Implementación CPU-GPU y comparativa de las bibliotecas BLAS-CUBLAS, LAPACK-CULA.*

—Reporte Técnico—

Misael Angeles Arce[1], Georgina Flores Becerra[1], and Antonio M. Vidal[2]

[1]DSC, Instituto Tecnológico de Puebla
Avenida Tecnológico 420, 72220 Puebla, México
misa_angeles@hotmail.es, kremhilda@gmail.com
[2]DSIC, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, España
avidal@dsic.upv.es

Abstract

Parallel programming has been available for a few decades using clusters of computers (sets of interconnected computers with shared memory and distributed memory); recently, it has been available using multicore CPUs and GPUs (Graphics Processing Unit). Parallel programming has been very useful in applications of science and engineering to reduce the sequential execution time by parallel numerical libraries for clusters, such as PBLAS and ScaLAPACK, which rely on the sequential numerical libraries BLAS and LAPACK. Parallel numerical libraries have been developed for GPUs, as CUBLAS and CULA (based on BLAS and LAPACK), developed in the CUDA programming platform, developed by NVIDIA. CUDA tries to exploit the GPUs potential. This report aims to introduce the configuration and use of BLAS, LAPACK, CUBLAS and CULA, from programs in C language, and to present a performance comparison among them, so the report can be used as a guide to support engineers and scientists who need this type of computation.

1. Introducción

Actualemente existe un conjunto de tecnologías de procesamiento masivo de datos que ayudan para la automatización de la resolución de problemas de cualquier índole. Estos problemas, dependiendo del número de operaciones que se realizan, pueden tardar un tiempo prolongado para su resolución. Los tiempos de procesamiento de estos problemas pueden ser mejorados con la programación paralela. Existen diferentes formas de aplicar la programación en paralelo, la mayoría depende del hardware que se utiliza. En la mayoría de las ocasiones la programación paralela solo es simulada debido al alto costo del hardware, aun así se obtienen resultados favorables.

Actualmente en el mercado se cuenta con una tecnología para la implementación de la programación en paralelo que utiliza unidades de procesamiento gráfico (GPU por sus siglas en Inglés): CUDA (Compute Unified Device Architecture). Esta tecnología mejora considerablemente los tiempos de ejecución en paralelo.

Este documento tiene como finalidad dar una introducción las bibliotecas secuenciales (BLAS, LAPACK) y paralelas (CUBLAS, CULA) para guiar en la instalación, introducción a la arquitectura, desarrollo y ejecución de programas que utilicen la tecnología CUDA, CULA, BLAS y LAPACK, como herramienta de apoyo para Computación de Altas Prestaciones, así como para personal docente, estudiantes y cualquier persona interesada en aprender estas tecnologías. En el caso de las bibliotecas paralelas se utilizarán las GPUs de las tarjetas NVIDIA, con el propósito de comparar sus tiempos de ejecución y aceleración para observar la disminución de tiempo de ejecución con la programación paralela con las GPUs.

En el sección 2 se ofrece una introducción a la computación paralela, se explican las diferentes arquitecturas que existen y algunas métricas para evaluar el desempeño de los programas paralelos.

*Este trabajo ha sido apoyado por la DGEST mediante el proyecto de investigación 2777.09-P.

En el sección 3 se da una introducción a la computación paralela con CUDA, se explica su arquitectura, modelo de programación y el lenguaje de programación que usa para escribir programas con CUDA.

En el sección 4 se presentan las operaciones básicas de matrices, sistemas de ecuaciones lineales y la descomposición de una matriz en sus valores singulares, también se describen las bibliotecas empleadas en este trabajo (BLAS, LAPACK, CUBLAS y CULA).

En el sección 5 se presenta la instalación de la plataforma CUDA y demás bibliotecas en una computadora con sistema operativo LINUX.

En el sección 6 se muestran programas que implementan algunas de las rutinas del gran conglomerado que integra cada una de las bibliotecas, como son: suma de vectores, multiplicación de matrices, resolución de sistemas de ecuaciones lineales, entre otras. Utilizando las GPUs de la tarjeta NVIDIA para las bibliotecas paralelas y la CPU de la computadora para las bibliotecas secuenciales.

En el sección 7 se muestran las tablas y gráficas comparativas de tiempos de ejecución de todas las rutinas implementadas en este trabajo. Por último se presentan las tablas y gráficas comparativas de precisión en la resolución de sistemas de ecuaciones lineales y la descomposición de una matriz en sus valores singulares.

Por último en la sección 8 se dan las conclusiones.

2. Introducción a la Programación Paralela

La programación paralela consiste en dividir un problema en sub-problemas, con el fin de optimizar el tiempo que tarda el mismo problema en ser resuelto como un todo. Para tal propósito fueron creadas las arquitecturas paralelas. Existen varias formas de clasificar el procesamiento paralelo. Puede considerarse a partir de la organización interna de los procesadores o desde el flujo de información a través del sistema.

2.1. Arquitecturas Paralelas

Michael J. Flynn propuso en 1966 [19] una taxonomía (considerando sistemas con uno o varios procesadores) que se basa en el flujo que siguen los datos dentro de la máquina y de las instrucciones sobre esos datos. Tomando en cuenta que la operación normal de una computadora es recuperar instrucciones de la memoria y ejecutarlas en el procesador, se define como *flujo de instrucciones* a la secuencia de instrucciones leída de la memoria y como *flujo de datos* a las operaciones ejecutadas sobre los datos en el procesador. El procesamiento paralelo puede ocurrir en el flujo de instrucciones, en el flujo de datos o en ambos.

2.2. Mecanismos de control

De acuerdo a los mecanismos de control, las computadoras se clasifican en los siguientes tipos [18]:

- Single Instruction stream, Single Data stream (SISD). Las computadoras de este tipo cuentan con una unidad de control, un procesador (PU) y una unidad de memoria, es decir, tienen un único flujo de instrucciones sobre un único flujo de datos, la ejecución de las instrucciones es secuencial (figura 1).

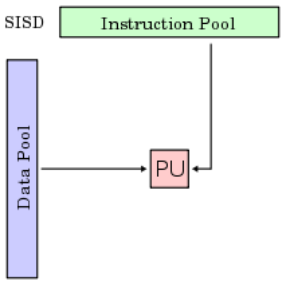


Figura 1: Arquitectura SISD

- Single Instruction stream, Multiple Data stream (SIMD). Estas computadoras tienen un único flujo de instrucciones que operan sobre múltiples flujos de datos, es decir, muchos procesadores (PU) bajo la supervisión de una unidad de control común. El procesamiento es síncrono, aunque la ejecución sigue siendo secuencial como en el caso anterior, todos los procesadores realizan la misma instrucción pero con diferentes conjuntos de datos. Por esta razón existirá concurrencia simulada[1], esta clasificación da origen a la máquina paralela (figura 2).

[1] La concurrencia se simula cuando se cuenta con una sola unidad de procesamiento

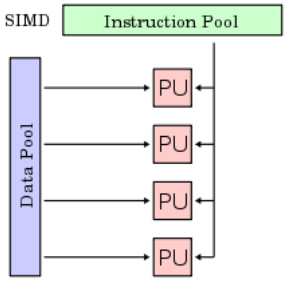


Figura 2: Arquitectura SIMD

- Multiple Instruction stream, Single Data stream (MISD). Las computadoras de este tipo (figura 3) cuentan con múltiples instrucciones que operan sobre un único flujo de datos, es decir, las instrucciones pasan a través de múltiples procesadores (PU). Estos sistemas operan de dos formas:
  - Varias instrucciones operando simultáneamente sobre un único dato.
  - Varias instrucciones operando sobre un dato que se va convirtiendo en un resultado que será la entrada para la siguiente etapa, todos los procesadores pueden trabajar de forma concurrente.

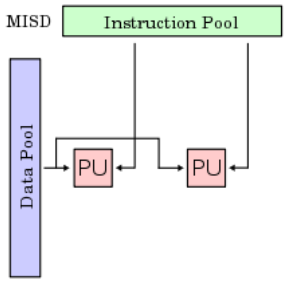


Figura 3: Arquitectura MISD

- Multiple Instruction stream, Multiple Data stream (MIMD). Estas computadoras tienen flujo de múltiples instrucciones que operan sobre múltiples datos. Son máquinas con memoria compartida que permiten ejecutar varios procesos simultáneamente (sistemas multiprocesador). Cada procesador (PU) es capaz de ejecutar su programa con diferentes datos, esto significa que los procesadores operan asíncronamente, es decir, pueden estar haciendo diferentes cosas en diferentes datos al mismo tiempo (figura 4).

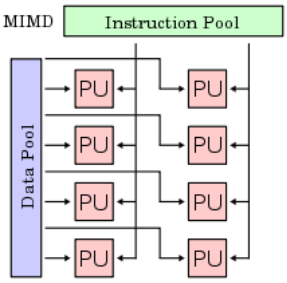


Figura 4: Arquitectura MIMD

2.2.1. Espacios de direccionamiento

Cuando se resuelve un problema entre varios procesadores es necesario que éstos se comuniquen entre sí para intercambiar información. El *paso de mensajes* y la *memoria compartida* proveen dos formas de comunicación [18]:

- Paso de mensajes. En la arquitectura de paso de mensajes, los procesadores usan una red para el paso de mensajes. Cada procesador tiene su propia memoria llamada *memoria privada* la cual es accesible únicamente por el mismo procesador. Los procesadores pueden interactuar sólo por el paso de mensajes. Esta arquitectura también es llamada *memoria distribuida* o *memoria privada*. Los sistemas que usan el paso de mensajes son llamados **multicomputadoras** o **sistemas de memoria distribuida**.
- Memoria compartida. En la arquitectura de memoria compartida es necesario que el hardware de soporte para que todos los procesadores dentro del sistema puedan leer y escribir, ya que los procesadores interactúan modificando datos almacenados en el sistema. Estas computadoras son conocidas como **multiprocesadores** o simplemente **sistemas de memoria compartida**.

2.3. Granularidad

Una computadora paralela puede estar compuesta por un pequeño número de procesadores muy potentes o por un gran número de procesadores poco potentes.

La Granularidad puede ser definida como la relación entre el tiempo requerido por una operación de comunicación básica y el tiempo requerido por un cómputo básico. Las computadoras paralelas en las que esta relación es pequeña son apropiadas para algoritmos que requieren comunicación frecuente, es decir, en los algoritmos en los que el tamaño de grano es pequeño (antes de la comunicación requerida). Estos algoritmos contienen paralelismo de grano fino, comúnmente estas computadoras son llamadas *computadoras de grano fino*. En contraste las computadoras en las que dicha relación es grande son adecuadas para algoritmos que no requieren de mucha comunicación, estas computadoras son referidas como *computadoras de grano grueso* [18].

También se considera el número y la potencia de los procesadores de una computadora para determinar la Granularidad.

Grano grueso	Pocos procesadores de gran potencia.
Grano mediano	Poco más de mil procesadores de mediana o baja potencia.
Grano fino	Del orden de las decenas de millar de procesadores de mediana o baja potencia.

Los procesadores de una computadora de grano grueso son considerablemente caros comparados con los procesadores para computadoras de grano mediano y de grano fino. Por esta razón los procesadores de grano grueso no se producen a gran escala, además requieren de técnicas de fabricación muy costosas. Por otra parte las computadoras de grano mediano suelen construirse con hardware externo de bajo costo.

El tipo de aplicaciones que pueden ser ejecutadas en las computadoras de granos grueso, mediano y fino son muy variadas, sin embargo se pueden mencionar las siguientes [21]:

Las aplicaciones con bajo grado de concurrencia tienen un mejor rendimiento en computadoras de grano grueso al no hacer un uso efectivo de un gran número de procesadores. Las aplicaciones con alto grado de concurrencia son más eficientes en computadoras de grano fino.

2.4. Métricas de desempeño

Los algoritmos secuenciales son evaluados en términos de su tiempo de ejecución en función del tamaño del problema [18]. Se define el tamaño del problema como el número de datos involucrados en una ejecución.

Cuando se trata de algoritmos paralelos el tiempo de ejecución depende no solamente del tamaño del problema sino también de la arquitectura y el número de procesadores de la(s) computadora(s). Por lo tanto para evaluar el rendimiento de un algoritmo dado, es necesario usar métricas de desempeño como:

- Tiempos de ejecución
- Aceleración

2.4.1. Tiempos de ejecución

El *tiempo de ejecución secuencial* de un programa, es el tiempo transcurrido entre el inicio y el fin de la ejecución del mismo en una computadora secuencial. El *tiempo de ejecución paralelo* de un programa, es el tiempo transcurrido entre el momento en que comienza el cómputo paralelo y el momento en el que el último procesador termina su trabajo. Se denota como T_s al tiempo de ejecución secuencial y como T_p al tiempo de ejecución paralelo [18].

2.4.2. Aceleración

Cuando se evalúan algoritmos paralelos el principal interés es conocer qué tan eficiente es con respecto a la implementación en forma secuencial. La aceleración es la métrica que captura el beneficio relativo al resolver un problema de forma paralela, entonces la *aceleración*, denotada con S, se define como el cociente del tiempo que se tarda en completar el cómputo de la tarea usando un sólo procesador entre el tiempo que necesita para realizarlo con p procesadores trabajando en paralelo. Se asume que los p procesadores usados por el algoritmo paralelo deben ser idénticos al procesador usado por el algoritmo secuencial [18].

$$S = \frac{T_s}{T_p}.$$

Es importante mencionar que al comparar el rendimiento de un algoritmo (secuencial vs paralelo) se debe considerar que pueden estar disponibles más de una versión, no siendo todas adecuadas para el paralelismo. Al usar una computadora secuencial se espera que el algoritmo resuelva el problema en la menor cantidad de tiempo posible. Entonces se debe comparar el rendimiento del algoritmo paralelo contra el rendimiento del algoritmo secuencial más rápido que resuelve el mismo problema. Para este documento las bibliotecas secuenciales utilizadas (BLAS y LAPACK) están optimizadas, por lo tanto los algoritmos en los que se basan son los más rápidos.

3. Introducción a CUDA

La llegada de las CPUs de multi-núcleo y las GPUs de múltiples núcleos significa que la principal corriente de procesadores son sistemas paralelos. Además su paralelismo continua a escala según la ley de Moore (Esta ley consiste en que cada 18 meses se duplicará el número de transistores en los circuitos integrados). El reto es desarrollar aplicaciones que sean transparentes en el uso de la programación en paralelo para aprovechar el creciente número de procesadores [13].

CUDA es un modelo de programación paralela y un entorno de desarrollo diseñado para superar este desafío manteniendo una baja curva de aprendizaje para los programadores que están familiarizados con el lenguaje de programación estándar C.

En su esencia CUDA tiene tres elementos clave: El primero es una jerarquía de grupos de hilos, el segundo es un modelo de recursos compartidos y el tercero es un grupo de instrucciones de sincronización, que son expuestos al programador como un conjunto mínimo de extensiones de C.

Estos elementos orientan al programador a dividir el problema en subproblemas secundarios que pueden ser resueltos de forma paralela y después en subprogramas más pequeños que pueden ser resueltos en forma cooperativa en paralelo. Esto permite una escalabilidad transparente, ya que cada subprograma puede ser programado para ser resuelto en cualquiera de los núcleos disponibles del procesador. Un programa compilado con CUDA se puede ejecutar en cualquier número de núcleos del procesador y el sistema de ejecución solo necesita saber el número de procesadores físicos.

3.1. Definición

En Noviembre de 2006 NVIDIA introduce CUDA (**C**ompute **U**nified **D**evice **A**rchitecture), que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por NVIDIA.

CUDA es una arquitectura de cómputo paralelo para fines generales que aprovecha el motor de cómputo paralelo de las unidades de procesamiento gráfico (GPU) de NVIDIA para resolver muchos de los problemas de cómputo más complejos en una fracción del tiempo requerido por la CPU. CUDA incluye una arquitectura de instrucciones (ISA en Inglés) y un motor de cómputo paralelo en la GPU [13].

Para programar en la arquitectura CUDA, actualmente los desarrolladores pueden usar C, uno de los lenguajes de programación de alto nivel más utilizados (figura 5), que entonces puede ejecutarse con un excelente rendimiento en un procesador compatible con CUDA [13].

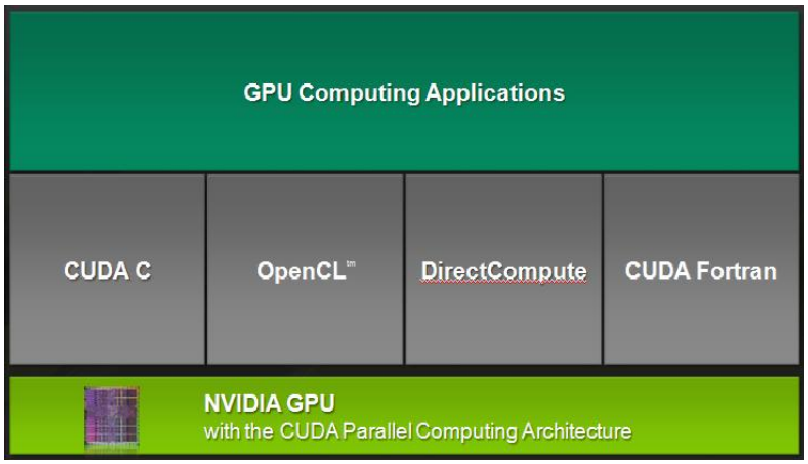


Figura 5: Soporte de CUDA para lenguajes

Sometidos a la demanda insaciable de mercado en cómputo en tiempo real, alta definición de gráficos en 3D, el procesador de gráficos programable o GPU se ha convertido en un procesador de múltiples núcleos altamente paralelos, multiproceso, con enorme poder de cómputo (figura 6) y ancho de banda de memoria muy alto.

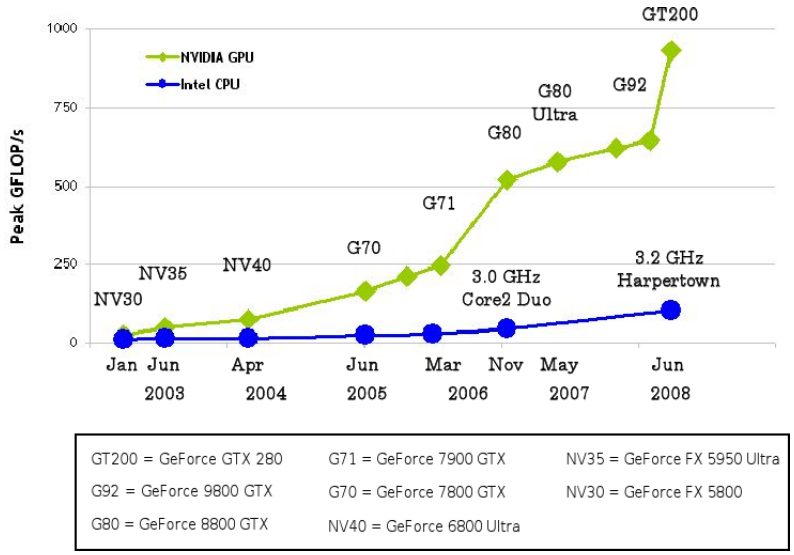


Figura 6: Operaciones de punto flotante por segundo (CPU y GPU)

La razón de la discrepancia en la capacidad de operaciones en punto flotante entre la CPU y la GPU es que la GPU está especializada en cómputo intensivo, computación altamente paralela y por lo tanto diseñada de manera que más transistores se dedican al procesamiento de datos (figura 7) en lugar de almacenamiento en cache de datos y control de flujo.

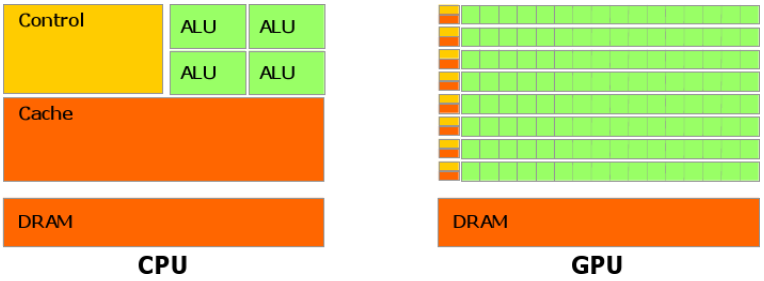
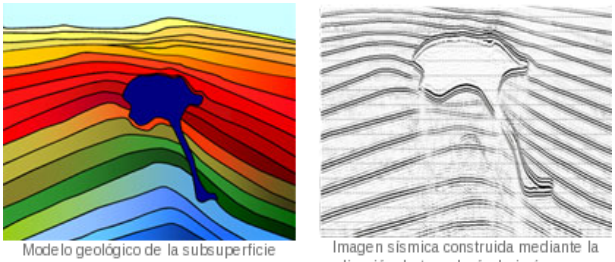


Figura 7: La GPU con más transistores para el procesamiento de datos

3.2. Aplicaciones de CUDA

En la actualidad, decenas de miles de desarrolladores, científicos, estudiantes, creadores de juegos e investigadores realizan aplicaciones que aprovechan la computación de la GPU en áreas tan diversas como juegos basados en la física, análisis de riesgo de activos, análisis de datos sísmicos y pronósticos climáticos. Algunas de ellas son las siguientes [1]:

SeismicCity[6] Utiliza CUDA para mejorar la posibilidad de descubrir petróleo. El costo de la perforación en la exploración profunda de pozos de petróleo puede llegar a cientos de millones de dólares. En muchos casos, existe apenas una posibilidad de perforar un pozo con éxito. La tecnología de imágenes en profundidad basada en CUDA (figura 8) de SeismicCity interpreta datos sísmicos que llevan a la selección de nuevas ubicaciones para perforación con mucho más rapidez de lo que podrían hacerlo los sistemas anteriores. Para mejorar la calidad y eficiencia de sus imágenes SeismicCity se inclinó por CUDA y las GPU NVIDIA Tesla de la serie 8. Con ello se logró un aumento de hasta 14 veces en el rendimiento con relación a la configuración anterior basada en la CPU.

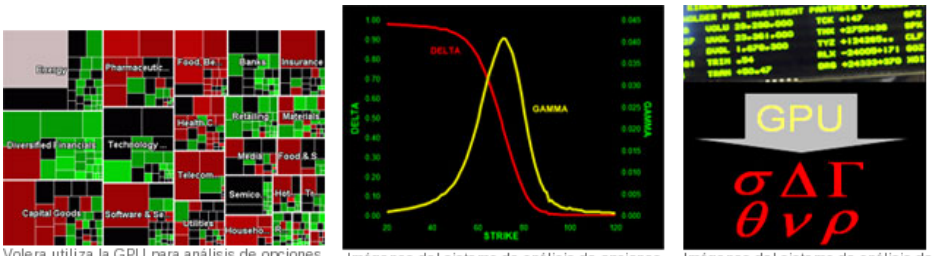


Modelo geológico de la subsuperficie que muestra objetivos de exploración bajo el cuerpo salino

Imagen sísmica construida mediante la aplicación de tecnología de imágenes en profundidad

Figura 8: Aplicaciones de CUDA (Energía)

Para **Hanweck Associates**[4] una firma de servicios financieros especializada en la administración de riesgo e inversiones, es esencial ofrecerles a los clientes una forma de recalcular las opciones en tiempo real. Hanweck lo logra a través de su línea Volera (figura 9) de análisis de opciones de alto rendimiento. Usando apenas 12 GPU aptas para CUDA, Volera analiza el mercado completo de opciones de capital de los EE UU. en tiempo real, una tarea que antes necesitaba más de 60 servidores convencionales.



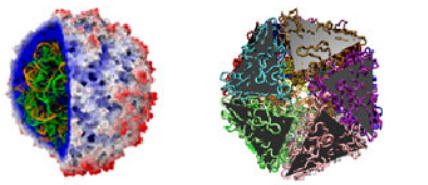
Volera utiliza la GPU para análisis de opciones en tiempo real

Imágenes del sistema de análisis de opciones de Volera

Imágenes del sistema de análisis de opciones de Volera

Figura 9: Aplicaciones de CUDA (Finanzas)

Los virus, causa de muchas enfermedades, son los organismos naturales más pequeños que se conocen. Debido a su simplicidad y al tamaño pequeño, los biólogos computacionales eligieron un virus para su primer intento de simular una forma de vida completa usando una computadora. Se trata del virus satélite de mosaico del tabaco, uno de los más pequeños. Los investigadores simularon el virus en una gota de agua salada usando un programa llamado NAMD (Nanoscale Molecular Dynamics o dinámica molecular en nano escala) de la **Universidad de Illinois en Urbana-Champaign**[8]. Las aplicaciones NAMD se han acelerado con CUDA (figura 10), logrando aumentos impresionantes de hasta 330 veces en la velocidad, en comparación con una CPU de núcleo único al ejecutarse en un clúster acelerado por la GPU en el Centro Nacional de Aplicaciones de Supercomputación (NCSA).

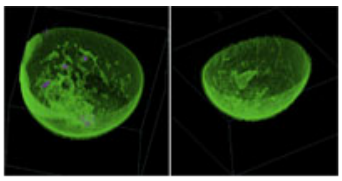


Capa protectora de la proteína que envuelve el genoma viral

Estructura icosahédrica de la capa de la proteína

Figura 10: Aplicaciones de CUDA (Investigación)

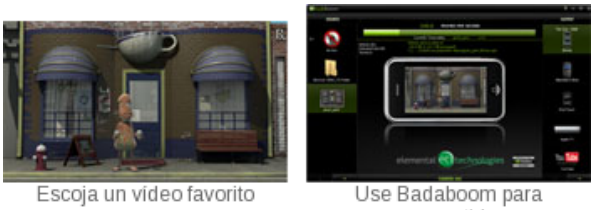
La capacidad de producir rápidamente imágenes bastante detalladas en un plazo corto tiene gran relevancia en el campo de las ecografías para detección del cáncer de mama. **TechniScan**[7], un desarrollador de sistemas de imágenes automatizadas por ultrasonido, cambió su algoritmo patentado de un sistema tradicional en la CPU a CUDA y a las GPU NVIDIA Tesla. El sistema basado en CUDA (figura 11) es capaz de procesar el algoritmo de TechniScan dos veces más rápido.



Renderizado volumétrico del sistema de ultrasonido para la mama completa (WBU) de TechniScan.

Figura 11: Aplicaciones de CUDA (Medicina)

A medida en que crece la popularidad de los dispositivos de medios digitales, los usuarios experimentan mayor frustración por la demora en la tarea de colocar vídeo en sus dispositivos. Por ejemplo, convertir una película de dos horas de duración puede tardar seis horas o más cuando se usa la CPU de la computadora. **Badaboom**[2] de Elemental es un programa de transcodificación de vídeo que convierte los archivos de vídeo estándar en formatos que se ejecutan en el Ipod y en otros dispositivos portátiles. Al aprovechar CUDA en las GPU NVIDIA, Badaboom puede acelerar el proceso de conversión para que resulte hasta 18 veces más rápido que los métodos tradicionales. La conversión de una película de dos horas de duración tarda cerca de 20 minutos (figura 12) en vez de varias horas.



Escoja un video favorito

Use Badaboom para convertirlo

Figura 12: Aplicaciones de CUDA (Vídeo y Fotografía)

Además, el mundo académico ha reconocido el increíble potencial de esta arquitectura de computación [9]. La computación de la GPU basada en CUDA ahora forma parte del plan de estudios de más de 20 universidades, entre las que se incluyen el MIT, Harvard, Cambridge, Oxford, los Institutos de Tecnología de la India, la Universidad Nacional de Taiwan y la Academia China de Ciencias.

En México actualmente CUDA se enseña en 2 Universidades (ver figura 13) de acuerdo a la página de Internet `http://www.nvidia.com/object/cuda_courses_and_map.html` consultada el 19 de Octubre del 2010, en la cual se pide a quienes estén enseñando CUDA con C/C++ que envíen información a NVIDIA para considerarlos en el conteo.

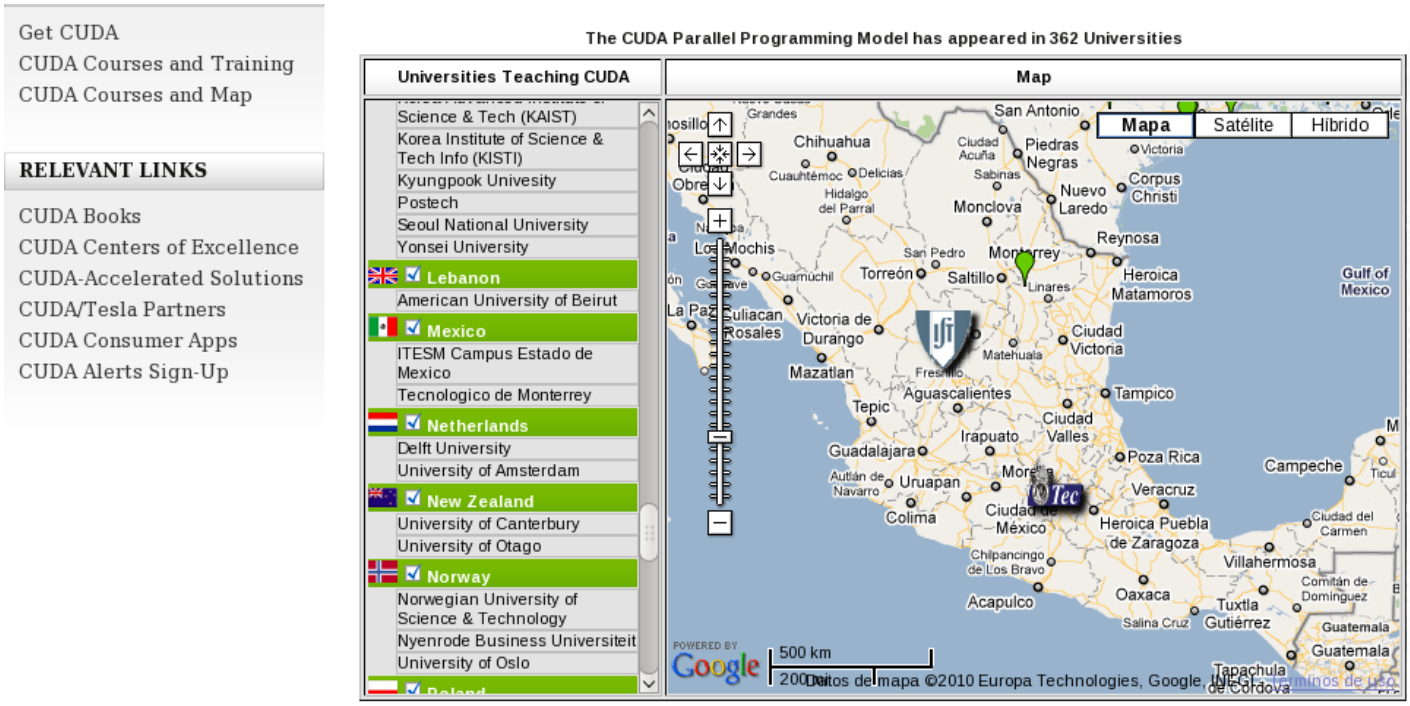


Figura 13: Universidades Mexicanas en las que se enseña CUDA

3.3. Arquitectura CUDA

La arquitectura de las tarjetas NVIDIA aptas para CUDA corresponden a un *multiporcesador*, como se ilustra en la figura 14. En esta figura puede apreciarse la arquitectura de una tarjeta NVIDIA con 8 mulprocesadores, con 16 núcleos cada uno y sus respectivas unidades de memoria.

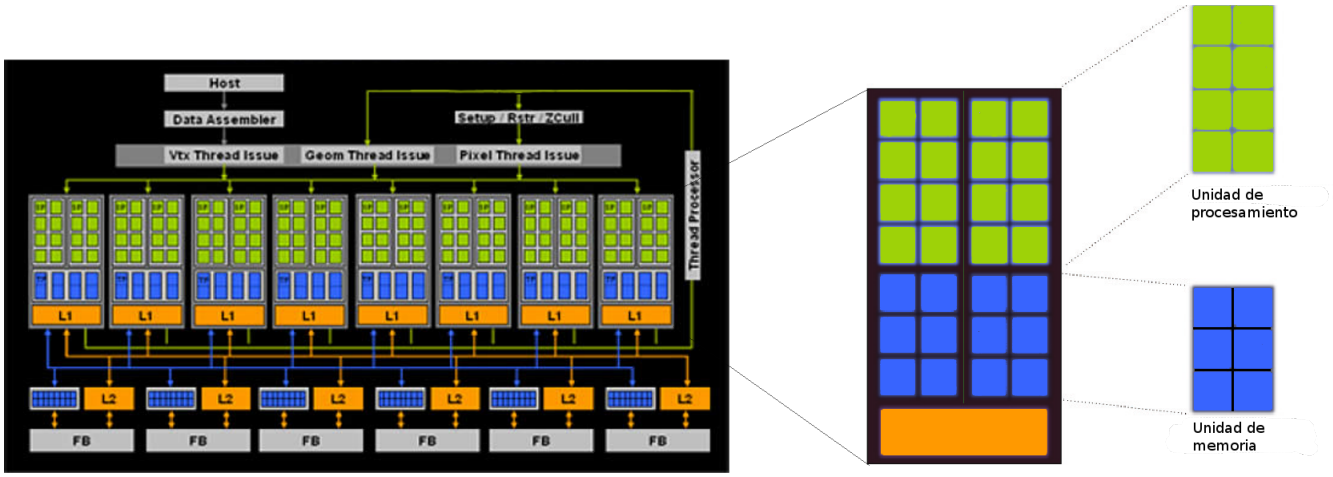


Figura 14: Arquitectura de Tarjeta NVIDIA con 128 núcleos

De acuerdo con lo visto en la sección 2.2 el mecanismo de control de la arquitectura CUDA corresponde al de las computadoras tipo SIMD, ya que la programación en paralelo en CUDA consiste en ejecutar un conjunto de hilos que realizan las mismas operaciones sobre múltiples datos.

Por otra parte, CUDA usa la memoria compartida como medio de comunicación entre los multiporcesadores (módulos L1 y L2 de la figura 14) que componen una tarjeta NVIDIA apta para CUDA.

De acuerdo a la teoría de las arquitecturas paralelas, las tarjetas NVIDIA se clasifican como mulitprocesadores, tomando en cuenta que una tarjeta NVIDIA está integrada por un conjunto de procesadores, entonces la granularidad de éstas es de **grano fino**.

3.4. CPU vs GPU

Para la programación en paralelo sobre GPUs se implementan programas en C con extensiones para CUDA. Algunas secciones de código de estos programas se ejecutan en CPU (las secciones de código secuencial), mientras que aquellas secciones de código paralelo se ejecutan en GPUs en forma de hilos, como se verá en los siguientes apartados.

3.5. Definición de Host y Device

El modelo de programación CUDA asume que los hilos CUDA se ejecutan en un dispositivo (device) físicamente separado que opera como un coprocesador al host donde se ejecuta el programa C. También da por hecho que tanto el dispositivo (device) como el host tienen su propia memoria dinámica de acceso aleatorio (DRAM) [13].

El **host** es un sistema con una o más CPUs y el **device** es una tarjeta de gráficos NVIDIA con GPUs apta para CUDA [13].

Comúnmente las tarjetas NVIDIA se asocian con el renderizado de gráficos, pero éstas son muy útiles para resolver cálculos aritméticos pudiendo computar miles de hilos en paralelo, esta característica las sitúa como un dispositivo ideal para el procesamiento en paralelo.

Es importante conocer las diferencias entre el host y el device para que el rendimiento de los programas CUDA que se escriban sea eficiente.

3.6. Hilos y RAM

Las diferencias principales entre un host y un dispositivo residen en los hilos y la memoria RAM [13]:

- **Hilos.** Los sistemas host pueden soportar un número limitado de hilos concurrentes. Por ejemplo, los servidores con cuatro procesadores (quad-core) pueden ejecutar solamente 16 hilos en paralelo. En contraste, la unidad paralela ejecutable más pequeña sobre un dispositivo (llamada warp o urdimbre[2]) comprende 32 hilos. Por ejemplo todas las GPUs de la tarjeta NVIDIA pueden soportar 768 hilos activos por multiporcesador. Los hilos de la CPU son generalmente entidades pesadas, el sistema operativo tiene que estar encendiendo y apagando hilos para proveer ejecución multi-hilo, es muy costoso pasar de un estado a otro. Por otro lado los hilos de la GPU son muy ligeros, en un sistema típico miles de hilos son encolados a razón de 32 hilos por urdimbre. Si la GPU debe esperar en una urdimbre, simplemente empieza a ejecutar otra, como los registros se asignan a hilos activos, ningún intercambio de registros y de estado ocurre entre hilos de la GPU, estando los recursos disponibles hasta que se complete la ejecución de los hilos.
- **RAM**. Ambos dispositivos (host y device) tienen su propia RAM. En el host generalmente la RAM es igualmente accesible para todo el código (con limitaciones establecidas por el sistema operativo). En el device la RAM es dividida virtual y físicamente en diferentes tipos (ver sección 3.7.3), cada una de ellas para satisfacer diferentes necesidades.

3.7. Modelo de Programación CUDA

3.7.1. Kernel

La extensión de C para CUDA [12] permite a los programadores escribir funciones llamadas *kernels*, que cuando son invocadas se ejecutan N veces en paralelo en diferentes *hilos de CUDA*, en contraste con las funciones de C que se ejecutan sólo una vez.

En el programa 1 se muestra un kernel, que es definido usando la palabra reservada **__global__** como sigue [13]:

Programa 1: Definición de kernel

```
1 // Definición del kernel
2 __global__ void nombre_kernel(tipo1 p_1, tipo2 p_2, ..., tipon p_n)
3 {
4   // sentencias del kernel
5 }
```

La estructura de una sentencia para invocar un kernel es:

nombre_kernel<<<configuración_de_ejecución>>>(parámetro_1, parámetro_2, . . . **, parámetro_n)**

donde **configuración_de_ejecución** juega un papel muy importante, ya que en ella se especifican valores como el número de hilos que ejecutan el kernel en paralelo y la forma en que se organizan, como se verá enseguida.

[2]Urdimbre, conjunto de hilos que se colocan paralelos unos a otros.

3.7.2. Jerarquía de hilos

Para la ejecución de programas paralelos a través de kernels CUDA emplea una jerarquía de hilos que está dada por tres categorías (ver figura 15) [13]:

- hilos (threads), los cuales se organizan en
- bloques (blocks), los cuales se organizan en
- mallas (grids), que son la jerarquía más alta.

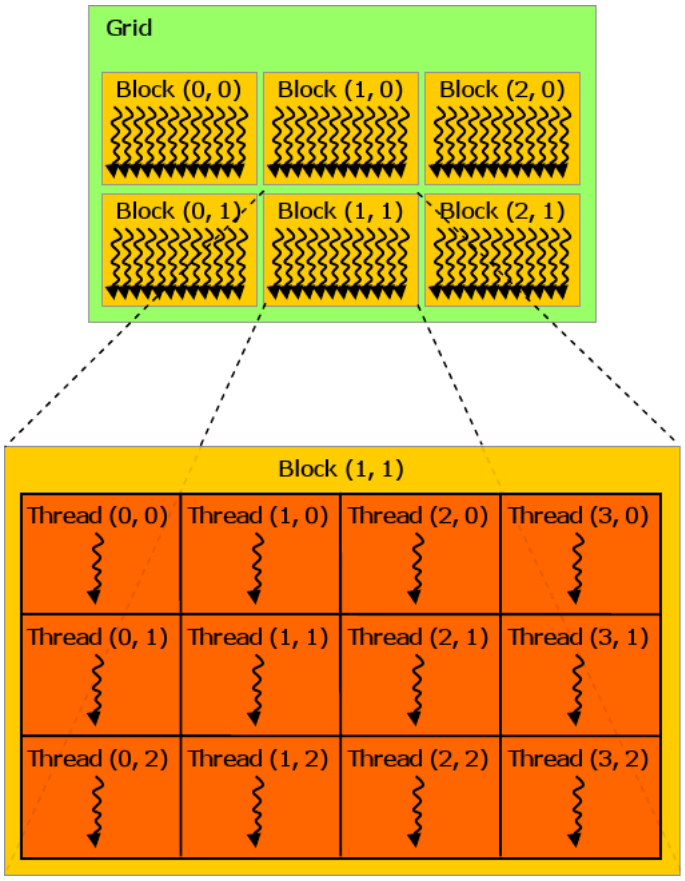


Figura 15: Jerarquía de Hilos CUDA

La dimensión de la malla se obtiene de la variable interconstruida **gridDim** y la dimensión de un bloque de la variable interconstruida **blockDim**; por otra parte, a cada hilo se le asigna un identificador único dado por la variable interconstruida **threadIdx**; de igual forma cada bloque se identifica con la variable interconstruida **blockIdx**:

threadIdx es una variable de tipo *unit3* que contiene el índice de un hilo dentro de un bloque.
blockIdx es una variable de tipo *unit3* que contiene el índice de un bloque dentro de una malla.
unit3 es un vector interconstruido de tipo entero, tiene una función constructor *unit3 nombre_variable(x, y, z).*
blockDim es una variable de tipo *dim3* que contiene las dimensiones de un bloque.
gridDim es una variable de tipo *dim3*, contiene la dimensión de una malla.
dim3 es un vector interconstruido, basado en *unit3*, es usado para especificar dimensiones, su función constructor es *dim3 nombre_variable(x, y, z).* Cualquier componente no especificado se inicializa con 1.

Los hilos pueden estar organizados, de acuerdo a la aplicación, en bloques de una dimensión (para manejar vectores), de dos dimensiones (para manejar matrices) y de tres dimensiones (para manejar arreglos tridimensionales). De acuerdo a esto, el identificador único de cada hilo puede calcularse de la siguiente forma:

Arreglo	**Índice**	**Identificador**
(D_x)	(x)	(x)
(D_x, D_y)	(x, y)	$(x + yD_x)$
(D_x, D_y, D_z)	(x, y, z)	$(x + yD_x + zD_xD_y)$

donde D_x corresponde a una dimensión (vectores), D_y corresponde a la segunda dimensión (matrices) y D_z corresponde a la tercera dimensión (arreglos tridimensionales). En la figura 15 se ilustra un conjunto de hilos organizados en un bloque de dos dimensiones, Por otra parte, los bloques están organizados en mallas de dos dimensiones.

El programa 2 es un ejemplo para ilustrar la definición e invocación de un kernel.

Programa 2: Invocación de un kernel

```
1 // Definición del kernel
2 __global__ void sumaVector(float *x, float *y, float *z)
3 {
4   int i = threadIdx.x;
5   z[i] = x[i] + y[i];
6 }
7
8 int main()
9 {
10   ...
11   // Invocación del kernel.
12   sumaVector<<<1, N>>>(x, y, z);
13 }
```

El ejemplo anterior realiza la suma de dos vectores guardando el resultado en un tercer vector (línea 5), como puede observarse el kernel debe ser invocado desde un programa C (líneas 8-13). Cada uno de los **N** hilos contenidos en **1** bloque (línea 12) realiza la suma de los elementos i-ésimo de **x** e **y** respectivamente (línea 4).

El programa 3 muestra el empleo **blockDim** y **blockIdx**.

Programa 3: Uso de blockDim y blockIdx

```
1 // Definición del kernel
2 __global__ void sumaMatriz(float A[N][N], float B[N][N], float C[N][N])
3 {
4   int i = blockIdx.x * blockDim + threadIdx.x;
5   int j = blockIdx.y * blockDim + threadIdx.y;
6   if( i < N && j < N )
7   C[i][j] = A[i][j] + B[i][j];
8 }
9
10 int main()
11 {
12   ...
13   // Invocación del kernel
14   int hilosPorBloque = 256;
15   int numBloques = (N + hilosPorBloque - 1) / hilosPorBloque;
16   sumaMatriz<<<numBloques, hilosPorBloque>>>(A, B, C);
17 }
```

El ejemplo anterior realiza la suma de dos matrices de tamaño NxN. El kernel **sumaMatriz** es ejecutado por un conjunto de hilos, donde cada hilo le corresponde sumar uno de los componentes de la matriz **A** con el corresponde componente de la matriz **B**, almacenando el resultado en el correspondiente de la matriz **C** (línea 7); para esto cada hilo calcula su correspondiente componente de la matriz **(i,j)** de acuerdo a la posición que ocupa dentro del bloque (es decir de acuerdo a **blockIdx**, **blockDim** y **threadIdx**, líneas 4 y 5).

Para invocar al kernel se definen en el ejemplo 256 hilos por bloque (línea 14), de acuerdo a esto se calcula el número de bloques requeridos para sumar NxN datos (línea 15). Con esto se invoca al kernel para ser ejecutado por **numBloques** bloques, cada uno de ellos con **hilosPorBloque** hilos.

A la vista de la jerarquía de hilos presentada, cabe señalar que la **configuración_de_ejecución** especificada en la invocación de un kernel juega un papel muy importante ya que en ella se especifican valores como la dimensión de los bloques y de los hilos por bloque. La configuración de ejecución, como ya se vio, tiene la siguiente forma:

<<<**Dm, Db**>>>

donde:

Dm Es la dimensión de la malla, igual al número de bloques.

Db Es la dimensión de los bloques, igual al número de hilos por bloque.

Existen otros parámetros en la configuración de ejecución, pero estos son los más usuales en la invocación de un kernel.

3.7.3. Jerarquía de memoria

En CUDA también se maneja una jerarquía de memoria, de modo que [13]:

- Cada hilo puede acceder a los datos de diferentes espacios de memoria (ver figura 16).
- Cada hilo tiene su propia memoria local.

- Cada bloque tiene su propia memoria compartida, accesible para todos los hilos del bloque, se caracteriza por tener el mismo tiempo de vida que los hilos que componen el bloque.
- Todos los hilos tienen acceso a la memoria global.

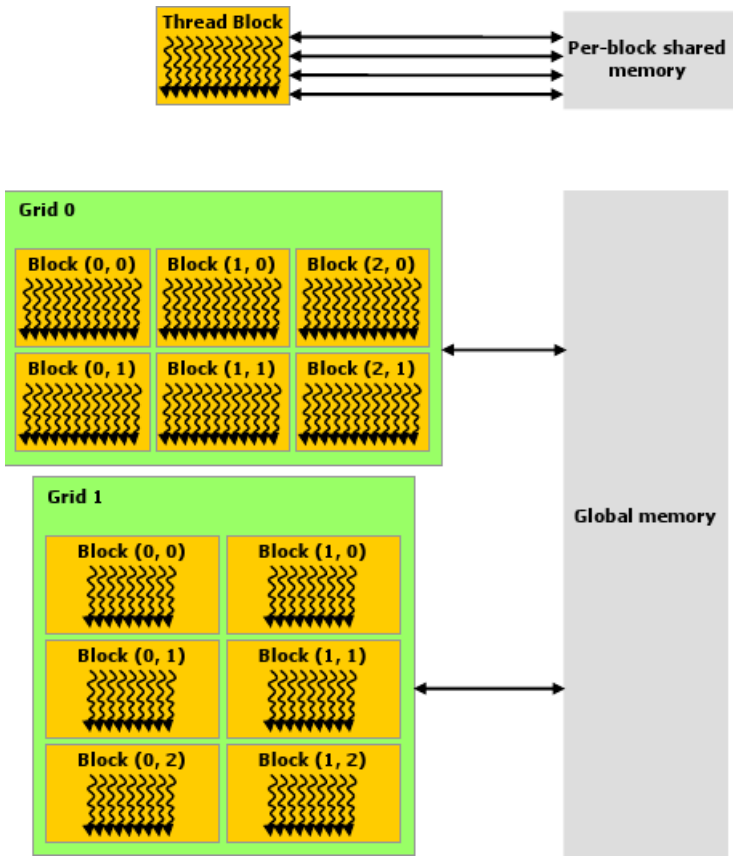


Figura 16: Jerarquía de memoria CUDA

4. Bibliotecas numéricas para operaciones básicas de vectores, matrices y sistemas de ecuaciones lineales

4.1. Operaciones básicas de vectores y matrices

El concepto de **vector** viene del mundo de la física, donde vector es una cantidad que tiene magnitud y dirección (por ejemplo fuerza y velocidad). El concepto matemático de *espacio vectorial* generaliza estas ideas, con aplicaciones en la teoría de códigos, la geometría finita, la criptografía y otras áreas de las matemáticas [20].

Un **espacio vectorial** consiste en un conjunto V y dos operaciones, suma de vectores y multiplicación por un escalar.

Un **Vector** es un objeto individual de un espacio vectorial.

Una **matriz** $A = (a_{ij})$ de $m \times n$, es un arreglo rectangular con números reales o complejos en cada a_{ij} dispuestos en *m* **filas** y *n* **columnas**.

La i-ésima fila de A, denotada como $A(i,:)$ es un arreglo $a_{i1}, a_{i2}, \ldots, a_{in}$. Los elementos de la i-ésima fila puede ser dispuestos como un **vector fila**. La j-ésima columna de A, denotada como $A(:,j)$ es un arreglo

$$\begin{matrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{matrix}$$

que puede ser identificado como un **vector columna**.

Generalmente las matrices son representadas con letras mayúsculas ($A, B, C, D, \dots$), los vectores con letras minúsculas ($x, y, z, \dots$) y los escalares con letras griegas minúsculas ($\alpha, \beta, \dots$).

4.1.1. Escalamiento de un vector: $\alpha \times x$

Sean x un vector y α un escalar, tales que:

$$\begin{aligned} \mathbf{x} &= (x_i, \dots, x_n) \quad x_i \in \mathbb{R} \quad para \quad todo \quad i = 1, \dots, n. \\ \alpha &= w \mid w \in \mathbb{R} \end{aligned}$$

entonces, el escalamiento por α del vector x se define comos sigue:

$$\alpha \times x = (\alpha \times x_i, \dots, \alpha \times x_n).$$

Por ejemplo, si

$$\begin{aligned} x &= (2, 4, 6, 8), \\ \alpha &= 9, \end{aligned}$$

entonces

$$\begin{aligned} \alpha \times x &= (9 \times 2, 9 \times 4, 9 \times 6, 9 \times 8) \\ &= (18, 36, 54, 72) \end{aligned}$$

4.1.2. Suma vector-vector: $\alpha \times x + y$

Sean x, y vectores y α un escalar, tales que:

$$\begin{aligned} \mathbf{x} &= (x_i, \dots, x_n) \\ \mathbf{y} &= (y_i, \dots, y_n) \quad x_i, y_i \in \mathbb{R} \quad para \quad todo \quad i = 1, \dots, n. \\ \alpha &= w \mid w \in \mathbb{R} \end{aligned}$$

entonces, la suma o adición de los vectores x, y por un escalar α se define como sigue:

$$\alpha \times x + y = (\alpha \times x_i + y_i, \dots, \alpha \times x_n + y_n).$$

Por ejemplo, si

$$\begin{aligned} x &= (2, 4, 6, 8), \\ y &= (1, 3, 5, 7), \\ \alpha &= 9 \end{aligned}$$

entonces

$$\begin{aligned} \alpha \times x + y &= (9 \times 2 + 1, 9 \times 4 + 3, 9 \times 6 + 5, 9 \times 8 + 7) \\ &= (18 + 1, 36 + 3, 54 + 5, 72 + 7) \\ &= (19, 39, 59, 79) \end{aligned}$$

4.1.3. Producto vector-vector: $x \times y$

Sean x, y vectores, tales que:

$$\begin{aligned}&\mathbf{x} = (x_i, \ldots, x_n)\\ &\mathbf{y} = (y_i, \ldots, y_n) \quad x_i, y_i \in \mathbb{R} \quad para \quad todo \quad i = 1, \ldots, n.\end{aligned}$$

entonces, el producto vector vector de x con y se define como sigue:

$$\mathbf{x} \times \mathbf{y} = (x_i \times y_i, + \cdots +, x_n \times y_n).$$

Por ejemplo, si

$$\begin{aligned}&x = (2, 4, 6, 8),\\ &y = (1, 3, 5, 7),\end{aligned}$$

entonces

$$\begin{aligned}x \times y &= (2 \times 1 + 4 \times 3 + 6 \times 5 + 8 \times 7)\\ &= (2 + 12 + 30 + 56)\\ &= 100\end{aligned}$$

4.1.4. Producto matriz-vector: $\alpha \times A \times x + \beta \times y$

Sean A una matriz; x, y vectores; α, β escalares, tales que:

$$\begin{aligned}&\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}\\ &\mathbf{x} = (x_i, \ldots, x_n)\\ &\mathbf{y} = (y_i, \ldots, y_n) \quad A_{ij}, x_i, y_i \in \mathbb{R} \quad para \quad todo \quad i = 1, \ldots, n \quad j = 1, \ldots, m.\\ &\alpha = w \mid w \in \mathbb{R}\\ &\beta = u \mid u \in \mathbb{R}\end{aligned}$$

entonces, el producto matriz-vector de la forma $\alpha \times A \times x + \beta \times y$ se define como sigue:

$$\alpha \times A \times x + \beta \times y = \begin{pmatrix} \alpha \times A_{11} \times x_1 + \beta \times y_1 + \alpha \times A_{12} \times x_2 + \beta \times y_2 & + \cdots + & \alpha \times A_{1n} \times x_n + \beta \times y_n \\ \alpha \times A_{21} \times x_1 + \beta \times y_1 + \alpha \times A_{22} \times x_2 + \beta \times y_2 & + \cdots + & \alpha \times A_{2n} \times x_n + \beta \times y_n \\ \vdots & \vdots & \\ \alpha \times A_{m1} \times x_1 + \beta \times y_1 + \alpha \times A_{m2} \times x_2 + \beta \times y_2 & + \cdots + & \alpha \times A_{mn} \times x_n + \beta \times y_n \end{pmatrix}.$$

Por ejemplo, si

$$\begin{aligned}&\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},\\ &x = (2, 4, 6),\\ &y = (1, 3, 5),\\ &\alpha = 9,\\ &\beta = 6,\end{aligned}$$

entonces

$$\begin{aligned}\alpha \times A \times x + \beta \times y &= \begin{pmatrix}(9\times 1\times 2+6\times 1)+(9\times 2\times 4+6\times 3)+(9\times 3\times 6+6\times 5)\\(9\times 4\times 2+6\times 1)+(9\times 5\times 4+6\times 3)+(9\times 6\times 6+6\times 5)\\(9\times 7\times 2+6\times 1)+(9\times 8\times 4+6\times 3)+(9\times 9\times 6+6\times 5)\end{pmatrix}\\ &= \begin{pmatrix}(18+6)+(72+18)+(162+30)\\(72+6)+(180+18)+(324+30)\\(126+6)+(288+18)+(486+30)\end{pmatrix}\\ &= \begin{pmatrix}306\\630\\954\end{pmatrix}\end{aligned}$$

4.1.5. Producto matriz-matriz: $\alpha \times A \times B + \beta \times C$

Sean A, B, C matrices; α, β escalares, tales que:

$$\mathbf{A} = \begin{pmatrix}a_{11} & a_{12} & \dots & a_{1k}\\ a_{21} & a_{22} & \dots & a_{2k}\\ \vdots & \vdots & \ddots & \vdots\\ a_{m1} & a_{m2} & \dots & a_{mk}\end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix}b_{11} & b_{12} & \dots & b_{1n}\\ b_{21} & b_{22} & \dots & b_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ b_{k1} & b_{k2} & \dots & b_{kn}\end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix}c_{11} & c_{12} & \dots & c_{1n}\\ c_{21} & c_{22} & \dots & c_{2n}\\ \vdots & \vdots & \ddots & \vdots\\ c_{m1} & c_{m2} & \dots & c_{mn}\end{pmatrix}$$

$$A_{ij}, B_{jl}, C_{il} \in \mathbb{R} \quad para \quad todo \quad i = 1, \dots, m; \quad j = 1, \dots, k; \quad l = 1, \dots, n$$

$$\alpha = w \mid w \in \mathbb{R}$$

$$\beta = u \mid u \in \mathbb{R}$$

entonces, el producto matriz-matriz de la forma $\alpha \times A \times C + \beta \times C$ se define como sigue:

$$\alpha \times A \times B + \beta \times C = \begin{pmatrix}
\begin{bmatrix}\alpha \times A_{11} \times B_{11} + \beta \times C_{11}\\ + \\ \alpha \times A_{12} \times B_{21} + \beta \times C_{12}\\ +\cdots+ \\ \alpha \times A_{1k} \times B_{k1} + \beta \times C_{1k}\end{bmatrix} & \begin{bmatrix}\alpha \times A_{11} \times B_{12} + \beta \times C_{11}\\ + \\ \alpha \times A_{12} \times B_{22} + \beta \times C_{12}\\ +\cdots+ \\ \alpha \times A_{1k} \times B_{k2} + \beta \times C_{1k}\end{bmatrix} & \dots & \begin{bmatrix}\alpha \times A_{1k} \times B_{kn} + \beta \times C_{1k}\\ + \\ \alpha \times A_{1k} \times B_{kn} + \beta \times C_{1k}\\ +\cdots+ \\ \alpha \times A_{1k} \times B_{kn} + \beta \times C_{1k}\end{bmatrix}\\
\begin{bmatrix}\alpha \times A_{21} \times B_{11} + \beta \times C_{21}\\ + \\ \alpha \times A_{22} \times B_{21} + \beta \times C_{22}\\ +\cdots+ \\ \alpha \times A_{2k} \times B_{k1} + \beta \times C_{2k}\end{bmatrix} & \begin{bmatrix}\alpha \times A_{21} \times B_{22} + \beta \times C_{21}\\ + \\ \alpha \times A_{22} \times B_{22} + \beta \times C_{22}\\ +\cdots+ \\ \alpha \times A_{2k} \times B_{k2} + \beta \times C_{2k}\end{bmatrix} & \dots & \begin{bmatrix}\alpha \times A_{2k} \times B_{kn} + \beta \times C_{2k}\\ + \\ \alpha \times A_{2k} \times B_{kn} + \beta \times C_{2k}\\ +\cdots+ \\ \alpha \times A_{2k} \times B_{kn} + \beta \times C_{2k}\end{bmatrix}\\
\vdots & & & \vdots\\
\begin{bmatrix}\alpha \times A_{m1} \times B_{1n} + \beta \times C_{m1}\\ + \\ \alpha \times A_{m2} \times B_{2n} + \beta \times C_{m2}\\ +\cdots+ \\ \alpha \times A_{mk} \times B_{kn} + \beta \times C_{mk}\end{bmatrix} & \begin{bmatrix}\alpha \times A_{m1} \times B_{2n} + \beta \times C_{m1}\\ + \\ \alpha \times A_{m2} \times B_{2n} + \beta \times C_{m2}\\ +\cdots+ \\ \alpha \times A_{mk} \times B_{kn} + \beta \times C_{mk}\end{bmatrix} & \dots & \begin{bmatrix}\alpha \times A_{mk} \times B_{kn} + \beta \times C_{mk}\\ + \\ \alpha \times A_{mk} \times B_{kn} + \beta \times C_{mk}\\ +\cdots+ \\ \alpha \times A_{mk} \times B_{kn} + \beta \times C_{mk}\end{bmatrix}
\end{pmatrix}.$$

Por ejemplo, si

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$
$$\mathbf{B} = \begin{pmatrix} 5 & 6 & 7 \\ 1 & 2 & 3 \\ 8 & 9 & 4 \end{pmatrix},$$
$$\mathbf{C} = \begin{pmatrix} 7 & 8 & 9 \\ 5 & 6 & 4 \\ 1 & 2 & 3 \end{pmatrix},$$
$$x = (2, 4, 6),$$
$$y = (1, 3, 5),$$
$$\alpha = 9,$$
$$\beta = 6,$$

entonces

$$\alpha \times A \times B + \beta \times C = \begin{pmatrix} \begin{bmatrix} 9\times1\times5+6\times7 \\ + \\ 9\times2\times1+6\times8 \\ + \\ 9\times3\times8+6\times9 \end{bmatrix} & \begin{bmatrix} 9\times1\times6+6\times7 \\ + \\ 9\times2\times2+6\times8 \\ + \\ 9\times3\times9+6\times9 \end{bmatrix} & \begin{bmatrix} 9\times1\times7+6\times7 \\ + \\ 9\times2\times2+6\times8 \\ + \\ 9\times3\times4+6\times9 \end{bmatrix} \\ \begin{bmatrix} 9\times4\times5+6\times5 \\ + \\ 9\times5\times1+6\times6 \\ + \\ 9\times6\times8+6\times4 \end{bmatrix} & \begin{bmatrix} 9\times4\times6+6\times5 \\ + \\ 9\times5\times2+6\times6 \\ + \\ 9\times6\times9+6\times4 \end{bmatrix} & \begin{bmatrix} 9\times4\times7+6\times5 \\ + \\ 9\times5\times2+6\times6 \\ + \\ 9\times6\times4+6\times4 \end{bmatrix} \\ \begin{bmatrix} 9\times7\times5+6\times1 \\ + \\ 9\times8\times1+6\times2 \\ + \\ 9\times9\times8+6\times3 \end{bmatrix} & \begin{bmatrix} 9\times7\times6+6\times1 \\ + \\ 9\times8\times2+6\times2 \\ + \\ 9\times9\times9+6\times3 \end{bmatrix} & \begin{bmatrix} 9\times7\times7+6\times1 \\ + \\ 9\times8\times2+6\times2 \\ + \\ 9\times9\times4+6\times3 \end{bmatrix} \end{pmatrix}$$
$$= \begin{pmatrix} 87+66+279 & 96+84+297 & 105+84+162 \\ 210+81+456 & 246+126+510 & 282+126+240 \\ 321+84+666 & 384+156+729 & 447+156+342 \end{pmatrix}$$
$$= \begin{pmatrix} 432 & 477 & 351 \\ 747 & 882 & 548 \\ 1071 & 1269 & 945 \end{pmatrix}$$

4.2. Problemas representativos a resolver con las bibliotecas LAPACK y CULA

Aquí se presentan algunos de los problemas que son capaces de resolver estas bibliotecas LAPACK y CULA (estas bibliotecas se abordan en secciones posteriores de este capítulo):

- **Resolución de sistemas de ecuaciones lineales** de la forma $Ax = b$.
  En matemáticas y álgebra lineal un sistema lineal de ecuaciones es un conjunto de ecuaciones lineales de la forma $Ax = b$, donde A es la matriz de coeficientes, b es el vector de los términos independientes y x es el vector solución, se debe encontrar x dados A y b. Por ejemplo el siguiente es un sistema de ecuaciones lineales: con *n* ecuaciones y *n* incógnitas [16]:

$$\begin{array}{lcccccccc} E_1 & : & a_{11}x_1 & + & a_{12}x_2 & +\cdots+ & a_{13}x_3 & = & b_1, \\ E_2 & : & a_{21}x_1 & + & a_{22}x_2 & +\cdots+ & a_{23}x_3 & = & b_2, \\ & & \vdots & & \vdots & & \vdots & & \vdots \\ E_n & : & a_{n1}x_1 & + & a_{n2}x_2 & +\cdots+ & a_{nn}x_n & = & b_n, \end{array}$$

que de la forma matricial se escribe como $Ax = b$, donde:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

El objetivo es encontrar los valores desconocidos $x_1 x_2 \dots x_n$ que satisface las n ecuaciones.

Si en el problema a resolver hay varios lados derechos (b con más de una columna) el sistema de ecuaciones lineales se escribe:

$$AX = B$$

donde las columnas de B son los lados derechos (individuales) y las columnas de X son las soluciones correspondientes.

- **Descomposición de una matriz en sus valores singulares** de la forma $A = U\Sigma V^T$.
  La factorización de una matriz A es la representación de A como el producto de varias matrices.

  Sea A una matriz de tamaño $m \times n$. La descomposición en valores singulares (SVD por sus siglas en Inglés) de A es la factorización $A = U\Sigma V^T$, donde U y V son matrices ortogonales de tamaño $m \times n$ y $n \times n$ respectivamente; y $\Sigma = diag(\sigma_1, \dots, \sigma_n)$ con $\sigma_1 \geq \dots \sigma_n \geq 0$. Los σ_i son los valores singulares de A. Las primeras n columnas de V son los vectores singulares derechos y las primeras n columnas de U son los vectores singulares izquierdos de A [17].

  - Matriz diagonal
    Una matriz diagonal es una matriz cuadrada que tiene 0 en todos sus componentes a excepción de la diagonal principal. Por ejemplo:

    $$\begin{pmatrix} 5 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 9 \end{pmatrix}$$

  - Matriz ortogonal
    Una matriz cuadrada se dice que es ortogonal si cumple que:

    $$A \times A^T = I,$$

    donde A es la matriz cuadrada, A^T es la transpuesta de A e I es la matriz identidad.
    La matriz identidad es en la que todos los elementos de la diagonal principal tienen 1 y en el resto tienen 0. Por ejemplo:

    $$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

En [16] se pueden encontrar diferentes métodos para resolver un sistema de ecuaciones lineales; por otra parte [17] se puede encontrar información acerca de cómo descomponer una matriz en sus valores singulares.

4.3. Norma vectorial y matricial

Las normas tienen el mismo propósito en los espacios vectoriales que el valor absoluto en la recta numérica: proporcionar una medida de distancia. Más específicamente, $\mathbb{R}^n$ junto con una norma en $\mathbb{R}^n$ define una métrica [17].

En la notación de las normas se usa la doble barra $||$ para no confundir con el valor absoluto que se indica con una barra $|$. Dentro de las normas vectoriales, algunas de las más importantes son las siguientes:

- 1-norma
  $||x||_1 \quad = \quad |x_1| + \dots + |x_n|$
- 2-norma
  $||x||_2 \quad = \quad \sqrt{|x_1|^2 + \dots + |x_n|^2}$
- ∞-norma (norma infinito)
  $||x||_\infty \quad = \quad \max\limits_{1 \leq i \leq n} |x_i|$

El análisis de algoritmos que involucran matrices frecuentemente requiere el uso de normas matriciales para evaluar la calidad del resultado (precisión). Por ejemplo al considerar la calidad en la solución de un sistema de ecuaciones lineales. Entonces para evaluar esto es necesario medir la distancia entre las matrices (error de precisión). Las normas matriciales proporcionan esta medida.

Aunque existen muchas normas matriciales, en este documento se usa únicamente la norma *Frobenius*

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

4.4. Error de precisión

Sea $\hat{x} \in \mathbb{R}^n$ una aproximación a $x \in \mathbb{R}^n$. Para una norma vectorial dada $||\cdot||$ se dice que [17]:

$$\varepsilon_{abs} = ||\hat{x} - x||$$

es el *error absoluto* en $\hat{x}$. Si $x \neq 0$, entonces

$$\varepsilon_{rel} = \frac{||\hat{x} - x||}{||x||}$$

establece el *error relativo* en $\hat{x}$. El error relativo en la $\infty - norma$ puede ser traducida como una declaración sobre el número de cifras significativas en $\hat{x}$. En particular, si

$$\frac{||\hat{x} - x||_\infty}{||x||_\infty} \approx 10^{-p}$$

se consideran p cifras significativas de $\hat{x}$.

4.5. BLAS

En esta sección se introduce la biblioteca BLAS y cómo se realizan con sus rutinas las operaciones vistas en secciones anteriores.

4.5.1. Definición

BLAS (Basic Linear Algebra Subprograms)[3] es un conjunto de rutinas computacionales escritas en Fortran 77 para operaciones básicas del Álgebra lineal, su primera publicación fue en 1979.

Por el uso tan extendido del lenguaje de programación C, se crearon versiones de todas las rutinas BLAS para dar soporte a este lenguaje (CBLAS fue escrita por Keita Teanishi en 1998), una diferencia entre Fortran y C en cuanto a la forma en que éstos asignan memoria para matrices es la siguiente, Fortran almacena matrices por columnas y C las almacena por filas. Esto se ilustra con un ejemplo:

Sea **A** la matriz:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

entonces:

(**Column-major-oder**, Orden columna-principal) Fortran asigna memoria por columnas, la matriz **A** se almacena de la siguiente forma en direcciones de memoria contiguas:

$$1 \quad 4 \quad 7 \quad 2 \quad 5 \quad 8 \quad 3 \quad 6 \quad 9.$$

(**Row-major-order**, Orden fila-principal) C asigna memoria por fila, la matriz **A** se almacena de la siguiente forma en direcciones de memoria contiguas:

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9.$$

BLAS se emplea a menudo para resolver problemas de áreas como:

- Problemas de la ingeniería.
- Problemas de ciencias de la computación.
- Tratamiento de señales.

- Simulación en ciencias de materiales.
- Minería de datos.
- Dinámica de fluidos.

BLAS se ha convertido en la interfaz de facto para construir bibliotecas robustas como LAPACK (Linear Algebra PACKage), biblioteca que se abordará en la sección 4.7.

La estructura del nombre de las rutinas BLAS es la siguiente.

1. $\overbrace{\textbf{s}}^{prefijo}\ \underbrace{\textbf{dot}}_{operación}$

   sdot Calcula el producto interno de dos vectores de precisión simple.

2. $\overbrace{\textbf{d}}^{prefijo}\ \underbrace{\textbf{ge}}_{tipo\ de\ matriz}\ \overbrace{\textbf{mv}}^{operación}$

   dgemv Realiza el producto matriz-vector de precisión doble, operando con una matriz general.

3. $\overbrace{\textbf{z}}^{prefijo}\ \underbrace{\textbf{sy}}_{tipo\ de\ matriz}\ \overbrace{\textbf{mm}}^{operación}$

   ztrmv Calcula el producto matriz-matriz de precisión doble compleja, operando con matrices simétricas.

Donde *prefijo* se refiere al tipo de datos del vector o matriz con los que opera; sus posibles valores están en la tabla 1.

Tabla 1: Tipos de precisión BLAS

Prefijo	**Precisión**
s	Simple
d	Doble
c	Compleja simple
z	Compleja doble

Tipo de matriz se refiere al tipo de matriz sobre la que opera la rutina; sus posibles valores están en la tabla 2

Tabla 2: Tipos de matrices

Tipo	**Descripción**	**Tipo**	**Descripción**
bd	Bidiagonal	po	Simétrica o Hermitiana definida positiva
di	Diagonal	pp	Simétrica o Hermitiana definida positiva (compacto)
gb	General definida por bandas	pt	Simétrica o Hermitiana definida positiva tridiagonal
ge	General (puede ser rectangular)	sb	Simétrica (real) definida por bandas
gg	Matriz general para un problema generalizado	sp	Simétrica con almacenamiento compacto
gt	General tridiagonal	st	Simétrica tridiagonal
hb	Hermitiana (compleja) definida por bandas	sy	Simétrica
he	Hermitiana compleja	tb	Triangular definida por bandas
hg	Matriz de Hessenberg superior (problema generalizado)	tg	Triangular de un problema generalizado
hp	Hermitiana (compleja) con almacenamiento compacto	tp	Triangular con almacenamiento compacto
hs	Matriz de Hessenberg superior	tr	Triangular
op	Ortogonal (real) con almacenamiento compacto	tz	Trapezoidal
or	Ortogonal real	un	Compleja unitaria
pb	Simétrica o Hermitiana definida positiva (por bandas)	up	Compleja unitaria con almacenamiento compacto

Y *operación* se refiere al cómputo que realiza la rutina, así:

- Las rutinas que terminan en **mv**, realizan operaciones matriz-vector.
- Las rutinas que terminan en **mm**, realizan operaciones matriz-matriz.

- Las rutinas que terminan con un término en Inglés, realizan dicha operación (scal-escalamiento, dot-producto, swap-intercambio, . . .).

Las rutinas BLAS se dividen en tres niveles que son:

- **Nivel 1.** En este nivel se realizan operaciones *vector-vector*. Algunas rutinas de este nivel se muestran en la tabla 3.
- **Nivel 2.** En este nivel se realizan operaciones *matriz-vector*. Algunas rutinas de este nivel se muestran en la tabla 4.
- **Nivel 3.** En este nivel se realizan operaciones *matriz-matriz*. Algunas rutinas de este nivel se muestran en la tabla 5.

Tabla 3: Rutinas BLAS nivel 1

Nombre	Operación	Prefijo #
#swap	$x \leftrightarrow y$	s, d, c, z
#scal	$x \leftarrow \alpha x$	s, d, c, z
#copy	$y \leftarrow x$	s, d, c, z
#axpy	$y \leftarrow \alpha x + y$	s, d, c, z
#dot	$dot \leftarrow xy$	s, d

Tabla 4: Rutinas BLAS nivel 2

Nombre	Operación	Prefijo #
#gemv	$y \leftarrow \alpha Ax + \beta y$	s, d, c, z
#gbmv	$y \leftarrow \alpha Ax + \beta y$	s, d, c, z
#hemv	$y \leftarrow \alpha Ax + \beta y$	c, z
#hbmv	$y \leftarrow \alpha Ax + \beta y$	c, z
#trmv	$x \leftarrow Ax$	s, d, c, z

Tabla 5: Rutinas BLAS nivel 3

Nombre	Operación	Prefijo #
#gemm	$C \leftarrow \alpha AB + \beta C$	s, d, c, z
#symm	$C \leftarrow \alpha AB + \beta C$	s, d, c, z
#syrk	$C \leftarrow \alpha AA^T + \beta C$	s, d, c, z
#syr2k	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$	s, d, c, z
#trmm	$B \leftarrow \alpha AB$	s, d, c, z

4.5.2. Operaciones básicas de matrices y vectores

En este apartado se muestran algunas de las rutinas de BLAS con las que se pueden hacer las operaciones básicas de vectores y matrices vistas en la sección 4.1. Estas rutinas se presentan de acuerdo a la interfaz de C de BLAS (CBLAS) y soportan todos los tipos de datos (ver tabla 1). En los ejemplos dados se utilizará la precisión simple **s**.

- **scal**, realiza el escalamiento de un vector. El prototipo de la rutina **cblas_sscal** es:

```
cblas_sscal(const int N, const float alpha, float *X, const int incX);
```

Entrada

N	es el número de elementos del vector de entrada.
alpha	es un escalar de precisión simple.
X	es un vector con n elementos de precisión simple.
incX	es el incremento entre los elementos de x.

Salida

x	$x \leftarrow \alpha x$

- **axpy**, realiza la suma vector-vector. El prototipo de la rutina **cblas_saxpy** es:

```
void cblas_saxpy(const int N, const float alpha, const float *X,
                 const int incX, float *Y, const int incY)
```

Entrada

N	es el número de elementos de los vectores de entrada.
alpha	es un escalar de precisión simple.
X	es un vector con n elementos de precisión simple.
incX	es el incremento entre los elementos de x.
Y	es un vector con n elementos de precisión simple.
incY	es el incremento entre los elementos de y.

Salida

y	$y \leftarrow \alpha x + y$

- **dot**, realiza el producto vector-vector. El prototipo de la rutina **cblas_sdot** es:

```
float cblas_sdot(const int N, const float *X, const int incX,
                 const float *Y, const int incY)
```

Entrada

N	es el número de elementos de los vectores de entrada.
X	es un vector con n elementos de precisión simple.
incX	es el incremento entre los elementos de x.
Y	es un vector con n elementos de precisión simple.
incY	es el incremento entre los elementos de y.

Salida

dot	$dot \leftarrow xy$

- **gemv**, realiza el producto matriz-vector. El prototipo de la rutina **cblas_sgemv** es:

```
void cblas_sgemv(const CBLAS_ORDER order, const CBLAS_TRANSPOSE TransA,
                 const int M, const int N, const float alpha,
                 const float  *A, const int lda, const float  *X,
                 const int incX, const float beta, float  *Y, const int incY)
```

Entrada

order	determina cómo será la matriz almacenada en memoria (ver sección 4.5.1).
TransA	especifica la operación que será realizada: Si TransA=='N' o 'n' se opera con A. Si TransA=='T' o 't' o 'C' o 'c' se opera con A^T.
M	es el número de filas de la matriz A.
N	es le número de columnas de la matriz A.
alpha	es una escalar de precisión simple.
A	es una matriz de precisión simple de dimensión (lda, m).
lda	dimensión de la matriz A.
X	es un vector con n elementos de precisión simple.
incX	es el incremento entre los elementos de x.
beta	es una escalar de precisión simple.
Y	es un vector con n elementos de precisión simple.
incy	es el incremento entre los elementos de y.

Salida

y	$y = \alpha \times A \times x + \beta \times y$

- **Producto matriz-matriz**
- **gemm**, realiza el producto matriz-matriz. El prototipo de la rutina **cblas_sgemm** es:

```
void cblas_sgemm(const CBLAS_ORDER Order, const CBLAS_TRANSPOSE TransA,
                 const CBLAS_TRANSPOSE TransB, const int M, const int N,
                 const int K, const void *alpha, const void  *A,
                 const int lda, const void  *B, const int ldb,
                 const void *beta, void  *C, const int ldc)
```

Entrada

order	determina cómo será la matriz almacenada en memoria (ver sección 4.5.1).
TransA	especifica la operación que será realizada: Si TransA==’N’ o ’n’ se opera con A. Si TransA==’T’ o ’t’ o ’C’ o ’c’ se opera con A^T.
TransB	especifica la operación que será realizada: Si TransB==’N’ o ’n’ se opera con B. Si TransB==’T’ o ’t’ o ’C’ o ’c’ se opera con B^T.
M	es el número de filas de la matriz A y el número de filas de la matriz C.
N	es el número de columnas de la matriz B y el número columnas de la matriz C.
K	es el número de columnas de la matriz A y el número de filas de la matriz B
alpha	es un escalar de precisión simple.
A	es una matriz de precisión simple de dimensión (lda, K) si TransA==’N’ o ’n’ y (lda, M) de otra manera.
lda	dimensión de la matriz A.
B	es una matriz de precisión simple de dimensión (lda, N) si TransB==’N’ o ’n’ y (ldb, K) de otra manera.
ldb	dimensión de la matriz B.
beta	es un escalar de precisión simple.
C	es una matriz de precisión simple de dimensión (lda, N).
ldc	dimensión que será usada para almacenar la matriz C.

Salida

C	$C = \alpha \times A \times B + \beta \times C$

4.6. CUBLAS

En esta sección se introduce la biblioteca paralela CUBLAS y cómo se realizan con sus rutinas las operaciones vistas en secciones anteriores.

4.6.1. Definición

CUBLAS [11] es una implementación de BLAS (Basic Linear Algebra Subprograms) para la plataforma NVIDIA-CUDA, que permite el acceso a los recursos de los dispositivos NVIDIA, las GPUs (Graphic Processing Unit). Esta biblioteca es independiente a nivel de API (Application Programming Interface) de esta manera CUBLAS no tiene que interactuar directamente con el driver CUDA.

El modelo básico de las aplicaciones CUBLAS consiste en:

- Crear matrices y vectores en memoria de CPU.
- Llenarlos con datos.
- Transferir los datos desde memoria de CPU a memoria de dispositivo (GPU).
- Invocar funciones CUBLAS para realizar operaciones.
- Transferir los resultados desde memoria de GPU a memoria de host (CPU).

CUBLAS puede ser usado desde Fortran y C/C++. Para máxima compatibilidad con el software de Álgebra Lineal existente (generalmente escrito en Fortran) CUBLAS usa el almacenamiento por columnas para las matrices y 1 como primer índice de los arreglos. Con C/C++ CUBLAS usa el almacenamiento por filas para las matrices (ver sección 4.5.1) y 0 como primer índice de los arreglos, para la declaración de arreglos **multi-dimensionales** no se puede usar la declaración estándar de C/C++, entonces las matrices deben ser implementadas como arreglos **unidimensionales**, esto implica crear macros o funciones capaces de leer correctamente los datos de las matrices. Para trabajar con CUBLAS se debe invocar **cublasInit** antes que cualquier otra función CUBLAS (ver sección 4.6.3).

La estructura del nombre de las rutinas CUBLAS es similar a la de las rutinas BLAS, con las siguientes diferencias:

- Prefijo **cublas**
- Prefijo del tipo de dato (ver tabla 1) en mayúscula.

Ejemplo:

1. $\underbrace{\textbf{cublas}}_{cublas}\ \overbrace{\textbf{S}}^{prefijo}\ \underbrace{\textbf{asum}}_{operación}$
   cublasSasum Calcula la suma de los valores absolutos de los elementos de un vector de precisión simple.
2. $\underbrace{\textbf{cublas}}_{cublas}\ \overbrace{\textbf{C}}^{prefijo}\ \underbrace{\textbf{tp}}_{tipo\ de\ matriz}\ \overbrace{\textbf{sv}}^{operación}$
   cublasCtpsv Resuelve un sistema de ecuaciones de la forma $Ax = b$. Con una matriz triangular con almacenamiento compacto (ver tabla 2) y un vector con elementos de precisión simple compleja.

3. $\underbrace{\textbf{cublas}}_{cublas}\ \overbrace{\textbf{Z}}^{prefijo}\ \underbrace{\textbf{ge}}_{tipo\ de\ matriz}\ \overbrace{\textbf{mm}}^{operación}$

cublasZgemm Realiza la multiplicación matriz matriz con elementos de precisión doble compleja operando con matrices generales (ver tabla 2).

Dado que CUBLAS es una implementación de BLAS, también se divide en tres niveles:

- **Nivel 1.** En este nivel se realizan operaciones *vector-vector*. Algunas rutinas de este nivel se muestran en la tabla 6.
- **Nivel 2.** En este nivel se realizan operaciones *matriz-vector*. Algunas rutinas de este nivel se muestran en la tabla 7.
- **Nivel 3.** En este nivel se realizan operaciones *matriz-matriz*. Algunas rutinas de este nivel se muestran en la tabla 8.

Tabla 6: Rutinas CUBLAS nivel 1

Nombre	Operación	Prefijo #
cublas#swap	$x \leftrightarrow y$	S, C, D, Z
cublas#scal	$x \leftarrow \alpha x$	S, C, D, Z
cublas#copy	$y \leftarrow x$	S, D, Z
cublas#axpy	$y \leftarrow \alpha x + y$	S, D, C, Z
cublas#dot	$dot \leftarrow xy$	S, D

Tabla 7: Rutinas CUBLAS nivel 2

Nombre	Operación	Prefijo #
cublas#gemv	$y \leftarrow \alpha Ax + \beta y$	S, D, C, Z
cublas#gbmv	$y \leftarrow \alpha Ax + \beta y$	S, D, C, Z
cublas#hemv	$y \leftarrow \alpha Ax + \beta y$	S, D, C, Z
cublas#hbmv	$y \leftarrow \alpha Ax + \beta y$	C, Z
cublas#trmv	$x \leftarrow Ax$	S, D, C, Z

Tabla 8: Rutinas CUBLAS nivel 3

Nombre	Operación	Prefijo #
cublas#gemm	$C \leftarrow \alpha AB + \beta C$	S, D, C, Z
cublas#symm	$C \leftarrow \alpha AB + \beta C$	S, D, C, Z
cublas#syrk	$C \leftarrow \alpha AA^T + \beta C$	S, D, C, Z
cublas#syr2k	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$	S, D, C, Z
cublas#trmm	$B \leftarrow \alpha AB$	S, D, C, Z

4.6.2. Tipos CUBLAS

cublasStatus es el único tipo CUBLAS. cublasStatus es usado para retornar el estado de una función. Las funciones auxiliares CUBLAS devuelven su estado directamente, mientras que el estado de las funciones básicas de CUBLAS puede ser recuperado a través de **cublasGetError()**. Actualmente están definidos los estados de la tabla 9

Tabla 9: Tipos cublasStatus

CUBLAS_STATUS_SUCCESS	Operación completada correctamente.
CUBLAS_STATUS_NOT_INITIALIZED	CUBLAS no se ha inicializado.
CUBLAS_STATUS_ALLOC_FAILED	Asignación de memoria fallida.
CUBLAS_STATUS_INVALID_VALUE	Valor numérico no soportado por la función.
CUBLAS_STATUS_ARCH_MISMATCH	La función requiere una característica ausente en la arquitectura de la tarjeta gráfica.
CUBLAS_STATUS_MAPPING_ERROR	Error al acceder a memoria de GPU.
CUBLAS_STATUS_EXECUTION_FAILED	Programa de GPU no se pudo ejecutar.
CUBLAS_STATUS_INTERNAL_ERROR	Error interno, operación fallida.

4.6.3. Funciones auxiliares CUBLAS

Función cublasInit()

```
   cublasStatus
cublasInit()
```

Inicializa la biblioteca CUBLAS, debe ser invocada antes que cualquier otra función CUBLAS.

Valores de retorno	
CUBLAS_STATUS_ALLOC_FAILED	Si no se pudieron reservar recursos
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

Función cublasShutdown()

```
   cublasStatus
cublasShutdown()
```

Libera los recursos que fueron usados en CUBLAS y en el host (CPU), finaliza la aplicación.

Valores de retorno	
CUBLAS_STATUS_NOT_INITIALIZED	Si la biblioteca CUBLAS no fue inicializada
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

Función cublasGetError()

```
   cublasStatus
cublasGetError()
```

Retorna el último error ocurrido en la invocación de cualquier función básica CUBLAS, mientras que las rutinas auxiliares retornan su estado directamente. Lee el error vía **cublasGetError** y restablece el estado interno con **CUBLAS_STATUS_SUCCESS**

Función cublasAlloc()

```
   cublasStatus
cublasAlloc (int n, int elemSize, void **devicePtr)
```

Reserva memoria en GPU para un arreglo de **n** elementos, donde cada elemento necesita **elemSize** bytes para ser almacenado. Si la función es invocada correctamente, el apuntador en GPU se posiciona en **devicePtr**. Nota: este es un apuntador creado en GPU por lo que no puede ser referenciado desde el host (CPU). La función **cublasAlloc** es un envoltorio[3] de **cudaMalloc()**[4]. Los apuntadores retornados por **cublasAlloc()** pueden ser pasados a cualquier kernel CUDA, no solo a las funciones CUBLAS.

Valores de retorno	
CUBLAS_STATUS_NOT_INITIALIZED	Si la biblioteca CUBLAS no fue inicializada.
CUBLAS_STATUS_INVALID_VALUE	Si n <= 0 o elemSize <= 0
CUBLAS_STATUS_ALLOC_FAILED	Si el objeto no pudo ser reservado en memoria
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

Función cublasFree()

```
   cublasStatus
cublasFree (const void *devicePtr)
```

Destruye un objeto en memoria GPU referenciado por **devicePtr**.

Valores de retorno	
CUBLAS_STATUS_NOT_INITIALIZED	Si la biblioteca CUBLAS no fue inicializada.
CUBLAS_STATUS_INTERNAL_ERROR	Si el objeto no se pudo destruir.
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

Función cublasSetVector()

```
   cublasStatus
cublasSetVector (int n, int elemSize, const void *x, int incx, void *y, int incy)
```

[3] Envoltorio es una función que desde dentro de ella invoca a otra.

[4] **cudaMalloc(void ** devPtr, size_t size)** reserva **size** bytes para un apuntador **devPtr** en memoria en dispositivo (GPU).

Copia un vector **x** de **n** elementos creado en host (CPU) a un vector **y** en GPU. Se asume que cada uno de los elementos de ambos vectores requiere **elemSize** bytes para ser almacenado. El espacio de almacenamiento entre los elementos del vector fuente **x** es **incx** e **incy** para el vector destino **y**. En general **y** apunta a un objeto reservado vía **cublasAlloc()**.

Valores de retorno	
CUBLAS_STATUS_NOT_INITIALIZED	Si la biblioteca CUBLAS no fue inicializada.
CUBLAS_STATUS_INVALID_VALUE	Si incx, incy o elemSize $<= 0$
CUBLAS_STATUS_MAPPING_ERROR	Error al acceder a la memoria de la GPU
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

Función cublasGetVector

```
cublasStatus
cublasGetVector (int n, int elemSize, const void *x, int incx, void *y, int incy)
```

Copia un vector **x** de **n** elementos desde memoria de la GPU a un vector **y** en memoria de host (CPU). Se asume que cada uno de los elementos de ambos vectores requiere **elemSize** bytes para ser almacenado. El espacio de almacenamiento entre los elementos del vector fuente **x** es **incx** e **incy** para el vector destino **y**. En general **x** apunta a un objeto reservado vía **cublasAlloc()**.

Valores de retorno	
CUBLAS_STATUS_NOT_INITIALIZED	Si la biblioteca CUBLAS no fue inicializada.
CUBLAS_STATUS_INVALID_VALUE	Si incx, incy o elemSize $<= 0$
CUBLAS_STATUS_MAPPING_ERROR	Error al acceder a la memoria de la GPU
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

Función cublasSetMatrix

```
cublasStatus
cublasSetMatrix (int rows, int cols, int elemSize, const void *A,
int lda, void *B, int ldb)
```

Copia una matriz **A** de **rows**$\times$**cols** elementos creado en host (CPU) a una matriz **B** en memoria de GPU. Cada elemento necesita **elemSize** bytes. Se asume que cada matriz es almacenada por columnas, con una dimensión principal (rows) de la matriz fuente **A** indicada en **lda** y la dimensión principal de la matriz **B** indicada en **ldb**. **B** es un apuntador en GPU y debe ser reservado vía **cublasAlloc()**.

Valores de retorno	
CUBLAS_STATUS_NOT_INITIALIZED	Si la biblioteca CUBLAS no fue inicializada.
CUBLAS_STATUS_INVALID_VALUE	Si row o col < 0; o elemSize, lda o ldb $<= 0$
CUBLAS_STATUS_MAPPING_ERROR	Error al acceder a la memoria de la GPU
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

Función cublasGetMatrix

```
cublasStatus
cublasGetMatrix (int rows, int cols, int elemSize, const void *A,
int lda, void *B, int ldb)
```

Copia una matriz **A** de **rows**$\times$**cols** desde mamoria GPU a una matriz **B** en memoria de host (CPU). Cada elemento requiere **elemSize** bytes. Se asume que cada matriz es almacenada por columnas, con una dimensión principal de la matriz **A** indicada en **lda**, y la dimensión principal de **B** indicada por **ldb**. **A** es un apuntador reservado vía **cublasAlloc()**.

Valores de retorno	
CUBLAS_STATUS_NOT_INITIALIZED	Si la biblioteca CUBLAS no fue inicializada.
CUBLAS_STATUS_INVALID_VALUE	Si row o col < 0; o elemSize, lda o ldb $<= 0$
CUBLAS_STATUS_MAPPING_ERROR	Error al acceder a la memoria de la GPU
CUBLAS_STATUS_SUCCESS	Si la aplicación finalizó correctamente.

4.6.4. Operaciones básicas de vectores y matrices

En este apartado se muestran algunas de las rutinas de CUBLAS con las que se pueden hacer las operaciones básicas de vectores y matrices vistas en la sección 4.1. En los ejemplos dados se utilizará la precisión simple **s**.

- **scal**, realiza el escalamiento de un vector. El prototipo de la rutina **cublasSscal** es:

```
void cublasSscal (int n, float alpha, float *x, int incx)
```

Entrada

n	es el número de elementos del vector de entrada.
alpha	es un escalar de precisión simple.
x	es un vector con n elementos de precisión simple.
incx	es el incremento entre los elementos de x.

Salida

x	$x \leftarrow \alpha x$

- **axpy**, realiza la suma vector-vector. El prototipo de la rutina **cublasSaxpy** es:

```
void cublasSaxpy (int n, float alpha, const float *x, int incx, float *y, int incy)
```

Entrada

n	es el número de elementos de los vectores de entrada.
alpha	es un escalar de precisión simple.
x	es un vector con n elementos de precisión simple.
incx	es el incremento entre los elementos de x.
y	es un vector con n elementos de precisión simple.
incy	es el incremento entre los elementos de y.

Salida

y	$y \leftarrow \alpha x + y$

- **dot**, realiza el producto vector-vector. El prototipo de la rutina **cublasSdot** es:

```
float cublasSdot (int n, const float *x, int incx, const float *y, int incy)
```

Entrada

n	es el número de elementos de los vectores de entrada.
x	es un vector con n elementos de precisión simple.
incx	es el incremento entre los elementos de x.
y	es un vector con n elementos de precisión simple.
incy	es el incremento entre los elementos de y.

Salida

dot	$dot \leftarrow xy$

- **gemv**, realiza el producto matriz-vector. El prototipo de la rutina **cublasSgemv** es:

```
void cublasSgemv(char trans, int m, int n, float alpha,
                 const float *A, int lda, const float *x,
                 int incx, float beta, float *y, int incy)
```

Entrada

trans	especifica la operación que será realizada: Si trans==’N’ o ’n’ se opera con A Si trans==’T’ o ’t’ o ’C’ o ’c’ se opera con A^T.
m	es el número de filas de la matriz A.
n	es le número de filas de la matriz A; m
alpha	es una escalar de precisión simple.
A	es una matriz de precisión simple de dimensión (lda, n). Si trans==’N’ o ’n’ y (lda, m) de otra manera.
lda	dimensión de la matriz A.
x	es un vector con n elementos de precisión simple.
incx	es el incremento entre los elementos de x.
beta	es una escalar de precisión simple.
y	es un vector con n elementos de precisión simple.
incy	es el incremento entre los elementos de y.

Salida

y	$y = \alpha \times A \times x + \beta \times y$

- **gemm**, realiza el producto matriz-matriz. El prototipo de la rutina **cublasSgemm** es:

```
void cublasSsgemm(char transa, char transb, int m, int n, int k,
                  float alpha, const float *A, int lda, const float *B,
                  int ldb, float beta, float *C, int ldc)
```

Entrada

transA	especifica la operación que será realizada: Si transA=='N' o 'n' se opera con A. Si transA=='T' o 't' o 'C' o 'c' se opera con A^T.
transB	especifica la operación que será realizada: Si transB=='N' o 'n' se opera con B. Si transB=='T' o 't' o 'C' o 'c' se opera con B^T.
m	es el número de filas de la matriz A y el número de filas de la matriz C.
n	es el número de columnas de la matriz B y el número columnas de la matriz C.
k	es el número de columnas de la matriz A y el número de filas de la matriz B
alpha	es un escalar de precisión simple.
A	es una matriz de precisión simple de dimensión (lda, K) si TransA=='N' o 'n' y (lda, m) de otra manera.
lda	dimensión de la matriz A.
B	es una matriz de precisión simple de dimensión (lda, N) si TransB=='N' o 'n' y (ldb, K) de otra manera.
ldb	dimensión de la matriz B.
beta	es un escalar de precisión simple.
C	es una matriz de precisión simple de dimensión (lda, N.
ldc	dimensión que será usada para almacenar la matriz C.

Salida

C	$C = \alpha \times A \times B + \beta \times C$

4.7. LAPACK

En esta sección se introduce la biblioteca LAPACK y cómo se realizan con sus rutinas la resolución de sistemas de ecuaciones lineales y la descomposición en valores singulares de matrices.

4.7.1. Definición

LAPACK [15] es un conjunto de rutinas computacionales escritas en Fortran 77, es capaz de resolver problemas básicos del Álgebra Lineal Numérica. Fue diseñado para funcionar eficientemente en la mayoría de las computadoras de alto rendimiento. LAPACK es acrónimo de Linear Algebra PACKage.

LAPACK puede resolver sistemas de ecuaciones lineales, problemas de mínimos cuadrados, problemas de valores propios y problemas de descomposición de valores singulares. LAPACK también puede realizar cálculos asociados a los anteriores, como: factorización de matrices y números de condición.

LAPACK se integra por tres tipos de rutinas que son las siguientes:

Rutinas de núcleo para resolver los problemas estándar.

Rutinas computacionales para resolver un gran número de tareas.

Rutinas auxiliares realizan algunas tareas de bajo nivel.

Típicamente cada rutina de núcleo hace una serie de llamados a rutinas computacionales, en su conjunto las rutinas computacionales pueden realizar un amplio rango de las operaciones cubiertas por las rutinas de núcleo. Muchas de las rutinas auxiliares pueden ser usadas para el análisis numérico o desarrollo de software, para esto se cuenta con documentación de código Fortran para estas rutinas, con el mismo nivel de detalle usado para las rutinas de núcleo.

La estructura del nombre de las rutinas LAPACK es muy similar a las de BLAS (ver sección 4.5.1) operando con las mismas precisiones (ver tabla 1) y los mismos tipos de matrices (ver tabla 2).

Otro detalle importante a mencionar es que, considerando que LAPACK fue escrito en Fortran, las matrices son almacenadas por columna (ver sección 4.5.1), esto quiere decir que desde un programa C se deben transponer las matrices antes de ser procesadas por alguna de las muchas rutinas que integran LAPACK.

4.7.2. Resolución de sistemas de ecuaciones lineales de la forma $Ax = b$

LAPACK cuenta con muchas rutinas para resolver sistemas de ecuaciones, una de ellas es **gesv**, dicha función ofrece soporte para todos los tipos de datos (ver tabla 1). En este ejemplo se usará la precisión simple **s**.

sgesv resuelve un sistema de ecuaciones lineales de la forma $AX = B$ (ver sección 4.2).

El prototipo de la rutina **sgesv_** es:

```
void sgesv_(int *n, int *nrhs, float *a, int *lda,
            int *ipiv, float *b, int *ldb, int *info)
```

Entrada

n	número de ecuaciones del sistema.
nrhs	número de columnas de la matriz b.
a	matriz de ecuaciones de (lda, n).
lda	número máximo de filas que se pueden almacenar en la matriz a. $lda \geq n$.
ipiv	arreglo de dimensión n, guarda los índices de las filas que son intercambiadas.
b	matriz de dimensión (ldb, nrhs), que de entrada contiene el lado derecho del problema y a la salida la matriz de soluciones X.
ldb	número máximo de filas que se pueden almacenar en la matriz b. $ldb \geq n$.
info	indica si la rutina tuvo éxito o no. Si `info=0` operación exitosa. Si `info=-i` el argumento i-ésimo pasado a la rutina era inválido.

Salida

X	B se sobrescribe con X.

4.7.3. Descomposición de una matriz en sus valores singulares de la forma $A = U\Sigma V^T$

LAPACK cuenta con una rutina para descomponer una matriz en sus valores singulares **gesvd**, dicha función ofrece soporte para todos los tipos de datos (ver tabla 1), en este ejemplo se usará la precisión simple **s**.

El prototipo de la rutina **sgesvd** es:

```
void sgesvd_(const char *jobu, const char *jobvt, int *m, integer *n,
             float *a, int *lda, float *s, float *u, int *ldu,
             float *vt, int *ldvt, float *work, int *lwork, int *info)
```

Entrada

jobu	determina si se calcula toda o parte de la matriz U. Si `jobu = 'A'` todas las m columnas de U son retornadas en el arreglo U. Si `jobu = 'S'` las primeras min(m, n) columnas de U (vectores singulares izquierdos) son retornados en el arreglo U. Si `jobu = 'O'` las primeras min(m, n) columnas de U (vectores singulares izquierdos) son sobrescritas en A. Si `jobu = 'N'` no se hacen cálculos de vectores singulares izquierdos.
jobvt	determina se se calcula toda o una parte de la matriz VT. Si `jobvt = 'A'` todas las n filas de VT son retornadas en el arreglo VT. Si `jobvt = 'S'` las primeras min(m, n) filas de VT (vectores singulares derechos) son retornados en el arreglo VT. Si `jobvt = 'O'` las primeras min(m, n) filas de VT son sobrescritas en A. Si `jobvt = 'N'` no se hacen cálculos de vectores singulares derechos. jobu y jobvt no pueden ser ambos 'O'
m	número de filas de la matriz A.
n	número de columnas de la matriz A.
a	matriz de ecuaciones de (m, n). Si `jobu = 'O'`, A es sobrescrita con las primeras columnas de U, que son almacenados por columna. Si `jobvt = 'O'`, A es sobrescrita con las primeras filas de la matriz VT, que son almacenados por fila. Si `jobu != 'O'` y `jobvt != 'O'`, el contenido de A es destruido.
lda	dimensión de la matriz a.
s	valores singulares de A.
u	matriz de dimensión (ldu, m) si `jobu = 'A'` o (ldu. min(m,n)) si `jobu = 'S'` o (m, m) si `jobu = 'A'`.
ldu	dimensión de la matriz S.
vt	matriz de dimensión (n , n) si `jobvt = 'A'` o los primeros min(m, n) filas de VT si `jobvt = 'S'`.
ldvt	dimensión de la matriz VT.
work	
lwork	
info	

Salida

`S, U, VT`	$A = U\Sigma V^T$

4.8. CULA

CULA fue liberado por EM Photonics en Agosto del 2009, EM Photonics es una empresa estadounidense en Newark.

4.8.1. Definición

CULA es una implementación de LAPACK para funcionar en plataformas NVIDIA CUDA masivamente paralelas sobre unidades de procesamiento gráfico (GPU). Actualmente CULA cuenta con tres versiones [10] **basic, premium y comercial**, siendo basic la única gratuita, por su carácter de gratuita no cubre toda la gama de rutinas LAPACK, pero sí las suficientes para llevar a buenos términos este trabajo.

CULA, al igual que LAPACK, almacena las matrices por columnas (ver sección 4.5.1). Entonces desde un programa C, antes de ser las matrices procesadas por una rutina CULA, deben ser transpuestas para obtener el resultado deseado.

A pesar de que CULA no fue desarrollado por NVIDIA, se tomó a CUBLAS como modelo para escribir todas las rutinas que integran CULA. Entonces se debe invocar **culaInitialize** antes que a cualquier otra función CULA.

La estructura del nombre de las rutinas CULA, es el siguiente:

1. $\overbrace{\textbf{cula}}^{prefijo}\ \underbrace{\textbf{S}}_{tipo\ de\ dato}\ \overbrace{\textbf{ge}}^{tipo\ de\ matriz}\ \underbrace{\textbf{sv}}_{operación}$

   culaSgesv Resuelve un sistema de ecuaciones lineales de la forma $Ax = b$, siendo A una matriz general y los datos reales de precisión simple.

2. $\overbrace{\textbf{cula}}^{prefijo}\ \underbrace{\textbf{D}}_{tipo\ de\ dato}\ \overbrace{\textbf{ge}}^{tipo\ de\ matriz}\ \underbrace{\textbf{svd}}_{operación}$

   culaDgesvd Descompone una matriz general de datos reales de precisión doble en sus valores singulares.

donde **cula** se refiere a que es una rutina de CULA, *tipo de dato* se refiere al tipo de precisión con la que opera la rutina (ver tabla 1), *tipo de matriz* se refiere al tipo de matriz con que opera la rutina (ver tabla 2) y *operación* al tipo de cómputo que realiza la rutina.

4.8.2. Tipos CULA

CULA crea sus propios tipos de datos usando el operador typedef [5] para realizar sus operaciones, que no son más que sinónimos de los tipos estándar de C, algunos son:

- `typedef int culaInt`
- `typedef float culaFloat`
- `typedef double culaDouble`
- `typedef culaInt culaDeviceInt`
- `typedef culaFloat culaDeviceFloat`
- `typedef culaDouble culaDeviceDouble`

4.8.3. Funciones auxiliares CULA

Se describen las funciones que están presentes en cualquier programa que use CULA.

culaInitialize inicializa CULA, debe ser invocada antes que cualquier otra función CULA.

```
culaStatus culaInitialize()
```

donde **culaStatus** monitorea el estado de invocación de CULA, el cual está definido como una enumeración [6]

```
typedef enum
{
  culaNoError,                      // Sin error
  culaNotInitialized,               // CULA no fue inicializado
  culaNoHardware,                   // No hay hardware apto para CULA
  culaInsufficientRuntime,          // Versión de driver CUDA no soportada
  culaInsufficientComputeCapability, // Las GPU no soportan las operaciones solicitadas
  culaInsufficientMemory,           // No hay memoria suficiente para continuar
  culaFeatureNotImplemented,        // Caraterística requerida no disponible
  culaArgumentError,                // Argumento incorrecto para la rutina
  culaDataError,                    // Error en el tipo de dato
  culaBlasError,                    // Error causado al invocar BLAS
  culaRuntimeError                  // Error de tiempo de ejecución
} culaStatus;
```

culaShutdown cierra CULA, además de que libera los recursos usados por la misma.

```
void culaShutdown()
```

culaGetErrorInfo esta función es usada para extender la funcionalidad de LAPACK en cuanto a los códigos de error.

```
info_t culaGetErrorInfo()
```

[5] El operador `typedef` permite a un programador crear un sinónimo de un tipo de dato definido por el usuario o de un tipo ya existente [14]

[6] Una enumeración es un tipo definido por el usuario con constantes de tipo entero. En la declaración de un tipo `enum` se escribe una lista de identificadores que internamente se asocian con las constantes 0, 1, 2, . . . [14]

4.8.4. Resolución de sistemas de ecuaciones lineales de la forma $Ax = b$

CULA cuenta con una rutina para resolver un sistema de ecuaciones: **gesv**, dicha función ofrece soporte para todos los tipos de datos (ver tabla 1), en este ejemplo se usará la precisión simple **S**.

El prototipo de la rutina **culaSgesv** es:

```
void culaSgesv(int n, int nrhs, culaFloat* a, int lda, culaInt* ipiv,
               culaFloat* b, int ldb)
```

Entrada

n	número de ecuaciones del sistema.
nrhs	número de columnas de la matriz b.
a	matriz de coeficientes de (lda, n).
lda	número máximo de filas que se pueden almacenar en la matriz a. lda $\geq n$.
ipiv	arreglo de dimensión n, guarda los índices de las filas que son intercambiadas.
b	matriz de dimensión (ldb, nrhs), que de entrada contiene el lado derecho del problema y a la salida la matriz de soluciones X.
ldb	número máximo de filas que se pueden almacenar en la matriz b. $ldb \geq n$.

Salida

X	$x = A^{-1} \times b$

4.8.5. Descomposición de una matriz en sus valores singulares de la forma $A = U\Sigma V^T$

CULA cuenta con una rutina para descomponer una matriz en sus valores singulares: **gesvd**, dicha función ofrece soporte para todos los tipos de datos (ver tabla 1), en este ejemplo se usará la precisión simple **s**.

El prototipo de la rutina **sgesvd** es:

```
culaStatus culaSgesvd(char jobu, char jobvt, int m, int n, culaFloat* a, int lda,
                      culaFloat* s, culaFloat* u, int ldu, culaFloat* vt, int ldvt)
```

Entrada

jobu	determina si se calcula toda o parte de la matriz U. Si `jobu = 'A'` todas las m columnas de U son retornadas en el arreglo U. Si `jobu = 'S'` las primeras min(m, n) columnas de U (vectores singulares izquierdos) son retornados en el arreglo U. Si `jobu = 'O'` las primeras min(m, n) columnas de U (vectores singulares izquierdos) son sobrescritas en A. Si `jobu = 'N'` no se hacen cálculos de vectores singulares izquierdos.
jobvt	determina se se calcula toda o una parte de la matriz VT. Si `jobvt = 'A'` todas las n filas de VT son retornadas en el arreglo VT. Si `jobvt = 'S'` las primeras min(m, n) filas de VT (vectores singulares derechos) son retornados en el arreglo VT. Si `jobvt = 'O'` las primeras min(m, n) filas de VT son sobrescritas en A. Si `jobvt = 'N'` no se hacen cálculos de vectores singulares derechos. jobu y jobvt no pueden ser ambos 'O'
m	número de filas de la matriz A.
n	número de columnas de la matriz A.
a	matriz de ecuaciones de (m, n). Si `jobu = 'O'`, A es sobrescrita con las primeras columnas de U, que son almacenados por columna. Si `jobvt = 'O'`, A es sobrescrita con las primeras filas de la matriz VT, que son almacenados por fila. Si `jobu != 'O'` y `jobvt != 'O'`, el contenido de A es destruido.
lda	dimensión de la matriz a.
s	valores singulares de A.
u	matriz de dimensión (ldu, m) si `jobu = 'A'` o (ldu. min(m,n)) si `jobu = 'S'` o (m, m) si `jobu = 'A'`.
ldu	dimensión de la matriz S.
vt	matriz de dimensión (n , n) si `jobvt = 'A'` o los primeros min(m, n) filas de VT si `jobvt = 'S'`.
ldvt	dimensión de la matriz VT.

Salida

S, U, VT	$A = U\Sigma V^T$

5. Instalación y configuración de CUDA, CUBLAS, CULA, BLAS y LAPACK

5.1. Instalación de CUDA en una computadora con sistema operativo LINUX

Para la instalación y configuración de CUDA y CUBLAS se usó el sistema operativo Linux Ubuntu 9.04 de 64 bits. Al momento de escribir este documento la versión actual de CUDA es **3.2**.

Para escribir programas utilizando la tecnología CUDA, es necesario cumplir con los siguientes requisitos:

1. Tarjeta gráfica apta para CUDA
2. Controlador (driver) de la tarjeta gráfica
3. Software CUDA
4. Sistema operativo LINUX (en este caso)
5. Compliador de C (gcc en este caso)

Los programas desarrollados se realizaron con el siguiente hardware:
Computadora de escritorio HP

- Procesador Intel Core Quad a 2.4 GHz.
- Memoria RAM de 4 Gb.
- Disco duro de 320 Gb.
- Tarjeta NVIDIA GeForce 8600 GTS
  - CUDA Cores: 32
  - Graphics Clock: 675 MHz.
  - Processor Clock: 1450 MHz.
- Sistema operativo: Ubuntu Linux 9.04 de 64 bits.

A continuación se explicarán los pasos que se siguieron para instalar CUDA en la computadora antes mencionada
Los requisitos para instalar CUDA son:

5.1.1. Verificación de requisitos

- **Tarjeta de gráficos NVIDIA apta para CUDA**. Para comprobar que se cuenta con una tarjeta apta para CUDA, desde una terminal o consola de Linux se ejecuta el siguiente comando (ver figura 17(a)):

```
lspci | grep -i NVIDIA
```

  Es necesario conocer el modelo de la tarjeta gráfica para descargar el driver o controlador correcto.

- **Distribución de Linux soportada por CUDA**. Para saber si la distribución de Linux es compatible con CUDA, desde la consola de comandos se ejecuta lo siguiente (ver figura 17(b)):

```
uname -n && cat /etc/*release
```

  Es importante conocer la distribución de Linux para descargar el controlador NVIDIA y el software CUDA adecuados.

- **Compilador de C**. Para verificar que ésta instalado el compilador de C, desde la consola de comandos se ejecuta lo siguiente (ver figura 17(c)):

```
gcc --version
```

```
misa@yo:~$ lspci | grep -i NVIDIA
01:00.0 VGA compatible controller: nVidia Corporation GeForce 8600 GTS (rev a1)
```

(a) Tarjeta de gráficos instalada

```
misa@yo:~$ uname -n && cat /etc/*release
yo
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=9.04
DISTRIB_CODENAME=jaunty
DISTRIB_DESCRIPTION="Ubuntu 9.04"
```

(b) Distribución de LINUX instalada

```
misa@yo:~$ gcc --version
gcc (Ubuntu 4.3.3-5ubuntu4) 4.3.3
Copyright (C) 2008 Free Software Foundation, Inc.
Esto es software libre; vea el código para las condiciones de copia.  NO hay
garantía; ni siquiera para MERCANTIBILIDAD o IDONEIDAD PARA UN PROPÓSITO EN
PARTICULAR
```

(c) Compilador de C (gcc) instalado

Figura 17: Verificación de requisitos para instalar CUDA

5.1.2. Obtención de software

Para utilizar CUDA se debe contar con los siguientes elementos básicos:

- Driver NVIDIA,
- CUDA Toolkit,
- CUDA SDK,

los cuales se obtienen de la página http://developer.nvidia.com/object/cuda_3_1_downloads.html#Linux (la fecha en que se realizó la consulta es 07 de Septiembre de 2010).

Driver NVIDIA Es el controlador de tarjeta gráfica, para descargarlo se deben especificar las características de la misma (ver figura 18). Se puede obtener accediendo a la dirección de Internet: http://www.nvidia.com/Download/index.aspx?lang=en-us

Figura 18: Página de descarga del driver NVIDIA

CUDA Toolkit. Es el compilador de C para CUDA así como un conjunto de bibliotecas para su implementación, hay versiones para Windows, Linux y MacOS (ver figura 19) de acuerdo a lo obtenido en la sección 5.1.1 descargar el CUDA Toolkit para la versión de Linux que está instalada (ver figura 20).

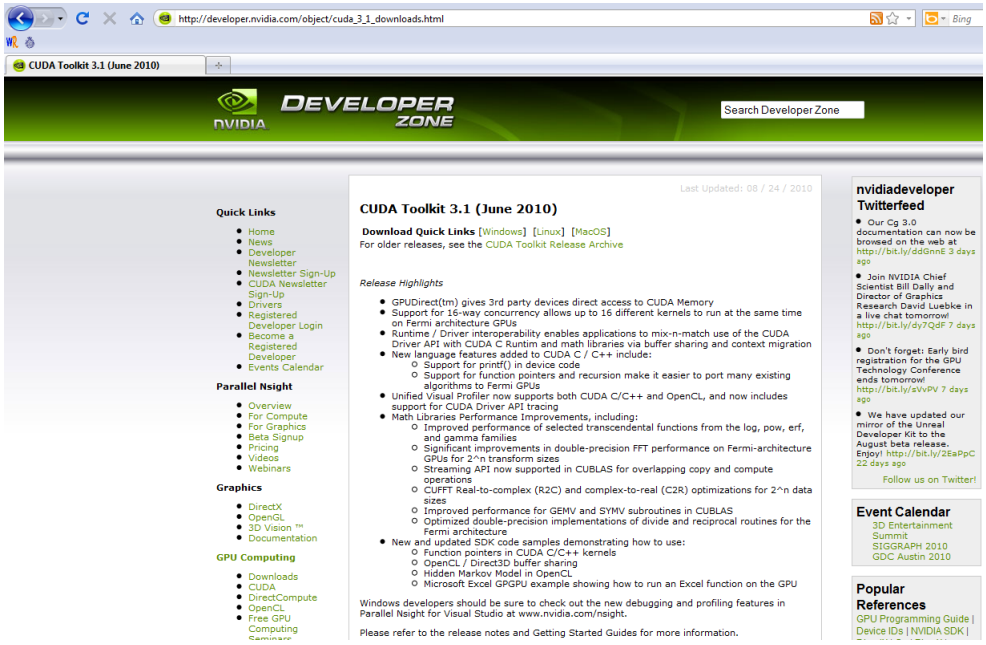


Figura 19: Página de descarga del software CUDA para diferentes sistemas operativos

CUDA SDK. Incluye proyectos de ejemplo que proporcionan código fuente y otros recursos para la construcción de los programas. Se descarga entrando a la dirección de Internet: `http://developer.nvidia.com/object/cuda_3_1_downloads.html#Linux`

de donde también es posible descargar documentación de las herramientas CUDA así como de CUBLAS, entre otras (ver figura 20).

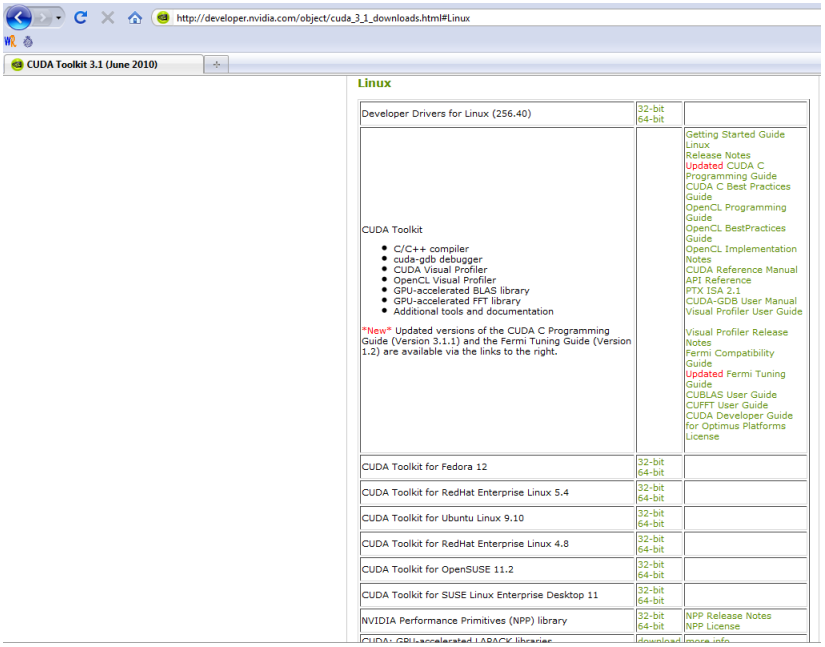


Figura 20: Página de descarga de software CUDA para Linux

5.1.3. Instalación y Configuración de software CUDA

Todo lo realizado en esta sección será desde la terminal de comandos. Para poder instalar y configurar el software CUDA es necesario conocer la contraseña de superusuario (root).

Primero se deben cambiar los permisos de los archivos descargados en la sección 5.1.2, esto se hace con el comando chmod.

```
sudo chmod 744 NVIDIA-Linux-x86_64-256.53.run
sudo chmod 744 cudatoolkit_3.1_linux_64_ubuntu9.10.run
sudo chmod 744 gpucomputingsdk_3.1_linux.run
```

Driver NVIDIA. Para instalar el controlador NVIDIA es necesario salir del modo gráfico, esto se hace con la combinación de teclas `Ctrl+Alt+(F1,F2, ..., F6)`

Una vez furea del entorno gráfico se debe detener el mismo, lo que se hace con el siguiente comando:

```
sudo /etc/init.d/gdm stop
```

Ahora, se puede proceder a instalar el controlador de NVIDIA con el siguiente comando:

```
sudo ./NVIDIA-Linux-x86_64-256.53.run
```

Este comando desplegará un asistente que guiará en la instalación del controlador, este asistente verifica que se cumpla con todos los requisitos, si no se cumplen se detiene reportando el motivo de la interrupción, en caso contrario continua con la instalación.Si todo ha salido bien se puede volver al modo gráfico tecleando el siguiente comando:

```
startx
```

En el menú sistema->preferencias->NVIDIA X Server Settings (ver figura 21) se puede verificar que el controlador se instaló correctamente. En el menú sistema->preferencias->NVIDIA X Server Settings (ver figura 21).

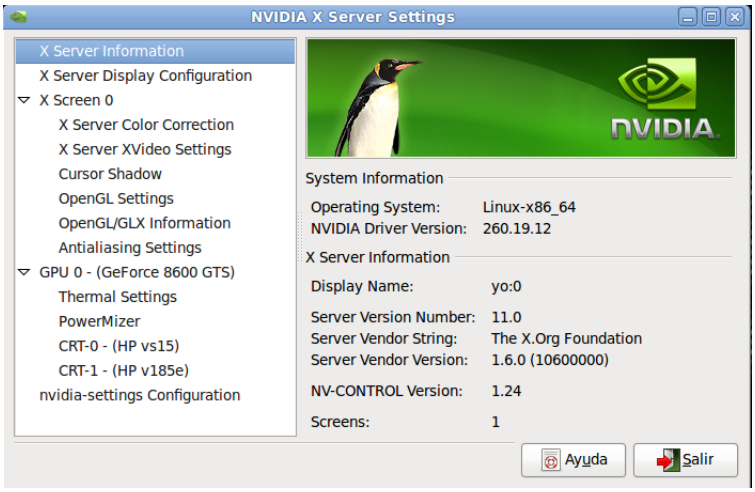


Figura 21: Panel de configuración NVIDIA

Errores más comunes: Si se han seguido los pasos para instalar el controlador NVIDIA no se tendrá ningún problema, si se tiene alguno, los errores más comunes son los siguientes:

1. Problemas al descargar el controlador de NVIDIA. Solución: Descargar otra vez el controlador.
2. La tarjeta no es compatible con el driver (ver sección 5.1.1). Solución: Adquirir una tarjeta NVIDIA compatible con CUDA.

CUDA Toolkit. Para instalar el Toolkit de CUDA se hace con el siguiente comando:

```
sudo ./cudatoolkit_3.1_linux_64_ubuntu9.10.run
```

Al comenzar la instalación se hace una comprobación del compilador de C (gcc) y de C++ (g++), si todo está bien se desplegará un asistente que guiará en la instalación.

CUDA_SDK. El SDK (Software Development Kit) de CUDA se instala de la misma forma que el CUDA Toolkit, con el comando:

```
sudo ./gpucomputingsdk_3.1_linux.run
```

Variables de Entorno. Una vez instalados el CUDA Toolkit y el CUDA SDK, es necesario configurar algunas variables de entorno imprescindibles para la ejecución de CUDA:

- PATH Contendrá la ruta en la que se instalo CUDA (/usr/local/cuda/bin)
- LD_LIBRARY_PATH Contendrá la ruta en la que se instalaron las bibliotecas CUDA (/usr/local/cuda/lib64/)

Para hacer esto se debe editar el archivo **~/.bashrc** (puede ser con vim, emacs, etc.) el cual contiene la definición de variables de entorno que utiliza el sistema. Se agregan las siguientes líneas al final del archivo:

```
# directorio de instalación de CUDA
export PATH=$PATH:/usr/local/cuda/bin
# si se tiene versión de sistema operativo de 32 bits
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib
# si se tiene versión de sistema operativo de 64 bits, agregar también
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64
```

5.1.4. Comporbación de la instalación

Tras haber hecho la instalación del software CUDA y la configuración de las variables de entorno, es necesario verificar si existe una comunicación correcta entre CUDA y el hardware (tarjeta NVIDIA), esto se logra compilando y ejecutando algunos programas del CUDA_SDK.

Desde la terminal de comandos se ejecuta:

```
nvcc -V
```

NVCC es el compilador de CUDA el cual se utiliza para compilar las aplicaciones realizadas con CUDA. Este compilador llama al compilador de C (gcc) para compilar código C, y al NVIDIA PTX (Parallel Thread Execution) para compilar código CUDA.

El comando anterior debe mostrar la versión del compilador CUDA, si aparece un mensaje de error, es necesario revisar las variables de entorno para constatar que tienen el valor correcto.

Una vez que el compilador es llamado correctamente, bastará con compilar algunos ejemplos para asegurar que se usan las GPUs. Estos programas se encuentran en el CUDA_SDK, el proceso de compilación es el siguiente:

```
cd ~/NVIDIA_GPU_Computing_SDK/C
make
```

Con esto se crean las bibliotecas necesarias para poder ejecutar los programas del CUDA_SDK

5.1.5. Ejecución de programas de prueba

Hasta ahora se ha comprobado la instalación y configuración del software CUDA, es momento de ejecutar algunos programas para confirmar que en verdad se está haciendo uso del hardware NVIDIA.

Los archivos fuente se encuentran en el directorio **src**, a continuación se muestra su ubicación:

```
cd ~/NVIDIA_GPU_Computing_SDK/C/src/
```

El programa **deviceQuery**, es el *hola mundo* de CUDA, muestra las características de la tarjeta NVIDIA (ver figura 22). Este programa está dentro del directorio deviceQuery y para compilarlo se ejecuta lo siguiente:

```
make
```

El programa se ejecuta con el siguiente comando:

```
./../../bin/linux/release/deviceQuery
```

```
misa@yo:~/NVIDIA_GPU_Computing_SDK/C/src/deviceQuery$ ./../../bin/linux/release/deviceQuery
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA

Device 0: "GeForce 8600 GTS"
  CUDA Driver Version:                           2.30
  CUDA Runtime Version:                          2.30
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         1
  Total amount of global memory:                 268107776 bytes
  Number of multiprocessors:                     4
  Number of cores:                               32
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.46 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       No
  Compute mode:                                  Default (multiple host threads can use this device simultaneously)

Test PASSED

Press ENTER to exit...
```

Figura 22: Características de Tarjeta NVIDIA

5.2. Instalación de CUBLAS

CUBLAS no necesita ser instalado, esta es una biblioteca que se crea al momento de instalar el CUDA Toolkit y para compilar y ejecutar programas que usen dicha biblioteca se debe especificar en la sentencia de compilación, por ejemplo:

```
nvcc miPrograma.cu -o miProgrma -lcublas
```

Esto se verá con detalle en la sección 6.1.4

5.3. Instalación de BLAS

Al igual que CUBLAS, BLAS es una biblioteca que se necesita para poder compilar los programas que hagan uso de la misma. La instación se describe a continuación, es necesario conocer la contraseña de superusuario (root).

Desde la consola de comandos se ejecuta lo siguiente:

```
sudo apt-cache search libblas
```

Con este comando se muestran todas las opciones que están disponibles para BLAS (ver figura 23).

```
misa@yo:~$ apt-cache search libblas
refblas3 - Basic Linear Algebra Subroutines 3, shared library
refblas3-dev - Basic Linear Algebra Subroutines 3, static library
libblas-dev - Subrutinas básicas de álgebra lineal 3, biblioteca estática
libblas-doc - Subrutinas de álgebra lineal básica 3, documentación
libblas3gf - Subrutinas de álgebra lineal básica 3, biblioteca compartida
libblas-test - Basic Linear Algebra Subroutines 3, testing programs
libatlas-base-dev - Programa de álgebra lineal de afinado automático, estático genérico
libatlas3gf-base - Automatically Tuned Linear Algebra Software,generic shared
misa@yo:~$ sudo apt-get install libblas-dev
```

Figura 23: Instalación de BLAS

Ahora se instala la opción **libblas-dev** con el siguiente comando:

```
sudo apt-get install libblas-dev
```

Por último se verifica que se haya instalado correctamente (ver figura 24), esto con una serie de comandos:

```
misa@yo:~$ dpkg -l | grep blas
ii  libblas-dev                                 1.2-2
ii  libblas-doc                                 1.2-2
ii  libblas-test                                1.2-2
ii  libblas3gf                                  1.2-2
misa@yo:~$ dpkg -L libblas-dev
/.
/usr
/usr/lib
/usr/lib/libblas.a
/usr/include
/usr/include/cblas.h
/usr/include/cblas_f77.h
/usr/share
/usr/share/doc
/usr/share/doc/libblas-dev
/usr/share/doc/libblas-dev/test_results.gz
/usr/share/doc/libblas-dev/copyright
/usr/share/doc/libblas-dev/README.Debian.gz
/usr/share/doc/libblas-dev/changelog.Debian.gz
/usr/lib/libblas.so
```

Figura 24: Vefificar instalación de BLAS

Primero, se busca todo lo relacionado con blas:

```
dpkg -l | grep blas
```

Segundo, se lista lo que contine el archivo antes instalado (libblas-dev):

```
dpkg -L libblas-dev
```

Con esto queda instalado BLAS, los archivos que son necesarios para compilar los programas desarrollados son: libblas.a y cblas.h (ver figura 24).

5.4. Instalación de CULA

Como se vio en la sección 4.8 actualmente CULA tiene tres versiones, en esta sección se describirá cómo instalar la versión **basic**. Antes de instalar CULA debe estar instalada y configurada CUDA.

CULA se encuentra disponible en la dirección electrónica www.culatools.com. Para poder descargar archivos desde este sitio, es necesario registrarse.

Como se puede ver en la figura 25 se han creado versiones para diferentes plataformas, en este caso se descargó la de Linux de 64 bits.

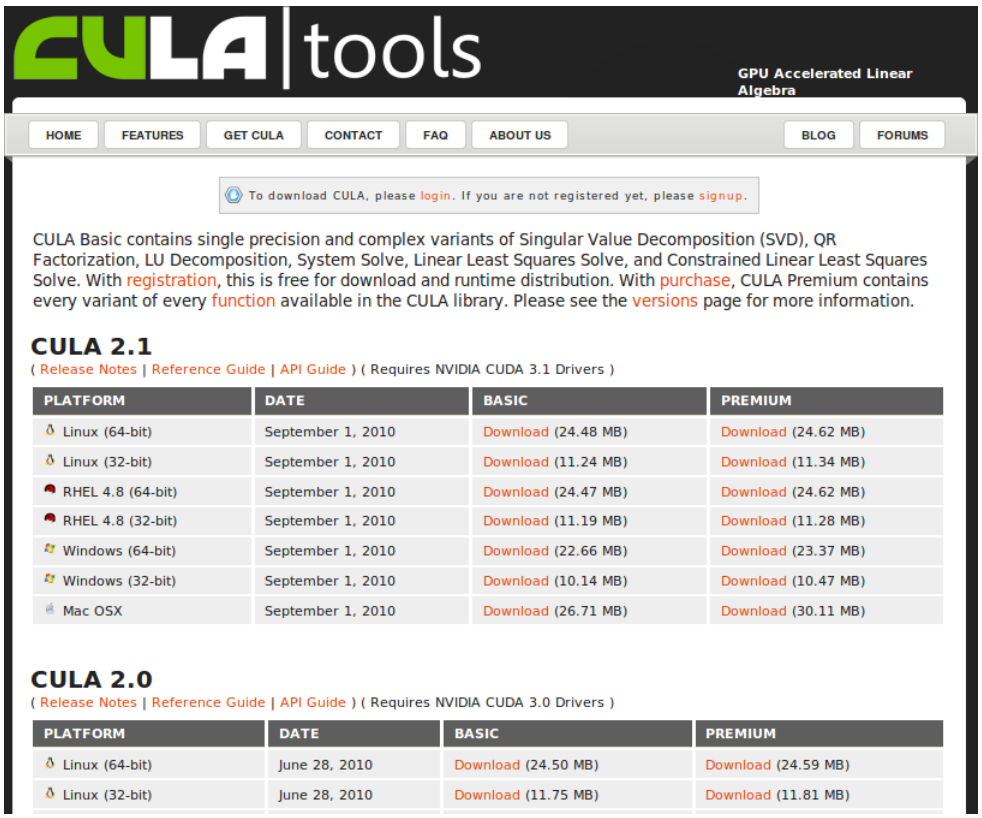


Figura 25: Página de CULA

Una vez descargado el archivo, se procede a instalarlo. Entonces, desde la consola de comandos se ejecuta lo siguiente:

```
sudo sh cula_2.1-linux64.run
```

Esta acción desplegará un asistente que guiará por la instalación.

Una vez instalado CULA es necesario configurar algunas variables de entorno. Se edita el archivo **~/.bashrc** y se agrega lo siguiente al final del archivo:

```
# directorio de instalación de CULA
export CULA_ROOT=/usr/local/cula
# archivos de cabecera de CULA
export CULA_INC_PATH=$CULA_ROOT/include
# archivos binarios de CULA
export CULA_BIN_PATH_32=$CULA_ROOT/bin
export CULA_BIN_PATH_64=$CULA_ROOT/bin64
# bibliotecas de CULA
export CULA_LIB_PATH_32=$CULA_ROOT/lib
export CULA_LIB_PATH_64=$CULA_ROOT/lib64
# se agregan bibliotecas de CULA a las bibliotecas del sistema
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CULA_LIB_PATH_32:$CULA_LIB_PATH_64
```

5.5. Instalación de LAPACK

Para instalar LAPACK se hace algo muy similar a lo hecho con BLAS.

Desde la consola de comandos se ejecuta lo siguiente:

```
sudo apt-cache search liblapack
```

Con este comando se muestran todas las opciones que están disponibles para LAPACK (ver figura 26).

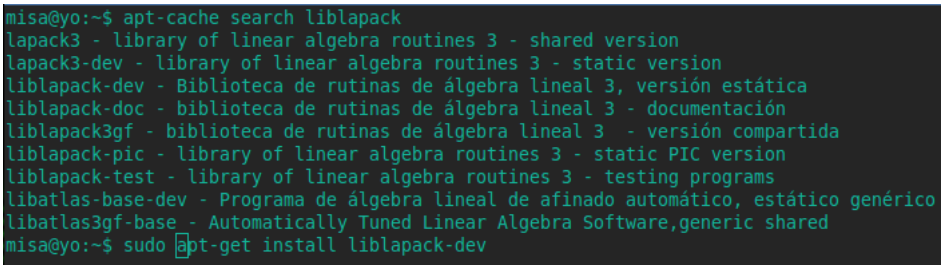


Figura 26: Instalación de LAPACK

Ahora se instala la opción **liblapack-dev** con el siguiente comando:

```
sudo apt-get install liblapack-dev
```

Por último se verifica que se haya instalado correctamente (ver figura 27), esto con una serie de comandos:

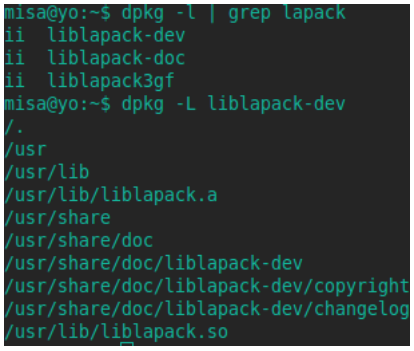


Figura 27: Verificar instalación de LAPACK

Primero, se busca todo lo relacionado con LAPACK:

```
dpkg -l | grep lapack
```

Segundo, se lista lo que contiene el archivo antes instalado (liblapack-dev):

```
dpkg -L liblapack-dev
```

Con esto queda instalado LAPACK, los archivos que son necesarios para compilar los programas desarrollados son: liblapack.a (ver figura 27).

6. Diseño codificación y pruebas de operaciones básicas

6.1. Elaboración, Compilación y Ejecución de Programas en C, BLAS, CUDA, CUBLAS

Todos los programas han sido escritos siguiendo las recomendaciones de [5].

A continuación se muestra la organización de los programas generados para C, BLAS, CUDA y CUBLAS:

|–bin/
|–include/
|——c.h
|——cubla.h
|——cuda.hcu
|——mat.h
|——vec.h
|–src/
|——c.c
|——cubla.cu
|——cuda.cu

|——mat.c
|——mat.cu
|——vec.c
|——vec.cu
|–test/
|——axpy.c
|——axpy.cu
|——gemm.c
|——gemm.cu
|——gemv.c
|——gemv.cu

bin	contiene los archivos ejecutables.
include	contiene los archivos de cabecera
src	contiene los archivos fuente, es decir, la implementación de las funciones definidas en los archivos de cabecera
test	contiene los archivos que prueban cada uno de los programas implementados.

6.1.1. Definición de las estructuras y funciones usadas por los programas en C, BLAS, CUDA y CUBLAS

En esta sección se muestra el contenido del directorio **include** que son todas las definiciones (archivos de cabecera) de las estructuras de datos y funciones para manipular matrices (programa 5) y vectores (programa 4), así como las funciones para operaciones básicas de matrices y vectores con C (programa 6), las funciones para operaciones básicas de matrices y vectores con CUDA (programa 8) y las funciones para operaciones básicas de matrices y vectores con CUBLAS (programa 7).

A excepción del programa 8 todos tienen extensión **.h**, la diferencia se debe a que en la implementación de cuda.hcu (programa cuda.cu) se usan kernels de CUDA y deben ser compilados con nvcc.

Programa 4: vec.h

```
1 // ============================================================================
2 // Filename: vec.h
3 //
4 // Description: Definición de las funciones para manipular vectores.
5 // Compiler: gcc
6 // ============================================================================
7
8 #ifndef  _VEC_H_
9 // Declara la estructura vector y sus funciones de acceso
10 #define  _VEC_H_
11
12 //-----------------------------------------------
13 // Estructura de Vector con elementos de precisión simple (float).
14 //-----------------------------------------------
15 typedef struct{
16   int n;  // Tamaño del vector
17   float *data;  // Elementos del vector
18 }Vector;  // ------- end of struct Vector -------
19
20 void llenaVector( Vector * );  // Llena el vector
21 void allocateVector( Vector *, int );  // Reserva recursos de forma dinámica
22 void freeVector( Vector * );  // Libera recursos
23 void setVectorValueAt( Vector *, int, float );  // Asigna valor al elemento i del vector
24 float getVectorValueAt( const Vector *, const int );  // retorna el valor del elemento i del vector
25 void imprimeVector( Vector * );  // imprime el vector
```

Programa 5: mat.h

```
1 // ============================================================================
2 // Filename: mat.h
3 //
4 // Description: Definición de las funciones para manipular matrices.
5 // Compiler: gcc
6 // ============================================================================
7
8 #ifndef  _MATRIZ_H_
9 #define _MATRIZ_H_
10
11 //-----------------------------------------------
12 // Estructura de Matriz con elementos de precisión simple (float).
13 //-----------------------------------------------
14 typedef struct{
15         int rows;  // Filas
```

```
16         int cols;  // Columnas
17         float *data;  // Datos
18 }Matriz;  // ------- end of struct Matriz -------
19
20 void allocateMatriz(Matriz *, int, int);  // Reserva recursos de forma dinámica
21 void llenaMatriz(Matriz *);  // Llena la matriz
22 void freeMatriz(Matriz *);  // Libera recursos
23 void setMatrizValueAt(Matriz *, int, int, float);  // Asigna valor al elemento i,j de la matriz
24 float getMatrizValueAt(const Matriz *, int, int);  // Retorna el valor del elemento i,j de una matriz
25 void transMatriz(Matriz *, int, int);  // Transpone una matriz
26 void imprimeMatriz(Matriz *);  // imprime la matriz
27 #endif   // termina la definición
```

Programa 6: c.h

```
1 // ============================================================================
2 // Filename: c.h
3 //
4 // Description: Definición de las funciones de operaciones básicas de vectores y
5 // matrices con C.
6 // Compiler: gcc
7 // ============================================================================
8
9 #ifndef _C_H_
10 #define _C_H_
11 #include  "vec.h"
12 #include  "mat.h"
13
14 typedef enum{ trans = 201, noTrans = 202 } Trans;  // determina si se toma A o A^T
15
16 void c_sscal( const int, const float, Vector *, const int );  // Escalamiento de un vector
17
18 void c_saxpy( const int, const float, const Vector *, const int, Vector *,
19     const int );  // Suma vector vector
20
21 float c_sdot( const int, const Vector *, const int, const Vector *,
22     const int );  // Producto vector vector
23
24 void c_sgemv( const  Trans, const int, const int, const float,
25   const Matriz *, const int, const Vector *, const int,
26   const float, Vector *, const int );  // Producto matriz vector
27
28 void c_sgemm( const  Trans, const  Trans, const int,
29   const int, const int, const float, const Matriz *,
30   const int, const Matriz *, const int, const float,
31   Matriz *, const int );  // Producto matriz matriz
32
33 #endif   // termina la definición
```

Programa 7: cubla.h

```
1 // ============================================================================
2 // Filename: cubla.h
3 //
4 // Description: Definición de las funciones de operaciones básicas de vectores y
5 // matrices con CUBLAS.
6 // Compiler: nnvcc
7 // ============================================================================
8 #ifndef _CUBLA_H_
9 #define _CUBLA_H_
10 #include  <cublas.h>
11 #include  "vec.h"
12 #include  "mat.h"
13
14 void check_cublas_status ( char *, cublasStatus );  // revisa el estado de invocación de cublas
15
16 void cublas_sscal( const int, const float, Vector *, const int );  // Escalamiento de un vector
17
18 void cublas_saxpy( const int, const float, const Vector *, const int, Vector *,
19     const int );  // Suma vector vector
20
21 float cublas_sdot( const int, const Vector *, const int, const Vector *,
22     const int );  // Producto vector vector
23
24 void cublas_sgemv( char, const int, const int, const float,
25   const Matriz *, const int, const Vector *, const int,
26   const float, Vector *, const int );  // Producto matriz vector
27
28 void cublas_sgemm( char, char, const int,
```

```
29   const int, const int, const float, const Matriz *,
30   const int, const Matriz *, const int, const float,
31   Matriz *, const int );  // Producto matriz matriz
32
33 #endif   // termina la definición
```

Programa 8: cuda.hcu

```
1  // ============================================================================
2  // Filename: cuda.hcu
3  //
4  // Description: Definición de las funciones de operaciones básicas de vectores y
5  // matrices con CUDA.
6  // Compiler: nvcc
7  // ============================================================================
8
9  #ifndef  _CUDA_H_
10 #define  _CUDA_H_
11
12 #include  "mat.h"
13 #include  "vec.h"
14
15 void cuda_sscal( const int, const float, Vector * );  // Escalamiento de un vector
16
17 void cuda_saxpy( const int, const float, const Vector *, Vector * );  // Suma vector vector
18
19 float cuda_sdot( const int, const Vector *, const Vector * );  // Producto vector vector
20
21 void cuda_sgemv( const int, const int, const float, const Matriz *, const Vector *,
22   const float, Vector * );  // Producto matriz vector
23
24 void cuda_sgemm( const int, const float, const Matriz *, const Matriz *,
25   const float, Matriz * );  // Producto matriz matriz
26
27 #endif   // termina la definición
```

6.1.2. Implementación de las estructuras y funciones para manipular matrices y vectores

A continuación se muestra el contenido del directorio **src**, que son las implementaciones de las funciones para manipular matrices (programa10) y vectores (programa 9). El resto de las implementaciones se aborda más adelante.

Para el caso de compilar los programas con CUDA se tienen los archivos **vec.cu** y **mat.cu** con exactamente el mismo contenido que **mat.c** y **vec.c** respectivamente, pero diferente extensión, esto para compilar con **nvcc**.

Programa 9: vec.c

```
1  // ============================================================================
2  // Filename: vec.c
3  //
4  // Description: Implementación de las funciones para manipular vectores
5  // Compiler: gcc
6  // ============================================================================
7
8
9  #include  <stdlib.h>
10 #include  <stdio.h>
11 #include  <malloc.h>
12
13 #include  "vec.h"  // funciones que manipulan vectores
14
15 time_t time ( time_t * timer );
16
17
18 // === FUNCTION ===============================================================
19 // Name: allocateVector
20 // Description: Reserva recursos en memoria para un vector de precisión simple.
21 // ============================================================================
22
23 void
24 allocateVector (
25   Vector *v,  // vector de n elementos
26   int n  // número de elementos del vector
27 )
28 {
29   v->data = (float *)malloc(sizeof(float)*n);
30   if ( v->data==NULL ) {
31     fprintf ( stderr, "\nError: no se ha podido reservar memoria para el vector\n" );
```

```
32      exit (EXIT_FAILURE);
33    }
34    else {
35      v->n = n;
36    }
37  }  // --- end of function allocateVector ---
38
39
40  // === FUNCTION ======================================================================
41  // Name: llenaVector
42  // Description: Llena un vector de precisión simple con números aleatorios.
43  // ====================================================================================
44
45  void
46  llenaVector ( Vector *v )
47  {
48    srand( time( NULL ));
49    int i;
50    for ( i = 0; i < v->n; i += 1 ) {
51      v->data[ i ] = rand()  % 9;
52    }
53  }  // --- end of function llenaVector ---
54
55
56  // === FUNCTION ======================================================================
57  // Name: freeVector
58  // Description: Libera recursos ocupados por el vector.
59  // ====================================================================================
60
61  void
62  freeVector ( Vector *v )
63  {
64    v->n = 0;
65    free (v->data);
66    v->data = NULL;
67  }  // --- end of function freeVector ---
68
69
70  // === FUNCTION ======================================================================
71  // Name: setVectorValueAt
72  // Description: Asigna un valor value a un elemento index de un vector v.
73  // ====================================================================================
74
75  void
76  setVectorValueAt (
77    Vector *v,  // vector al que se le asignan valores
78    int index,  // índice del vector en que se guarda value
79    float value  // valor
80  )
81  {
82    if ( index < 0 || index >= v->n ) {
83      printf ( "sV_Indice fuera de rango\n" );
84      exit( EXIT_FAILURE );
85    }
86    else {
87      v->data[index] = value;
88    }
89  }  // --- end of function setVectorValueAt ---
90
91
92  // === FUNCTION ======================================================================
93  // Name: getVectorValueAt
94  // Description: Retorna el valor float de un vector v del índice index
95  // ====================================================================================
96
97  float
98  getVectorValueAt (
99    const Vector *v,  // vector del que se obtiene index
100   const int index  // índice para obtener su valor
101 )
102 {
103   if ( index < 0 || index >= v->n ) {
104     printf ( "gV_Indice fuera de rango\n" );
105     exit( EXIT_FAILURE );
106   }
107   else {
108     return v->data[index];
109   }
110 }  // --- end of function getVectorValueAt ---
111
112
```

```
113 // === FUNCTION ======================================================================
114 // Name: imprimeVector
115 // Description: Imprime el contenido de un vector con elementos de precisión simple.
116 // ====================================================================================
117
118 void
119 imprimeVector ( Vector *v )
120 {
121   int i;
122   printf ( "*******************************************************\n" );
123   for ( i = 0; i < v->n; i += 1 ) {
124     printf ( "[%d]:\t%.3f\n", i, v->data[i] );
125   }
126 }  // --- end of function imprimeVector ---
```

Programa 10: mat.c

```
1 // ====================================================================================
2 // Filename: mat.c
3 //
4 // Description: Implementación de las funciones para manipular matrices
5 // Compiler: gcc
6 // ====================================================================================
7
8
9 #include  <stdio.h>
10 #include  <stdlib.h>
11 #include  <malloc.h>
12 #include  <time.h>
13
14 #include  "mat.h"  // funciones que manipulan matrices
15
16 time_t time ( time_t * timer );  // ayuda a generar números aleatorios
17
18
19 // === FUNCTION ======================================================================
20 // Name: allocateMatriz
21 // Description: Reserva memoria para una matriz de presición simple (float).
22 // ====================================================================================
23
24 void
25 allocateMatriz (
26   Matriz *m,  // matriz a la que se le reserva memoria
27   int rows,  // filas de la matriz
28   int cols  // columnas de la matriz
29 )
30 {
31   m->data = (float *)malloc(sizeof(float)*rows*cols);
32   if ( m==NULL ) {
33     fprintf ( stderr, "\nError: no se ha podido reservar memoria para la matriz\n" );
34     exit (EXIT_FAILURE);
35   }
36   else {
37     m->rows = rows;
38     m->cols = cols;
39   }
40 }  // --- end of function allocateMatriz ---
41
42
43 // === FUNCTION ======================================================================
44 // Name: llenaMatriz
45 // Description: Llena una matriz de presición simple con números aleatorios.
46 // ====================================================================================
47
48 void
49 llenaMatriz ( Matriz *m )
50 {
51   srand( time(NULL));
52   int i, j;
53   for ( i = 0; i < m->rows; i += 1 ) {
54     for ( j = 0; j < m->cols; j += 1 ) {
55       setMatrizValueAt(m, i, j, rand() % 9);
56     }
57   }
58 }  // --- end of function llenaMatriz ---
59
60
61 // === FUNCTION ======================================================================
62 // Name: freeMatriz
63 // Description: Libera los recursos ocupados por la matriz.
```

```
64 // ===========================================================================================
65
66 void
67 freeMatriz ( Matriz *m )
68 {
69   m->rows = m->cols = 0;
70   free(m->data);
71   m->data = NULL;
72 }  // --- end of function freeMatriz ---
73
74 // === FUNCTION ============================================================================
75 // Name: setMatrizValueAt
76 // Description: Asigna un valor (float) a un elemento i,j de una matriz de
77 // presición simpe.
78 // ===========================================================================================
79
80 void
81 setMatrizValueAt (
82   Matriz *m,  // matriz a la que se asignan valores
83   int i,  // i-esima fila de la matriz
84   int j,  // j-esima columna de la matriz
85   float value  // valor que se gaurada en m(i,j)
86 )
87 {
88   m->data[j * m->cols + i] = value;
89
90 }  // --- end of function setMatrizValueAt ---
91
92
93 // === FUNCTION ============================================================================
94 // Name: getMatrizValueAt
95 // Description: Retorna el valor de la posición i,j de precisión simple (float)
96 // de una matriz
97 // ===========================================================================================
98
99 float
100 getMatrizValueAt (
101   const Matriz *m, // matriz de la que se obtienen valores
102   int i,  // i-esima fila de ma matriz
103   int j  // j-esima columna de la matriz
104 )
105 {
106   return m->data[j * m->cols + i];
107 }  // --- end of function getMatrizValueAt ---
108
109 // === FUNCTION ============================================================================
110 // Name: imprimeMatriz
111 // Description: Imprime una matriz.
112 // ===========================================================================================
113
114 void
115 imprimeMatriz ( Matriz *m )
116 {
117   int i, j;
118   printf ( "***********************************************\n" );
119   for ( i = 0; i < m->rows; i += 1 ) {
120     for ( j = 0; j < m->cols; j += 1 ) {
121       printf ( "%.3f\t", getMatrizValueAt(m, i, j));
122     }
123     printf ( "\n" );
124   }
125 }  // --- end of function imprimeMatriz ---
```

6.1.3. Implementación, Compilación y Ejecución de progamas con C y BLAS

En este apartado se muestran las implementaciones de operaciones básicas de matrices y vectores con C, la suma vector-vector (programa 11), el producto matriz-vector (programa 13) y el producto matriz-matriz (programa 15).

En el caso de BLAS se hizo lo siguiente:

- Para usar BLAS como se se usó, se necesitan dos archivos (ver sección 5.3) **libblas.a** para la compilación y **cblas.h** para ser incluido en los programas que usen BLAS, esto mediante la sentencia **#include <cblas.h>**.
- Se decidió juntar los programas de C y BLAS proque usan el mismo compilador (gcc).

De esta manera no fue necesario crear funciones para implementar las operaciones básicas de matrices y vectores, sino únicamente es necesaria la biblioteca como tal (4.5) y así invocar a las rutinas necesarias. Ahora bien, las operaciones con BLAS se muestran en los programas de prueba, que son, suma vector-vector (programa 12 en la línea 57), producto matriz-vector (programa 14 en la línea 68) y el producto matriz-matriz (programa 16 en la línea 62).

Programa 11: Suma vector-vector con C

```
// === FUNCTION ======================================================================
// Name: c_saxpy
// Description: Suma vector vector, de precisión simple (float).
// y = α × x + y
// ============================================================================================

void
c_saxpy (
  const int n,  // número de elementos del vector de entrada
  const float alpha,  // escalar
  const Vector *x,  // vector de n elementos
  const int incX,  // espacio de almacenamiento entre los elementos de x
  Vector *y,  // vector de n elementos
  const int incY  // espacio de almacenamiento entre los elementos de y
)
{
  int m, i, iX, iY;
  m = i = iX = iY = 0;
  if ( n != 0 || alpha != 0.0 ) {
    if ( incX == 1 && incY == 1 ) {
      m = n % 4;
      if ( m == 0 ) {
        for ( i = m; i < n; i += 4 ) { // desenvolvimiento de ciclo
          // y(i + 1, 2, 3) += x(i + 1, 2, 3) * alpha
          setVectorValueAt(y, i, (getVectorValueAt(y, i) + getVectorValueAt(x, i)*alpha));
          setVectorValueAt(y, i+1, (getVectorValueAt(y, i+1) + getVectorValueAt(x, i+1)*alpha));
          setVectorValueAt(y, i+2, (getVectorValueAt(y, i+2) + getVectorValueAt(x, i+2)*alpha));
          setVectorValueAt(y, i+3, (getVectorValueAt(y, i+3) + getVectorValueAt(x, i+3)*alpha));
        }
      }
      else {
        for ( i = 0; i < n; i += 1 ) {
          // y(i) += x(i) * alpha
          setVectorValueAt(y, i, (getVectorValueAt(y, i) + getVectorValueAt(x, i)*alpha));
        }
      }
    }
    else {
      if ( incX <= 0 ) iX = (-n+1) * incX+1;
      if ( incY <= 0 ) iY = (-n+1) * incY+1;
      for (i = 0; i < n; i += 1 ) {
        if( iX < n && iY < n ) {
          // y(iY) += x(iX) * alpha
          setVectorValueAt(y, iY, (getVectorValueAt(y, iY) + getVectorValueAt(x, iX)*alpha));
          iX += incX;
          iY += incY;
        }
      }
    }
  }
}  // --- end of function c_saxpy ---
```

En el siguiente programa se prueba la operación axpy (suma vector-vector) con C y BLAS, para compilar el mismo se hace con la siguiente sentencia:

```
gcc src/vec.c src/mat.c src/c.c test/axpy.c -o bin/axpy_cpu -I include/ -lblas
```

donde [7]

gcc	es el compilador de C. (ver sección 5.1.1)
src/vec.c	funciones para manipular vectores.
src/mat.c	funciones para manipular matrices (aunque aquí no se opera con matrices, es necesario para compialr src/c.c).
src/c.c	funciones de operaciones de matrices y vectores con C.
test/axpy.c	prueba axpy con C y BLAS.
-o	le indica al compilador que genere un ejecutable axpy_cpu y lo guarde en el directorio bin.
bin/axpy_cpu	ejecutable generado.
-I	le indica al compilador que debe considerar archivos de cabecera creados por el usuario (directorio include).
include/	directorio donde estan los archivos de cabecera.
-lblas	biblioteca libblas.a para reconocer las rutinas BLAS.

Para ejecutar el programa que prueba la suma vector-vector con C y BLAS se hace con la siguiente sentencia:

[7] En la descripción de las sentencias de compilación y de ejecución de los siguientes programas de prueba se omite lo que es común con éstas, como son: compilador, archivos, bibliotecas, etc.

```
./bin/axpy_cpu 15 3.6
```

./	le indica al sistema que se trata de un archivo ejecutable dentro del directorio actual
bin/axpy_cpu	archivo ejecutable que esta dentro del directorio bin.
15	argumento que recibe el programa, indica el tamaño del vector (ver programa 12 línea 36).
3.6	argumento que recibe el programa, indica el escalar (ver programa 12 línea 37).

Programa 12: Prueba suma vector-vector con C y BLAS

```
1  // ================================================================================
2  // Filename: axpy.c
3  //
4  // Description: pueba la rutina axpy con C y BLAS que realiza la suma de dos vectores
5  // y = α × x + y
6  // Compiler: gcc
7  // ================================================================================
8
9
10 #include  <stdlib.h>
11 #include  <stdio.h>
12
13 #include  <cblas.h>  // rutinas con BLAS
14
15
16 #include  "vec.h"  // funciones que manipulan vectores
17 #include  "c.h"  // funciones de matrices y vectores con C
18
19 // === FUNCTION ===============================================================
20 // Name: main
21 // Description: Prueba la rutina axpy con C y BLAS que realiza la suma de dos vectores.
22 // y = α × x + y
23 // ================================================================================
24
25 int
26 main (
27   int argc,  // contador de argumentos
28   char *argv[]  // valor de argumentos
29 )
30 {
31   if ( argc != 3 ) {
32     printf ( "Argumentos: n\talpha\n" );
33     exit(EXIT_FAILURE);
34   }
35
36   // toma argumentos de la sentencia de ejecución
37   int n = atoi(argv[1]);  // número de elementos de los vectores de entrada
38   float alpha = atof(argv[2]);  // escalar
39
40   // vectores para realizar operaciones
41   Vector xc, yc;  // para C
42   Vector xb, yb;  // para BLAS
43
44   // reserva memoria en CPU
45   allocateVector(&xc, n);
46   allocateVector(&yc, n);
47   allocateVector(&xb, n);
48   allocateVector(&yb, n);
49
50   // llena los vectores con números aleatorios
51   llenaVector(&xc);
52   llenaVector(&yc);
53   llenaVector(&xb);
54   llenaVector(&yb);
55
56   // realiza la suma de vectoes
57   c_saxpy(n, alpha, &xc, 1, &yc, 1);  // con C
58   cblas_saxpy(n, alpha, xb.data, 1, yb.data, 1);  // con BLAS
59
60   // imprime resultado
61   imprimeVector(&yc);  // con C
62   imprimeVector(&yb);  // con BLAS
63
64   // libera memoria
65   freeVector(&xc);
66   freeVector(&yc);
67   freeVector(&xb);
68   freeVector(&yb);
```

```
69
70   // finaliza programa
71   return EXIT_SUCCESS;
72 }  // ------- end of function main -------
```

Programa 13: Producto matriz-vector con C

```
1 // === FUNCTION ======================================================================
2 // Name: c_sgemv
3 // Description: Realiza el producto matriz vector, de precisión simple (float), operando
4 // con una matriz general. y = α × A × x + β × y
5 // ====================================================================================
6
7 void
8 c_sgemv(
9   const  Trans Trans,  // determina si se opera con A o con AT (transpuesta)
10  const int m,  // número de filas de la matriz A
11  const int n,  // número de columnas de la matriz A
12  const float alpha,  // escalar
13  const Matriz *A,  // matriz de m*n elementos
14  const int lda,  // dimensión de A
15  const Vector *x,  // vector de n elementos
16  const int incX,  // espacio de almacenamiento entre los elementos de x
17  const float beta,  // escalar
18  Vector *y,  // vector de n elementos
19  const int incY // espacio de almacenamiento entre los elementos de y
20 )
21 {
22  int lenX, lenY, i, j, iX, iY;
23  lenX = lenY = i = j = iX = iY = 0;
24  float temp;
25  temp = 0.0f;
26
27  // si alguno de los siguientes argumentos no cumple la condición, sale de la rutina
28  if ( m != 0 || n != 0 || (alpha != 0.0 && beta != 1.0) ) {
29    if ( Trans == noTrans ) {
30      lenX = n;
31      lenY= m;
32    }
33    else {
34      lenX = m;
35      lenY = n;
36    }
37    // Realiza la operación y = β × y
38    if ( beta != 1 ) {
39      if ( incY == 1 ) {
40        if ( beta == 0.0  ) {
41          for ( i = 0; i < lenY; i += 1 ) {
42            // y(i) = 0
43            setVectorValueAt(y, i, 0.0f);
44          }
45        }
46        else {
47          for ( i = 0; i < lenY; i += 1 ) {
48            // y(i) *= beta
49            setVectorValueAt(y, i, getVectorValueAt(y, i)*beta);
50          }
51        }
52      }
53      else {
54        if ( beta == 0.0 ) {
55          for ( iY = 0; iY < lenY; iY += incY ) {
56            // y(iY) = 0
57            setVectorValueAt(y, iY, 0.0f);
58          }
59        }
60        else {
61          for ( iY = 0; iY < lenY; iY += incY ) {
62            // y(iY) *= beta
63            setVectorValueAt(y, iY, getVectorValueAt(y, iY)*beta);
64          }
65        }
66      }
67    }
68    // Realiza la operación α × A × x + y
69    iX = iY = 0;
70    if ( Trans == trans ) {
71      if ( incY == 1 ) {
72        for ( j = 0; j < n; j += 1 ) {
73          // si x(iX) != 0
```

```
74            if ( getVectorValueAt(x, iX) != 0.0f ) {
75              // temp = alpha * x(iX)
76              temp = alpha * getVectorValueAt(x, iX);
77              for ( i = 0; i < m; i += 1 ) {
78                //y(i) += temp * A(i,j)
79                setVectorValueAt(y, i, getVectorValueAt(y, i) + temp * getMatrizValueAt(A, i, j));
80              }
81            }
82            iX += incX;
83          }
84        }
85        else {
86          for ( j = 0; j < n; j += 1 ) {
87            if ( getVectorValueAt(x, iX) != 0.0f ) { // si x(iX) != 0
88              // temp = alpha * x(iX)
89              temp = alpha * getVectorValueAt(x, iX);
90              iY = 0;
91              for ( i = 0; i < m; i += 1 ) {
92                // y(iY) += temp * A(i,j)
93                setVectorValueAt(y, iY, getVectorValueAt(y, iY) + temp * getMatrizValueAt(A, i, j));
94                iY += incY;
95              }
96            }
97            iX += incX;
98          }
99        }
100     }
101     else {
102       // Realiza la operación alpha * A^T * x + y
103       if ( incX == 1 ) {
104         for ( j = 0; j < n; j += 1 ) {
105           temp = 0.0f;
106           for ( i = 0; i < m; i += 1 ) {
107             // temp += A(i,j) * x(i)
108             temp += getMatrizValueAt(A, i, j) * getVectorValueAt(x, i);
109           }
110           // y(iY) += alpha * temp
111           setVectorValueAt(y, iY, getVectorValueAt(y, iY) + alpha * temp);
112           iY += incY;
113         }
114       }
115       else {
116         for ( j = 0; j < n; j += 1 ) {
117           temp = 0.0f;
118           iX = 0;
119           for ( i = 0; i < m; i += 1 ) {
120             // temp += A(i,j) * x(iX)
121             temp += getMatrizValueAt(A, i, j) * getVectorValueAt(x, iX);
122             iX += incX;
123           }
124           // y(iY) += alpha * temp
125           setVectorValueAt(y, iY, getVectorValueAt(y, iY) + alpha * temp);
126           iY += incY;
127         }
128       }
129     }
130   }
131 }  // --- end of function c_sgemv ---
```

En el siguiente programa se prueba la operación gemv (producto matriz-vector) con C y BLAS, para compilar el mismo se hace con la siguiente sentencia:

```
gcc src/vec.c src/mat.c src/c.c test/gemv.c -o bin/gemv_cpu -I include/ -lblas
```

donde (para ver la descripción de todos los elementos de las sentencias de Compilación y ejecución ver página 47).

test/gemv.c	prueba gemv con C y BLAS.
bin/gemv_cpu	ejecutable generado.

Para ejecutar el programa que prueba el producto matriz-vector con C y BLAS se hace con la siguiente sentencia:

```
./bin/gemv_cpu 50 0.36 2.47
```

donde

bin/gemv_cpu	archivo ejecutable que esta dentro del directorio bin.
50	argumento que recibe el programa, indica el tamaño vector y la dimensión de la matriz (ver programa 14 línea 36).
0.36	argumento que recibe el programa, indica el escalar alpha (ver programa 14 línea 37).
2.47	argumento que recibe el programa, indica el escalar beta (ver programa 14 línea 38).

Programa 14: Prueba producto matriz-vector con C y BLAS

```c
// ================================================================================
// Filename: gemv.c
//
// Description: pueba la rutina gemv con C y BLAS que realiza el producto
// matriz-vector de la forma $\alpha \times A \times x + \beta \times y$
// Compiler: gcc
// ================================================================================


#include  <stdlib.h>
#include  <stdio.h>

#include  <cblas.h>  // rutinas con BLAS


#include  "vec.h"  // funciones que manipulan vectores
#include  "mat.h"  // funciones que manipulan matrices
#include  "c.h"  // funciones de matrices y vectores con C

// === FUNCTION ===================================================================
// Name: main
// Description: Prueba la rutina gemv con C y BLAS que realiza el producto
// matriz-vector operando con una matriz general $y = \alpha \times A \times x + \beta \times y$
// ================================================================================

int
main (
  int argc,  // contador de argumentos
  char *argv[]  // valor de argumentos
)
{
  if ( argc != 4 ) {
    printf ( "Argumentos: n\talpha\tbeta\n" );
    exit(EXIT_FAILURE);
  }
  // toma argumentos de la sentencia de ejecución
  int n = atoi(argv[1]);  // número de elementos de los vectores de entrada
  float alpha = atof(argv[2]);  // escalar
  float beta = atof(argv[3]);  // escalar

  // matrices para realizar operaciones
  Matriz Ac;  // para C
  Matriz Ab;  // para BLAS

  // vectores para realizar operaciones
  Vector xc, yc;  // para C
  Vector xb, yb;  // para BLAS

  // reserva memoria en CPU
  allocateMatriz(&Ac, n, n);
  allocateMatriz(&Ab, n, n);

  allocateVector(&xc, n);
  allocateVector(&yc, n);
  allocateVector(&xb, n);
  allocateVector(&yb, n);

  // llena las matrices y los vectores con números aleatorios
  llenaMatriz(&Ac);
  llenaMatriz(&Ab);

  llenaVector(&xc);
  llenaVector(&yc);
  llenaVector(&xb);
  llenaVector(&yb);

  // realiza el producto matriz vector
  c_sgemv(noTrans, n, n, alpha, &Ac, n, &xc, 1, beta, &yc, 1);  // con C
  cblas_sgemv(CblasRowMajor, CblasNoTrans, n, n, alpha, Ab.data, n,
      xb.data, 1, beta, yb.data, 1);  // con BLAS

  // imprime resultado
  imprimeVector(&yc);  // con C
  imprimeVector(&yb);  // con BLAS

  // libera memoria
  freeMatriz(&Ac);
  freeMatriz(&Ab);

  freeVector(&xc);
```

```
81   freeVector(&yc);
82   freeVector(&xb);
83   freeVector(&yb);
84
85   // finaliza programa
86   return EXIT_SUCCESS;
87 }  // ------- end of function main -------
```

Programa 15: Producto matriz-matriz con C

```
1 // === FUNCTION ======================================================================
2 // Name: c_sgemm
3 // Description: Realiza el producto matriz matriz, de precisión simple (float),
4 // operando con matrices generales. C = α × A × B + β × C
5 // ====================================================================================
6
7 void
8 c_sgemm (
9   const  Trans transA,  // determina si se opera con A o con AT (transpuesta)
10  const  Trans transB,  // determina si se opera con B o con BT (transpuesta)
11  const int m,  // número de filas de la matriz A y de la matriz C
12  const int n,  // número de columnas de la matriz A y de la matriz C
13  const int k,  // número de columnas de la matriz A y filas de la matriz B
14  const float alpha,  // escalar
15  const Matriz *A,  // matriz de dimensión lda*k si A y lda*m si AT
16  const int lda,  // dimensión de A
17  const Matriz *B,  // matriz de dimensión ldb*n si B y ldb*k si BT
18  const int ldb,  // dimensión de B
19  const float beta,  // escalar
20  Matriz *C,  // matriz de dimensión ldb*n
21    const int ldc  // dimensión de C
22 )
23 {
24  float temp;
25  int nrowA, ncolA, nrowB, i, j, l;
26  nrowA = ncolA = nrowB = i = j = l = 0;
27  if ( transB == noTrans )
28    nrowB = k;
29  else
30    nrowB = n;
31  // Realiza la operación β × C
32  if ( alpha == 0.0 ) {
33    if ( beta == 0.0 ) { // Si alpha y beta == 0, C(i,j)=0
34      for ( i = 0; i < n; i += 1 ) {
35        for ( j = 0; j < m; j += 1 ) {
36          // C(i,j) = 0
37          setMatrizValueAt(C, i, j, 0.0f);
38        }
39      }
40    }
41    else {
42      for ( i = 0; i < n; i += 1 ) {
43        for ( j = 0; j < m; j += 1 ) {
44          // C(i,j) = beta * C(i,j)
45          setMatrizValueAt(C, i, j,
46            beta * getMatrizValueAt(C, i, j));
47        }
48      }
49    }
50  }
51  // Realiza la operación α × A × B + β × C
52  if ( transB == noTrans ) {
53    if ( transA == noTrans ) {
54      for ( j = 0; j < n; j += 1 ) {
55        if ( beta == 0.0 ) {
56          for ( i = 0; i < m; i += 1 ) {
57            // C(i,j) = 0
58            setMatrizValueAt(C, i, j, 0.0f);
59          }
60        }
61        else if ( beta != 1.0f ) {
62          for ( i = 0; i < m; i += 1 ) {
63            // C(i,j) = beta * C(i,j)
64            setMatrizValueAt(C, i, j,
65              beta * getMatrizValueAt(C, i, j));
66          }
67        }
68        for ( l = 0; l < k; l += 1 ) {
69          if ( getMatrizValueAt(B, l, j) != 0.0  ) { // Si B(l,j) != 0
70            // temp = alpha * B(l,j)
```

```
71            temp = alpha * getMatrizValueAt(B, l, j);
72            for ( i = 0; i < m; i += 1 ) {
73              // C(i,j) = C(i,j) + temp * A(i,l)
74              setMatrizValueAt(C, i, j,
75                getMatrizValueAt(C, i, j) +
76                temp * getMatrizValueAt(A, i, l));
77            }
78          }
79        }
80      }
81    }
82    else {  // Realiza la operación alpha × A^T × B + β × C
83      for ( j = 0; j < n; j += 1 ) {
84        for ( i = 0; i < m; i += 1 ) {
85          temp = 0.0f;
86          for ( l = 0; l < k; l += 1 ) {
87            // temp += A(l,i) * B(l,j)
88            temp += getMatrizValueAt(A, l, i) *
89              getMatrizValueAt(B, l, j);
90          }
91          if ( beta == 0.0 ) {
92            // C(i,j) = alpha * temp
93            setMatrizValueAt(C, i, j,
94              alpha * temp);
95          }
96          else {
97            // C(i,j) = alpha * temp + beta * C(i,j)
98            setMatrizValueAt(C, i, j,
99              alpha * temp + beta * getMatrizValueAt(C, i, j));
100         }
101       }
102     }
103   }
104 }
105 else { // Realiza la operación α × A × B^T + β × C
106   if ( transA == noTrans ) {
107     for ( j = 0; j < n; j += 1 ) {
108       if ( beta == 0.0 ) {
109         for ( i = 0; i < m; i += 1 ) {
110           // C(i,j) = 0
111           setMatrizValueAt(C, i, j, 0.0f);
112         }
113       }
114       else if ( beta != 1.0 ) {
115         for ( i = 0; i < m; i += 1 ) {
116           // C(i,j) = beta * C(i,j)
117           setMatrizValueAt(C, i, j,
118             beta * getMatrizValueAt(C, i, j ));
119         }
120       }
121       for ( l = 0; l < k; l += 1 ) {
122         if ( getMatrizValueAt(B, j, l) != 0.0 ) { // Si B(j,l) != 0
123           // temp = alpha * B(j,l)
124           temp = alpha  * getMatrizValueAt(B, j, l);
125           for ( i = 0; i < m; i += 1 ) {
126             // C(i,j) = C(i,j) + temp * A(i,l)
127             setMatrizValueAt(C, i, j,
128               getMatrizValueAt(C, i, j) + temp *
129               getMatrizValueAt(A, i, l));
130           }
131         }
132       }
133     }
134   }
135   else { // Realiza la operación alpha × A^T × B^T + beta × C
136     for ( j = 0; j < n; j += 1 ) {
137       for ( i = 0; i < m; i += 1 ) {
138         temp = 0.0f;
139         for ( l = 0; l < k; l += 1 ) {
140           // temp += A(l,i) * B(j,l)
141           temp += getMatrizValueAt(A, l, i) *
142             getMatrizValueAt(B, j, l);
143         }
144         if ( beta == 0.0 ) {
145           // C(i,j) = alpha * temp
146           setMatrizValueAt(C, i, j, alpha * temp);
147         }
148         else {
149           // C(i,j) = alpha * temp + beta * C(i,j)
150           setMatrizValueAt(C, i, j, alpha * temp +
151             beta * getMatrizValueAt(C, i, j));
```

```
152             }
153           }
154         }
155       }
156     }
157 }   // --- end of function c_sgemm ---
```

En el siguiente programa se prueba la operación gemv (producto matriz-matriz) con C y BLAS, para compilar el mismo se hace con la siguiente sentencia:

```
gcc src/vec.c src/mat.c src/c.c test/gemm.c -o bin/gemm_cpu -I include/ -lblas
```

donde (para ver la descripción de todos los elementos de las sentencias de Compilación y ejecución ver página 47).

test/gemm.c	prueba gemm con C y BLAS.
bin/gemm_cpu	ejecutable generado.

Para ejecutar el programa que prueba el producto matriz-matriz con C y BLAS se hace con la siguiente sentencia:

```
./bin/gemm_cpu 1000 15.65 0.056
```

donde

bin/gemm_cpu	archivo ejecutable que esta dentro del directorio bin.
1000	argumento que recibe el programa, indica la dimensión de la matriz (ver programa 16 entre línea 34).
15.65	argumento que recibe el programa, indica el escalar alpha (ver programa 16 línea 35).
0.056	argumento que recibe el programa, indica el escalar beta (ver programa 16 línea 36).

Programa 16: Prueba producto matriz-matriz con C y BLAS

```
1  // ================================================================================
2  // Filename: gemm.c
3  //
4  // Description: pueba la rutina gemm con C y BLAS que realiza el producto
5  // matriz-matriz de la forma α × A × B + β × C
6  // Compiler: gcc
7  // ================================================================================
8
9
10 #include  <stdlib.h>
11 #include  <stdio.h>
12
13 #include  <cblas.h>  // rutinas con BLAS
14
15 #include  "mat.h"  // funciones que manipulan matrices
16 #include  "c.h"  // funciones de matrices y vectores con C
17
18 // === FUNCTION ===============================================================
19 // Name: main
20 // Description: Prueba la rutina gemm con C y BLAS que realiza el producto matriz-matriz
21 // operando con matrices generales C = α × A × B + β × C
22 // ================================================================================
23
24 int
25 main (
26   int argc,  // contador de argumentos
27   char *argv[]  // valor de argumentos
28 )
29 {
30   if ( argc != 4 ) {
31     printf ( "Argumentos: n\talpha\tbeta\n" );
32     exit(EXIT_FAILURE);
33   }
34   // toma argumentos de la sentencia de ejecución
35   int n = atoi(argv[1]);  // número de elementos de los vectores de entrada
36   float alpha = atof(argv[2]);  // escalar
37   float beta = atof(argv[3]);  // escalar
38
39   // matrices para realizar operaciones
40   Matriz Ac, Bc, Cc;  // para C
41   Matriz Ab, Bb, Cb;  // para BLAS
42
43   // reserva memoria en CPU
44   allocateMatriz(&Ac, n, n);
45   allocateMatriz(&Bc, n, n);
46   allocateMatriz(&Cc, n, n);
47
48   allocateMatriz(&Ab, n, n);
49   allocateMatriz(&Bb, n, n);
50   allocateMatriz(&Cb, n, n);
51
52   // llena las matrices y los vectores con números aleatorios
```

```
53   llenaMatriz(&Ac);
54   llenaMatriz(&Bc);
55   llenaMatriz(&Cc);
56
57   llenaMatriz(&Ab);
58   llenaMatriz(&Bb);
59   llenaMatriz(&Cb);
60
61   // realiza la el producto matriz matriz
62   c_sgemm(noTrans, noTrans, n, n, n, alpha, &Ac, n, &Bc, n, beta, &Cc, n );  // con C
63   cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, n, n, alpha,
64       Ab.data, n, Bb.data, n, beta, Cb.data, n);  // con BLAS
65
66   // imprime resultado
67   imprimeMatriz(&Cc);  // con C
68   imprimeMatriz(&Cb);  // con BLAS
69
70   // libera memoria
71   freeMatriz(&Ac);
72   freeMatriz(&Bc);
73   freeMatriz(&Cc);
74
75   freeMatriz(&Ab);
76   freeMatriz(&Bb);
77   freeMatriz(&Cb);
78
79   // finaliza programa
80   return EXIT_SUCCESS;
81 }  // ------- end of function main -------
```

6.1.4. Implementación, Compilación y Ejecución de programas con CUDA y CUBLAS

En este apartado se muestran las implementaciones de operaciones básicas de matrices y vectores con CUDA y CUBLAS: la suma vector-vector con CUDA (programa 17) y con CUBLAS (programa 18), el producto matriz-vector con CUDA (programa 20) y con CUBLAS (programa 21), y el producto matriz-matriz con CUDA (programa 23) y con CUBLAS (programa 24).

También se presentan los programas que prueban tanto los programas de CUDA como los de CUBLAS, y son: programa 19, prueba la suma vector-vector con CUDA y CUBLAS; programa 22, prueba producto matriz-vector con CUDA y CUBLAS; y programa 25, prueba el producto matriz-matriz con CUDA y CUBLAS.

Se decidió juntar los programas de CUDA y CUBLAS ya que usan el mismo compilador (nvcc).

Programa 17: Suma vector-vector con CUDA

```
1 // === FUNCTION ======================================================================
2 // Name: cuda_saxpy
3 // Description: Suma vector vector, de precisión simple (float).
4 // y = α × x + y
5 // ============================================================================================
6
7 __global__ void
8 saxpy(
9   const int n,  // número de elementos del vector de entrada
10  const float alpha,  // escalar
11  const float *x,  // vector de n elementos
12  float *y  // vector de n elementos
13 )
14 {
15  int i = blockIdx.x * blockDim.x + threadIdx.x;
16  if ( i < n ) {
17    y[i] = alpha*x[i] + y[i];
18  }
19
20 }
21
22 void
23 cuda_saxpy(
24  const int n,  // número de elementos del vector de entrada
25  const float alpha,  // escalar
26  const Vector *x,  // vector de n elementos
27  Vector *y  // vector de n elementos
28 )
29 {
30  // calcula los bytes que x e y ecuparán en memoria
31  size_t size = n*sizeof(float);
32
33  // reserva memoria en GPU
34  float *dx; cudaMalloc(&dx, size);
35  float *dy; cudaMalloc(&dy, size);
```

```
36
37   // copia el vector de CPU a GPU
38   cudaMemcpy(dx, x->data, size, cudaMemcpyHostToDevice);
39   cudaMemcpy(dy, y->data, size, cudaMemcpyHostToDevice);
40
41   // calcula los bloques que serán necesarios la operación
42   int hilosPorBloque = 512;
43   int bloquesPorMalla = (n + hilosPorBloque - 1) / hilosPorBloque;
44
45   // invoca kernel
46   saxpy<<<hilosPorBloque, bloquesPorMalla>>>(n, alpha, dx, dy);
47
48   // copia los vectores de GPU a CPU
49   cudaMemcpy(y->data, dy, size, cudaMemcpyDeviceToHost);
50
51   // libera memoria
52   cudaFree(dx);
53   cudaFree(dy);
54 }  // --- end of function cuda_saxpy ---
```

Programa 18: Suma vector-vector con CUBLAS

```
1 // === FUNCTION ======================================================================
2 // Name: cublas_saxpy
3 // Description: Suma vector vector, de precisión simple (float).
4 // y = α × x + y
5 // ====================================================================================
6
7 void
8 cublas_saxpy (
9   const int n,  // número de elementos del vector de entrada
10  const float alpha,  // escalar
11  const Vector *x,  // vector de n elementos
12  const int incX,  // espacio de almacenamiento entre los elementos de x
13  Vector *y,  // vector de n elementos
14  const int incY  // espacio de almacenamiento entre los elementos de y
15 )
16 {
17   // vectores para GPU
18   float *dx;
19   float *dy;
20
21   // controla el estado de CUBLAS
22   cublasStatus status;
23
24   // inicia CUBLAS
25   status = cublasInit();
26   check_cublas_status("inicia", status);
27
28   // reserva memoria en GPU
29   status = cublasAlloc(n, sizeof(float), (void **)&dx);
30   check_cublas_status("alloc dx", status);
31   status = cublasAlloc(n, sizeof(float), (void **)&dy);
32   check_cublas_status("alloc dy", status);
33
34   // copia los vectores de CPU a GPU
35   status = cublasSetVector(n, sizeof(float), x->data, 1, dx, 1 );
36   check_cublas_status("set dx", status);
37   status = cublasSetVector(n, sizeof(float), y->data, 1, dy, 1 );
38   check_cublas_status("set dy", status);
39
40   // realiza la suma de dos vectores
41   cublasSaxpy(n, alpha, dx, incX, dy, incY);
42   status = cublasGetError();
43   check_cublas_status("saxpy", status);
44
45   // copia vector de GPU a CPU
46   status = cublasGetVector(n, sizeof(float), dy, 1, y->data, 1);
47   check_cublas_status("get dy", status);
48
49   // libera memoria
50   status = cublasFree(dx);
51   check_cublas_status("free dx", status);
52   status = cublasFree(dy);
53   check_cublas_status("free dy", status);
54
55   // cierra CUBLAS
56   status = cublasShutdown();
57   check_cublas_status("cerrar", status);
58 }  // --- end of function cublas_saxpy ---
```

En el siguiente programa se prueba la operación axpy (suma vector-vector) con CUDA y CUBLAS, para compilar el mismo se hace con la siguiente sentencia:

```
nvcc src/vec.cu src/mat.cu src/cuda.cu src/cubla.cu test/axpy.cu -o bin/axpy_gpu -I include/ -lcublas
```

donde [8]

nvcc	es el compilador de NVIDIA para CUDA.
src/vec.cu	funciones para manipular vectores.
src/mat.cu	funciones para manipular matrices.
src/cuda.cu	funciones de matrices y vectores con CUDA.
src/cubla.cu	funciones de matrices y vectores con CUBLAS.
test/axpy.cu	prueba axpy con CUDA y CUBLAS.
-o	le indica al compilador que genere un ejecutable axpy_gpu y lo guarde en el directorio bin.
bin/axpy_gpu	ejecutable generado.
-I	le indica al compilador que debe considerar archivos de cabecera creados por el usuario (directorio include).
include/	directorio donde están los archivos de cabecera.
-lcublas	biblioteca libcublas.a para compilar CUBLAS.

Para ejecutar el programa que prueba la suma vector-vector con CUDA y CUBLAS se hace con la siguiente sentencia:

```
./bin/axpy_gpu 15 3.6
```

donde

./	le indica al sistema que se trata de un archivo ejecutable dentro del directorio actual
bin/axpy_gpu	archivo ejecutable que esta dentro del directorio bin.
15	argumento que recibe el programa, indica el tamaño del vector (ver programa 19 línea 34).
3.6	argumento que recibe el programa, indica el escalar (ver programa 19 línea 35).

Programa 19: Prueba suma vector-vector con CUDA y CUBLAS

```
1  // ================================================================================
2  // Filename: axpy.cu
3  //
4  // Description: pueba la rutina axpy con CUDA y CUBLAS que realiza la suma de dos vectores
5  // y = α × x + y
6  // Compiler: nvcc
7  // ================================================================================
8
9
10 #include  <stdlib.h>
11 #include  <stdio.h>
12
13 #include  "vec.h"  // funciones que manipulan vectores
14 #include  "cuda.hcu"  // rutinas con cuda
15 #include  "cubla.h"  // rutinas con cublas
16
17 // === FUNCTION ======================================================================
18 // Name: main
19 // Description: Prueba la rutina axpy con CUDA y CUBLAS que realiza la suma de dos
20 // vectores. y = α × x + y
21 // ================================================================================
22
23 int
24 main (
25   int argc,  // contador de argumentos
26   char *argv[]  // valor de argumentos
27 )
28 {
29   if ( argc != 3 ) {
30     printf ( "Argumentos: n\talpha\n" );
31     exit(EXIT_FAILURE);
32   }
33
34   // toma argumentos de la sentencia de ejecución
35   int n = atoi(argv[1]);  // número de elementos de los vectores de entrada
36   float alpha = atof(argv[2]);  // escalar
37
38   // vectores para realizar operaciones
39   Vector xc, yc;  // para C
```

[8] En la descripción de las sentencias de compilación y de ejecución de los siguientes programas de prueba se omite lo que es común con éstas, como son: compilador, archivos, bibliotecas, etc.

```
  Vector xb, yb;  // para BLAS

  // reserva memoria en CPU
  allocateVector(&xc, n);
  allocateVector(&yc, n);
  allocateVector(&xb, n);
  allocateVector(&yb, n);

  // llena los vectores con números aleatorios
  llenaVector(&xc);
  llenaVector(&yc);
  llenaVector(&xb);
  llenaVector(&yb);

  // realiza la suma de vectoes
  cuda_saxpy(n, alpha, &xc, &yc);  // con CUDA
  cublas_saxpy(n, alpha, &xb, 1, &yb, 1);  // con BLAS

  // imprime resultado
  imprimeVector(&yc);  // con C
  imprimeVector(&yb);  // con BLAS

  // libera memoria
  freeVector(&xc);
  freeVector(&yc);
  freeVector(&xb);
  freeVector(&yb);

  // finaliza programa
  return EXIT_SUCCESS;
}  // ------- end of function main -------
```

Programa 20: Producto matriz-vector con CUDA

```
// === FUNCTION ======================================================================
// Name: cuda_sgemv
// Description: Realiza el producto matriz vector, de precisión simple (float), operando
// con una matriz general. y = α × A × x + β × y
// ===================================================================================

__global__ void
sgemv(
  const int m,  // número de filas de la matriz A
  const int n,  // número de columnas de la matriz A
  const float alpha,  // escalar
  const float *A,  // matriz de m*n elementos
  const float *x,  // vector de n elementos
  const float beta,  // escalar
  float *y  // vector de n elementos
)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float temp = 0.0;
  if ( i < n ) {
    for (int a = 0; a < m; a++)
      temp += alpha * A[a+n*i] * x[a];
    y[i] = temp + beta * y[i];
  }
}

void
cuda_sgemv(
  const int m,  // número de filas de la matriz A
  const int n,  // número de columnas de la matriz A
  const float alpha,  // escalar
  const Matriz *A,  // matriz de m*n elementos
  const Vector *x,  // vector de n elementos
  const float beta,  // escalar
  Vector *y  // vector de n elementos
)
{
  // calcula los bytes que A, x e y ecuparán en memoria
  size_t sizev = n*sizeof(float);
  size_t sizem = m*n*sizeof(float);

  // reserva memoria en GPU
  float *dA; cudaMalloc(&dA, sizem);
  float *dx; cudaMalloc(&dx, sizev);
  float *dy; cudaMalloc(&dy, sizev);

```

```
47   // copia la matriz y los vectores de CPU a GPU
48   cudaMemcpy(dA, A->data, sizem, cudaMemcpyHostToDevice);
49   cudaMemcpy(dx, x->data, sizev, cudaMemcpyHostToDevice);
50   cudaMemcpy(dy, y->data, sizev, cudaMemcpyHostToDevice);
51
52   // calcula los bloques que serán necesarios la operación
53   int hilosPorBloque = 512;
54   int bloquesPorMalla = (n + hilosPorBloque - 1) / hilosPorBloque;
55
56   // invoca kernel
57   sgemv<<<hilosPorBloque, bloquesPorMalla>>>(m, n, alpha, dA, dx, beta, dy);
58
59   // copia el vector de GPU a CPU
60   cudaMemcpy(y->data, dy, sizev, cudaMemcpyDeviceToHost);
61
62   // libera memoria
63   cudaFree(dA);
64   cudaFree(dx);
65   cudaFree(dy);
66 }  // --- end of function cuda_sgemv ---
```

Programa 21: Producto matriz-vector con CUBLAS

```
1  // === FUNCTION ======================================================================
2  // Name: cublas_sgemv
3  // Description: Realiza el producto matriz vector, de precisión simple (float), operando
4  // con una matriz general. y = α × A × x + β × y
5  // ===================================================================================
6
7  void
8  cublas_sgemv (
9    const  char trans,  // determina si se opera con A o con AT (transpuesta)
10   const int m,  // número de filas de la matriz A
11   const int n,  // número de columnas de la matriz A
12   const float alpha,  // escalar
13   const Matriz *A,  // matriz de m*n elementos
14   const int lda,  // dimensión de A
15   const Vector *x,  // vector de n elementos
16   const int incX,  // espacio de almacenamiento entre los elementos de x
17   const float beta,  // escalar
18   Vector *y,  // vector de n elementos
19   const int incY  // espacio de almacenamiento entre los elementos de y
20 )
21 {
22   // matriz y vectores para GPU
23   float *dA;
24   float *dx;
25   float *dy;
26
27   // controla el estado de CUBLAS
28   cublasStatus status;
29
30   // inicia CUBLAS
31   status = cublasInit();
32   check_cublas_status("inicia", status);
33
34   // calcula las dimensiones del A, x e y
35   int elemA, elemx, elemy;
36
37   if ( trans == 'N' || trans == 'n' ) {// A
38     elemA = lda*n;
39     elemx = 1 + (n-1)*abs(incX);
40     elemy = 1 + (m-1)*abs(incY);
41   }
42   else {// AT
43     elemA = lda*m;
44     elemx = 1 + (m-1)*abs(incX);
45     elemy = 1 + (n-1)*abs(incY);
46   }
47
48   // reserva memoria en GPU
49   status = cublasAlloc(elemA, sizeof(float), (void **)&dA);
50   check_cublas_status("alloc dA", status);
51   status = cublasAlloc(elemx, sizeof(float), (void **)&dx);
52   check_cublas_status("alloc dx", status);
53   status = cublasAlloc(elemy, sizeof(float), (void **)&dy);
54   check_cublas_status("alloc dy", status);
55
56   // copia la matriz y los vectores de CPU a GPU
57   status = cublasSetMatrix(m, n, sizeof(float), A->data, m, dA, m );
```

```
58   check_cublas_status("set d_A", status);
59   status = cublasSetVector(elemx, sizeof(float), x->data, 1, dx, 1 );
60   check_cublas_status("set dx", status);
61   status = cublasSetVector(elemy, sizeof(float), y->data, 1, dy, 1 );
62   check_cublas_status("set dy", status);
63
64   // realiza el producto matriz vector
65   cublasSgemv(trans, m, n, alpha, dA, lda, dx, incX, beta, dy, incY);
66   status = cublasGetError();
67   check_cublas_status("sgemv", status);
68
69   // copia vector de GPU a CPU
70   status = cublasGetVector(n, sizeof(float), dy, 1, y->data, 1);
71   check_cublas_status("get dC", status);
72
73   // libera memoria
74   status = cublasFree(dA);
75   check_cublas_status("free dA", status);
76   status = cublasFree(dx);
77   check_cublas_status("free dx", status);
78   status = cublasFree(dy);
79   check_cublas_status("free dy", status);
80
81   // cierra CUBLAS
82   status = cublasShutdown();
83   check_cublas_status("cerrar", status);
84 }  // --- end of function cublas_sgemv ---
```

En el siguiente programa se prueba la operación gemv (producto matriz-vector) con CUDA y CUBLAS, para compilar el mismo se hace con la siguiente sentencia:

```
nvcc src/vec.cu src/mat.cu src/cuda.cu src/cubla.cu test/gemv.cu -o bin/gemv_gpu -I include/ -lcublas
```

donde (para ver la descripción de todos los elementos de las sentencias de Compilación y ejecución ver página 57).

test/gemv.cu	prueba gemv con CUDA y CUBLAS.
bin/gemv_gpu	ejecutable generado.

Para ejecutar el programa que prueba el producto matriz-vector con CUDA y CUBLAS se hace con la siguiente sentencia:

```
./bin/gemv_gpu 50 0.36 2.47
```

donde

bin/gemv_gpu	archivo ejecutable que esta dentro del directorio bin.
50	argumento que recibe el programa, indica el tamaño vector y la dimensión de la matriz (ver programa 22 línea 35).
0.36	argumento que recibe el programa, indica el escalar alpha (ver programa 22 línea 36).
2.47	argumento que recibe el programa, indica el escalar beta (ver programa 22 línea 37).

Programa 22: Prueba producto matriz-vector con CUDA y CUBLAS

```
1  // ============================================================================
2  // Filename: gemv.cu
3  //
4  // Description: pueba la rutina gemv con CUDA y CUBLAS que realiza el producto
5  // matriz-vector de la forma α × A × x + β × y
6  // Compiler: nvcc
7  // ============================================================================
8
9
10 #include  <stdlib.h>
11 #include  <stdio.h>
12
13 #include  "vec.h"  // funciones que manipulan vectores
14 #include  "mat.h"  // funciones que manipulan matrices
15 #include  "cuda.hcu"  // rutinas con cuda
16 #include  "cubla.h"  // rutinas con cublas
17
18 // === FUNCTION ===============================================================
19 // Name: main
20 // Description: Prueba la rutina gemv con CUDA y CUBLAS que realiza el producto
21 // matriz-vector operando con una matriz general y = α × A × x + β × y
22 // ============================================================================
23
24 int
25 main (
26   int argc,  // contador de argumentos
27   char *argv[]  // valor de argumentos
28 )
29 {
```

```
30  if ( argc != 4 ) {
31    printf ( "Argumentos: n\talpha\tbeta\n" );
32    exit(EXIT_FAILURE);
33  }
34  // toma argumentos de la sentencia de ejecución
35  int n = atoi(argv[1]);  // número de elementos de los vectores de entrada
36  float alpha = atof(argv[2]);  // escalar
37  float beta = atof(argv[3]);  // escalar
38
39  // matrices para realizar operaciones
40  Matriz Ac;  // para CUDA
41  Matriz Ab;  // para CUBLAS
42
43  // vectores para realizar operaciones
44  Vector xc, yc;  // para CUDA
45  Vector xb, yb;  // para CUBLAS
46
47  // reserva memoria en CPU
48  allocateMatriz(&Ac, n, n);
49  allocateMatriz(&Ab, n, n);
50
51  allocateVector(&xc, n);
52  allocateVector(&yc, n);
53  allocateVector(&xb, n);
54  allocateVector(&yb, n);
55
56  // llena las matrices y los vectores con números aleatorios
57  llenaMatriz(&Ac);
58  llenaMatriz(&Ab);
59
60  llenaVector(&xc);
61  llenaVector(&yc);
62  llenaVector(&xb);
63  llenaVector(&yb);
64
65  // realiza el producto matriz vector
66  cuda_sgemv(n, n, alpha, &Ac, &xc, beta, &yc);  // con CUDA
67  cublas_sgemv('T', n, n, alpha, &Ab, n, &xb, 1, beta, &yb, 1);  // con CUBLAS
68
69  // imprime resultado
70  imprimeVector(&yc);  // con CUDA
71  imprimeVector(&yb);  // con CUBLAS
72
73  // libera memoria
74  freeMatriz(&Ac);
75  freeMatriz(&Ab);
76
77  freeVector(&xc);
78  freeVector(&yc);
79  freeVector(&xb);
80  freeVector(&yb);
81
82  // finaliza programa
83  return EXIT_SUCCESS;
84 }  // ------- end of function main -------
```

Programa 23: Producto matriz-matriz con CUDA

```
1 // === FUNCTION ======================================================================
2 // Name: cuda_sgemm
3 // Description: Realiza el producto matriz matriz, de precisión simple (float),
4 // operando con matrices generales. C = α × A × B + β × C
5 // ====================================================================================
6
7 __global__ void
8 sgemm(
9   const int n,  // número de columnas de la matriz A y de la matriz C
10  const float alpha,  // escalar
11  const float *A,  // matriz de dimensión lda*k si A y lda*m si AT
12  const float *B,  // matriz de dimensión ldb*n si B y ldb*k si BT
13  const float beta,  // escalar
14  float *C  // matriz de dimensión ldb*n
15 )
16 {
17  int row = blockIdx.y * blockDim.y + threadIdx.y;
18  int col = blockIdx.x * blockDim.x + threadIdx.x;
19  float temp = 0.0f;
20  for (int i = 0; i < n; ++i)
21    temp += alpha * A[row*n+i] * B[i*n+col];
22  C[row*n+col] = temp + beta * C[row*n+col];
```

```
23 }
24
25 void
26 cuda_sgemm(
27   const int n,  // número de columnas de la matriz A y de la matriz C
28   const float alpha,  // escalar
29   const Matriz *A,  // matriz de dimensión lda*k si A y lda*m si AT
30   const Matriz *B,  // matriz de dimensión ldb*n si B y ldb*k si BT
31   const float beta,  // escalar
32   Matriz *C  // matriz de dimensión ldb*n
33 )
34 {
35   // calcula los bytes que A, B y C ecuparán en memoria
36   size_t size = n*n*sizeof(float);
37
38   // reserva memoria en GPU
39   float *dA; cudaMalloc(&dA, size);
40   float *dB; cudaMalloc(&dB, size);
41   float *dC; cudaMalloc(&dC, size);
42
43   // copia las matrices de CPU a GPU
44   cudaMemcpy(dA, A->data, size, cudaMemcpyHostToDevice);
45   cudaMemcpy(dB, B->data, size, cudaMemcpyHostToDevice);
46   cudaMemcpy(dC, C->data, size, cudaMemcpyHostToDevice);
47
48   // calcula los bloques que serán necesarios la operación
49   int hilosPorBloque = 512;
50   int bloquesPorMalla = (n + hilosPorBloque - 1) / hilosPorBloque;
51
52   // invoca kernel
53   sgemm<<<hilosPorBloque, bloquesPorMalla>>>(n,alpha, dA, dB, beta, dC);
54
55   // copia la matriz de GPU a CPU
56   cudaMemcpy(C->data, dC, size, cudaMemcpyDeviceToHost);
57
58   // libera memoria
59   cudaFree(dA);
60   cudaFree(dB);
61   cudaFree(dC);
62 }  // --- end of function cuda_sgemm ---
```

Programa 24: Producto matriz-matriz con CUBLAS

```
1 // === FUNCTION ======================================================================
2 // Name: cublas_sgemm
3 // Description: Realiza el producto matriz matriz, de precisión simple (float),
4 // operando con matrices generales. C = α × A × B + β × C
5 // ====================================================================================
6
7 void
8 cublas_sgemm(
9   const  char transA,  // determina si se opera con A o con AT (transpuesta)
10  const  char transB,  // determina si se opera con B o con BT (transpuesta)
11  const int m,  // número de filas de la matriz A y de la matriz C
12  const int n,  // número de columnas de la matriz A y de la matriz C
13  const int k,  // número de columnas de la matriz A y filas de la matriz B
14  const float alpha,  // escalar
15  const Matriz *A,  // matriz de dimensión lda*k si A y lda*m si AT
16  const int lda,  // dimensión de A
17  const Matriz *B,  // matriz de dimensión ldb*n si B y ldb*k si BT
18  const int ldb,  // dimensión de B
19  const float beta,  // escalar
20  Matriz *C,  // matriz de dimensión ldb*n
21  const int ldc  // dimensión de C
22 )
23 {
24  // matrices para GPU
25  float *dA;
26  float *dB;
27  float *dC;
28
29  // controla el estado de CUBLAS
30  cublasStatus status;
31
32  // inicia CUBLAS
33  status = cublasInit();
34  check_cublas_status("inicia", status);
35
36  // calcula las dimensiones del A, B y C
37  int elemA, elemB, elemC;
```

```
38
39  if ( transA == 'N' || transA == 'n' ) // A
40    elemA = lda*k;
41  else // AT
42    elemA = lda*m;
43
44  if ( transB == 'N' || transB == 'n' ) // B
45    elemB = lda*n;
46  else // BT
47    elemB = lda*k;
48  elemC = ldc*n;
49
50  // reserva memoria en GPU
51  status = cublasAlloc(elemA, sizeof(float), (void **)&dA);
52  check_cublas_status("alloc dA", status);
53  status = cublasAlloc(elemB, sizeof(float), (void **)&dB);
54  check_cublas_status("alloc dB", status);
55  status = cublasAlloc(elemC, sizeof(float), (void **)&dC);
56  check_cublas_status("alloc dC", status);
57
58  // copia las matrices de CPU a GPU
59  status = cublasSetMatrix(m, n, sizeof(float), A->data, m, dA, m );
60  check_cublas_status("set d_A", status);
61  status = cublasSetMatrix(m, n, sizeof(float), B->data, m, dB, m );
62  check_cublas_status("set d_B", status);
63  status = cublasSetMatrix(m, n, sizeof(float), C->data, m, dC, m );
64  check_cublas_status("set d_C", status);
65
66  // realiza el producto matriz matriz
67  cublasSgemm(transA, transB, m, n, k, alpha, dA, lda, dB, ldb, beta, dC, ldc);
68  status = cublasGetError();
69  check_cublas_status("sgemv", status);
70
71  // copia matriz C de GPU a CPU
72  status = cublasGetMatrix(m, n, sizeof(float), dC, m, C->data, m);
73  check_cublas_status("get dy", status);
74
75  // libera memoria
76  status = cublasFree(dA);
77  check_cublas_status("free dA", status);
78  status = cublasFree(dB);
79  check_cublas_status("free dB", status);
80  status = cublasFree(dC);
81  check_cublas_status("free dC", status);
82
83  // cierra CUBLAS
84  status = cublasShutdown();
85  check_cublas_status("cerrar", status);
86 }  // --- end of function cublas_sgemm ---
```

En el siguiente programa se prueba la operación gemv (producto matriz-matriz) con CUDA y CUBLAS, para compilar el mismo se hace con la siguiente sentencia:

```
nvcc src/vec.cu src/mat.cu src/cuda.cu src/cubla.cu test/gemm.cu -o bin/gemm_gpu -I include/ -lcublas
```

donde (para ver la descripción de todos los elementos de las sentencias de Compilación y ejecución ver página 57).

test/gemm.cu prueba gemm con CUDA y CUBLAS.

bin/gemm_gpu ejecutable generado.

Para ejecutar el programa que prueba el producto matriz-matriz con CUDA y CUBLAS se hace con la siguiente sentencia:

```
./bin/gemm_gpu 1000 15.65 0.056
```

donde

bin/gemm_gpu archivo ejecutable que esta dentro del directorio bin.

1000 argumento que recibe el programa, indica la dimensión de la matriz (ver programa 25 entre línea 34).

15.65 argumento que recibe el programa, indica el escalar alpha (ver programa 25 línea 35).

0.056 argumento que recibe el programa, indica el escalar beta (ver programa 25 línea 36).

Programa 25: Prueba producto matriz-matriz con CUDA y CUBLAS

```
1 // ================================================================================
2 // Filename: gemm.cu
3 //
4 // Description: pueba la rutina gemm con CUDA y CUBLAS que realiza el producto
5 // matriz-matriz de la forma α × A × B + β × C
6 // Compiler: nvcc
7 // ================================================================================
8
9
```

```
#include  <stdlib.h>
#include  <stdio.h>

#include  "mat.h"  // funciones que manipulan matrices
#include  "cuda.hcu"  // rutinas con CUDA
#include  "cubla.h"  // rutinas con CUBLAS

// === FUNCTION ======================================================================
// Name: main
// Description: Prueba la rutina gemm con CUDA y CUBLAS que realiza el producto
// matriz-matriz operando con matrices generales C = α × A × B + β × C
// ===================================================================================

int
main (
  int argc,  // contador de argumentos
  char *argv[]  // valor de argumentos
)
{
  if ( argc != 4 ) {
    printf ( "Argumentos: n\talpha\tbeta\n" );
    exit(EXIT_FAILURE);
  }
  // toma argumentos de la sentencia de ejecución
  int n = atoi(argv[1]);  // número de elementos de los vectores de entrada
  float alpha = atof(argv[2]);  // escalar
  float beta = atof(argv[3]);  // escalar

  // matrices para realizar operaciones
  Matriz Ac, Bc, Cc;  // para CUDA
  Matriz Ab, Bb, Cb;  // para CUBLAS

  // reserva memoria en CPU
  allocateMatriz(&Ac, n, n);
  allocateMatriz(&Bc, n, n);
  allocateMatriz(&Cc, n, n);

  allocateMatriz(&Ab, n, n);
  allocateMatriz(&Bb, n, n);
  allocateMatriz(&Cb, n, n);

  // llena las matrices y los vectores con números aleatorios
  llenaMatriz(&Ac);
  llenaMatriz(&Bc);
  llenaMatriz(&Cc);

  llenaMatriz(&Ab);
  llenaMatriz(&Bb);
  llenaMatriz(&Cb);

  // realiza la el producto matriz matriz
  cuda_sgemm(n, alpha, &Ac, &Bc, beta, &Cc);  // con CUDA
  cublas_sgemm('N', 'N', n, n, n, alpha, &Ab, n, &Bb, n,
      beta, &Cb, n);  // con CUBLAS

  // imprime resultado
  imprimeMatriz(&Cc);  // con CUDA
  imprimeMatriz(&Cb);  // con CUBLAS

  // libera memoria
  freeMatriz(&Ac);
  freeMatriz(&Bc);
  freeMatriz(&Cc);

  freeMatriz(&Ab);
  freeMatriz(&Bb);
  freeMatriz(&Cb);

  // finaliza programa
  return EXIT_SUCCESS;
}  // ------- end of function main -------
```

6.2. Elaboración, Compilación y Ejecución de Programas en LAPACK y CULA

A continuación se muestra la organización de los programas generados para LAPACK y CULA:

|–bin/

|–include/

|——cu_lapack.h

|——micula.h
|——milapack.h
|–sh/
|——revisaEntorno.sh
|–src/
|——cu_lapack.c
|——micula.c
|——milapack.c
|–test/
|——gesv.c
|——gesvd.c

Para escribir los programas correspondientes a LAPACK y CULA se creó el mismo árbol de directorios como en el caso de la sección 6.1, pero se agrega el directorio **sh** que contiene el script **revisaEntorno.sh**, este script verifica que estén configuradas variables de entorno para que CULA pueda funcionar sin problemas, de lo contrario reporta el motivo de la interrupción (programa 26).

Programa 26: revisaEntorno.sh

```
#!/bin/bash

# Revisa si las variables de entorno para CULA estan configuradas.

revisa_variable_entorno()
{
   var="$1"
   val='$'$1
   if [ -z  `eval "echo $val"` ]; then
     echo "Advertencia: $var no esta definida"
      eval $2=true
   fi
}

alerta=false
revisa_variable_entorno "CULA_ROOT" alerta
revisa_variable_entorno "CULA_INC_PATH" alerta
revisa_variable_entorno "CULA_BIN_PATH_32" alerta
revisa_variable_entorno "CULA_BIN_PATH_64" alerta
revisa_variable_entorno "CULA_LIB_PATH_32" alerta
revisa_variable_entorno "CULA_LIB_PATH_64" alerta

if $alerta ; then
    echo ""
    echo "------------------------------------------------------------------------"
    echo "Advertencia: Algunas variables para que CULA funcione no se encontraron "
    echo "        Esto impide la correcta compilacion y ejecucion de los programas"
    echo "------------------------------------------------------------------------"
    echo ""
fi
```

6.2.1. Definición de las funciones usadas por los programas en LAPACK y CULA

En esta sección se muestran los archivos de cabecera (directorio **include**) mismos que contienen las funciones que hacen uso de las rutinas **sgesv_** y **sgesvd_** (programa 29), **culaSgesv** y **culaSgesvd** (programa 28), así como las funciones para calcular las diferentes normas (las normas calculadas tienen como propósito medir el error de precisión en los cálculos realizados por las rutinas LAPACK y CULA probadas), imprimir una matriz, transponer una matriz y llenar una matriz (programa 27), estas últimas se usan en los programas de prueba correspondientes.

Programa 27: cu_lapack.h

```
1 // ============================================================================
2 // Filename: cu_lapack.h
3 //
4 // Description: Definición de las funciones para manipular matrices y calcular la
5 // norma 2 y la norma Frobenius con C.
6 // Compiler: gcc
7 // ============================================================================
8
9 #ifndef  _CU_LAPACK_H_
10 #define  _CU_LAPACK_H_
11 void imprime_matriz( char *, int, int, float * );  // imprime una matriz o un vector
```

```
12 void transpone_matriz( int, int, float *, float * );  // transpone una matriz
13 void llenarm( int, int, float *);  // llena una matriz con números aleatorios
14
15 // calcula la norma2 error de precisión con BLAS
16 float calcula_norma2_A_usv( int, int, float *, float *, float *, float *);  // norma Frobenius ||A - USVT||_F
17 float calcula_norma2_Ax_b( int, float *, float *, float * );  // 2-norma ||Ax - b||_2
18 float calcula_norma2_x_x( int, float *, float * );  // 2-norma ||x' - x||_2
19 #endif   // termina la definición
```

Programa 28: micula.h

```
1 // ============================================================================
2 // Filename: micula.h
3 //
4 // Description: Definición de las funciones para manipular culaSgesv y culaSgesvd
5 // Compiler: gcc
6 // ============================================================================
7
8 #ifndef  _MICLUA_H_
9 #define  _MICLUA_H_
10 #include  <culapack.h>
11 void check_cula_status ( char *, culaStatus );  // revisa el estado de invoación de CULA
12 // funciones para sgesv
13 void culaSgesv_ejemplo ( int, float *, float * );  // usa culaSgesv.
14
15 // funciones para sgesvd
16 void culaSgesvd_ejemplo( int, int, float *, int );  // usa culaSgesvd
17 #endif   // termina la definición
```

Programa 29: milapack.h

```
1 // ============================================================================
2 // Filename: milapack.h
3 //
4 // Description: Definición de las funciones para manipular sgesv_ y sgesvd_
5 // Compiler: gcc
6 // ============================================================================
7
8 #ifndef  _MILAPACK_H_
9 #define  _MILAPACK_H_
10 // funciones para sgesv
11 void genera_b( int, float *, float *, float * );  // genera el vector de constantes b, con A y x b=Ax
12 void lapackSgesv_ejemplo( int, float *, float * );  // usa sgesv_
13
14 // funciones para sgesvd
15 void lapackSgesvd_ejemplo ( int, int, float *, int );  // usa sgesvd_
16 #endif   // termina la definición
```

6.2.2. Implementación de las funciones usadas por los programas en LAPACK y CULA

En los programas que se presentan a continuación están las implementaciones de las funciones para manipular matrices y calcular las diferentes normas para comparar el error de precisión generado por las rutinas LAPACK y CULA probadas (programa 30), así como una función que se encarga de monitorear el estado de invocación de CULA (programa 31) y por último la implementación de una función que se encarga de generar el vector de constantes b a partir de una matriz de coeficientes A y un vector de incógnitas x dados (programa 32) con el fin de conocer de antemano la solución del sistema de ecuaciones $Ax = b$ y calcular el error de la solución que da LAPACK y CULA mediante la 2-norma $||x_{calculada} - x_{solución}||_2$. El resto de las implementaciones se verán más adelante.

Programa 30: cu_lapack.c

```
1 // ============================================================================
2 // Filename: cu_lapack.c
3 //
4 // Description: Definición de las funciones para manipular matrices y calcular la
5 // norma 2 y la norma Frobenius con C.
6 // Compiler: gcc
7 // ============================================================================
8
9 #include  <stdio.h>
10 #include  <stdlib.h>
11 #include  <string.h>
12 #include  <cblas.h>  // blas
13
14 #include  "cu_lapack.h"
```

```
15
16 // === FUNCTION ======================================================================
17 // Name: imprime_matriz
18 // Description: imprime una matriz o vector dadas sus dimensiones
19 // ====================================================================================
20
21 void
22 imprime_matriz (
23   char *obj,  // nombre de la matriz
24   int m,  // filas de la matriz
25   int n,  // columnas de ma martiz
26   float *A // matriz
27 )
28 {
29   int i, j;
30   printf ( "\n %s\n", obj );
31
32   for ( i = 0; i < m; i += 1 ) {
33     for ( j = 0; j < n; j += 1 )
34       // imprime el elemento x(i,j)
35       printf ( "%.4f\t", A[j+n*i] );
36     printf ( "\n" );
37   }
38 }  // --- end of function imprime_matriz ---
39
40 // === FUNCTION ======================================================================
41 // Name: transpone_matriz
42 // Description: transpone una matriz A ->A^T
43 // ====================================================================================
44
45 void
46 transpone_matriz(
47   int m,  // filas de la matriz
48   int n,  // columnas de la matriz
49   float *A,  // matriz origen
50   float *AT // mariz destino
51 )
52 {
53   int i, j;
54   for ( i = 0; i < n; i++ )
55       for( j = 0; j < m; j++ )
56         AT[j+m*i] = A[i+n*j];  // A->AT
57 }  // --- end of function transpone_matriz ---
58
59 // === FUNCTION ======================================================================
60 // Name: llenarm
61 // Description: llena una matriz de m*n elementos con números aleatorios
62 // ====================================================================================
63
64 void
65 llenarm (
66   int m,  // filas de la matriz
67   int n,  // columnas de la matriz
68   float *A // matriz
69 )
70 {
71   srand(time(NULL));
72   int i;
73   for ( i = 0; i < m*n; i += 1 )
74     A[i] = (float)(rand()%9*.1);
75 }  // --- end of function llenarm ---
76
77 // === FUNCTION ======================================================================
78 // Name: calcula_norma2_Ax_b
79 // Description: Calcula la norma 2. El error de precisión en el resultado ||Ax - b||_2
80 // ====================================================================================
81
82 float // norma calculada
83 calcula_norma2_Ax_b (
84   int n,  // número de ecuaciones
85   float *A,  // matriz ecuaciones. AT por compatibilidad con Fortran
86   float *x,  // vector de resultados
87   float *b // vector solucion
88 )
89 {
90   // se calcula Ax-b, retorna el vector residuo.
91   cblas_sgemv ( CblasRowMajor, CblasNoTrans, n, n, 1.0f, A, n, x, 1, -1.0f, b, 1 );
92
93   // calcula la norma 2 del vector b.
94   return cblas_snrm2 ( n, b, 1 );
95 }  // --- end of function calcula_norma2 ---
```

```
96
97
98 // === FUNCTION ======================================================================
99 // Name: calcula_norma2_x_x
100 // Description: Calcula la norma 2. El error de precisión en el resultado ||x' - x||_2
101 // ===================================================================================
102
103 float // norma calculada
104 calcula_norma2_x_x (
105   int n,  // elementos de x1 y x2
106   float *x1,  // vector de resultados x'
107   float *x2  //vector de resultados x
108 )
109 {
110   // realiza operacion x2 = x1-x2
111   cblas_saxpy(n, -1.0f, x1, 1, x2, 1);
112
113   // calcula la norma 2 de x2
114   return cblas_snrm2(n, x2, 1);
115 }  // --- end of function calcula_norma2_x_x ---
116
117
118 // === FUNCTION ======================================================================
119 // Name: calcula_norma2_A_usv
120 // Description: calcula la norma 2. El error de precisión en el resultado ||A - USV^T||_F
121 // ===================================================================================
122
123 float
124 calcula_norma2_A_usv (
125   int m,  // filas de la matriz
126   int n,  // columnas de la matriz
127   float *A,  // matriz
128   float *U,  // vectores singulares izquierdos
129   float *S,  // valores singulares
130   float *VT // vectores singulares derechos
131 )
132 {
133   float *US = (float *)malloc(m*n*sizeof(float));
134   float *USVT = (float *)malloc(m*n*sizeof(float));
135
136   // calcula US
137   cblas_sgemm(CblasRowMajor, CblasTrans, CblasNoTrans, m, n, m, 1.0f, U, m, S, n, 1.0f, US, n);
138
139   // calcula USVT
140   cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, n, 1.0f, US, n, VT, n, 1.0f, USVT, n);
141
142   // calcula A-USVT
143   cblas_saxpy(m*n, -1.0f, A, 1, USVT, 1);
144
145   return cblas_snrm2(m*n, A, 1);
146 }  // --- end of function calcula_norma_A_uxv ---
```

Programa 31: Chequeo del estado de invocación de CULA

```
1 // === FUNCTION ======================================================================
2 // Name: check_cula_status
3 // Description: Revisa el estado de invocación de CULA.
4 // ===================================================================================
5
6 void
7 check_cula_status (
8   char *obj,  // nombre del objeto que realiza alguna operación con cula
9   culaStatus status // controla el estado de cula
10 )
11 {
12   if(!status) // si no hay errores regresa sin cambios
13     return;
14
15   // reporta el tipo de error y cierra cula
16   if(status == culaArgumentError)
17     printf("%s: Valor no valido para el parametro %d\n", obj, culaGetErrorInfo());
18       else if(status == culaDataError)
19         printf("%s: Error de datos (%d)\n", obj, culaGetErrorInfo());
20         else if(status == culaBlasError)
21         printf("%s, Error de BLAS (%d)\n", obj, culaGetErrorInfo());
22       else if(status == culaRuntimeError)
23         printf("%s: Error de ejecucion (%d)\n", obj, culaGetErrorInfo());
24   else
25     printf("%s\n", culaGetStatusString(status));
26
```

```
27   culaShutdown();
28   exit(EXIT_FAILURE);
29 }  // --- end of function check_cula_status ---
```

Programa 32: Genera el vector b

```
1 // === FUNCTION ======================================================================
2 // Name: genera_b
3 // Description: genera el vector de constantes, dados A y x. b=Ax
4 // ====================================================================================
5
6 void
7 genera_b (
8   int n,  // número de ecuaciones de A
9   float *A,  // matriz de ecuaciones
10  float *x,  // vector de resultados
11  float *b // vector solución
12 )
13 {
14  cblas_sgemv( CblasRowMajor, CblasNoTrans, n, n, 1.0f, A, n, x, 1, 1.0f, b, 1 );
15 }  // --- end of function genera_b ---
```

6.2.3. Implementación, Compilación y Ejecución de programas para la resolución de un sistema de ecuaciones lineales

A continuación se muestran los programas para la resolución de un sistema de ecuaciones tanto en LAPACK (programa 33) como en CULA (programa 34) y para concluir éste tema se presenta un programa que prueba los dos anteriores (programa 35). Se creó un sólo programa de prueba porque ambos (LAPACK y CULA) usan el mismo compilador, gcc en éste caso.

Programa 33: Resolución de sistema de ecuaciones lineales con LAPACK

```
1 // === FUNCTION ======================================================================
2 // Name: lapackSgesv_ejemplo
3 // Description: Prueba sgesv_, que resuelve un sistema de ecuaciones de la forma
4 // Ax = b.
5 // ====================================================================================
6
7 void
8 lapackSgesv_ejemplo (
9   int n,  // número de ecuaciones
10  float *A,  // matriz de ecuaciones
11  float *b // vector de constantes
12 )
13 {
14  int nrhs = 1;  // columnas del vector b
15  // vector para guardar el pivotamiento
16  int *pivot = (int *)malloc(n*sizeof(int));
17  int info;  // código de posibles errores
18
19  float *AT = (float *)malloc(sizeof(float)*n*n);
20  float *A2 = (float *)malloc(sizeof(float)*n*n);
21  float *x = (float *)malloc(sizeof(float)*n);
22  float *b2 = (float *)malloc(sizeof(float)*n);
23
24  printf( "-------------------\n" );
25  printf( "       sgesv_\n" );
26  printf( "-------------------\n" );
27
28  // si A o b o pivot apuntan a NULL, sale del programa
29  if(!A || !b || !pivot)
30  exit(EXIT_FAILURE);
31
32  // se copia b en b2 para el calculo de la norma
33  cblas_scopy(n, b, 1, b2, 1);
34
35  // se transpone A para compatibilidad con fortran.
36  transpone_matriz(n, n, A, AT);
37
38  // copia A en A2 para el cálculo de la norma
39  cblas_scopy(n*n, A, 1, A2, 1);
40
41  // resuelve el sistema de ecuaciones. Retorna el vector de incógnitas
42  // sobre escribiendo b
43  sgesv_(&n, &nrhs, AT, &n, pivot, b, &n, &info);
44
45  // se copia el vector solución x=b, para calcular la norma
```

```
46   cblas_scopy(n, b, 1, x, 1);
47
48   calcula_norma2_Ax_b ( n, A2, x, b2 );
49
50   // libera memoria
51   free(pivot);
52   free(AT);
53   free(A2);
54   free(x);
55   free(b2);
56 }  // --- end of function lapackSgesv_ejemplo ---
```

Programa 34: Resolución de sistema de ecuaciones lineales con CULA

```
1 // Name: culaSgesv_ejemplo
2 // Description: Prueba culaSgesv, que resuelve un sistemas de ecuaciones de la forma
3 // Ax = b.
4 // ============================================================================================
5
6 void
7 culaSgesv_ejemplo (
8   int n,  // número de ecuaciones
9   float *A,  // matriz de ecuaciones
10  float *b // vector de constantes
11 )
12 {
13  int nrhs = 1;  // columnas del vector b
14  // vector para guardar el pivotamiento
15  culaInt *pivot = (culaInt *)malloc(n*sizeof(culaInt));
16
17  culaStatus status;  // estado de CULA
18
19  culaFloat *AT = (culaFloat*)malloc(n*n*sizeof(culaFloat));
20  culaFloat *A2 = (culaFloat *)malloc(n*n*sizeof(culaFloat));
21  culaFloat *x = (culaFloat *)malloc(n*sizeof(culaFloat));
22  culaFloat *b2 = (culaFloat *)malloc(n*sizeof(culaFloat));
23
24  printf("-------------------\n");
25  printf("       culaSgesv\n");
26  printf("-------------------\n");
27
28  // si A o b o pivot apuntan a NULL, sale del programa
29  if(!A || !b || !pivot)
30  exit(EXIT_FAILURE);
31
32  // inicia CULA
33  status = culaInitialize();
34  check_cula_status("inicio", status);
35
36  // se copia b en b2 para el calculo de la norma
37  cblas_scopy(n, b, 1, b2, 1);
38
39  // tanspone A para compatibilidad con Fortran, A->AT
40  transpone_matriz(n, n, A, AT);
41
42  // copia A en A2 para el cálculo de la norma
43  cblas_scopy(n*n, A, 1, A2, 1);
44
45  // resuelve el sistema de ecuaciones. Retorna el vector de incógnitas
46  // sobre escribiendo b
47  status = culaSgesv(n, nrhs, AT, n, pivot, b, n);
48  check_cula_status("culaSgesv", status);
49
50  // se copia el vector solución x=b, para calcular la norma
51  cblas_scopy(n, b, 1, x, 1);
52
53  // se calcula la norma
54  calcula_norma2_Ax_b(n, A2, x, b2);
55
56  // cierra cula
57  culaShutdown();
58
59  // libera memoria
60  free(pivot);
61  free(AT);
62  free(A2);
63  free(x);
64  free(b2);
65 }  // --- end of function culaSgesv_ejemplo ---
```

En el siguiente programa se prueba la resolución de un sistema de ecuaciones (gesv) con LAPACK y CULA, para compilarlo se escribe la siguiente sentencia:

```
sh sh/revisaEntorno.sh
gcc src/cu_lapack.c src/micula.c src/milapack.c test/gesv.c -o bin/gesv -I/usr/local/cula/include -Iinclude/
    -I/usr/local/cula/lib64 -lcublas -lcudart -lcula -llapack
```

donde

sh	intérprete de comandos.
sh/revisaEntorno.sh	ver programa 26.
gcc	es el compilador de C. (ver sección 5.1.1).
src/cu_lapack.c	funciones para manipular matrices y calcular las diferentes normas.
src/micula.c	funciones para manipular **culaSgesv**.
src/milapack.c	funciones para manipular **sgesv_**.
test/gesv.c	prueba **culaSgesv** y **sgesv_**.
-o	le indica al compilador que genere un ejecutable gesv y lo guarde en el directorio **bin**.
bin/gesv	ejecutable generado.
-I	le indica al compilador que debe considerar archivos de cabecera creados por el usuario (directorio **include**).
-L	agrega directorios para buscar bibliotecas, en éste caso las de CULA.
/usr/local/cula/lib64	directorio donde están las bibliotecas de CULA.
-lcublas	biblioteca libcublas.a para usar CUBLAS.
-lcudart	biblioteca libcudart.a para usar CUDA.
-lcula	biblioteca libcula.a para usar CULA.
-llapack	biblioteca liblapack.a para usar LAPACK.

Para ejecutar el programa que prueba la resolución de un sistema de ecuaciones con LAPACK y CULA se hace con la siguiente sentencia:

```
./bin/gesv 2000
```

donde

./	le indica al sistema que se trata de un archivo ejecutable dentro del directorio actual
bin/gesv	archivo ejecutable que esta dentro del directorio **bin**.
2000	argumento que recibe el programa, indica el número de ecuaciones (ver programa 35 línea 36).

Programa 35: Prueba la resolución de sistema de ecuaciones lineales

```
1  // ================================================================================
2  // Filename: gesv.c
3  //
4  // Description: Prueba la rutinas sgesv_ y culaSgesv
5  // Compiler: gcc
6  // ================================================================================
7
8  #include  <stdio.h>
9  #include  <stdlib.h>
10
11 #include        <culapack.h>  // cula
12 #include        <cblas.h>  // blas
13
14 #include  "micula.h"  // funciones para manipular culaSgesv y culaSgesvd
15 #include  "milapack.h"  // funciones para manipular sgesv_ y sgesvd_
16 #include  "cu_lapack.h"  // funciones para manipular matrices y calcular norma 2
17
18 // === FUNCTION ===============================================================
19 // Name: main
20 // Description: prueba la rutina sgesv que resuelve un sistema ecuaciones de la forma
21 // Ax=b, con implementaciones de LAPACK y CULA.
22 // ================================================================================
23
24 int
25 main (
26   int argc,  // contador de argumentos
27   char *argv[]  // valores de argumentos
28 )
29 {
30   if ( argc != 2 ) {
31     printf ( "Argumentos:Numero de ecuaciones -> N:\n" );
32     exit(EXIT_FAILURE);
33   }
34
```

```
35  // toma los argumentos de la sentencia de ejecución
36  int n = atoi(argv[1]);  // número de ecuaciones
37  int i, j;
38
39  float *A = (float *)malloc(n*n*sizeof(float));  // matriz de ecuaciones
40  float *Al = (float *)malloc(n*n*sizeof(float));  // matriz para lapack
41  float *Ac = (float *)malloc(n*n*sizeof(float));  // matriz para cula
42
43  float *x = (float *)malloc(n*sizeof(float));  // vector resultado
44  float *xl = (float *)malloc(n*sizeof(float));  // vector para lapack
45  float *xc = (float *)malloc(n*sizeof(float));  // vector para cula
46
47  float *b = (float *)malloc(n*sizeof(float));  // vector solucion
48  float *bl = (float *)malloc(n*sizeof(float));  // vector para lapack
49  float *bc = (float *)malloc(n*sizeof(float));  // vector para cula
50
51
52  llenarm(n, n, A);  // genera la matriz de ecuaciones
53  llenarm(1, n, x);  // genera el vector de incognitas
54
55  // copia matriz para lapack y cula
56  cblas_scopy(n*n, A, 1, Al, 1);  // copia A en Al Al=A
57  cblas_scopy(n*n, A, 1, Ac, 1);  // copia A en Ac Ac=A
58
59  // cpoia vector resultado para lapack y cula
60  cblas_scopy(n, x, 1, xl, 1);  // copia x en xl xl=x
61  cblas_scopy(n, x, 1, xc, 1);  // copia x en xc xc=x
62
63  // genera el vector de constantes b=Ax
64  genera_b(n, A, x, b);
65
66  cblas_scopy(n, b, 1, bl, 1);
67  cblas_scopy(n, b, 1, bc, 1);
68
69  // operaciones con lapack
70  lapackSgesv_ejemplo(n ,Al, bl);
71  printf ( "Norma2 x'-x\t %.15f\n", calcula_norma2_x_x(n, x, bl) );
72
73  // operaciones con cula
74  culaSgesv_ejemplo(n, Ac, bc);
75  printf ( "Norma2 x'-x\t %.15f\n", calcula_norma2_x_x(n, x, bc) );
76
77  // libera memoria
78  free(A); free(Al); free(Ac);
79  free(x); free(xl); free(xc);
80  free(b); free(bl); free(bc);
81
82  // finaliza programa
83  return EXIT_SUCCESS;
84 }  // ------- end of function main -------
```

6.2.4. Implementación, Compilación y Ejecución de programas para la descomposición de una matriz en sus valores singulares

A continuación se muestran los programas para la descomposición de una matriz en sus valores singulares en LAPACK (programa 36) y en CULA (programa 37), al final se presenta un programa que prueba los dos anteriores (programa 38).

Programa 36: Descomposición de una matriz en sus valores singulares con LAPACK

```
1 // === FUNCTION ======================================================================
2 // Name: lapackSgesvd_ejemplo
3 // Description: prueba sgesvd_ que calcula los valores singulares de una matriz.
4 // A = USV^T
5 // ====================================================================================
6
7 void
8 lapackSgesvd_ejemplo (
9   int m,  // filas de la matriz
10  int n,  // columnas de la matriz
11  float *A,  // matriz
12  int lda // dimensión de la matriz (filas)
13 )
14 {
15  int ldu = m;  // dimensión de la matriz u
16  int ldvt = n;  // dimensión de la matriz vt
17  int info = 0;  // código de posibles errores
18  int lwork = 0;  // dimensión del espacio de trabajo
19
```

```
  // se transpone A para compatibilidad con fortran.
  float *AT = (float *)malloc(m*n*sizeof(float));
  transpone_matriz(m, n, A, AT);

  // vector que contiene los valores singulares ordenados
  // de tal manera que s(i) >= s(i+1)
  float *s = (float *)malloc(n*sizeof(float));

  // matriz que contiene los vectores singulares izquierdos de A
  // almacenados por columna
  float *u = (float *)malloc(ldu*m*sizeof(float));

  // matriz que contiene los vectores singulares derechos de A
  // almacenados por fila
  float *vt = (float *)malloc(ldvt*n*sizeof(float));

  printf("-------------------\n");
  printf("       sgesvd_\n");
  printf("-------------------\n");

  lwork = -1;
  // la primera vez que se invoca sgesvd_ se calcula la dimensión óptima de work
  // que se guarda en wkopt
  float wkopt;
  sgesvd_( "A", "A", &m, &n, AT, &lda, s, u, &ldu, vt, &ldvt, &wkopt, &lwork, &info );

  lwork = (int)wkopt;  // asigna la dimensión de work

  float *work = (float *)malloc(lwork*sizeof(float));

  // calcula los valores singulares de una matriz
  sgesvd_( "A", "A", &m, &n, AT, &lda, s, u, &ldu, vt, &ldvt, work, &lwork, &info );

  imprime_matriz("valores singulares", 1, n, s);

  // dado que s es retornado como vector porsgesvd_. Para calcular la norma ||A - USVT||2
  // es necesario tomarlo como matriz, los valoes singulares quedan ubicados en la diagonal
  // principal de la matriz ms
  float *ms = (float *)malloc(m*n*sizeof(float));

  // ms, se llena con ceros a excepción de la diagonal principal que es llenada con los valores
  // singulares almacenados en s
  int diag = 0, i;
  for ( i = 0; i < m*n; i += 1 ) {
    if ( i == n*diag + diag ) {
      ms[i] = s[diag];  // ms(n*diag+diag) = s(i)
      diag++;
    }
    else {
      ms[i] = 0.0f;
    }
  }

  // se calcula la norma 2
  calcula_norma2_A_usv(m, n, A, u, ms, vt);

  // libera memoria
  free(AT);
  free(s);
  free(u);
  free(vt);
  free(work);
  free(ms);
}  // --- end of function lapackSgesvd_ejemplo ---
```

Programa 37: Descomposición de una matriz en sus valores singulares con CULA

```
// === FUNCTION ======================================================================
// Name: culaSgesvd_ejemplo
// Description: prueba culaSgesvd que calcula los valores singulares de una matriz.
// A = USV^T
// ===================================================================================

void
culaSgesvd_ejemplo(
  int m,  // filas de la matriz
  int n,  // columnas de la matriz
  float *A,  // matriz
  int lda // dimensión de la matriz (filas)
)
```

```
14 {
15   int ldu = m;  // dimension de la matriz u
16   int ldvt = n;  // dimension de la matriz vt
17 
18   culaStatus status;  // estado de CULA
19 
20   // inicia CULA
21   status = culaInitialize();
22   check_cula_status("inicia", status);
23 
24   // se transpone A para compatibilidad con fortran.
25   culaFloat *AT = (culaFloat*)malloc(m*n*sizeof(culaFloat));
26   transpone_matriz(m ,n, A, AT);
27 
28   // vector que contiene los valores singulares ordenados
29   // de tal manera que s(i) >= s(i+1)
30   culaFloat *s = (float *)malloc(n*sizeof(float));
31 
32   // matriz que contiene los vectores singulares izquierdos de A
33   // almacenados por columna
34   culaFloat *u = (float *)malloc(ldu*m*sizeof(float));
35 
36   // matriz que contiene los vectores singulares derechos de A
37   // almacenados por fila
38   culaFloat *vt = (float *)malloc(ldvt*n*sizeof(float));
39 
40   printf("-------------------\n");
41   printf("       culaSgesvd\n");
42   printf("-------------------\n");
43 
44   // calcula los valores singulares de una matriz
45   status = culaSgesvd('A', 'A', m, n, AT, lda, s, u, ldu, vt, ldvt);
46   check_cula_status("culaSgesvd", status);
47 
48   imprime_matriz("valores singulares", 1, n, s);
49 
50   // dado que s es retornado como vector por culaSgesvd. Para calcular la norma ||A - USV^T||_2
51   // es necesario tomarlo como matriz, los valoes singulares quedan ubicados en la diagonal
52   // principal de la matriz ms
53   culaFloat *ms = (culaFloat *)malloc(m*n*sizeof(float));
54 
55   int diag = 0, i;
56   for ( i = 0; i < m*n; i += 1 ) {
57     if ( i == n*diag + diag ) {
58       ms[i] = s[diag];
59       diag++;
60     }
61     else {
62       ms[i] = 0.0f;
63     }
64   }
65 
66   // se calcula la norma 2
67   calcula_norma2_A_usv(m, n, A, u, ms, vt);
68 
69   // cierra cula
70   culaShutdown();
71 
72   // libera memoria
73   free(AT);
74   free(u);
75   free(s);
76   free(vt);
77 }  // --- end of function calcula_norma2_A_usv ---
```

A continuación se presenta el programa se prueba la descomposición de una matriz en sus valroes singulares (gesvd) con LAPACK y CULA, para compilarlo se escribe la siguiente sentencia:

```
sh sh/revisaEntorno.sh
gcc -o bin/gesvd src/cu_lapack.c src/micula.c src/milapack.c test/gesvd.c -I/usr/local/cula/include -Iinclude/
     -I/usr/local/cula/lib64 -lcublas -lcudart -lcula -llapack -lblas
```

donde (para ver la descripción de todos los elementos de la sentencias de compilación y ejecución, ver página 71).

test/gesvd.c	prueba culaSgesvd y sgesvd_.
bin/gesvd	ejecutable generado.
-lblas	biblioteca libblas.a para usar BLAS.

Para ejecutar el programa que prueba la descomposición de una matriz en sus valores singulares con LAPACK y CULA se hace con la siguiente sentencia:

```
./bin/gesvd 1500 1800
```

bin/gesvd	archivo ejecutable que esta dentro del directorio bin.
1500	argumento que recibe el programa, indica el número filas de la matriz (ver programa 38 línea 36).
1800	argumento que recibe el programa, indica el número columnas de la matriz (ver programa 38 línea 37).

Programa 38: Prueba la descomposición de una matriz en sus valores singulares

```
1 // ============================================================================
2 // Filename: gesvd.c
3 //
4 // Description: Prueba la rutinas sgesvd_ y culaSgesvd
5 // Compiler: gcc
6 // ============================================================================
7
8 #include  <stdio.h>
9 #include  <stdlib.h>
10
11 #include        <culapack.h>  // cula
12 #include        <cblas.h>  // blas
13
14 #include  "micula.h"  // funciones para manipular culaSgesv y culaSgesvd
15 #include  "milapack.h"  // funciones para manipular sgesv_ y sgesvd_
16 #include  "cu_lapack.h"  // funciones para manipular matrices y calcular norma 2
17
18 // === FUNCTION ===============================================================
19 // Name: main
20 // Description: prueba la rutina sgesvd que calcula los valores singulares de una
21 // matriz. A = UΣV^T
22 // ============================================================================
23
24 int
25 main (
26   int argc,  // contador de argumentos
27   char *argv[]  // valores de argumentos
28 )
29 {
30   // toma los argumentos de la sentencia de ejecución
31   if ( argc != 3 ) {
32     printf ( "Argumentos:filasM columnasM\n" );
33     exit(EXIT_FAILURE);
34   }
35
36   int m = atoi(argv[1]);  // filas de la matriz
37   int n = atoi(argv[2]);  // columnas de la matriz
38
39   float *A = (float *)malloc(m*n*sizeof(float));  // matriz A
40   float *Al = (float *)malloc(m*n*sizeof(float));  // matriz A para lapack
41   float *Ac = (float *)malloc(m*n*sizeof(float));  // matriz A para cula
42
43   llenarm(m, n, A);  // llena la matriz
44
45   // copia matriz para lapack y cula
46   cblas_scopy(m*n, A, 1, Al, 1);  // copia A en Al Al=A
47   cblas_scopy(m*n, A, 1, Ac, 1);  // copia A en Ac Ac=A
48
49   // operaciones con lapack
50   lapackSgesvd_ejemplo(m, n, Al, m);
51
52   // operaciones con cula
53   culaSgesvd_ejemplo(m, n, Ac, m);
54
55   // libera memoria
56   free(A); free(Al); free(Ac);
57
58   // finaliza programa
59   return EXIT_SUCCESS;
60 }  // ------- end of function main -------
```

7. Toma de tiempo de operaciones básicas

7.1. Gráficas comparativas de tiempos de ejecución

En las siguientes gráficas se muestra el tiempo que se tarda una aplicación en realizar determinado número de operaciones en forma secuencial utilizando la CPU de la computadora para C, BLAS y LAPACK, y por otra parte utilizando las GPUs de

la tarjeta gráfica para CUDA, CUBLAS y CULA. Para conocer las características de la CPU y la tarjeta gráfica ver sección 5.1. Para cada programa se promediaron los tiempos de ejecución de 10 muestras.

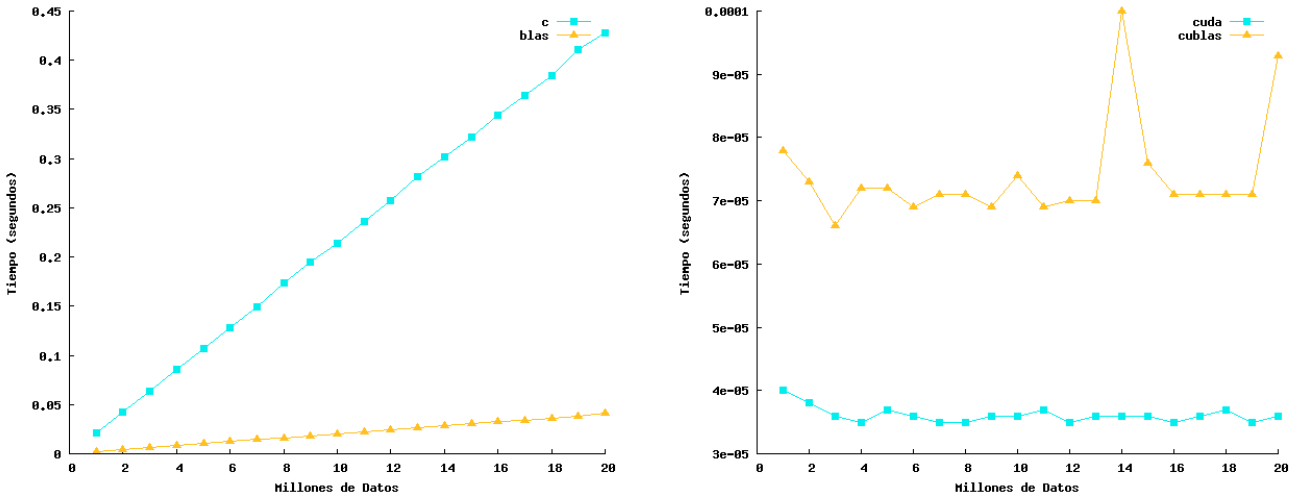
Es importante mencionar que para efectos prácticos en las tablas de tiempos de ejecución se presentan los datos con los que fue ejecutado dicho programa, pero para cada operación se debe considerar lo siguiente:

- Suma vector-vector $\alpha \times x + y$
  - en esta operación intervienen dos vectores, esto quiere decir que si en la tabla correspondiente se indica que los datos son 100, en realidad se está operando con 200 datos. Vector x de 100 datos y vector y de 100 datos. Sumados dan 200 datos operados en esa ejecución.
- Producto matriz-vector $\alpha \times A \times x + \beta \times y$
  - en esta operación intervienen dos vectores y una matriz, esto quiere decir que si en la tabla correspondiente se indica que los datos son 200, en realidad se está operando con 40,800 datos. Matriz de 200x200 datos y dos vectores de 200 datos cada uno. Sumados dan 40,800 datos operados en esa ejecución.
- Producto matriz-matriz $\alpha \times A \times B + \beta \times C$
  - en esta operación intervienen tres matrices, esto quiere decir que si en la tabla correspondiente se indica que los datos son 1500, en realidad se está operando con 6,750,000 datos. Tres matrices de 1500x1500 datos cada una. Sumados dan 6,750,000 datos operados en esa ejecución.
- Resolución de un sistema de ecuaciones lineales $Ax = b$
  - en esta operación intervienen una matriz y un vector, esto quiere decir que si en la tabla correspondiente se indica que los datos son 1800, en realidad se está operando con 3,241,800. Una matriz de 1800x1800 datos y un vector de 1800 datos. Sumados dan 3,241,800 datos operados en esa ejecución.
- Descomposición de una matriz en sus valores singulares $A = U\Sigma V^T$
  - en esta operación intervienen tres matrices y un vector, esto quiere decir que si en la tabla correspondiente se indica que los datos son 600, en realidad se está operando con 1,080,600 datos. Tres matrices de 600x600 datos cada una y un vector de 600 datos. Sumados dan 1,080,600 datos operados en esa ejecución.

En la tabla 10 se muestran los tiempos de ejecución en segundos de la suma vector-vector en relación al número de datos, tanto para C y BLAS (procesados en CPU), como para CUDA y CUBLAS (procesados en GPU). Se aprecia que, a mayor número de datos, el procesamiento en C se tarda más comparado con BLAS, y el procesamiento en CUBLAS se toma más tiempo comparado con CUDA; esto puede verse gráficamente en las figuras 28(a) y 28(b), también se observa que de las cuatro implementaciones, la de CUDA en GPU es la más eficiente.

Tabla 10: Tiempos de ejecución en segundos de la suma vector-vector

Datos en millones	**C**	**BLAS**	**CUDA**	**CUBLAS**
1	0.021643	0.002050	0.000040	0.000078
2	0.042839	0.004178	0.000038	0.000073
3	0.063991	0.006238	0.000036	0.000066
4	0.085601	0.008230	0.000035	0.000072
5	0.107084	0.010241	0.000037	0.000072
6	0.128478	0.012335	0.000036	0.000069
7	0.149578	0.014381	0.000035	0.000071
8	0.173134	0.016358	0.000035	0.000071
9	0.194890	0.018387	0.000036	0.000069
10	0.213373	0.020500	0.000036	0.000074
11	0.236187	0.022419	0.000037	0.000069
12	0.256899	0.024800	0.000035	0.000070
13	0.281570	0.026761	0.000036	0.000070
14	0.301335	0.028553	0.000036	0.000100
15	0.321626	0.030369	0.000036	0.000076
16	0.343961	0.032527	0.000035	0.000071
17	0.364049	0.034080	0.000036	0.000071
18	0.384838	0.036482	0.000037	0.000071
19	0.411192	0.038367	0.000035	0.000071
20	0.427978	0.040867	0.000036	0.000093



(a) C y BLAS

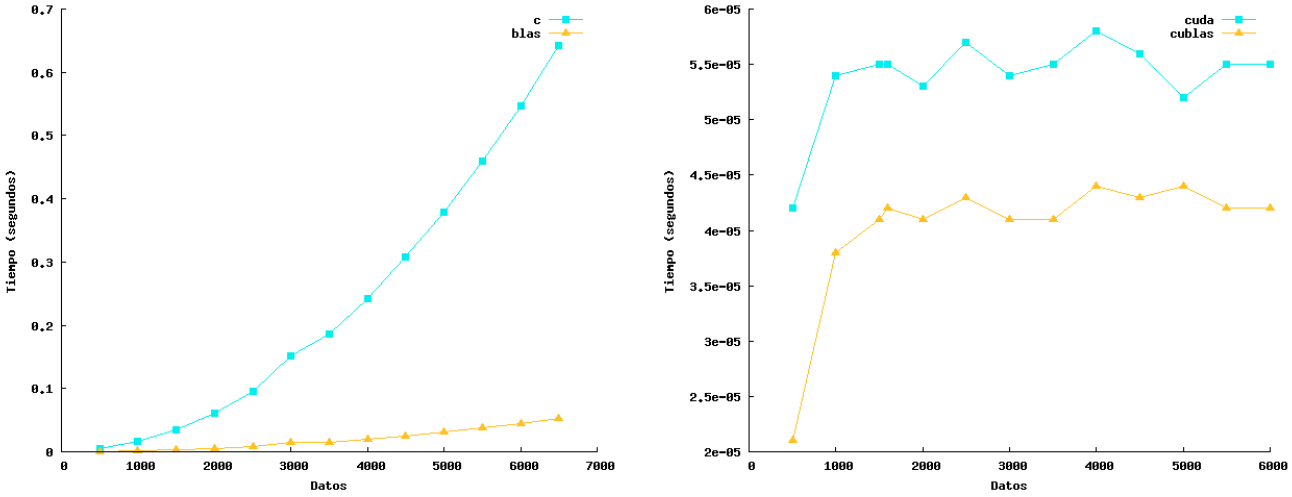
(b) CUDA y CUBLAS

Figura 28: Tiempos de ejecución en segundos de la suma vector-vector

En la tabla 11 se muestran los tiempos de ejecución en segundos del producto matriz-vector en relación al número de datos, tanto para C y BLAS (procesados en CPU), como para CUDA y CUBLAS (procesados en GPU). Se aprecia que a mayor número de datos, tanto para el procesamiento en C se tarda más comparado con BLAS, y el procesamiento con CUDA se toma más tiempo comparado con CUBLAS; esto puede verse gráficamente en las figuras 29(a) y 29(b), también se aprecia que de las cuatro implementaciones, la de CUBLAS en GPU es la más eficiente.

Tabla 11: Tiempos de ejecución en segundos del producto matriz-vector

Datos	**C**	**BLAS**	**CUDA**	**CUBLAS**
500	0.005057	0.000408	0.000042	0.000021
1000	0.016634	0.001278	0.000054	0.000038
1500	0.034150	0.002840	0.000055	0.000041
2000	0.060688	0.005031	0.000055	0.000042
2500	0.094846	0.007861	0.000053	0.000041
3000	0.151033	0.014702	0.000057	0.000043
3500	0.185785	0.015413	0.000054	0.000041
4000	0.242592	0.020066	0.000055	0.000041
4500	0.307384	0.025396	0.000058	0.000044
5000	0.379527	0.031338	0.000056	0.000043
5500	0.459511	0.037987	0.000052	0.000044
6000	0.546764	0.045141	0.000055	0.000042
6500	0.641677	0.052977	0.000055	0.000042



(a) C y BLAS

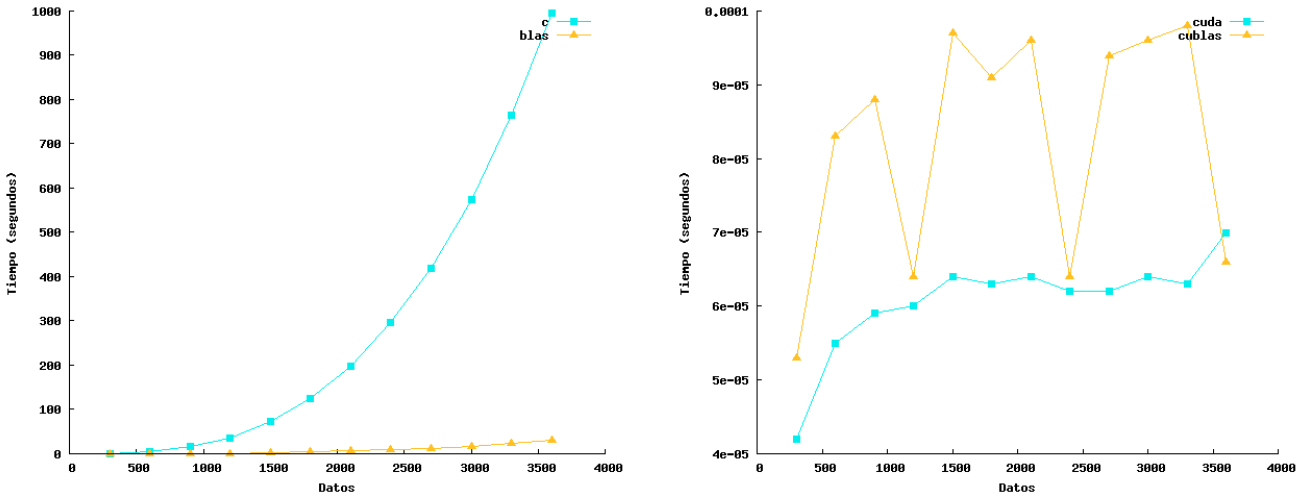
(b) CUDA y CUBLAS

Figura 29: Tiempos de ejecución en segundos del producto matriz-vector

En la tabla 12 se muestran los tiempos de ejecución en segundos del producto matriz-matriz en relación al número de datos, tanto para C y BLAS (procesados en CPU), como para CUDA y CUBLAS (procesado en GPU). Se aprecia que, a mayor número de datos, el procesamiento en C se tarda más comparado con BLAS, y el procesamiento en CUBLAS se toma más tiempo comparado con CUDA; esto puede verse gráficamente en las figuras 30(a) y 30(b), también se observa que de las cuatro implementaciones, la de CUDA en GPU es la más eficiente.

Tabla 12: Tiempos de ejecución en segundos del producto matriz-matriz

Datos	**C**	**BLAS**	**CUDA**	**CUBLAS**
300	0.543052	0.011647	0.000042	0.000053
600	4.419110	0.087529	0.000055	0.000083
900	15.393505	0.371912	0.000059	0.000088
1200	36.219259	1.088280	0.000060	0.000064
1500	71.795655	2.153596	0.000064	0.000097
1800	125.151296	3.710877	0.000063	0.000091
2100	197.723727	5.953435	0.000064	0.000096
2400	296.437968	8.829339	0.000062	0.000064
2700	419.761462	12.612607	0.000062	0.000094
3000	574.546915	17.226728	0.000064	0.000096
3300	763.680276	23.091469	0.000063	0.000098
3600	994.900824	29.854655	0.000070	0.000066



(a) C y BLAS

(b) CUDA y CUBLAS

Figura 30: Tiempos de ejecución en segundos del producto matriz-matriz

En la tabla 13 se muestran los tiempos de ejecución de la resolución de un sistema de ecuaciones lineales en relación al número de datos, calculada con LAPACK en CPU y con CULA en GPU. En la figura 31 puede observarse que con pocos datos el tiempo de ambos es casi el mismo y a medida que la cantidad de datos incrementa, también lo hace el tiempo de ejecución de LAPACK hasta el final de esta medición.

Tabla 13: Tiempos de ejecución en segundos de la resolución de un sistema de ecuaciones lineales con LAPACK y CULA

Datos	**LAPACK**	**CULA**
200	0.000	0.010
400	0.020	0.020
600	0.040	0.040
800	0.110	0.040
1000	0.200	0.060
1200	0.310	0.100
1400	0.480	0.130
1600	0.700	0.160
1800	0.980	0.210
2000	1.340	0.260
2200	1.760	0.340
2400	2.250	0.400
2600	2.870	0.500
2800	3.540	0.610
3000	4.350	0.720
3200	5.230	0.810
3400	6.280	0.990
3600	7.380	1.150
3800	8.700	1.320
4000	10.030	1.500
4200	11.780	1.680
4400	13.610	1.890
4600	15.860	2.130
4800	18.010	2.250
5000	20.800	2.720

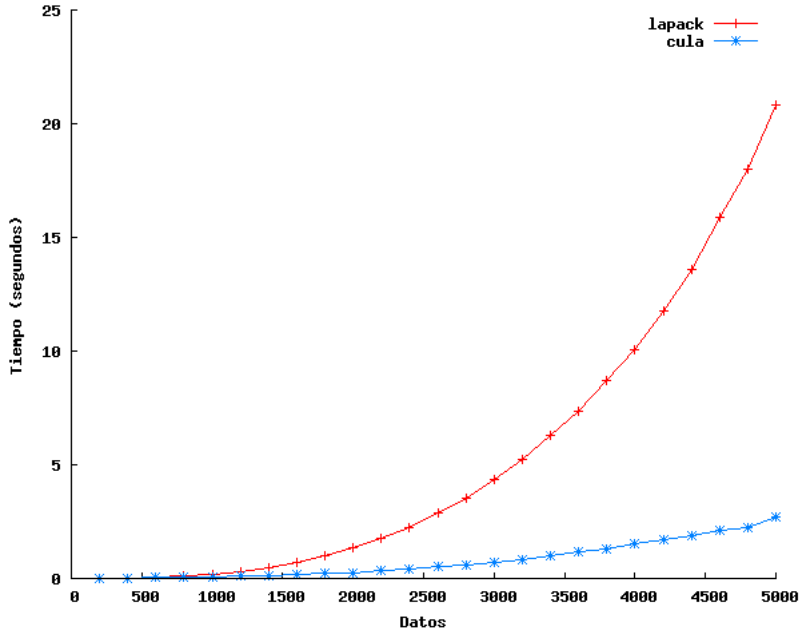


Figura 31: Tiempos de ejecución en segundos de la resolución de un sistema de ecuaciones lineales con LAPACK y CULA

En la tabla 14 se muestran los tiempos de ejecución de la descomposición de una matriz en sus valores singulares en relación al número de datos, calculada con LAPACK en CPU y con CULA en GPU. En la figura 32 puede observarse que al inicio el tiempo de ambos es casi el mismo y a medida que la cantidad de datos incrementa, también lo hace el tiempo de ejecución de LAPACK hasta el final de esta medición.

Tabla 14: Tiempos de ejecución en segundos de la descomposición de una matriz en sus valores singulares con LAPACK y CULA

Datos	**LAPACK**	**CULA**
200	0.050	0.210
400	0.410	0.560
600	1.550	1.300
800	3.870	2.530
1000	7.920	4.580
1200	13.950	7.290
1400	22.140	11.260
1600	33.160	15.670
1800	47.520	22.300
2000	65.230	30.460
2200	87.310	40.580

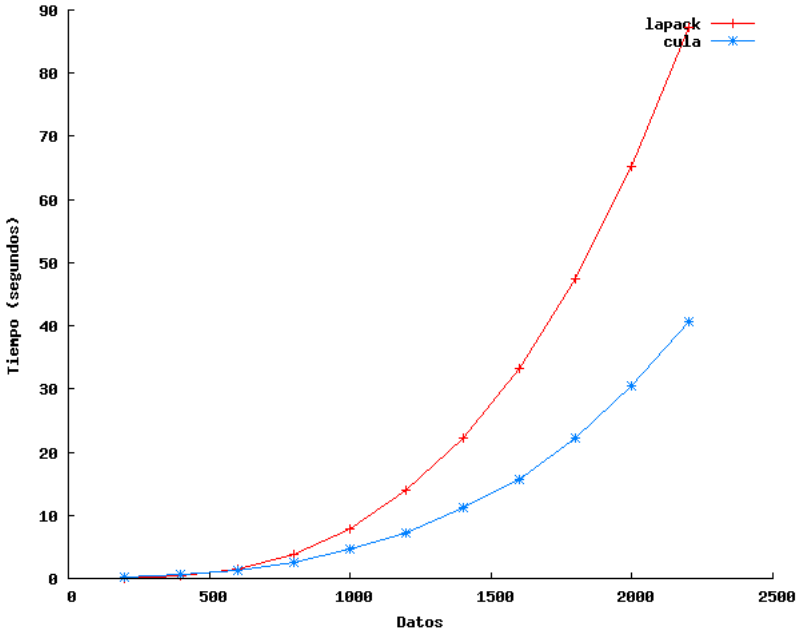


Figura 32: Tiempos de ejecución en segundos de la descomposición de una matriz en sus valores singulares con LAPACK y CULA

7.2. Gráficas comparativas de precisión

A continuación se muestran las gráficas del error (absoluto) de precisión (ver sección 4.4) obtenidas después de probar las rutinas de resolución de un sistema de ecuaciones lineales, **sgesv_** con LAPACK y **culasSgesv** con CULA, así como de la descomposición de una matriz en sus valores singulares, **sgesvd_** con LAPACK y **culaSgesvd** con CULA.

Para la resolución de sistemas de ecuaciones lineales se calculó la **2-norma** de las siguientes formas:

1. $||Ax - b||_2$
   Dados A y b se calcula x (programas 33 y 34). Una vez calculado el vector de soluciones se realiza el cálculo de la 2-norma con la matriz A multiplicada por el vector solución calculado x menos el lado derecho b. En la tabla 15 están los errores obtenidos y en la figura 33 se aprecia que a medida que incrementa el volumen de datos también lo hace el error de LAPACK.
2. $||x' - x||_2$
   Dados A y x se genera b (programa 32), una vez generado el lado derecho se calcula el vector de soluciones x'. Por último se calcula la 2-norma con x' calculado (programas 33 y 34) y x generado anteriormente. En la tabla 16 están los errores obtenidos y en la figura 34 se aprecia que LAPACK es mejor en algunos casos y CULA en otros.

En el caso de la descomposición de una matriz en sus valores singulares se calculó la norma **Frobenius** de la siguiente forma:

1. $||A - U\Sigma V^T||_F$
   Dada A se calculan los valores singulares Σ, los vectores singulares izquierdos U y los vectores singulares derechos V^T (programas 36 y 37). Por último se calcula la norma de Frobenius con la matriz A menos el producto de $U\Sigma V^T$. En la tabla 17 están los errores obtenidos y en la figura 35 se aprecia que el error de LAPACK y CULA son los mismos.

Tabla 15: Error de precisión $||Ax - b||_2$

Datos	**LAPACK**	**CULA**
200	0.0125631671	0.0106530404
400	0.0609354861	0.0439697243
600	0.1775842458	0.0978757367
800	0.2543732822	0.1908898801
1000	0.5173198581	0.2373087257
1200	0.7246696949	0.3751424253
1400	0.9136813879	0.5790876150
1600	1.4232378006	0.6865450144
1800	1.5044667721	0.7463110685
2000	1.9107308388	0.9146737456
2200	2.5800547600	1.1580615044
2400	2.8131930828	1.5290611982
2600	3.3206603527	1.7144433260
2800	5.1204848289	2.2013757229
3000	5.0487017632	2.2739510536
3200	7.0594434738	2.6864297390
3400	7.3425140381	2.7969024181
3600	7.7406258583	3.1721324921
3800	11.0346117020	3.4877324104
4000	8.8278856277	3.6917390823
4200	13.0904273987	4.1139025688
4400	11.1504449844	4.7407393456
4600	14.9798278809	5.6308784485
4800	13.7823114395	6.1132903099
5000	17.9968013763	6.7911257744

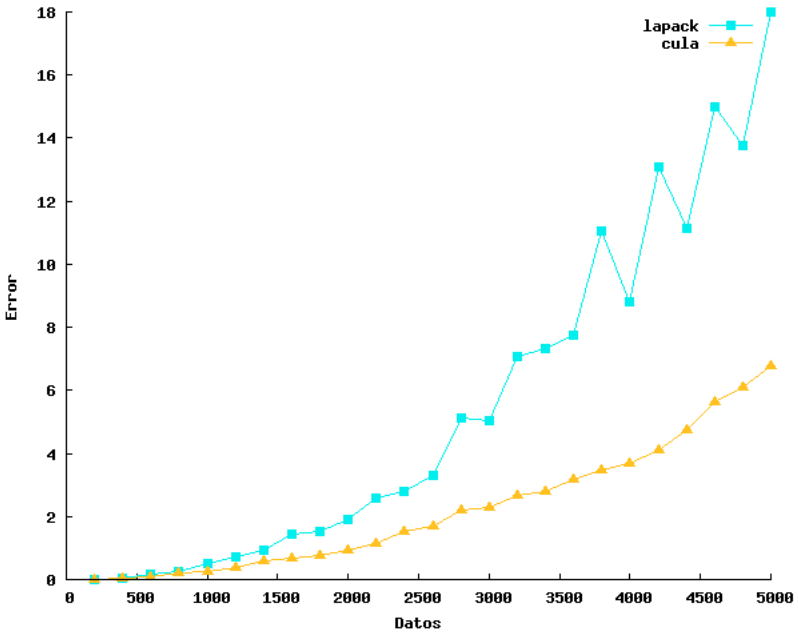


Figura 33: Error de precisión $||Ax - b||_2$

Tabla 16: Error de precisión $||x' - x||_2$

Datos	**LAPACK**	**CULA**
200	0.0028549773	0.0018318129
400	0.0028367571	0.0098393485
600	0.0846495107	0.9508686066
800	0.0190547165	0.0184939876
1000	1.5163561106	0.9447230101
1200	0.1360099465	0.0520490073
1400	0.0270938631	0.1015914381
1600	0.0503252447	0.0855503529
1800	0.0992848799	0.1017128900
2000	0.0944773331	0.2058163285
2200	1.5007501841	0.1001991779
2400	0.3128896356	0.0800881833
2600	0.3948256075	0.3300400674
2800	0.2790013850	0.4653852880
3000	1.3039089441	0.1582597345
3200	0.1806106418	0.2167799622
3400	0.2957033813	0.4053837359
3600	3.4129896164	0.9251242876
3800	0.1915441006	2.2761318684
4000	0.2063082010	0.3260892034
4200	0.3070977926	0.4070806503
4400	0.7783013582	1.0630515814
4600	15.1867141724	28.2282047272
4800	0.4206043184	0.5896192193
5000	1.1197817326	0.3390541673

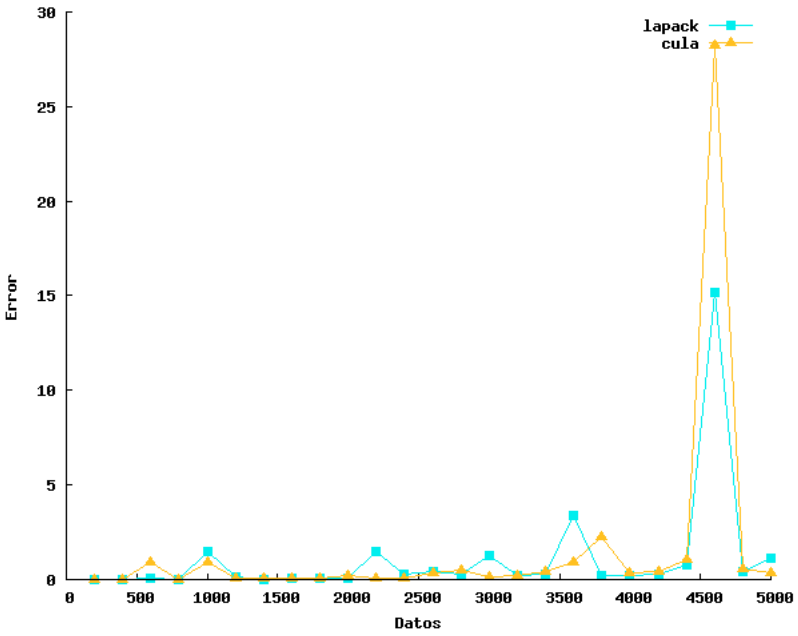


Figura 34: Error de precisión $||x' - x||_2$

Tabla 17: Error de precisión $||A - U\Sigma V^T||_F$

Datos	**LAPACK**	**CULA**
200	95.5473709106	95.5473709106
400	190.4156951904	190.4156951904
600	286.1187133789	286.1187133789
800	381.3372497559	381.3372497559
1000	475.4443054199	475.4443054199
1200	568.5411987305	568.5411987305
1400	663.0902709961	663.0902709961
1600	756.6641845703	756.6641845703
1800	850.1774291992	850.1774291992
2000	943.4375000000	943.4375000000
2200	1036.0960693359	1036.0960693359

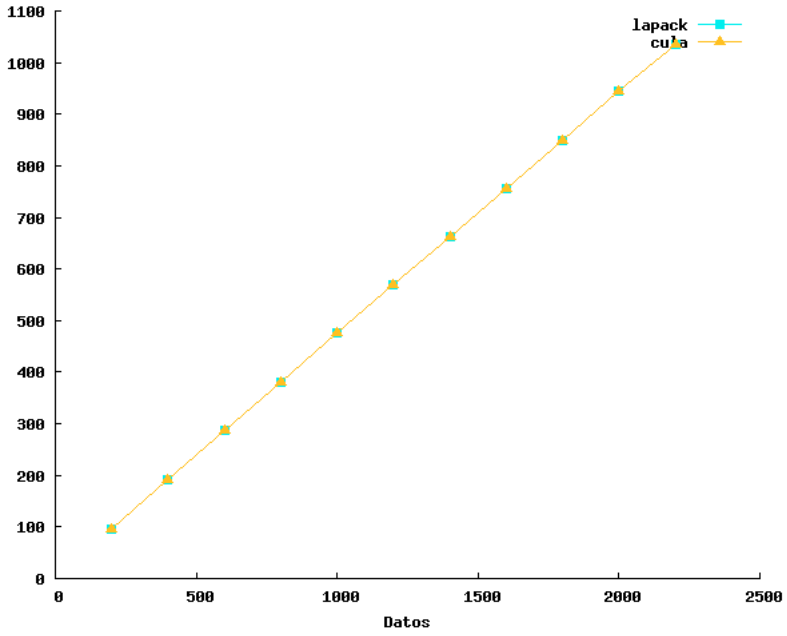


Figura 35: Error de precisión $||A - U\Sigma V^T||_F$

8. Conclusiones

- Este reporte técnico servirá como herramienta de apoyo para cualquier interesado en aprender las bases del manejo de las bibliotecas numéricas secuenciales BLAS y LAPACK (que trabajan en CPUs) y de las bibliotecas numéricas paralelas CUBLAS y CULA (que trabajan en GPUs).
- En este reporte se ha presentado la forma de instalar y configurar las bibliotecas numéricas secuenciales y paralelas mencionadas, así como la forma de resolver un conjunto de operaciones básicas del álgebra lineal mediante el llamado a rutinas de estas bibliotecas.
- Con el estudio numérico experimental presentado en este reporte se ha constatado que se pueden obtener programas paralelos más eficientes con las rutinas numéricas de CUBLAS y CULA, aprovechando así la potencialidad de un conjunto de núcleos de GPUs utilizados como procesadores de propósito general.

Referencias

[1] Aplicaciones de cuda. http://la.nvidia.com/object/cuda_in_action_la.html. [consultado 19-Octubre-2010].

[2] Badaboom. www.badaboomit.com. [consultado 19-Octubre-2010].

[3] Blas. http://www.netlib.org/blas/faq.html#1. [consultado 10-Agosto-2010].

[4] Hanweck associates. www.hanweckassoc.com. [consultado 19-Octubre-2010].

[5] Recommended C Style and Coding Standards. http://www.psgd.org/paul/docs/cstyle/cstyle.htm. [consultado 19-Octubre-2010].

[6] Seismiccity. www.seismiccity.com. [consultado 19-Octubre-2010].

[7] Techniscan. www.techniscanmedicalsystems.com. [consultado 19-Octubre-2010].

[8] Universidad de Illinois. www.ks.uiuc.edu/Research/namd/. [consultado 19-Octubre-2010].

[9] Universidades donde se enseña CUDA. http://www.nvidia.com/object/cuda_courses_and_map.html. [consultado 19-Octubre-2010].

[10] *CULA Programmer's Guide 2.1*. EM Photonics, 2010.

[11] *NVIDIA CUBLAS Guide 3.1*. NVIDIA, 2010.

[12] *NVIDIA CUDA C Programming Guide 3.1*. NVIDIA, 2010.

[13] *NVIDIA CUDA Reference Manual 3.1*. NVIDIA, 2010.

[14] Luis Joyanes Aguilar and Ignacio Zahonero Martínez. *Algoritmos y Estructuras de Datos Una perspectiva en C*. Mc Graw Hill, 2004.

[15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1999.

[16] Richard L. Burden and J. Douglas Faires. *Análisis Numérico*. Thomson Editores, 1998.

[17] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. SIAM, 1996.

[18] Vipin Kumar. *Introduction to parallel computing*. The Benjamin/Cummings Publishing Company, 1994.

[19] M. Morris Mano. *Arquitectura de Computadoras*. Prentice Hall, 1994.

[20] Kenneth H. Rosen. *Handbook of Discrete and Combinatorial Mathematics*. CRC-Press, 2000.

[21] Andrew Stuart Tanenbaum. *Organización de Computadoras un Enfoque Estructurado*. Pearson Educación, 2000.