muestra la gráfica de las bases de datos vs tiempo de duración de cada uno de los procesos, tanto del CPU como el GPU. El tiempo de los datos también se muestra en forma logarítmica.

Tamaño de la matriz	Tiempo de procesamiento, log(seg.)	
	Proceso en serie (CPU)	Proceso en paralelo (GPU)
500x500	0.113943352	0.041392685
1000x1000	0.959041392	0.505149978
1500x1500	1.491361694	0.954242509
2000x2000	1.860936621	1.31386722
2500x2500	2.151676231	1.547774705
3000x3000	2.388633969	1.756636108
3500x3500	2.588831726	1.946943271
4000x4000	2.762678564	2.107888025

Tabla 4.7: Tiempos de procesamiento para la inversa de una matriz

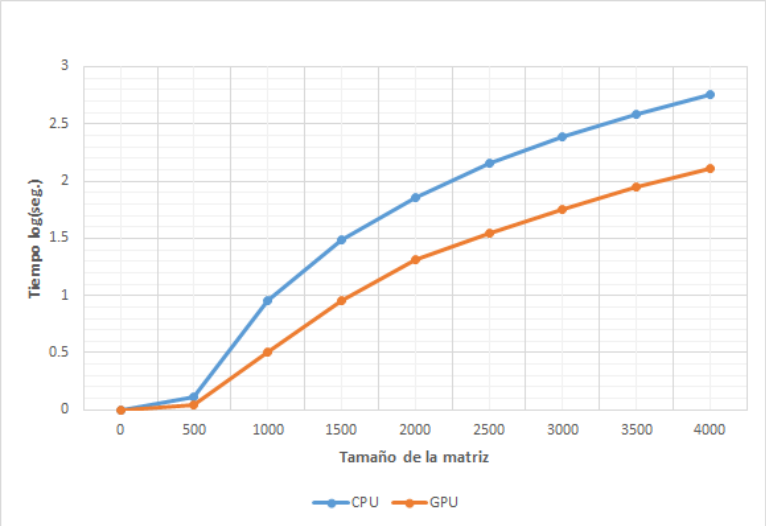

Figura 4.3: Comparación de tiempos para la inversa de una matriz

4.2.4 Experimentación en la covarianza de una matriz

La tabla 4.8 muestra los tiempos de procesamiento obtenidos al realizar experimentos de la Covarianza de una matriz procesados con el CPU y el GPU. La figura 4.4 muestra la gráfica de las bases de datos vs tiempo de duración de cada uno de los procesos, tanto del CPU como el GPU. El tiempo de los datos también se muestra en forma logarítmica.

Tamaño de la matriz	Tiempo de procesamiento, log(seg.)	
	Proceso en serie (CPU)	Proceso en paralelo (GPU)
500x500	0.255272505	0
1000x1000	0.991226076	0.204119983
1500x1500	1.614897216	0.397940009
2000x2000	2.046495164	0.612783857
2500x2500	2.361916619	0.698970004
3000x3000	2.600646236	0.924279286
3500x3500	2.833338389	1.025305865
4000x4000	2.979275148	1.161368002

Tabla 4.8: Tiempos de procesamiento para la covarianza de una matriz

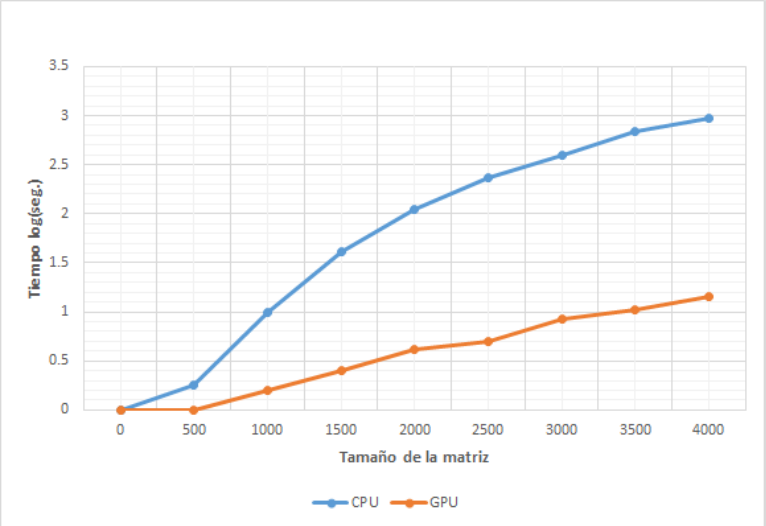

Figura 4.4: Comparación de tiempos para la covarianza de una matriz

4.2.5 Experimentación en la determinante de una matriz

La tabla 4.9 muestra los tiempos de procesamiento obtenidos al realizar experimentos de la Determinante de una matriz procesados con el CPU y el GPU. La figura 4.5 muestra la gráfica de las bases de datos vs tiempo de duración de cada uno de los procesos, tanto del CPU como el GPU. El tiempo de los datos también se muestra en forma logarítmica.

Tamaño de la matriz	Tiempo de procesamiento, log(seg.)	
	Proceso en serie (CPU)	Proceso en paralelo (GPU)
500x500	-0.251811973	-0.744727495
1000x1000	0.621176282	0.08278537
1500x1500	1.143639235	0.551449998
2000x2000	1.512417549	0.911690159
2500x2500	1.79940948	1.224014811
3000x3000	2.035669835	1.405517107
3500x3500	2.236234863	1.627775375
4000x4000	2.410186529	1.761852694

Tabla 4.9: Tiempos de procesamiento para la determinante de una matriz

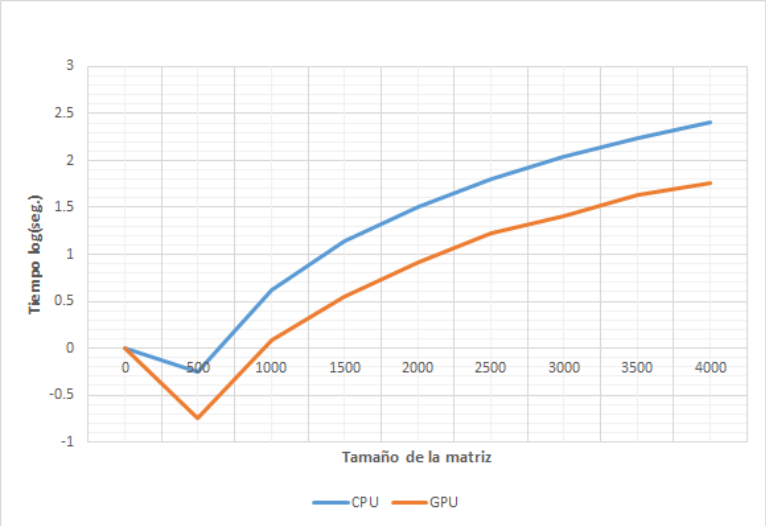

Figura 4.5: Comparación de tiempos para la determinante de una matriz

4.3 Experimentación de tiempos de procesamiento en algoritmos de clasificación

Se realizó una corrida de experimentos en cada uno de los 3 algoritmos de clasificación utilizados en minería de datos, estos son: vecinos cercanos (KNN),Análisis discriminante lineal (LDA) y Análisis discriminante cuadrático (QDA). Todas las bases de datos analizadas fueron dividas en un 70% para fase de entrenamiento y un 30% para fase de prueba.Cabe mencionar que el uso de la herramienta MATLAB solo fue utilizada para comprobar que los resultados que se obtuvieron de los experimentos realizados fueran correctos, pero la comparación de tiempos entre la CPU y GPU se hizo únicamente con el cluster de computación con las especificacioens mencionadas anteriormente.

4.3.1 Experimentación en vecinos cercanos

Para este estudio se opto por hacer 2 experimentos, uno con bases de datos que contuvieran pocos atributos, y otro, con muchos. Para la obtención de la distancia entre vecinos, se puede utilizar la distancia euclidiana que se define de la siguiente

forma:

$$D(x,y) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2} \tag{4.1}$$

La implementación en OpenCL con la librería creada se utilizó en el cálculo de esta distancia. El pseudocódigo de este proceso se muestra a continuacíon:

Algoritmo 1 Cálculo de la distancia euclidiana

Entrada: $TRN = \{Trn[1], ..., Trn[j]\}$ datos de entrenamiento, $TST = \{Tst[1], ..., Tst[i]\}$ datos de prueba,N=Tamaño de datos de entrenamiento
Salida: D[j]=Vector de distancias euclidianas

1: **para** $j = 1$: N **hacer**
2: $D[j] = \sqrt{\sum_{k=1}^{m}(Trn[j]_k - Tst[i]_k)^2}$
3: **fin para**

De donde se obtienen las distancias de cada una de los datos de entrenamiento con respecto al dato de test analizado. En la tabla 4.10 y gráfica 4.6 se muestran los resultados de procesamiento obtenidos con pocos atributos para este clasificador (el tiempo de los datos se muestra de forma logarítmica). En la tabla 4.11 y gráfica 4.7 se muestran los resultados para muchos atributos.

Muestras	Tiempo CPU (log(seg.))	Tiempo GPU (log(seg.))
1000	0.22	2.26
5000	2.15	3.67
10000	3.03	4.29
15000	3.56	4.59

Tabla 4.10: Comparación de tiempos KNN (100 atributos)

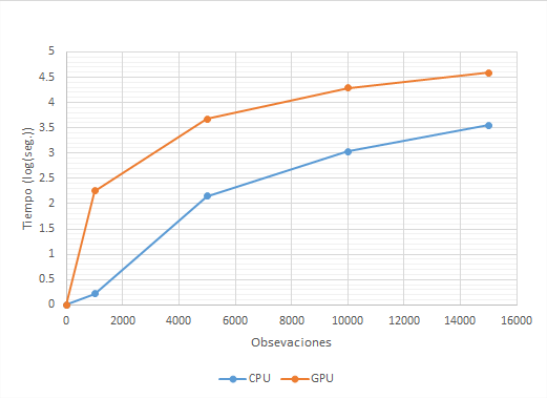

Figura 4.6: Comparación de tiempos para algoritmo de clasificación KNN con pocos atributos

Muestras	Atributos	Tiempo CPU (log(seg.))	Tiempo GPU (log(seg.))
1000	500	0.531478917	2.341236623
1500	1000	1.105510185	2.78008678
2200	1500	1.623042434	3.182799697
3000	2000	1.990072335	3.522452059
3600	2500	2.235275877	3.751245222
4500	3000	2.511067458	3.994726613
5020	3430	2.68003621	4.13159974

Tabla 4.11: Comparación de tiempos KNN (muchos atributos)

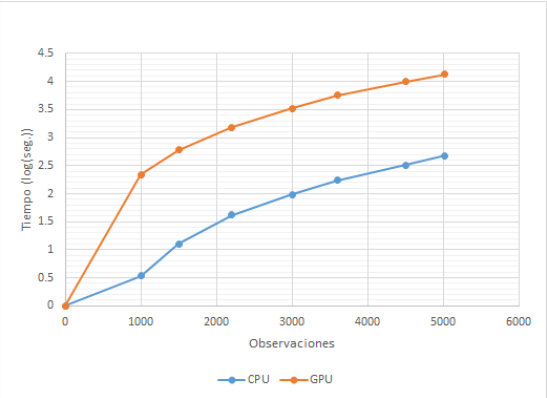

Figura 4.7: Comparación de tiempos para algoritmo de clasificación KNN con muchos atributos

4.3.2 Experimentación en análisis discriminante lineal

Las funciones LDA para cada clase están dadas por la siguiente expresión:

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T + \log \pi_k \tag{4.2}$$

El pseudocódigo para la obtención del discriminante por clase es el siguiente:

Algoritmo 2 Cálculo del discriminante lineal por clase

Entrada: $TST = \{Tst[1], ..., Tst[i]\}$ datos de prueba,π_k =probabilidad de clase k, μ_k = media de la clase k , N = cantidad de clases
Salida: $\delta(Tst[i])$ = Vector de discriminantes
para $k = 1$: N **hacer**
2: $\delta_k(Tst[i]) = Tst[i]^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k^T + \log \pi_k$
fin para

En la tabla 4.12 y gráfica 4.8 se muestran los resultados de procesamiento obtenidos para este clasificador, en este experimento, los datos de tiempo se encuentran en segundos. La implementación en OpenCL con la librería creada se utilizó en el cálculo de la covarianza de los datos (Σ) y su inversa (Σ^{-1}) con las funciones ParCov y ParInv.

Muestras	Atributos	Tiempo CPU (seg.)	Tiempo GPU (seg.)
1000	500	6.57	4.39
1500	1000	40.94	26.05
2200	1500	169.47	104.74
3000	2000	449	284.03
3600	2500	894.19	557.37
4500	3000	1619.97	1017.93
5020	3430	2965.31	1952.02

Tabla 4.12: Comparación de tiempos LDA

4.3.3 Experimentación en análisis discriminante cuadrático

Para este clasificador, al igual que el clasificador KNN se realizaron 2 experimentos con bases de datos de pocos y muchos atributos. Las funciones QDA para

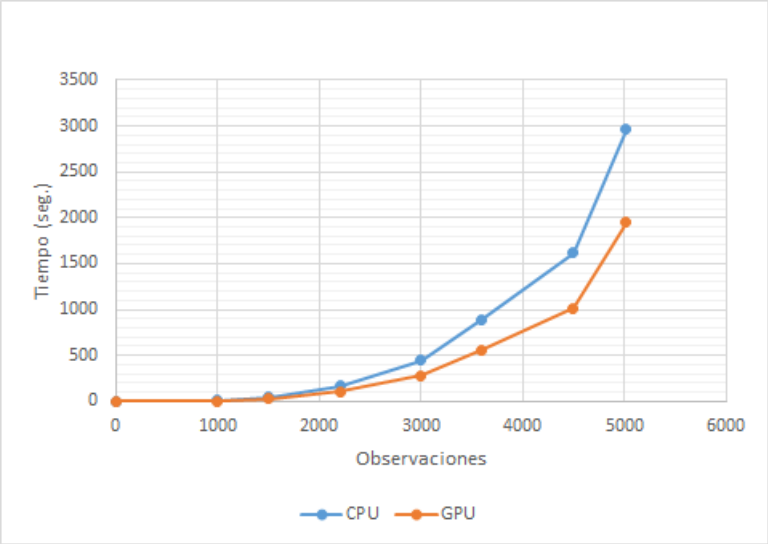

Figura 4.8: Comparación de tiempos para algoritmo de clasificación LDA

cada clase están dadas por la siguiente expresión:

$$\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log\pi_k \tag{4.3}$$

El pseudocódigo para la obtención del discriminante por clase es el siguiente:

Algoritmo 3 Cálculo del discriminante cuadrático por clase

Entrada: $TST = \{Tst[1], ..., Tst[i]\}$ datos de prueba,π_k =probabilidad de clase k, μ_k = media de la clase k , N = cantidad de clases

Salida: $\delta(Tst[i])$ = Vector de discriminantes

para $k = 1$: N **hacer**

$\delta_k(Tst[i]) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(Tst[i]-\mu_k)^T\Sigma_k^{-1}(Tst[i]-\mu_k) + \log\pi_k$

3: **fin para**

Para el algoritmo con pocos atributos, se implementó la librería para el cálculo de la inversa y determinante de la covarianza (Σ^{-1} y $|\Sigma_k|$ respectivamente) por clase; Posteriormente se calcula el discriminante de cada clase. En la tabla 4.13 y gráfica 4.9 se muestran los resultados de procesamiento obtenidos con pocos atributos para este clasificador (el tiempo de los datos se muestra en segundos).

Muestras	Tiempo CPU (seg.)	Tiempo GPU (seg.)
1000	28.90	21.00
5000	601.81	391.65
10000	2582.40	1468.38
15000	6487.42	3516.65

Tabla 4.13: Comparación de tiempos QDA (100 atributos)

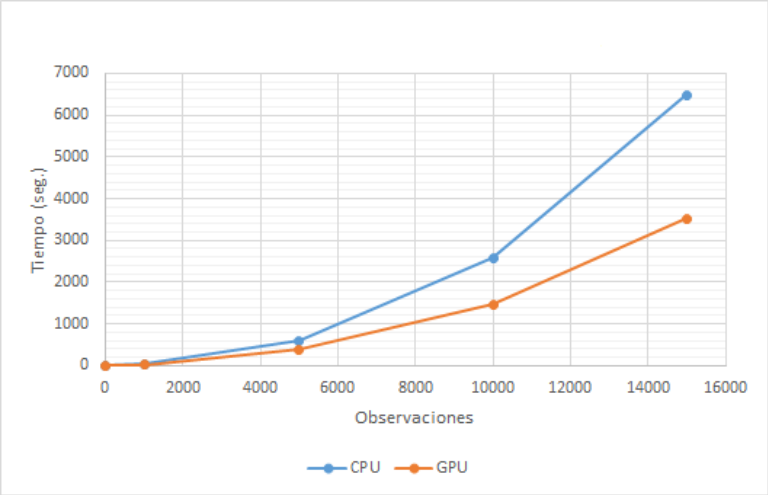

Figura 4.9: Comparación de tiempos para algoritmo de clasificación QDA con pocos atributos

Para el algoritmo con muchos atributos, se obtuvo un desbordamiento en el cálculo de la determinante de la covarianza, imposibilitando el cálculo del discriminante, por lo que se necesitó de la descomposición QR para evitar este problema y resolver el logaritmo de la determinante de la covarianza por clase, para esto se utilizó la función CPUqr de la librería. Después se puede obtener el valor del logaritmo del determinante de la covarianza sumando los logaritmos de los datos de la diagonal principal de la matriz triangular superior R obtenida con la descomposición QR. Cabe señalar que en este trabajo se realizó una implementación de esta descomposición QR con el metódo de Gram-Schmidt utilizando cálculo paralelo, sin embargo, los experimentos mostraron un mejor desempeño de este algoritmo utilizando CPU a diferencia del GPU, por lo que se implemento la descomposición QR de forma secuencial, dejándose una línea de investigación para la optimización de este algoritmo u otros métodos diferentes para futuras inves-

tigaciones. Posteriormente se puede calcular la discriminante sustituyendo este valor en la función discriminante.

En la tabla 4.14 y gráfica 4.10 se muestran los resultados para muchos atributos.

Muestras	Atributos	Tiempo CPU (seg.)	Tiempo GPU (seg.)
500	100	15.59	12.05
1200	300	843.53	472.14
2000	500	5731.4	3232.91
2600	700	20086.42	11435.31
3200	1000	68425.08	39404.12

Tabla 4.14: Comparación de tiempos QDA (muchos atributos)

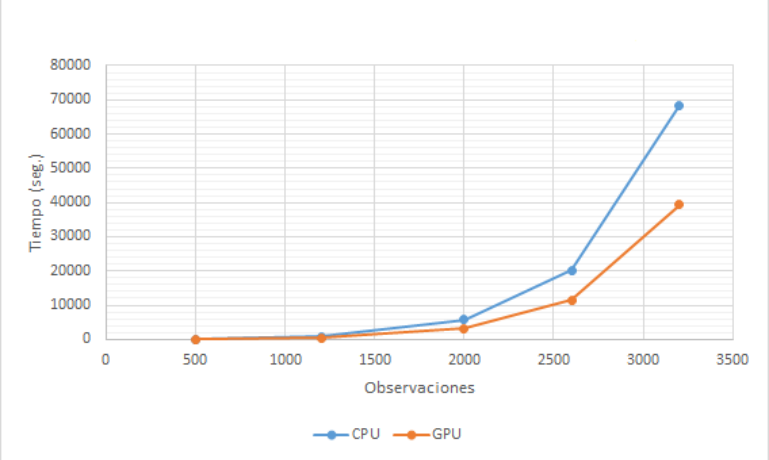

Figura 4.10: Comparación de tiempos para algoritmo de clasificación QDA con muchos atributos

4.4 Discusión de resultados

En la figura 4.11 se muestran las dependencias que cada una de las funciones tiene dentro de la librería.

Se observa que todas las funciones necesitan de la obtención de las dimensiones y de sus datos (funciones Dimensions y Data respectivamente), y que para el cálculo de una operación matricial es necesario la función de inicialización

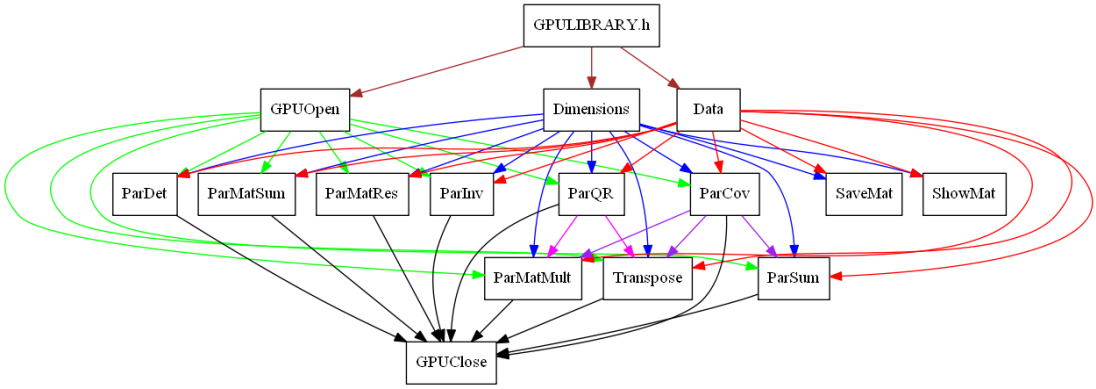

Figura 4.11: Dependencias de funciones dentro de la librería GPULIBRARY.h

GPUOpen para utilizar la tarjeta GPU como también el uso de la función GPU-Close para finalizar los comandos que utiliza OpenCL. La función ParQR requiere de las funciones Transpose y ParMatMult para realizar el cálculo, como también las utiliza la función ParCov con la inclusión de la función ParSum.

Con respecto a las funciones de la librería,se nota que para una suma de matrices la función realizada con CPU es más eficiente que la hecha en GPU, esto es porque el procesamiento de esta operación es muy sencillo, la diferencia radica en que cuando se utiliza la función de la librería, se necesita inicializar la plataforma, la tarjeta GPU, el contexto y la cola de comandos para empezar el procesamiento, esto consume una cierta cantidad de tiempo de procesamiento.

En la multiplicación de matrices si hay una gran diferencia entre ambas funciones, se observa que entre mas grande la base de datos la eficiencia de la función con GPU es mucho mejor que en CPU, en la última base de datos analizada logro ser hasta 100 veces mas rápida aproximadamente.

En la inversa de una matriz también la eficiencia de la función con GPU es mejor, obteniéndose una velocidad de hasta 4.5 veces mas rápida en comparación con la función en CPU.

En la covarianza de una matriz nuevamente la función con GPU es mucho mejor, debido a que en el cálculo de la covarianza se necesita multiplicar matrices, el resultado de la eficiencia de esta función se esperaba, teniendo una velocidad de hasta 66 veces mas rápida en comparación con la función con CPU.

Por último en la determinante de una matriz tambíen se obtuvo una velocidad de procesamiento más rápida con la GPU, teniendo un resultado de hasta 4.5 veces mejor.

En el algoritmo de clasificación KNN no se obtuvieron resultados positivos con el uso de esta librería, esto es por que las operaciones de este clasificador consisten en la obtención de distancias que se obtienen principalmente con sumas entre vectores, se notó anteriormente que la suma de matrices no es de gran desempeño en el uso de la GPU, y considerando que dentro de este algoritmo hay que estar llamando varias veces a esta función de suma, el gasto de procesamiento se acumula y por ende el procesamiento es mayor.

En el clasificador LDA, por el contrario, se obtuvieron buenos resultados con el uso de esta librería, el porcentaje de desempeño del GPU contra el CPU para cada base de datos utilizada se muestra en la tabla 4.15, se puede ver una velocidad de procesamiento superior comparado al de funciones con CPU de hasta un 38%.

Muestras	Atributos	%Velocidad
1000	500	33.18
1500	1000	36.37
2200	1500	38.20
3000	2000	36.74
3600	2500	37.67
4500	3000	37.16
5020	3430	34.17

Tabla 4.15: Eficiencia de funciones con GPU contra funciones CPU para LDA

El análisis del algoritmo de clasificación QDA fue un poco mas extenso, debido a que el algoritmo cambió con el uso de base de datos con muchos atributos. En la tabla 4.16 se puede ver (al igual que el clasificador LDA) una velocidad de procesamiento superior comparado al de funciones con CPU de hasta un 45%.

Se tuvieron problemas para la obtención del determinante de una base de datos de muchos atributos para el cálculo del discriminante en el QDA, ya que este valor se desbordaba, dando un valor muy grande, por lo que se utilizó la descomposición QR, utilizándose la suma del logaritmo de cada uno de los valores de la diagonal principal de la matriz triangular superior R, que es equivalente a la manera de calcular el logaritmo de la determinante para cada clase($log|\Sigma_k|$)

Muestras	%Velocidad
1000	27.34
5000	34.92
10000	43.14
15000	45.79

Tabla 4.16: Desempeño de funciones con GPU contra funciones CPU para QDA con 100 atributos

y así se solucionó este problema. En la tabla 4.17 se puede ver la velocidad de procesamiento que supera a funciones con CPU hasta un 44%.

Muestras	Atributos	%Velocidad
500	100	22.71
1200	300	44.03
2000	500	43.59
2600	700	43.07
3200	1000	42.41

Tabla 4.17: Desempeño de funciones con GPU contra funciones CPU para QDA con muchos atributos

En general, se obtuvieron muy buenos resultados en cuanto a los tiempos de procesamiento de cálculo matricial con la tarjeta GPU, demostrando que la librería es muy útil para la mayoría de las funciones creadas en OpenCL para cálculos matriciales.

Capítulo 5

Conclusiones

5.1 Observaciones de la experimentación

Dentro de los resultados obtenidos durante el desarrollo de esta investigación, observamos lo siguiente:

1. El realizar una función de suma de matrices para que trabaje con GPU no es de gran utilidad, ya que la operación no requiere mucho procesamiento y aparte esta tiene que cargar cierta información antes de empezar lo que provoca más tiempo de procesamiento. Para esta operación es mejor usar funciones realizadas con CPU.

2. Funciones tales como la multiplicación de matrices, inversa, covarianza y determinante de una matriz mostraron un mejor desempeño trabajando con GPU, observandose tiempos de procesamiento mucho menores a diferencia de utilizarlas con CPU.

3. Utilizar la librería en el algoritmo de clasificación KNN no fue mejor que utilizando funciones con CPU, este clasificador requiere en su mayoría sumas de vectores, y como se mencionó anteriormente, estas operaciones no tienen un buen desempeño en GPU.

4. La utilización de la librería en el algoritmo de clasificación LDA mostró un mejor desempeño a diferencia de funciones en CPU, mostrando velocidades de procesamiento mejores en las bases de datos de hasta un 38%.

5. También la utilización de la librería en el algoritmo de clasificación QDA mostró un mejor desempeño a diferencia de funciones en CPU, mostrando velocidades de procesamiento mejores en las bases de datos con pocos atributos de hasta un 45%. Para bases de datos que contengan muchos atributos en este clasificador, es necesario utilizar procesos para solucionar ($log|\Sigma_k|$) ya que el determinante se puede desbordar y por ende no tener un resultado correcto. En este trabajo se utilizó la descomposición QR para solucionar este problema. La velocidad de procesamiento para bases de datos con muchos atributos en este clasificador fue mejor con la utilización de esta librería, mejorando su desempeño hasta un 44% a diferencia de funciones en CPU.

La ventaja de esta librería es que fue creada para un lenguaje de programación abierto (OpenCL) en el que su versión es gratuita y cualquiera dispone de ella, dando la posibilidad de trabajar con tarjetas GPU para cálculo matricial sin necesidad de utilizar herramientas que cuestan dinero como MATLAB.

5.2 Trabajo Futuro

Como trabajo futuro se plantea lo siguiente:

1. Implementación de más operaciones matriciales para que trabajen con GPU tales como la raíz cuadrada de una matriz (Higham, 1986) o la descomposición en valores singulares (Wall *et al.*, 2003) , utlizadas en muchos algoritmos de minería datos, realizando estudios de procesamiento con bases de datos grandes y comparar los resultados con funciones que trabajen con CPU, agregando estas nuevas funciones a la librería creada.

2. Experimentar el desempeño de esta librería en más algoritmos de clasificación, como lo son el bayesiano ingenuo (naive bayes) (Rish, 2001), máquinas de vectores de soporte(SVM) (Cortes & Vapnik, 1995) o en el clasificador discriminante adaptativo de vecinos cercanos (DANN) (Hastie & Tibshirani, 1996).

3. Los clasificadores mencionados, forman parte de algoritmos supervisados, en los que ya se tiene un conjunto de ejemplos clasificados. También se puede experimentar el desempeño de esta librería en algoritmos no supervisados, tales como el de agrupamiento (clustering)(Kaufman & Rousseeuw, 1990) o el k-medias(Hartigan & Wong, 1979).

Referencias

Arvizu, H. (2012). *Programando para el GPU CUDA*. Consultado el 7 de enero de 2015 en `http://sg.com.mx/revista/programando-para-el-gpu-cuda`. 12

Banger, R. & Bhattacharyya, K. (2013). Memory model. In *OpenCL Programming by Example*, 52–54, PACKT. 15

Calameo (2011). *Matrices y determinantes*. Consultado el 20 de enero de 2015 en `http://en.calameo.com/read/0009436373a9d5d7b1a50`. 21

Cortes, C. & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, **20**, 273. 47

Cruz, M. (2010). *Determinante de una matriz*. Consultado el 20 de enero de 2015 en `http://www.gridmorelos.uaem.mx/ mcruz//cursos/mn/determinante.pdf`. 20

Escolano, S. (2011). *Ventajas y desventajas del GPU*. Consultado el 27 de diciembre de 2014 en `http://www.dtic.ua.es/jgpu11/material/sesion1_jgpu11.pdf`. 3

Fix, E. & Hodges, J. (1989). Commentary on fix and hodges. In *An Important Contribution to Nonparametric Discriminant Analysis and Density Estimation*, 233–238, International Statistical Review. 24

Fixstars (2009). *Calling the kernel*. Consultado el 13 de enero de 2015 en `http://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/calling-the-kernel/`. 16, 18

Flautner, K., Uhlig, R., Reinhardt, S. & Mudge, T. (2000). Thread-level parallelism and interactive performance of desktop applications. *ACM*, **28**, 129–138. 6

Flynn, M. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, **21**, 948–960. 6

Godfrey, M. (1993). First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, **15**, 27–75. 7, 8

Gutiérrez, A. (2012). *Comparativo de pantallas*. Consultado el 6 de enero de 2015 en `http://www.ciberninja.com/2012_10_01_archive.html`. 11

Hartigan, J. & Wong, M. (1979). A k-means clustering algorithm. *Journal of the Royal Statistical Society*, **28**, 100–108. 48

Hastie, T. & Tibshirani, R. (1996). Discriminant adaptive nearest neighbor classification. *IEEE transactions on pattern analysis and machine intelligence*, **18**, 607–615. 47

Higham, N. (1986). Newton's method for the matrix square root. *Mathematics of Computation*, **46**, 537–549. 47

IXBT Labs (2008). *Diferencias entre GPU y CPU*. Consultado el 26 de diciembre de 2014 en `http://ixbtlabs.com/articles3/video/cuda-1-p1.html`. 1

James, M. (1990). *Rapid Application Development*. MacMillan. 26

Jaume, I. (2010). *Arquitectura CUDA*. Consultado el 26 de diciembre de 2014 en `https://aulavirtual.uji.es/pluginfile.php/774610/mod_resource/content/0/GPU/memoria_fran_gpu.pdf`. 2

Kaufman, L. & Rousseeuw, P. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Online Library. 48

Khronos Group (2008). *OpenCL – Portable Heterogeneous Computing*. Consultado el 7 de enero de 2015 en `https://www.khronos.org/opencl/`. 13

LASS (2009). *The Von Neuman model*. Consultado el 2 de enero de 2015 en `http://none.cs.umass.edu/ dganesan/courses/fall09/handouts/Chapter4.pdf`. 7

López, C. (2009). *Ortogonalización de Gram-Schmidt*. Consultado el 22 de enero de 2015 en `http://issuu.com/kmels/docs/ortogonalizacion-de-gram-schmidt`. 22

Martin, H. (2003). A third survey of domestic electronic digital computing systems. *Computer Arithmetic*, **3**, 104–111. 6

Meneses, L. (2008). *Matrices y determinantes*. Consultado el 12 de enero de 2015 en `http://www2.eco.uva.es/lmeneses/Guia_de_Trabajo/Esquemas_teoricos/tema3.pdf`. 16

Merayo, G. (1995). Matriz traspuesta. In *Lecciones Prácticas de Cálculo Numérico*, 96, Universidad Pontifica Comillas. 18

Morell, V. (2011). *Sistemas GPU*. Consultado el 6 de enero de 2015 en `http://www.dtic.ua.es/jgpu11/material/sesion1_jgpu11.pdf/`. 10

Munshi, A., Gaster, B., Mattson, G., James, T. & Ginsburg, D. (2012). Matrix multiplication with opencl. In *OpenCL Programming Guide*, 499–513, Addison-Wesley. 17

NVIDIA (2008). *NVIDIA acelera Considerablemente la busqueda de una cura*. Consultado el 26 de diciembre de 2014 en `http://la.nvidia.com/object/prla_080708.html`. 1

NVIDIA (2010). *GPU-computing*. Consultado el 7 de enero de 2015 en `http://www.nvidia.es/object/gpu-computing-es.html`. 11, 12

Pai, V. & Ranganathan, P. (1997). The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. *High-Performance Computer Architecture*, **2**, 72–83. 6

Pang-Ning, T., Stein, M. & Kumer, V. (2006). *Introduction to Statistical Pattern Recognition*. Addison-Wesley. 24

Rish, I. (2001). An empirical study of the naive bayes classifier. *Computer Science*, 6. 47

S.A.B.I.A. (2011). *ATIvsCUDA*. Consultado el 8 de enero de 2015 en `http://sabia.tic.udc.es/gc/trabajos%202011-12/ATIvsCUDA/arquitectura.html`. 14, 15

Scarpino, M. (2012). Matrices and qr descompisition. In *OpenCL in action*, 258–277, Manning. 22

Sharma, G., Agarwala, A. & Bhattacharya, B. (2013). A fast parallel gauss jordan algorithm for matrix inversion using CUDA. *Elsevier*, **3**, 31–37. 20

Snecdocter, G. & Cochran, W. (1980). Covariance matrix. In *Statistical Methods*, 342, Ames. 21

Torres, J., Spola, N., Alvarez, E. & Monard, M. (2014). A framework to generate synthetic multi-label datasets. *Elsevier*, **5**, 155–176. 29

Torres, J. (2013). *Que es y como funciona un GPU*. Consultado el 5 de enero de 2015 en `http://hipertextual.com/archivo/2013/12/hardware-gpu-grafica/`. 10

Wall, M., Rechtsteiner, A. & Rocha, L. (2003). Singular value decomposition and principal component analysis. *SIAM*, **35**, 551–566. 47

William, R., Peter, A. & Barabara, B. (1979). How non-normality affects the quadratic discriminant function. *Communications in Statistics-Theory and Methods*, **8**, 1285–1301. 25

Xataca (2014). *Pasado,presente y futuro del GPU*. Consultado el 3 de enero de 2015 en `http://www.xataka.com/componentes-de-pc/las-gpu-como-pasado-presente-y-futuro-de-la-computacion`. 9

Anexo A

Manual de uso para librería matricial con GPU

En este anexo se muestra una descripción de cada una de las funciones y variables de la librería para que el usuario tenga un fácil aprendizaje de su utilización. Esta documentación fue realizada por la herramienta doxygen, el cuál es un generador de documentación para C++, C, Java, Objective-C, Python, IDL, VHDL y en cierta medida para PHP, C# y D. La documentación se puede encontrar en internet en el sitio http://nova.mxl.uabc.mx/~ lroberto/.

A.1 GPUOpen

Sintaxis:
GPUOpen()

Descripción:
Inicializa la plataforma, el dispositivo (GPU), el contexto y las colas de comandos a utilizar en el código.

A.2 GPUClose

Sintaxis:
GPUClose()

Descripción:
Cierra las colas de comandos y el contexto a utilizar en el código.

A.3 Dimensions

Sintaxis:
Dimensions(FILE Archivo, int Dimensiones)

Parámetros:
Archivo:

Variable de tipo File utilizado para guardar la información del archivo .txt a analizar.

Dimensiones:

Variable de tipo *int array* de 3 elementos utilizado para guardar las dimensiones de la matriz contenidas en el archivo *.txt* en donde el primer elemento guarda el número de filas, el segundo el número de columnas y el tercero la cantidad de datos que contiene el archivo.

Descripción:
Obtiene las dimensiones del archivo *.txt* a analizar (número de filas, número de columnas y cantidad de datos del archivo), guardándolos en la variable "Dimensiones". El archivo a analizar debe ser de una Matriz $M\ x\ N$ espaciada ya sea por espacios, comas o tabuladores.

A.4 Data

Sintaxis:
Data(FILE Archivo, int Dim, double Datos)

Parámetros:
Archivo:

Variable de tipo File que guarda la información del archivo .txt a analizar.

Dim:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenidas en el archivo .txt.

Datos:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores del archivo .txt.

Descripción:
Obtiene los datos del archivo .txt a analizar, guardándolos en la variable "Datos". El archivo a analizar debe ser de una Matriz M x N espaciada ya sea por espacios, comas o tabuladores.

A.5 ShowMat

Sintaxis:
ShowMat(double Datos, int Dim)

Parámetros:
Datos:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores del archivo .txt.

Dim:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "Datos".

Descripción:
Muestra el resultado en matriz de la variable "Datos" en pantalla.

A.6 SaveMat

Sintaxis:
SaveMat(FILE Archivo, double Datos, int Dim)

Parámetros:
Archivo:

Variable de tipo File en donde se guardará la variable "Datos" en archivo .txt.

Datos:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores del archivo .txt.

Dim:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "Datos".

Descripción:
Escribe los valores de la variable "Datos" en un archivo .txt que el usuario decida utilizar, los acomodará en forma de matriz espaciada por tabuladores.

A.7 Transpose

Sintaxis:
Transpose(double DatosIn, int DimDatosIn, double DatosOut, int DimDatosOut)

Parámetros:
DatosIn:

Variable de tipo double (o variable dinámica tipo double) que será la variable que se quiere procesar.

DimDatosIn:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "DatosIn".

DatosOut:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores de la traspuesta de la matriz contenida en la variable "DatosIn".

DimDatosOut:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "DatosOut".

Descripción:
Realiza la traspuesta de una matriz contenida en la variable "DatosIn" guardándola en la variable "DatosOut".

A.8 ParSum

Sintaxis:
ParSum(double DatosIn, int DimDatosIn, double DatosOut, int DimDatosOut)

Parámetros:
DatosIn:

Variable de tipo double (o variable dinámica tipo double) que será la variable que se quiere procesar.

DimDatosIn:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "DatosIn".

DatosOut:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores de la suma de las columnas de la matriz contenida en la variable "DatosIn".

DimDatosOut:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "DatosOut".

Descripción:
Realiza la suma de cada uno los elementos de cada columna contenidas en la matriz de la variable "DatosIn"; el resultado será un vector que contiene los valores de la sumatoria de cada una de las columnas de la matriz, guardándose en la variable "DatosOut".

A.9 ParMatSum

Sintaxis:
ParMatSum(double DatosIn1, double DatosIn2, int DimDatos, double DatosOut)

Parámetros:
DatosIn1:

Primera variable de tipo double (o variable dinámica tipo double) a procesar.

DatosIn2:

Segunda variable de tipo double (o variable dinámica tipo double) a procesar.

DimDatos:

Variable de tipo int array de 3 elementos que guarda las dimensiones de las matrices.

DatosOut:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores de la suma de las matrices contenidas en las variables "DatosIn1" y "DatosIn2".

Descripción:
Realiza la suma de las matrices contenidas en las variables "DatosIn1" y "DatosIn2". Las dimensiones de la matriz resultante serán las mismas que la de las iniciales.

A.10 ParMatRes

Sintaxis:
ParMatRes(double DatosIn1, double DatosIn2, int DimDatos, double DatosOut)

Parámetros:
DatosIn1:

Primera variable de tipo double (o variable dinámica tipo double) a procesar.

DatosIn2:

Segunda variable de tipo double (o variable dinámica tipo double) a procesar.

DimDatos:

Variable de tipo int array de 3 elementos que guarda las dimensiones de las matrices.

DatosOut:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores de la resta de las matrices contenidas en las variables "DatosIn1" y "DatosIn2".

Descripción:
Realiza la resta de las matrices contenidas en las variables "DatosIn1" y "DatosIn2". Las dimensiones de la matriz resultante serán las mismas que la de las iniciales.

A.11 ParMatMult

Sintaxis:
ParMatMult(double DatosIn1, double DatosIn2, int DimDatosIn1, int DimDatosIn2, double DatosOut, int DimDatosOut)

Parámetros:
DatosIn1:

Primera variable de tipo double (o variable dinámica tipo double) a procesar.

DatosIn2:

Segunda variable de tipo double (o variable dinámica tipo double) a procesar.

DimDatosIn1:

Variable de tipo int array de 3 elementos que guarda las dimensiones de las matriz contenida en la variable "DatosIn1".

DimDatosIn2:

Variable de tipo int array de 3 elementos que guarda las dimensiones de las matriz contenida en la variable "DatosIn2".

DatosOut:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores de la multiplicación entre las matrices contenidas en las variables "DatosIn1" y "DatosIn2".

DimDatosOut:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz resultante contenida en la variable "DatosOut".

Descripción:
Realiza la multiplicación de las matrices contenidas en las variables "DatosIn1" y "DatosIn2", guardándola en la variable "DatosOut". Sean las dimensiones de la matriz contenida en la variable "DatosIn1" como M x P y la matriz contenida en la variable "DatosIn2" como P x N, las dimensiones resultantes en la matriz contenida en la variable "DatosOut" serán de orden M x N.

A.12 ParInv

Sintaxis:
ParInv(double DatosIn, int DimDatosIn, double DatosOut)

Parámetros:
DatosIn:

Variable de tipo double (o variable dinámica tipo double) que será la variable que se quiere procesar.

DimDatosIn:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "DatosIn". Las dimensiones de la matriz a procesar para una inversa debe ser de orden M x M.

DatosOut:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores de la inversa de la matriz contenida en la variable "DatosIn".

Descripción:
Realiza la inversa de una matriz contenida en la variable "DatosIn" guardándola en la variable "DatosOut". Las dimensiones de la matriz resultante serán las mismas que la de las iniciales.

A.13 ParDet

Sintaxis:
ParDet(double DatosIn, int DimDatosIn, double DatosOut)

Parámetros:
DatosIn:

Variable de tipo int double (o variable dinámica tipo double) que será la variable que se quiere procesar.

DimDatosIn:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "DatosIn". Las dimensiones de la matriz a procesar para un determinante debe ser de orden M x M.

DatosOut:

Variable de tipo double que guarda el valor del determinante de la matriz contenida en la variable "DatosIn".

Descripción:
Realiza la determinante de una matriz contenida en la variable "DatosIn" guardándola en la variable "DatosOut".

A.14 ParCov

Sintaxis:
ParCov(double DatosIn, int DimDatosIn, double DatosOut, int DimDatosOut)

Parámetros:
DatosIn:

Variable de tipo double (o variable dinámica tipo double) que será la variable que se quiere procesar.

DimDatosIn:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz contenida en la variable "DatosIn".

DatosOut:

Variable de tipo double (o variable dinámica tipo double) que guarda los valores de la covarianza de la matriz contenida en la variable "DatosIn".

DimDatosOut:

Variable de tipo int array de 3 elementos que guarda las dimensiones de la matriz resultante contenida en la variable "DatosOut".

Descripción:
Realiza la covarianza de las matriz contenida en la variable "DatosIn", guardándola en la variable "DatosOut". Sean las dimensiones de la matriz contenida en la variable "DatosIn" como M x N, las dimensiones resultantes en la matriz contenida en la variable "DatosOut" serán de orden N x N.

A.15 ParQR

Sintaxis:
ParQR(double DatosIn, double Q, double R, int DimDatos)

Parámetros:
DatosIn:

Variable de tipo double (o variable dinámica tipo double) que será la variable que se quiere procesar.

Q:

Variable de tipo double (o variable dinámica tipo double) que guarda la matriz ortogonal "Q" de la matriz contenida en la variable "DatosIn".

R:

Variable de tipo double (o variable dinámica tipo double) que guarda la matriz triangular superior "R" de la matriz contenida en la variable "DatosIn".

DimDatos:

Variable de tipo int array de 3 elementos que guarda las dimensiones de las matrices.

Descripción:
Realiza la descomposición QR de la matriz contenida en la variable "DatosIn". Las dimensiones de las matrices resultantes serán las mismas que la de las iniciales.

Anexo B

Librería para trabajo matricial con GPU

```
#ifdef __APPLE__
#include <CL/opencl.h>
#else
#include <CL/cl.h>
#endif
#define MAX_SOURCE_SIZE (0x100000)

cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret;
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_context context = NULL;
cl_command_queue command_queue = NULL;
cl_command_queue command_queue2 = NULL;
```

```
void GPUOpen()
{
    /* Get Platform Information */
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);

    /* Get Device Information */
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id,
&ret_num_devices);

    /* Create OpenCL Context */
context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);

    /* Create command queue */
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
ret = clFlush(command_queue);

command_queue2 = clCreateCommandQueue(context, device_id,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &ret);
ret = clFlush(command_queue2);
}


void GPUClose()
{
ret = clFinish(command_queue);
ret = clFinish(command_queue2);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseCommandQueue(command_queue2);
ret = clReleaseContext(context);
}
```

```
void ShowMat(double *Datos,int *Dim) {
int i,j;

for (i=0; i < Dim[0]; i++){
for (j=0; j < Dim[1]; j++)
printf("%7.4f      ",Datos[i*Dim[1]+j]);
printf("\n");
}
printf("\n");
}

void SaveMat(FILE *archivo,double *Datos,int *Dim)
{
int i,j;

for(j=0;j<Dim[0];j++){
for(i=Dim[1]*j;i<(Dim[1]*j+Dim[1]);i++){
if (i!=Dim[1]*j+Dim[1]-1)
fprintf(archivo,"%.15f      ",Datos[i]);
else
fprintf(archivo,"%.15f",Datos[i]);
}
fprintf(archivo,"\n");
}
}


void Dimensions(FILE *archivo,int *Dim1)
{
int Dato=0,Filas=0,Columnas=0,ban=0,i,j;
char caracteres[50000];
char *pch;
```

```
if (archivo == NULL){
printf("File not found\n");
exit(1);
}
while (feof(archivo) == 0){
fgets(caracteres,50000,archivo);
pch=strtok(caracteres,"      ,");
while (pch!=NULL)
{
pch = strtok(NULL,"      ,");
Dato++;
if(ban==0)
Columnas++;
}
ban=1;
}
Dato - -;
Filas=Dato/Columnas;

Dim1[0]=Filas;
Dim1[1]=Columnas;
Dim1[2]=Dato;
}


void Data(FILE *archivo,int *Dim,double *Datos)
{
int i,Dato=0;
char caracteres[50000];
char *pch2;

if (archivo == NULL){
printf("File not found\n");
```

```
exit(1);
}

while (feof(archivo) == 0){
fgets(caracteres,50000,archivo);
pch2=strtok(caracteres,"       ,");
while (pch2!=NULL)
{
Datos[Dato]=atof(pch2);
pch2 = strtok(NULL,"       ,");
Dato++;
}
if (Dato==Dim[2])
break;
}
}


void Transpose(double *Datos, int *Dim, double *C, int *Dim2)
{

cl_mem Amobj = NULL;
cl_mem Cmobj = NULL;
cl_program program = NULL;
cl_kernel kernel[2] = NULL, NULL;

int i,j;
double* A;

A = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));

FILE *fp;
const char fileName[] = "./TRASPUESTA.cl";
```

```
size_t source_size;
char *source_str;

    /* Load kernel source file */
fp = fopen(fileName, "rb");
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

    /* Initialize input data */
for (i=0; i < Dim[2]; i++)
A[i] = Datos[i];

    /* Create buffer object */
Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);

    /* Copy input data to memory buffer */
ret = clEnqueueWriteBuffer(command_queue2, Amobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), A, 0, NULL, NULL);

    /* Create kernel from source */
program = clCreateProgramWithSource(context, 1,
(const char **)&source_str, (const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
```

```
    /* Create task parallel OpenCL kernel */
kernel[0] = clCreateKernel(program, "TRASPUESTA1", &ret);
kernel[1] = clCreateKernel(program, "TRASPUESTA2", &ret);

    /* Set OpenCL kernel arguments */
for (i=0; i < 2; i++) {
ret = clSetKernelArg(kernel[i], 0, sizeof(cl_mem), (void *)&Amobj);
ret = clSetKernelArg(kernel[i], 1, sizeof(cl_mem), (void *)&Cmobj);
ret = clSetKernelArg(kernel[i], 2, sizeof(int), &Dim[0]);
ret = clSetKernelArg(kernel[i], 3, sizeof(int), &Dim[1]);
}

    /* Execute OpenCL kernel as task parallel */
for (i=0; i < 2; i++) {
ret = clEnqueueTask(command_queue2, kernel[i], 0, NULL, NULL);
}

    /* Copy result to host */
ret = clEnqueueReadBuffer(command_queue2, Cmobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), C, 0, NULL, NULL);

Dim2[0]=Dim[1];
Dim2[1]=Dim[0];
Dim2[2]=Dim[2];

    /* Finalization */
ret = clReleaseKernel(kernel[0]);
ret = clReleaseKernel(kernel[1]);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Amobj);
ret = clReleaseMemObject(Cmobj);
```

```
free(source_str);
free(A);
}


void ParSum(double *Datos, int *Dim, double *C, int *Dim2)
{

cl_mem Cmobj = NULL;
cl_mem Matrizmobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;

int i, j;
double* Matriz;

Matriz = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));

FILE *fp;
const char fileName[] = "./SUMAMATRIZ.cl";
size_t source_size;
char *source_str;

    /* Load kernel source file*/
fp = fopen(fileName, "rb");
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);
```

```
    /* Initialize input data */
for (i=0; i < Dim[2]; i++)
Matriz[i] = Datos[i];

    /* Create buffer object */
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,Dim[1]*sizeof(double),
NULL, &ret);
Matrizmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);

    /* Copy input data to memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Matrizmobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), Matriz, 0, NULL, NULL);

    /* Create kernel from source */
program = clCreateProgramWithSource(context, 1,
(const char **)&source_str, (const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    /* Create task parallel OpenCL kernel */
kernel = clCreateKernel(program, "SUMAMATRIZ", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&Matrizmobj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&Cmobj);
ret = clSetKernelArg(kernel, 2, sizeof(int), &Dim[0]);
ret = clSetKernelArg(kernel, 3, sizeof(int), &Dim[1]);

size_t global_item_size = Dim[1];
size_t local_item_size = 1;

    /* Execute OpenCL kernel as data parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
```

```
&global_item_size, &local_item_size, 0, NULL, NULL);

    /* Copy result to host */
ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE, 0,
Dim[1]*sizeof(double), C, 0, NULL, NULL);

Dim2[0]=1;
Dim2[1]=Dim[1];
Dim2[2]=Dim[1];

    /* Finalization */
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Cmobj);
ret = clReleaseMemObject(Matrizmobj);

free(source_str);
free(Matriz);
}


void ParMatSum(double *MatrizA,double *MatrizB,int *Dim,double *C)
{

cl_mem Amobj = NULL;
cl_mem Bmobj = NULL;
cl_mem Cmobj = NULL;
cl_program program = NULL;
cl_kernel kernel[2] = NULL,NULL;

int i, j;
double* Matriz1,*Matriz2;
```

```
Matriz1 = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));
Matriz2 = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));

FILE *fp;
const char fileName[] = "./SUMA.cl";
size_t source_size;
char *source_str;

    /* Load kernel source file*/
fp = fopen(fileName, "rb");
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

    /* Initialize input data */
for (i=0; i < Dim[2]; i++){
Matriz1[i] = *(MatrizA+i);
Matriz2[i] = *(MatrizB+i);
}

    /* Create buffer object */
Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
Bmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
```

```
    /* Copy input data to memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Amobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), Matriz1, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, Bmobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), Matriz2, 0, NULL, NULL);

    /* Create kernel from source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

if (Dim[0] >= Dim[1]){
/* Create task parallel OpenCL kernel */
kernel[0] = clCreateKernel(program, "SUMA1", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), (void *)&Amobj);
ret = clSetKernelArg(kernel[0], 1, sizeof(cl_mem), (void *)&Bmobj);
ret = clSetKernelArg(kernel[0], 2, sizeof(cl_mem), (void *)&Cmobj);
ret = clSetKernelArg(kernel[0], 3, sizeof(int), &Dim[1]);

size_t global_item_size = Dim[0];

    /* Execute OpenCL kernel as data parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel[0], 1, NULL,
&global_item_size,NULL, 0, NULL, NULL);
}

else{
/* Create task parallel OpenCL kernel */
kernel[1] = clCreateKernel(program, "SUMA2", &ret);
```

```
    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel[1], 0, sizeof(cl_mem), (void *)&Amobj);
ret = clSetKernelArg(kernel[1], 1, sizeof(cl_mem), (void *)&Bmobj);
ret = clSetKernelArg(kernel[1], 2, sizeof(cl_mem), (void *)&Cmobj);
ret = clSetKernelArg(kernel[1], 3, sizeof(int), &Dim[0]);
ret = clSetKernelArg(kernel[1], 4, sizeof(int), &Dim[1]);
size_t global_item_size = Dim[1];

    /* Execute OpenCL kernel as data parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel[1], 1, NULL,
&global_item_size,NULL, 0, NULL, NULL);
}

    /* Copy result to host */
ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE,
0,Dim[0]*Dim[1]*sizeof(double), C, 0, NULL, NULL);

    /* Finalization */
ret = clReleaseKernel(kernel[0]);
ret = clReleaseKernel(kernel[1]);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Amobj);
ret = clReleaseMemObject(Bmobj);
ret = clReleaseMemObject(Cmobj);

free(source_str);
free(Matriz1);
free(Matriz2);

    }
```

```
void ParMatRes(double *MatrizA,double *MatrizB,int *Dim,double *C)
{

cl_mem Amobj = NULL;
cl_mem Bmobj = NULL;
cl_mem Cmobj = NULL;
cl_program program = NULL;
cl_kernel kernel[2] = NULL,NULL;

int i, j;
double* Matriz1,*Matriz2;

Matriz1 = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));
Matriz2 = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));

FILE *fp;
const char fileName[] = "./RESTA.cl";
size_t source_size;
char *source_str;

    /* Load kernel source file*/
fp = fopen(fileName, "rb");
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

    /* Initialize input data */
for (i=0; i < Dim[2]; i++){
Matriz1[i] = *(MatrizA+i);
```

```
Matriz2[i] = *(MatrizB+i);
}

    /* Create buffer object */
Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
Bmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);

    /* Copy input data to memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Amobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), Matriz1, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, Bmobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), Matriz2, 0, NULL, NULL);

    /* Create kernel from source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

if (Dim[0] >= Dim[1]){
/* Create task parallel OpenCL kernel */
kernel[0] = clCreateKernel(program, "RESTA1", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), (void *)&Amobj);
ret = clSetKernelArg(kernel[0], 1, sizeof(cl_mem), (void *)&Bmobj);
ret = clSetKernelArg(kernel[0], 2, sizeof(cl_mem), (void *)&Cmobj);
ret = clSetKernelArg(kernel[0], 3, sizeof(int), &Dim[1]);
```

```
size_t global_item_size = Dim[0];

    /* Execute OpenCL kernel as data parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel[0], 1, NULL,
&global_item_size,NULL, 0, NULL, NULL);
}

else{
/* Create task parallel OpenCL kernel */
kernel[1] = clCreateKernel(program, "RESTA2", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel[1], 0, sizeof(cl_mem), (void *)&Amobj);
ret = clSetKernelArg(kernel[1], 1, sizeof(cl_mem), (void *)&Bmobj);
ret = clSetKernelArg(kernel[1], 2, sizeof(cl_mem), (void *)&Cmobj);
ret = clSetKernelArg(kernel[1], 3, sizeof(int), &Dim[0]);
ret = clSetKernelArg(kernel[1], 4, sizeof(int), &Dim[1]);
size_t global_item_size = Dim[1];

    /* Execute OpenCL kernel as data parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel[1], 1, NULL,
&global_item_size,NULL, 0, NULL, NULL);
}

    /* Copy result to host */
ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE,
0,Dim[0]*Dim[1]*sizeof(double), C, 0, NULL, NULL);

    /* Finalization */
ret = clReleaseKernel(kernel[0]);
ret = clReleaseKernel(kernel[1]);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Amobj);
```

```
ret = clReleaseMemObject(Bmobj);
ret = clReleaseMemObject(Cmobj);

free(source_str);
free(Matriz1);
free(Matriz2);

    }


void ParMatMult(double *MatrizA,double *MatrizB, int *DimA,int *DimB,double
*C, int *DimC)
{

cl_mem Amobj = NULL;
cl_mem Bmobj = NULL;
cl_mem Cmobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;

int i, j, szA, szB, szC;
double *Matriz1,*Matriz2;

szA=DimA[0]*DimA[1];
szB=DimB[0]*DimB[1];
szC=DimA[0]*DimB[1];

Matriz1 = (double *)malloc(szA*sizeof(double));
Matriz2 = (double *)malloc(szB*sizeof(double));

FILE *fp;
const char fileName[] = "./MULTIPLICACION.cl";
size_t source_size;
```

```
char *source_str;

    /* Load kernel source file */
fp = fopen(fileName, "r");
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}

source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

    /* Initialize input data */
for (i=0; i < DimA[2]; i++)
Matriz1[i] = *(MatrizA+i);
for (i=0; i < DimB[2]; i++)
Matriz2[i] = *(MatrizB+i);

    /* Create Buffer Object */
Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
szA*sizeof(double), NULL, &ret);
Bmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
szB*sizeof(double), NULL, &ret);
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
szC*sizeof(double), NULL, &ret);

    /* Copy input data to the memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Amobj, CL_TRUE, 0,
szA*sizeof(double), Matriz1, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, Bmobj, CL_TRUE, 0,
szB*sizeof(double), Matriz2, 0, NULL, NULL);
```

```
    /* Create kernel program from source file*/
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    /* Create data parallel OpenCL kernel */
kernel = clCreateKernel(program, "MULTIPLICACION", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&Amobj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&Bmobj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&Cmobj);
ret = clSetKernelArg(kernel, 3, sizeof(int), &DimB[1]);
ret = clSetKernelArg(kernel, 4, sizeof(int), &DimA[0]);
ret = clSetKernelArg(kernel, 5, sizeof(int), &DimA[1]);
ret = clSetKernelArg(kernel, 6, sizeof(double)*DimA[1], NULL);

size_t global_item_size;
global_item_size = DimA[0];

    /* Execute OpenCL kernel as data parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size,NULL,0, NULL, NULL);

    /* Transfer result to host */
ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE, 0,
szC*sizeof(double), C, 0, NULL, NULL);

DimC[0]=DimA[0];
DimC[1]=DimB[1];
DimC[2]=szC;
```

```
    /* Finalization */
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Amobj);
ret = clReleaseMemObject(Bmobj);
ret = clReleaseMemObject(Cmobj);

free(source_str);
free(Matriz1);
free(Matriz2);
}


void ParInv(double *Datos, int *Dim, double *Datos2)
{

cl_mem Cmobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;

int i,j,ban=0, Aux=0;
double* C;

C = (double*)malloc(2*Dim[0]*Dim[1]*sizeof(double));

FILE *fp;
const char fileName[] = "./INVERSA.cl";
size_t source_size;
char *source_str;

    /* Load kernel source file*/
fp = fopen(fileName, "rb");
if (!fp) {
```

```
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

    /* Initialize input data */
for (i=0;i < 2*Dim[2]; i++)
C[i]=0;

for (i=0;i<Dim[0];i++)
C[Dim[0]+2*i*Dim[0]+i]=1;

for (i=0; i < Dim[2]; i++){
C[i+Aux] = *(Datos+i);
ban++;
if (ban == Dim[0]){
Aux = Aux + Dim[0];
ban=0;
}
}
Aux=0;ban=0;

    /* Create buffer object */
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
2*Dim[0]*Dim[1]*sizeof(double), NULL, &ret);

    /* Copy input data to memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Cmobj, CL_TRUE, 0,
2*Dim[0]*Dim[1]*sizeof(double), C, 0, NULL, NULL);
```

```
    /* Create kernel from source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    /* Create task parallel OpenCL kernel */
kernel = clCreateKernel(program, "INVERSA", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),(void *)&Cmobj);
ret = clSetKernelArg(kernel, 1, sizeof(int), &Dim[0]);
ret = clSetKernelArg(kernel, 2, sizeof(double)*6000, NULL);

size_t global_item_size = Dim[0];

    /* Execute OpenCL kernel as data Parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size,NULL, 0, NULL, NULL);

    /* Copy result to host */
ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE,
0,2*Dim[0]*Dim[1]*sizeof(double), C, 0, NULL, NULL);

    /* Display result*/
for (i=0; i <Dim[2]; i++){

Datos2[i] = C[Dim[0]+i+Aux];
ban++;
if (ban == Dim[0]){
Aux = Aux + Dim[0];
ban=0;
}
```

```
}

    /* Finalization */
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Cmobj);

free(C);
free(source_str);
}


void ParDet(double *Datos, int *Dim, double *Determinante)
{

cl_mem Cmobj = NULL;
cl_mem Dmobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;

int i,j;
double *C, *Det;

C = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));
Det = (double*)malloc(1*sizeof(double));

FILE *fp;
const char fileName[] = "./DET.cl";
size_t source_size;
char *source_str;

    /* Load kernel source file*/
fp = fopen(fileName, "rb");
```

```
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

    /* Initialize input data */
for (i=0;i < Dim[2]; i++)
C[i]=Datos[i];

    /* Create buffer object */
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
Dmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
1*sizeof(double), NULL, &ret);

    /* Copy input data to memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Cmobj, CL_TRUE, 0,
Dim[0]*Dim[1]*sizeof(double), C, 0, NULL, NULL);

    /* Create kernel from source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    /* Create task parallel OpenCL kernel */
kernel = clCreateKernel(program, "DETERMINANTE", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),(void *)&Cmobj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),(void *)&Dmobj);
```

```
ret = clSetKernelArg(kernel, 2, sizeof(int), &Dim[0]);
ret = clSetKernelArg(kernel, 3, sizeof(double)*6000, NULL);

size_t global_item_size = Dim[0];

    /* Execute OpenCL kernel as data Parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size,NULL, 0, NULL, NULL);

    /* Copy result to host */
ret = clEnqueueReadBuffer(command_queue, Dmobj, CL_TRUE, 0,
1*sizeof(double), Det, 0, NULL, NULL);

    /* Display result*/
Determinante[0]=Det[0];

    /* Finalization */
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Cmobj);
ret = clReleaseMemObject(Dmobj);

free(C);
free(Det);
free(source_str);
}


void ParCov(double *Datos, int *Dim, double *Datos2, int *Dim2)
{

if (Dim[0]==1){
Datos2[0]=0;
```

```
Dim2[0]=1;
Dim2[1]=1;
Dim2[2]=1;
}

else{

int i,j,Dim3[3],Dim4[3],Dim5[3];
double *Datos1,*Datos3,*Datos4;
float tiempo;
clock_t t1,t2;

Datos1 = (double*)malloc(Dim[2]*sizeof(double));
Datos3 = (double*)malloc(Dim[1]*sizeof(double));
Datos4 = (double*)malloc(Dim[2]*sizeof(double));

for(i=0;i<Dim[2];i++)
Datos1[i]=Datos[i];

ParSum(Datos1,Dim,Datos3,Dim3);

for (j=0;j<Dim[1];j++){
for (i=0;i<Dim[0];i++)
Datos1[i*Dim[1]+j]=Datos1[i*Dim[1]+j]-(Datos3[j]/Dim[0]);
}

Transpose(Datos1,Dim,Datos4,Dim4);

ParMatMult(Datos4,Datos1,Dim4,Dim,Datos2,Dim5);

for(i=0;i<Dim5[2];i++)
Datos2[i]=Datos2[i]/(Dim[0]-1);
```

```
Dim2[0]=Dim4[0];
Dim2[1]=Dim[1];
Dim2[2]=Dim5[2];

free(Datos1);
free(Datos3);
free(Datos4);
}
}


void ParQR(double *Datos, double *q, double *r, int *Dim)
{

cl_mem Cmobj = NULL;
cl_mem Qmobj = NULL;
cl_program program = NULL;
cl_kernel kernel = NULL;

int i,j;
double *C;

C = (double*)malloc(Dim[0]*Dim[1]*sizeof(double));

FILE *fp;
const char fileName[] = "./QR.cl";
size_t source_size;
char *source_str;

    /* Load kernel source file*/
fp = fopen(fileName, "rb");
if (!fp) {
fprintf(stderr, "Failed to load kernel.\n");
```

```
exit(1);
}
source_str = (char *)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

    /* Initialize input data */
for (i=0;i < Dim[2]; i++)
C[i]=Datos[i];

    /* Create buffer object */
Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);
Qmobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
Dim[0]*Dim[1]*sizeof(double), NULL, &ret);

    /* Copy input data to memory buffer */
ret = clEnqueueWriteBuffer(command_queue, Cmobj, CL_TRUE,0,
Dim[0]*Dim[1]*sizeof(double), C, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, Qmobj, CL_TRUE,0,
Dim[0]*Dim[1]*sizeof(double), q, 0, NULL, NULL);

    /* Create kernel from source */
program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    /* Create task parallel OpenCL kernel */
kernel = clCreateKernel(program, "QR", &ret);

    /* Set OpenCL kernel arguments */
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),(void *)&Cmobj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem),(void *)&Qmobj);
```

```
ret = clSetKernelArg(kernel, 2, sizeof(int), &Dim[1]);
ret = clSetKernelArg(kernel, 3, sizeof(double)*6000, NULL);

size_t global_item_size = Dim[0];

    /* Execute OpenCL kernel as data Parallel */
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size,NULL, 0, NULL, NULL);

    /* Copy result to host */
ret = clEnqueueReadBuffer(command_queue, Qmobj, CL_TRUE,0,
Dim[0]*Dim[1]*sizeof(double), q, 0, NULL, NULL);

Transpose(q,Dim,q,Dim);
ParMatMult(q,Datos,Dim,Dim,r,Dim);
Transpose(q,Dim,q,Dim);

    /* Finalization */
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(Cmobj);
ret = clReleaseMemObject(Qmobj);

free(C);
free(source_str);
}
```

Anexo C

Kernels para operaciones matriciales con GPU

C.1 Kernel Traspuesta (TRASPUESTA.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void TRASPUESTA1(__global double* A,__global double* C,
const int Filas,const int Columnas)
{
int i,j;

for (i=0;i<Filas;i++){
for (j=0;j<Columnas-i;j++)
C[Filas*j+i*(Filas+1)]=A[Columnas*i+j+i];
}
}

__kernel void TRASPUESTA2(__global double* A, __global double* C,
Filas,const int Columnas)
```

```
{

int i,j;

for (i=0;i<Columnas;i++){
for (j=0;j<Filas-i;j++)
C[Filas*i+j+i]=A[Columnas*j+i*(Columnas+1)];
}
}
```

C.2 Kernel Suma de matriz a vector (SUMAMATRIZ.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void SUMAMATRIZ(__global double* Matriz,__global double* C,
const int Filas,const int Columnas)
{
int base = get_global_id(0);
int i;
C[base]=0;

for (i=0;i<Filas;i++)
C[base] += Matriz[base+i*Columnas];
}
```

C.3 Kernel Suma de matrices (SUMA.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void SUMA1(__global double* A, __global double* B,
__global double* C,const int Columnas)
```

```
{
int base = get_global_id(0);
int iloc = get_local_id(0);
int nloc = get_local_size(0);
int i,k;

for (i=0;i<Columnas;i++)
C[Columnas*base+i] = A[Columnas*base+i] + B[Columnas*base+i];

    }

__kernel void SUMA2(__global double* A, __global double* B,
__global double* C,const int Filas,const int Columnas)
{
int base = get_global_id(0);
int i;

for (i=0;i<Filas;i++)
C[base+i*Columnas] = A[base+i*Columnas] + B[base+i*Columnas];
}
```

C.4 Kernel Resta de matrices (RESTA.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void RESTA1(__global double* A, __global double* B,
__global double* C,const int Columnas)
{
int base = get_global_id(0);
int iloc = get_local_id(0);
int nloc = get_local_size(0);
int i,k;
```

```
for (i=0;i<Columnas;i++)
C[Columnas*base+i] = A[Columnas*base+i] - B[Columnas*base+i];

    }

__kernel void RESTA2(__global double* A, __global double* B,
_ _global double* C,const int Filas,const int Columnas)
{
int base = get_global_id(0);
int i;

for (i=0;i<Filas;i++)
C[base+i*Columnas] = A[base+i*Columnas] - B[base+i*Columnas];
}
```

C.5 Kernel Multiplicación de matrices (MULTIPLICACION.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void MULTIPLICACION(__global double* A,__global double* B,
__global double* C,const int Mdim,const int Ndim,const int Pdim,__local double* Bwrk)
{
int i = get_global_id(0);
int iloc = get_local_id(0);
int nloc = get_local_size(0);

int k,j;
double Awrk[24000];
double value;
```

```
for (k=0;k < Pdim; k++) Awrk[k] = A[i*Pdim+k];

for (j = 0; j < Mdim; j++)
{
for (k=iloc;k < Pdim;k=k+nloc) Bwrk[k] = B[k*Mdim+j];
barrier(CLK_LOCAL_MEM_FENCE);
value=0.0;
for (k=0;k < Pdim; k++) value += Awrk[k] * Bwrk[k];
barrier(CLK_GLOBAL_MEM_FENCE);
C[i*Mdim+j] = value;
}
}
```

C.6 Kernel Inversa de una matriz (INVERSA.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void INVERSA(__global double* C,const int T,__local double* Aux2)
{
int base = get_global_id(0);
int iloc = get_local_id(0);
int nloc = get_local_size(0);
int G = get_global_size(0);

int a=0,i,j=0,k,ban=0,l;
double Aux[1000],X,Y,b,tmp;

for (i=0;i<T;i++){

for (k=iloc;k<G;k=k+nloc)
Aux2[k]=C[k + 1 + i + (2*i*G)];
barrier(CLK_LOCAL_MEM_FENCE);
```

```
X=C[i + (2*i*G)];
a=0;ban=0;

while(X==0 && j<T+1){
l=0;
if(ban==0){
j=i+1;
for (k=0;k<2;k++){
Aux[l]=C[base + k*G + (2*i*G)];
l++;
}
ban=1;l=0;
}

for (k=iloc;k<2*G;k=k+nloc)
C[k + (2*i*G)]= C[k + (2*j*G)];
for (k=0;k<2;k++){
C[base + k*G + (2*j*G)]= Aux[l];
l++;
}

X=C[i + (2*i*G)];
if(X==0){
l=0;
for (k=iloc;k<2*G;k=k+nloc)
C[k + (2*j*G)]=C[k + (2*i*G)];
for (k=0;k<2;k++){
C[base + k*G + (2*i*G)]=Aux[l];
l++;
}
}
else{
for (k=iloc;k<G;k=k+nloc)
```

```
Aux2[k]= C[k + 1 + i + (2*i*G)];
barrier(CLK_LOCAL_MEM_FENCE);
C[j + G + (2*i*G)]=C[j + G + (2*i*G)]/X;
b = C[j + G + (2*i*G)];
a=j;
break;
}

j++;
}

l=0;
for (k=iloc;k<G;k=k+nloc){
Aux2[k]=Aux2[k]/X;
Aux[l]=Aux2[k];
C[k + 1 + i + (2*i*G)]=Aux2[k];
l++;
}

for (j=0;j<T;j++){
for (k=iloc;k<G;k=k+nloc)
Aux2[k]=C[k + 1 + i + (2*j*G)];
barrier(CLK_LOCAL_MEM_FENCE);
if(j!=i){
Y=C[i +(2*j*G)];
if(Y!=0){
l=0;
for (k=iloc;k<G;k=k+nloc){
Aux2[k]= Aux2[k]-Y*Aux[l];
C[k + 1 + i + (2*j*G)]=Aux2[k];
l++;
}
}
```

```
}
}

if (a!=0){
C[a+G+(2*base*G)] = C[a+G+(2*base*G)] - C[i+(2*base*G)]*C[a+G+(2*i*G)];
C[a+G+(2*i*G)] = b;
}
}
}
```

C.7 Kernel Determinante de una matriz (DET.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void DETERMINANTE(__global double* C,__global double* Det,const
int T,__local double* Aux2)
{
int base = get_global_id(0);
int iloc = get_local_id(0);
int nloc = get_local_size(0);
int G = get_global_size(0);

int i,j=0,k,ban=0,l;
double Aux[1000],X,Y;

if(T==1)
Det[0]=C[0];

else{
Det[0]=1;
for (i=0;i<T;i++){
```

```
for (k=iloc;k<T;k=k+nloc)
Aux2[k]=C[k + (i*T)];
barrier(CLK_LOCAL_MEM_FENCE);

X=C[i + (i*T)];

ban=0;

while(X==0 && j<T+1){
if(ban==0){
j=i+1;
Aux[0]=C[base + (i*T)];
ban=1;
}

for (k=iloc;k<T;k=k+nloc)
C[k + (i*T)]= C[k + (j*T)];

C[base + (j*T)]= Aux[0];

X=C[i + (i*T)];
if(X==0){
for (k=iloc;k<T;k=k+nloc)
C[k + (j*T)]=C[k + (i*T)];

C[base + (i*T)]=Aux[0];
}
else{
Det[0]= (-1*Det[0]);
for (k=iloc;k<T;k=k+nloc)
Aux2[k]= C[k + (i*T)];
barrier(CLK_LOCAL_MEM_FENCE);
break;
```

```
}

j++;
}

Det[0]=Det[0]*X;

l=0;
for (k=iloc;k<T;k=k+nloc){
Aux2[k]=Aux2[k]/X;
Aux[l]=Aux2[k];
C[k + (i*T)]=Aux2[k];
l++;
}

for (j=i+1;j<T;j++){
for (k=iloc;k<T;k=k+nloc)
Aux2[k]=C[k + (j*T)];
barrier(CLK_LOCAL_MEM_FENCE);
Y=C[i +(j*T)];
if(Y!=0){
l=0;
for (k=iloc;k<T;k=k+nloc){
Aux2[k]= Aux2[k]-Y*Aux[l];
C[k + (j*T)]=Aux2[k];
l++;
}
}
}
}
}
}
```

C.8 Kernel Descompisición QR de una matriz (QR.cl)

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void QR(__global double* C, __global double* Q,const int T,__local
double* qcol)
{

int base = get_global_id(0);
int iloc = get_local_id(0);
int nloc = get_local_size(0);
int G = get_global_size(0);

int i,j,k;
double Aux,Aux2[6000],Aux3[6000],dcol[6000],proj[6000];

for(k=0;k<T;k++) dcol[k]= C[k + base*T];

for(i=0;i<T;i++) {

proj[base] = 0.0;

for(j=0;j<i;j++) {

Aux = 0.0;

qcol[base] = dcol[i] * Aux3[j];
barrier(CLK_LOCAL_MEM_FENCE);

for(k=0;k<G;k++)
Aux += qcol[k];

for(k=0;k<G;k++)
proj[k] = proj[k] + (Aux * Aux3[j]);
```

```
barrier(CLK_GLOBAL_MEM_FENCE);

}
Aux = 0.0;

qcol[base]= dcol[i] - proj[base];

Aux2[base] = qcol[base];

for (k=iloc;k<G;k=k+nloc)
qcol[k] = qcol[k] * qcol[k];
barrier(CLK_LOCAL_MEM_FENCE);

for(k=0;k<G;k++)
Aux += qcol[k];

Aux = sqrt(Aux);

Q[i + base*T]= Aux2[base]/Aux;

for (k=iloc;k<G;k=k+nloc)
qcol[k] = Q[i + k*T];
barrier(CLK_LOCAL_MEM_FENCE);

Aux3[i] = qcol[base];
}
}
```

Anexo D

Algoritmos de clasificación implementados en OpenCL

D.1 Librería CPUFUNCTIONS.h

En esta sección se presentan las funciones secuenciales que se utilizaron en los algoritmos de clasificación y se guardaron en una librería de nombre CPUFUNCTIONS.h.

```
void CPUSum(double *Datos, int *Dim, double *Datos2, int *Dim2)
{

int i,j;
double sum=0.0;

for(j=0;j<Dim[1];j++){
sum=0.0;
for(i=0;i<Dim[0];i++)
sum+= Datos[i*Dim[1]+j];
Datos2[j]=sum;
```

```
}
Dim2[0]=1;
Dim2[1]=Dim[1];
Dim2[2]=Dim[1];
}

void CPUMatMult(double *MatrizA,double *MatrizB, int *DimA,int *DimB,double
*Datos, int *DimC)
{

int i, j, k;
double tmp;

for (i=0; i<DimA[0]; i++){
for (j=0; j<DimB[1]; j++){
tmp = 0.0;
for(k=0;k<DimA[1];k++)
tmp += MatrizA[i*DimA[1]+k] * MatrizB[k*DimB[1]+j];
Datos[i*DimA[0]+j] = tmp;
}
}

DimC[0]=DimA[0];
DimC[1]=DimB[1];
DimC[2]=DimA[0]*DimB[1];
}

void CPUqr(double *Datos, double *q, double *r, int *Dim)
{

int i,j,k,z,Dimcol[3],Dimcol2[3],DimAux[3],Dim2[3];
double *proj,*dcol,*qcol,*VecAux,Aux;
proj = (double*)malloc(Dim[0]*sizeof(double));
```

```
dcol = (double*)malloc(Dim[0]*sizeof(double));
qcol = (double*)malloc(Dim[0]*sizeof(double));
VecAux = (double*)malloc(Dim[0]*sizeof(double));

Dimcol2[0] = 1;Dimcol2[1] = Dim[0];Dimcol2[2] = Dim[0];
Dimcol[0] = Dim[0];Dimcol[1] = 1;Dimcol[2] = Dim[0];

for(i=0;i<Dim[1];i++){
for(z=0;z<Dim[0];z++)
proj[z] = 0;

for(k=0;k<Dim[0];k++)
dcol[k]= Datos[i + k*Dim[1]];

for(j=0;j<i;j++) {
for(k=0;k<Dim[0];k++)
qcol[k]= q[j + k*Dim[1]];

CPUMatMult(dcol,qcol,Dimcol2,Dimcol,&Aux,DimAux);
CPUMatMult(&Aux,qcol,DimAux,Dimcol2,VecAux,Dimcol2);
CPUMatSum(proj,VecAux,Dimcol,proj);

}

CPUMatRes(dcol,proj,Dimcol,qcol);
CPUMatMult(qcol,qcol,Dimcol2,Dimcol,&Aux,DimAux);
Aux = pow(Aux,0.5);

for(k=0;k<Dim[0];k++) {

qcol[k] = qcol[k]/Aux;
q[i + k*Dim[1]] = qcol[k];
```

```
}
}

Transpose(q,Dim,q,Dim);
ParMatMult(q,Datos,Dim,Dim,r,Dim);
Transpose(q,Dim,q,Dim);

free(proj);
free(dcol);
free(qcol);
free(VecAux);
}
```

D.2 Algoritmo vecinos cercanos

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "GPULIBRARY.h"
#include "CPUFUNCTIONS.h"
#ifdef __APPLE__
#include <CL/opencl.h>
#else
#include <CL/cl.h>
#endif

int main(){

double temp,Exactitud,Aciertos=0.0;
i=0,j=0,ban1=0,ban2=0,k=5,Dim1[3],Dim2[3],NumClass=0,
```

```
Class[10],ContClass[10],Mayor,Mayores,l;
float tiempo=0;

FILE *archivo;
clock_t t1,t2;
GPUOpen();
t1=clock();

    /* EXTRAYENDO DATOS DE TRAIN */
archivo = fopen("./MATEXPQDA/TR15000X100.txt","r");
Dimensions(archivo,Dim1);
fclose(archivo);

double *Datos1;
Datos1 = (double*)malloc(Dim1[2]*sizeof(double));
archivo = fopen("./MATEXPQDA/TR15000X100.txt","r");
Data(archivo,Dim1,Datos1);
fclose(archivo);

    /* EXTRAYENDO DATOS DE TEST */
archivo = fopen("./MATEXPQDA/TS15000X100.txt","r");
Dimensions(archivo,Dim2);
fclose(archivo);

double *Datos2;
Datos2 = (double*)malloc(Dim2[2]*sizeof(double));
archivo = fopen("./MATEXPQDA/TS15000X100.txt","r");
Data(archivo,Dim2,Datos2);
fclose(archivo);

    /* DECLARANDO VARIABLES PARA DATOS Y ETIQUETAS
DE TRAIN Y DE TEST */
double *Train,*ClassTr,*Test,*ClassTs,*ClassTrAux;
```

```
Train = (double*)malloc((Dim1[2]-Dim1[0])*sizeof(double));
ClassTr = (double*)malloc(Dim1[0]*sizeof(double));
ClassTrAux = (double*)malloc(Dim1[0]*sizeof(double));
Test = (double*)malloc((Dim2[2]-Dim2[0])*sizeof(double));
ClassTs = (double*)malloc(Dim2[0]*sizeof(double));

    /* OBTENIENDO ETIQUETAS Y DATOS DE TRAIN */
for(i=0;i<Dim1[2];i++) {
if (i==Dim1[1]-1+j) {
j=j+Dim1[1];
ClassTr[ban1]=Datos1[i];
ban1++;
}
else{
Train[ban2]=Datos1[i];
ban2++;
}
}

    /* OBTENIENDO NUMERO DE CLASES */
for (i=0;i<Dim1[0];i++)
if(ClassTr[i]>NumClass)
NumClass=ClassTr[i];

    /* OBTENIENDO ETIQUETAS Y DATOS DE TEST */
j=0;ban1=0;ban2=0;
for(i=0;i<Dim2[2];i++) {
if (i==Dim2[1]-1+j){
j=j+Dim2[1];
ClassTs[ban1]=Datos2[i];
ban1++;
}
else{
```

```
Test[ban2]=Datos2[i];
ban2++;
}
}

int Dim3[3],Dim4[3],DimAux[3],DimVec[3],DimVecT[3];
Dim3[0]=Dim1[0];Dim3[1]=Dim1[1]-1;Dim3[2]=Dim1[2]-Dim1[0];
Dim4[0]=Dim2[0];Dim4[1]=Dim2[1]-1;Dim4[2]=Dim2[2]-Dim2[0];
DimVec[0]=1;DimVec[1]=Dim3[1];DimVec[2]=Dim3[1];
DimVecT[0]=Dim3[1];DimVecT[1]=1;DimVecT[2]=Dim3[1];

double *Distancias,*Prediccion,*Tst,*Obs,*ResAux,Aux;
Distancias = (double*)malloc(Dim3[0]*sizeof(double));
Prediccion = (double*)malloc(Dim2[0]*sizeof(double));
Tst = (double*)malloc(DimVec[2]*sizeof(double));
Obs = (double*)malloc(DimVec[2]*sizeof(double));
ResAux = (double*)malloc(DimVec[2]*sizeof(double));

for(l=0;l<Dim4[0];l++) {
for(i=0;i<Dim1[0];i++)
ClassTrAux[i]=ClassTr[i];

for(i=0;i<Dim4[1];i++)
Tst[i]=Test[l*Dim3[1]+i];

    /* CALCULO DE LA DISTANCIA EUCLIDEA */
for(j=0;j<Dim3[0];j++){
Distancias[j]=0.0;

for(i=0;i<Dim3[1];i++)
Obs[i]=Train[j*Dim3[1]+i];
```

```
ParMatRes(Obs,Tst,DimVec,ResAux);
ParMatMult(ResAux,ResAux,DimVec,DimVecT,&Aux,DimAux);
Distancias[j]=Aux;
Distancias[j]=pow(Distancias[j],0.5);
}

    /* ACOMODANDO DISTANCIAS DE FORMA ASCENDENTE */
for (i=1; i<Dim3[0]; i++){
for (j=0 ; j<Dim3[0] - 1; j++)
if (Distancias[j] > Distancias[j+1]){
temp = Distancias[j];
Distancias[j] = Distancias[j+1];
Distancias[j+1] = temp;
temp = ClassTrAux[j];
ClassTrAux[j] = ClassTrAux[j+1];
ClassTrAux[j+1] = temp;
}
}

    /* INICIALIZANDO CLASES Y CONTADORES DE CLASES*/
for(i=0;i<10;i++){
Class[i]=i+1;
ContClass[i]=0;
}

    /* OBTENIENDO CONTADORES DE CLASES*/
for (j=0; j<NumClass; j++){
for (i=0; i<k; i++)
if(ClassTrAux[i]==Class[j])
ContClass[j]++;
}
```

```
    /* ACOMODANDO CONTADORES DE FORMA DESCENDENTE */
for (i=1; i<NumClass; i++){
for (j=0; j<NumClass - 1; j++)
if (ContClass[j] < ContClass[j+1]){
temp = ContClass[j];
ContClass[j] = ContClass[j+1];
ContClass[j+1] = temp;
temp = Class[j];
Class[j] = Class[j+1];
Class[j+1] = temp;
}
}

    /* OBTENIENDO CLASE QUE OBTUVO MAYOR CONTEO */
Mayores=0;
for(i=1;i<NumClass;i++)
if(ContClass[0]==ContClass[i])
Mayores++;

if(Mayores==0)
Mayor=Class[0];
else{
Mayor=rand() % (Mayores+1);
Mayor=Class[Mayor];
}

    /* ESCRIBIENDO PREDICCION */
Prediccion[l]=Mayor;
}

for(i=0;i<Dim4[0];i++)
if(Prediccion[i]==ClassTs[i])
```

```
Aciertos++;

Exactitud=(Aciertos*100)/Dim4[0];

//for(i=0;i<Dim2[0];i++)
printf("la exactitud de los datos es de %.4f \n",Exactitud);

for (i=0; i<Dim4[0]; i++)
printf("%.2f\n",Prediccion[i]);
GPUClose();

t2=clock();
tiempo=(t2-t1)/(float)CLOCKS_PER_SEC;
printf("el tiempo transcurrido del programa fue de %.4f\n\n",tiempo);

free(Datos1);
free(Datos2);
free(Train);
free(Test);
free(ClassTr);
free(ClassTrAux);
free(ClassTs);
free(Distancias);
free(Prediccion);
free(Obs);
free(Tst);
free(ResAux);
}
```

D.3 Algoritmo análisis discriminante lineal

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "GPULIBRARY.h"
#include "CPUFUNCTIONS.h"
#ifdef __APPLE__
#include <CL/opencl.h>
#else
#include <CL/cl.h>
#endif

int main(){

    double Exactitud,Aciertos=0.0,PClass,Res1,Res2,Mayor;
int i=0,j=0,k,l,ban1=0,ban2=0,ObsClass,NumObs,Dim1[3],Dim2[3],
NumClass=0,Class[10],tmp;
float tiempo=0;

FILE *archivo;
clock_t t1,t2;
GPUOpen();
t1=clock();

    /* EXTRAYENDO DATOS DE TRAIN */
archivo = fopen("./MATEXPLDA/TR1000X500.txt","r");
Dimensions(archivo,Dim1);
fclose(archivo);
```

```
double *Datos1;
Datos1 = (double*)malloc(Dim1[2]*sizeof(double));
archivo = fopen("./MATEXPLDA/TR1000X500.txt","r");
Data(archivo,Dim1,Datos1);
fclose(archivo);

    /* EXTRAYENDO DATOS DE TEST */
archivo = fopen("./MATEXPLDA/TS1000X500.txt","r");
Dimensions(archivo,Dim2);
fclose(archivo);

double *Datos2;
Datos2 = (double*)malloc(Dim2[2]*sizeof(double));
archivo = fopen("./MATEXPLDA/TS1000X500.txt","r");
Data(archivo,Dim2,Datos2);
fclose(archivo);

    /* DECLARANDO VARIABLES PARA DATOS Y ETIQUETAS
DE TRAIN Y DE TEST */
double *Train,*ClassTr,*Test,*ClassTs;
Train = (double*)malloc((Dim1[2]-Dim1[0])*sizeof(double));
ClassTr = (double*)malloc(Dim1[0]*sizeof(double));
Test = (double*)malloc((Dim2[2]-Dim2[0])*sizeof(double));
ClassTs = (double*)malloc(Dim2[0]*sizeof(double));

    /* OBTENIENDO ETIQUETAS Y DATOS DE TRAIN */
for(i=0;i<Dim1[2];i++) {
if (i==Dim1[1]-1+j){
j=j+Dim1[1];
ClassTr[ban1]=Datos1[i];
ban1++;
}
else{
```

```
Train[ban2]=Datos1[i];
ban2++;
}
}

    /* OBTENIENDO NUMERO DE CLASES */
for (i=0;i<Dim1[0];i++)
if(ClassTr[i]>NumClass)
NumClass=ClassTr[i];

    /* OBTENIENDO ETIQUETAS Y DATOS DE TEST */
j=0;ban1=0;ban2=0;
for(i=0;i<Dim2[2];i++) {
if (i==Dim2[1]-1+j){
j=j+Dim2[1];
ClassTs[ban1]=Datos2[i];
ban1++;
}
else{
Test[ban2]=Datos2[i];
ban2++;
}
}

    /* DIMENSIONES DE TRAIN Y TEST SIN ETIQUETAS*/
int Dim3[3],Dim4[3],Dim5[3],DimTRClass[3],DimMean[3],DimAux[3],*Ind;
Dim3[0]=Dim1[0];Dim3[1]=Dim1[1]-1;Dim3[2]=Dim1[2]-Dim1[0];
Dim4[0]=Dim2[0];Dim4[1]=Dim2[1]-1;Dim4[2]=Dim2[2]-Dim2[0];

    /* DECLARACION DE VARIABLES */
double *Prediccion,*MediaTr,*Discriminante,*CovTotal,*InvCovClass,
TRClass,*Obs,*ResAux;
Discriminante = (double*)malloc(NumClass*sizeof(double));
```

```
Prediccion = (double*)malloc(Dim4[0]*sizeof(double));
MediaTr = (double*)malloc(Dim3[1]*sizeof(double));
Obs = (double*)malloc(Dim4[1]*sizeof(double));
CovTotal = (double*)malloc(Dim3[1]*Dim3[1]*sizeof(double));
InvCovClass = (double*)malloc(Dim3[1]*Dim3[1]*sizeof(double));
Ind = (int*)malloc(Dim3[0]*sizeof(int));

    /* OBTENIENDO LA COVARIANZA TOTAL*/
ParCov(Train,Dim3,CovTotal,Dim5);

    /* OBTENIENDO LA INVERSA DE LA COVARIANZA*/
ParInv(CovTotal,Dim5,InvCovClass);

    /* INICIALIZANDO CLASES */
for(i=0;i<10;i++)
Class[i]=i+1;

    /* NUMERO DE OBESRVACIONES A PREDECIR */
for(NumObs=0;NumObs<Dim4[0];NumObs++) {

    /* LOCALIZANDO NUMERO DE OBSERVACION */
for(i=0;i<Dim3[1];i++)
Obs[i]=Test[NumObs*Dim3[1]+i];

    /* OBTENIENDO EL VALOR DEL DISCRIMINANTE
DE LA CLASE */
for(j=0;j<NumClass;j++) {
ObsClass=0;

    /* OBTENIENDO LOS INDICES Y NUMERO DE
OBSERVACIONES DE CLASE */
for(i=0;i<Dim3[0];i++)
if (ClassTr[i]==Class[j])
```

```
{
Ind[ObsClass]=i;
ObsClass++;
}

    /* OBTENIENDO LAS OBSERVACIONES DE CLASE */
DimTRClass[0]=ObsClass;
DimTRClass[1]=Dim3[1];
DimTRClass[2]=DimTRClass[0]*DimTRClass[1];
TRClass = (double*)malloc(DimTRClass[2]*sizeof(double));
for (k=0;k<ObsClass;k++)
for (l=0;l<Dim3[1];l++)
TRClass[k*Dim3[1] + l]=Train[Ind[k]*Dim3[1] + l];

    /* OBTENIENDO LA PROBABILIDAD DE CLASE */
PClass=(ObsClass*1.0)/Dim3[0];

    /* OBTENIENDO MEDIA DE LA CLASE */
CPUSum(TRClass,DimTRClass,MediaTr,DimMean);
for(i=0;i<DimMean[2];i++)
MediaTr[i]=MediaTr[i]/DimTRClass[0];

    /* CALCULANDO EL DISCRIMINANTE DE LA CLASE */
ResAux = (double*)malloc(DimMean[0]*Dim5[1]*sizeof(double));

CPUMatMult(MediaTr,InvCovClass,DimMean,Dim5,ResAux,DimAux);

    tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(ResAux,Obs,DimAux,DimMean,&Res1,DimAux);
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;

    CPUMatMult(MediaTr,InvCovClass,DimMean,Dim5,ResAux,DimAux);
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
```

```
CPUMatMult(ResAux,MediaTr,DimAux,DimMean,&Res2,DimAux);

Discriminante[j]= Res1 - (0.5*Res2) + log(PClass);

}

    /* ESCRIBIENDO PREDICCION */
Mayor=Discriminante[0];
Prediccion[NumObs]=1;
for (i=1;i<NumClass;i++)
if(Discriminante[i]>Mayor){
Prediccion[NumObs]=Class[i];
Mayor=Discriminante[i];
}
}

for(i=0;i<Dim4[0];i++)
printf("%f\n",Prediccion[i]);

    /* CALCULANDO ACIERTOS*/
for(i=0;i<Dim4[0];i++)
if(Prediccion[i]==ClassTs[i])
Aciertos++;

    /* CALCULANDO EXACTITUD*/
Exactitud=(Aciertos*100.0)/Dim4[0];

for(i=0;i<Dim2[0];i++)
printf("la exactitud de los datos es de %.4f\n",Exactitud);
GPUClose();

t2=clock();
tiempo=(t2-t1)/(float)CLOCKS_PER_SEC;
```

```
printf("el tiempo transcurrido del programa fue de %.4f\n\n",tiempo);

free(Datos1);
free(Datos2);
free(Train);
free(Test);
free(ClassTr);
free(ClassTs);
free(Discriminante);
free(Prediccion);
free(MediaTr);
free(CovTotal);
free(InvCovClass);
free(Ind);
free(TRClass);
free(Obs);
free(ResAux);
}
```

D.4 Algoritmo análisis discriminante cuadrático para pocos atributos

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "GPULIBRARY.h"
#include "CPUFUNCTIONS.h"
#ifdef __APPLE__
```

D.4 Algoritmo análisis discriminante cuadrático para pocos atributos

```
#include <CL/opencl.h>
#else
#include <CL/cl.h>
#endif

int main(){

double Exactitud,Aciertos=0.0,PClass,Res1,Res2,Res3,Mayor;
int i=0,j=0,k,l,ban1=0,ban2=0,ObsClass,NumObs;
int Dim1[3],Dim2[3],NumClass=0,Class[10],tmp;
float tiempo=0;

FILE *archivo;
clock_t t1,t2;
GPUOpen();
t1=clock();
/* EXTRAYENDO DATOS DE TRAIN */
archivo = fopen("./MATEXPQDA/TR15000X100.txt","r");
Dimensions(archivo,Dim1);
fclose(archivo);

double *Datos1;
Datos1 = (double*)malloc(Dim1[2]*sizeof(double));
archivo = fopen("./MATEXPQDA/TR15000X100.txt","r");
Data(archivo,Dim1,Datos1);
fclose(archivo);

    /* EXTRAYENDO DATOS DE TEST */
archivo = fopen("./MATEXPQDA/TS15000X100.txt","r");
Dimensions(archivo,Dim2);
fclose(archivo);
```

```
double *Datos2;
Datos2 = (double*)malloc(Dim2[2]*sizeof(double));
archivo = fopen("./MATEXPQDA/TS15000X100.txt","r");
Data(archivo,Dim2,Datos2);
fclose(archivo);

    /* DECLARANDO VARIABLES PARA DATOS Y ETIQUETAS
DE TRAIN Y DE TEST */
double *Train,*ClassTr,*Test,*ClassTs;
Train = (double*)malloc((Dim1[2]-Dim1[0])*sizeof(double));
ClassTr = (double*)malloc(Dim1[0]*sizeof(double));
Test = (double*)malloc((Dim2[2]-Dim2[0])*sizeof(double));
ClassTs = (double*)malloc(Dim2[0]*sizeof(double));

    /* OBTENIENDO ETIQUETAS Y DATOS DE TRAIN */
for(i=0;i<Dim1[2];i++) {
if (i==Dim1[1]-1+j){
j=j+Dim1[1];
ClassTr[ban1]=Datos1[i];
ban1++;
}
else{
Train[ban2]=Datos1[i];
ban2++;
}
}

    /* OBTENIENDO NUMERO DE CLASES */
for (i=0;i<Dim1[0];i++)
if(ClassTr[i]>NumClass)
NumClass=ClassTr[i];
```

```
    /* OBTENIENDO ETIQUETAS Y DATOS DE TEST */
j=0;ban1=0;ban2=0;
for(i=0;i<Dim2[2];i++) {
if (i==Dim2[1]-1+j){
j=j+Dim2[1];
ClassTs[ban1]=Datos2[i];
ban1++;
}
else{
Test[ban2]=Datos2[i];
ban2++;
}
}

    /* DIMENSIONES DE TRAIN Y TEST SIN ETIQUETAS*/
int Dim3[3],Dim4[3],Dim5[3],DimTRClass[3],DimMean[3],DimAux[3],*Ind;
Dim3[0]=Dim1[0];Dim3[1]=Dim1[1]-1;Dim3[2]=Dim1[2]-Dim1[0];
Dim4[0]=Dim2[0];Dim4[1]=Dim2[1]-1;Dim4[2]=Dim2[2]-Dim2[0];

    /* DECLARACION DE VARIABLES */
double *Prediccion,*MediaTr,*Discriminante,*CovClass;
double *InvCovClass,*TRClass,*Obs,*ResAux,DetClass;
Discriminante = (double*)malloc(NumClass*sizeof(double));
Prediccion = (double*)malloc(Dim4[0]*sizeof(double));
MediaTr = (double*)malloc(Dim3[1]*sizeof(double));
Obs = (double*)malloc(Dim4[1]*sizeof(double));
Ind = (int*)malloc(Dim3[0]*sizeof(int));

    /* INICIALIZANDO CLASES */
for(i=0;i<10;i++)
Class[i]=i+1;
```

```
    /* NUMERO DE OBESRVACIONES A PREDECIR */
for(NumObs=0;NumObs<Dim4[0];NumObs++) {

    /* LOCALIZANDO NUMERO DE OBSERVACION */
for(i=0;i<Dim3[1];i++)
Obs[i]=Test[NumObs*Dim3[1]+i];

    /* OBTENIENDO EL VALOR DEL DISCRIMINANTE DE LA CLASE */
for(j=0;j<NumClass;j++) {
ObsClass=0;

    /* OBTENIENDO LOS INDICES Y NUMERO DE OBSERVACIONES DE
CLASE */
for(i=0;i<Dim3[0];i++)
if (ClassTr[i]==Class[j])
{
Ind[ObsClass]=i;
ObsClass++;
}

    /* OBTENIENDO LAS OBSERVACIONES DE CLASE */
DimTRClass[0]=ObsClass;
DimTRClass[1]=Dim3[1];
DimTRClass[2]=DimTRClass[0]*DimTRClass[1];
TRClass = (double*)malloc(DimTRClass[2]*sizeof(double));
for (k=0;k<ObsClass;k++)
for (l=0;l<Dim3[1];l++)
TRClass[k*Dim3[1] + l]=Train[Ind[k]*Dim3[1] + l];

    /* DECLARANDO VARIABLES PARA LA COVARIANZA DE CLASE Y
SU INVERSA */
CovClass = (double*)malloc(DimTRClass[1]*DimTRClass[1]*sizeof(double));
```

```
InvCovClass = (double*)malloc(DimTRClass[1]*DimTRClass[1]*sizeof(double));

    /* OBTENIENDO LA PROBABILIDAD DE CLASE */
PClass=(ObsClass*1.0)/Dim3[0];

    /* OBTENIENDO MEDIA DE LA CLASE */
CPUSum(TRClass,DimTRClass,MediaTr,DimMean);
for(i=0;i<DimMean[2];i++)
MediaTr[i]=MediaTr[i]/DimTRClass[0];

    /* OBTENIENDO LA COVARIANZA DE LA CLASE */
ParCov(TRClass,DimTRClass,CovClass,Dim5);

    /* OBTENIENDO LA INVERSA DE LA COVARIANZA DE LA CLASE */
ParInv(CovClass,Dim5,InvCovClass);

    /* OBTENIENDO LA DETERMINANTE DE LA COVARIANZA DE LA
CLASE */
ParDet(CovClass,Dim5,&DetClass);

    /* CALCULANDO EL DISCRIMINANTE DE LA CLASE */
ResAux = (double*)malloc(DimMean[0]*Dim5[1]*sizeof(double));
CPUMatMult(Obs,InvCovClass,DimMean,Dim5,ResAux,DimAux);
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(ResAux,Obs,DimAux,DimMean,&Res1,DimAux);

tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(MediaTr,InvCovClass,DimMean,Dim5,ResAux,DimAux);
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(ResAux,Obs,DimAux,DimMean,&Res2,DimAux);

tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(MediaTr,InvCovClass,DimMean,Dim5,ResAux,DimAux);
```

```
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(ResAux,MediaTr,DimAux,DimMean,&Res3,DimAux);

Discriminante[j]= (-0.5*Res1) + Res2 - (0.5*Res3) - 0.5*log(DetClass) + log(PClass);

}

    /* ESCRIBIENDO PREDICCION */
Mayor=Discriminante[0];
Prediccion[NumObs]=1;
for (i=1;i<NumClass;i++)
if(Discriminante[i]>Mayor){
Prediccion[NumObs]=Class[i];
Mayor=Discriminante[i];
}
}

for(i=0;i<Dim4[0];i++)
printf("%f\n",Prediccion[i]);

    /* CALCULANDO PREDICCION*/
for(i=0;i<Dim4[0];i++)
if(Prediccion[i]==ClassTs[i])
Aciertos++;

    /* CALCULANDO EXACTITUD*/
Exactitud=(Aciertos*100.0)/Dim4[0];

for(i=0;i<Dim2[0];i++)
printf("la exactitud de los datos es de %.4f \n",Exactitud);
GPUClose();
t2=clock();
tiempo=(t2-t1)/(float)CLOCKS_PER_SEC;
```

```
printf("el tiempo transcurrido del programa fue de %.4f\n\n",tiempo);

free(Datos1);
free(Datos2);
free(Train);
free(Test);
free(ClassTr);
free(ClassTs);
free(Discriminante);
free(Prediccion);
free(MediaTr);
free(CovClass);
free(InvCovClass);
free(Ind);
free(TRClass);
free(Obs);
free(ResAux);
}
```

D.5 Algoritmo análisis discriminante cuadrático para muchos atributos

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "GPULIBRARY.h"
#include "CPUFUNCTIONS.h"
#ifdef __APPLE__
```

```
#include <CL/opencl.h>
#else
#include <CL/cl.h>
#endif

int main(){

double Exactitud,Aciertos=0.0,PClass,logDet,Res1,Res2,Res3,Mayor;
int i=0,j=0,k,l,ban1=0,ban2=0,ObsClass,NumObs;
int Dim1[3],Dim2[3],NumClass=0,Class[10],tmp;
float tiempo=0;

FILE *archivo;
clock_t t1,t2;
GPUOpen();
t1=clock();
/* EXTRAYENDO DATOS DE TRAIN */
archivo = fopen("./MATEXPQDA/TR3200X1000.txt","r");
Dimensions(archivo,Dim1);
fclose(archivo);

double *Datos1;
Datos1 = (double*)malloc(Dim1[2]*sizeof(double));
archivo = fopen("./MATEXPQDA/TR3200X1000.txt","r");
Data(archivo,Dim1,Datos1);
fclose(archivo);

    /* EXTRAYENDO DATOS DE TEST */
archivo = fopen("./MATEXPQDA/TS3200X1000.txt","r");
Dimensions(archivo,Dim2);
fclose(archivo);
```

```
double *Datos2;
Datos2 = (double*)malloc(Dim2[2]*sizeof(double));
archivo = fopen("./MATEXPQDA/TS3200X1000.txt","r");
Data(archivo,Dim2,Datos2);
fclose(archivo);

    /* DECLARANDO VARIABLES PARA DATOS Y ETIQUETAS
DE TRAIN Y DE TEST */
double *Train,*ClassTr,*Test,*ClassTs;
Train = (double*)malloc((Dim1[2]-Dim1[0])*sizeof(double));
ClassTr = (double*)malloc(Dim1[0]*sizeof(double));
Test = (double*)malloc((Dim2[2]-Dim2[0])*sizeof(double));
ClassTs = (double*)malloc(Dim2[0]*sizeof(double));

    /* OBTENIENDO ETIQUETAS Y DATOS DE TRAIN */
for(i=0;i<Dim1[2];i++) {
if (i==Dim1[1]-1+j){
j=j+Dim1[1];
ClassTr[ban1]=Datos1[i];
ban1++;
}
else{
Train[ban2]=Datos1[i];
ban2++;
}
}

    /* OBTENIENDO NUMERO DE CLASES */
for (i=0;i<Dim1[0];i++)
if(ClassTr[i]>NumClass)
NumClass=ClassTr[i];
```

```
    /* OBTENIENDO ETIQUETAS Y DATOS DE TEST */
j=0;ban1=0;ban2=0;
for(i=0;i<Dim2[2];i++) {
if (i==Dim2[1]-1+j){
j=j+Dim2[1];
ClassTs[ban1]=Datos2[i];
ban1++;
}
else{
Test[ban2]=Datos2[i];
ban2++;
}
}

    /* DIMENSIONES DE TRAIN Y TEST SIN ETIQUETAS*/
int Dim3[3],Dim4[3],Dim5[3],DimTRClass[3],DimMean[3],DimAux[3],*Ind;
Dim3[0]=Dim1[0];Dim3[1]=Dim1[1]-1;Dim3[2]=Dim1[2]-Dim1[0];
Dim4[0]=Dim2[0];Dim4[1]=Dim2[1]-1;Dim4[2]=Dim2[2]-Dim2[0];

    /* DECLARACION DE VARIABLES */
double *Prediccion,*MediaTr,*Discriminante,*CovClass,*Q,*R;
double *InvCovClass,*TRClass,*Obs,*ResAux,DetClass;
Discriminante = (double*)malloc(NumClass*sizeof(double));
Prediccion = (double*)malloc(Dim4[0]*sizeof(double));
MediaTr = (double*)malloc(Dim3[1]*sizeof(double));
Obs = (double*)malloc(Dim4[1]*sizeof(double));
Ind = (int*)malloc(Dim3[0]*sizeof(int));

    /* INICIALIZANDO CLASES */
for(i=0;i<10;i++)
Class[i]=i+1;
```

```
    /* NUMERO DE OBESRVACIONES A PREDECIR */
for(NumObs=0;NumObs<Dim4[0];NumObs++) {

    /* LOCALIZANDO NUMERO DE OBSERVACION */
for(i=0;i<Dim3[1];i++)
Obs[i]=Test[NumObs*Dim3[1]+i];

    /* OBTENIENDO EL VALOR DEL DISCRIMINANTE DE LA CLASE */
for(j=0;j<NumClass;j++) {
ObsClass=0;logDet=0;

    /* OBTENIENDO LOS INDICES Y NUMERO DE OBSERVACIONES DE
CLASE */
for(i=0;i<Dim3[0];i++)
if (ClassTr[i]==Class[j])
{
Ind[ObsClass]=i;
ObsClass++;
}

    /* OBTENIENDO LAS OBSERVACIONES DE CLASE */
DimTRClass[0]=ObsClass;
DimTRClass[1]=Dim3[1];
DimTRClass[2]=DimTRClass[0]*DimTRClass[1];
TRClass = (double*)malloc(DimTRClass[2]*sizeof(double));
for (k=0;k<ObsClass;k++)
for (l=0;l<Dim3[1];l++)
TRClass[k*Dim3[1] + l]=Train[Ind[k]*Dim3[1] + l];

    /* DECLARANDO VARIABLES PARA LA COVARIANZA DE CLASE Y
SU INVERSA */
CovClass = (double*)malloc(DimTRClass[1]*DimTRClass[1]*sizeof(double));
InvCovClass = (double*)malloc(DimTRClass[1]*DimTRClass[1]*sizeof(double));
```

```
Q = (double*)malloc(DimTRClass[1]*DimTRClass[1]*sizeof(double));
R = (double*)malloc(DimTRClass[1]*DimTRClass[1]*sizeof(double));

    /* OBTENIENDO LA PROBABILIDAD DE CLASE */
PClass=(ObsClass*1.0)/Dim3[0];

    /* OBTENIENDO MEDIA DE LA CLASE */
CPUSum(TRClass,DimTRClass,MediaTr,DimMean);
for(i=0;i<DimMean[2];i++)
MediaTr[i]=MediaTr[i]/DimTRClass[0];

    /* OBTENIENDO LA COVARIANZA DE LA CLASE */
ParCov(TRClass,DimTRClass,CovClass,Dim5);

    /* OBTENIENDO LA INVERSA DE LA COVARIANZA DE LA CLASE */
ParInv(CovClass,Dim5,InvCovClass);

    /* REALIZANDO DECOSCOMPOSICION QR DE LA CLASE PARA EVI-
TAR DESBORDAMIENTO */
CPUqr(CovClass,Q,R,Dim5);
for(i=0;i<Dim5[1];i++)
logDet += log(R[i + i*Dim5[1]]);

    /* CALCULANDO EL DISCRIMINANTE DE LA CLASE */
ResAux = (double*)malloc(DimMean[0]*Dim5[1]*sizeof(double));
CPUMatMult(Obs,InvCovClass,DimMean,Dim5,ResAux,DimAux);
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(ResAux,Obs,DimAux,DimMean,&Res1,DimAux);

tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(MediaTr,InvCovClass,DimMean,Dim5,ResAux,DimAux);
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
```

```
CPUMatMult(ResAux,Obs,DimAux,DimMean,&Res2,DimAux);

tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(MediaTr,InvCovClass,DimMean,Dim5,ResAux,DimAux);
tmp=DimMean[1];DimMean[1]=DimMean[0];DimMean[0]=tmp;
CPUMatMult(ResAux,MediaTr,DimAux,DimMean,&Res3,DimAux);

Discriminante[j]= (-0.5*Res1) + Res2 - (0.5*Res3) - 0.5*logDet + log(PClass);

}

    /* ESCRIBIENDO PREDICCION */
Mayor=Discriminante[0];
Prediccion[NumObs]=1;
for (i=1;i<NumClass;i++)
if(Discriminante[i]>Mayor){
Prediccion[NumObs]=Class[i];
Mayor=Discriminante[i];
}
}

for(i=0;i<Dim4[0];i++)
printf("%f\n",Prediccion[i]);

    /* CALCULANDO PREDICCION*/
for(i=0;i<Dim4[0];i++)
if(Prediccion[i]==ClassTs[i])
Aciertos++;

    /* CALCULANDO EXACTITUD*/
Exactitud=(Aciertos*100.0)/Dim4[0];
```

```
for(i=0;i<Dim2[0];i++)
printf("la exactitud de los datos es de %.4f \n",Exactitud);
GPUClose();
t2=clock();
tiempo=(t2-t1)/(float)CLOCKS_PER_SEC;
printf("el tiempo transcurrido del programa fue de %.4f\n\n",tiempo);

free(Datos1);
free(Datos2);
free(Train);
free(Test);
free(ClassTr);
free(ClassTs);
free(Discriminante);
free(Prediccion);
free(MediaTr);
free(CovClass);
free(InvCovClass);
free(Ind);
free(TRClass);
free(Obs);
free(ResAux);
free(Q);
free(R);
}
```