UNIVERSIDAD NACIONAL DE HUANCAVELICA

(Creado por Ley 25265)

FACULTAD DE CIENCIAS DE INGENIERÍA

ESCUELA ACADÉMICA PROFESIONAL DE CIVIL (HUANCAVELICA)

TESIS

"TÉCNICAS NUMÉRICAS SOBRE PROCESADOR GRÁFICO, PARA LA SOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES EN ANÁLISIS ESTRUCTURAL"

LÍNEA DE INVESTIGACIÓN

ANÁLISIS ESTRUCTURAL

PRESENTADO POR:

Bach. CASTELLARES PAUCAR, William

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO CIVIL

HUANCAVELICA, PERÚ

2019

UNIVERSIDAD NACIONAL DE HUANCAVELICA

FACULTAD DE CIENCIAS DE INGENIERÍA

ACTA DE SUSTENTACIÓN DE TESIS

En el Auditórium de la Facultad de Ciencias de Ingeniería, a los 12 días del mes de diciembre del año 2019, a horas 12:00 m., se reunieron los miembros del Jurado Calificador conformado por los siguientes: **M.Sc. Iván Arturo AYALA BIZARRO (PRESIDENTE), Arq. Abdón Dante OLIVERA QUINTANILLA (SECRETARIO), M.Sc. Hugo Rubén LUJAN JERI (VOCAL),** designados con Resolución de Consejo de Facultad N° 338-2017-FCI-UNH, de fecha 03 de agosto del 2017, así mismo se cambió al Asesor y han sido reestructurado los miembros de Jurados Evaluadores con Resolución de Decano N° 196-2019-FCI-UNH, de fecha 30 de octubre del 2019 y con Resolución de Decano N° 281-2019-FCI-UNH de fecha 06 de diciembre del 2019 se ratifica al Asesor y Jurados evaluadores, a fin de proceder con la calificación de la sustentación del informe final de tesis titulado: "TÉCNICAS NUMÉRICAS SOBRE PROCESADOR GRÁFICO, PARA LA SOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES EN ANÁLISIS ESTRUCTURAL", presentado por el Bachiller **William CASTELLARES PAUCAR,** para optar el **Título Profesional de Ingeniero Civil**; en presencia del **Ing. Carlos GASPAR PACO** como Asesor del presente trabajo de tesis. Finalizado la evaluación a horas 1:20 p.m.; se invitó al público presente y al sustentante abandonar el recinto. Luego de una amplia deliberación por parte de los Jurados, se llegó al siguiente resultado:

APROBADO [X] POR...UNANIMIDAD.....

DESAPROBADO [ ]

En señal de conformidad, firmamos a continuación:

Presidente

Secretario

Vocal

Vº Bº Decano

“TÉCNICAS NUMÉRICAS SOBRE PROCESADOR GRÁFICO, PARA LA SOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES EN ANÁLISIS ESTRUCTURAL"

Autor:

William Castellares Paucar

Asesor:

Ing. Carlos Gaspar Paco

DEDICATORIA:

A mi madre, que en vida fue Irene Paucar Araujo, por el cariño, la paciencia, y todo el apoyo que me brindó.

A mi amigo, que en vida fue Ray Sugar Molina Huamaní, por todos esos años que pasamos, la amistad que me brindó, su compromiso y predilección hacia la investigación, y por su lucha incansable contra el cáncer.

AGRADECIMIENTOS

- A la Escuela Profesional de Ingeniería Civil- Huancavelica; a los docentes que intentan dar lo mejor de sí mismos en la formación de buenos profesionales en esta difícil carrera, y a los amigos que hice durante mi permanencia en esta escuela.
- A mi familia, en especial a mi hermana y sobrino por su apoyo y paciencia.

UNIVERSIDAD NACIONAL DE HUANCAVELICA
WANKA WILLKA
PERU

Contenido

Índice de tablas:

Índice de gráficos

Índice de figuras:

RESUMEN

El presente trabajo de investigación se centra principalmente en el uso del procesador gráfico (GPU) para paralelizar instrucciones en la solución de sistemas de ecuaciones lineales obtenidos de problemas de análisis estructural, es decir, dicho esto, nos limitamos a la solución de sistemas que tienen como matriz de coeficientes (matriz de rigidez): una matriz dispersa, cuadrada, simétrica y definida positiva. Los métodos considerados son: la factorización de Cholesky, la factorización LDL^T, el método de los gradientes conjugados y el método de Gauss Jordan.

Se han usado los formatos de almacenamiento CSC (*Compressed Sparse Column*) y SKS (*Skyline Storage*) para comprimir la matriz, adaptándolos para almacenar matrices simétricas (en donde solo la parte triangular inferior es almacenada), ya que la matriz, en su forma normal, almacena una gran cantidad de ceros, que en general, son innecesarios, con esto se busca mejorar tanto el rendimiento como el uso de memoria.

Se ha usado el OpenFEM, un conjunto de herramientas de elementos finitos, para analizar un problema determinado y obtener así el sistema de ecuaciones, es decir, la matriz de rigidez, luego, se ha comprimido la matriz usando uno de los formatos de almacenamiento, para luego resolver el sistema usando el kernel correspondiente. Los kernels (funciones que se ejecutan sobre la GPU) se han escrito usando la interfaz de OpenCL en C++, pero ejecutados por comodidad en Julia, mediante el uso de bibliotecas compartidas. Para poder trabajar con la memoria directamente en la GPU se usó la biblioteca de ArrayFire en C++, y el paquete ArrayFire.jl en Julia. Se ha medido la memoria utilizada y el tiempo de ejecución para diferentes órdenes de la matriz, y finalmente, se han preparado tablas para comparar resultados.

Palabras clave: Técnicas numéricas, procesador gráfico, GPU, Sistemas de ecuaciones lineales, análisis estructural.

ABSTRACT

The present research work focuses mainly on the use of the graphic processor (GPU) to parallelize instructions in the solution of systems of linear equations obtained from structural analysis problems, i.e. with that said, we limit ourselves to the solution of systems that have as a coefficient matrix (stiffness matrix): a dispersed, square, symmetric and positive-definite matrix. The methods considered are: Cholesky factorization, LDL^T factorization, the conjugate gradient method and the Gauss Jordan method.

The CSC (Compressed Sparse Column) and SKS (Skyline Storage) storage formats have been used to compress the matrix, adapting them to store symmetric matrices (where only the lower triangular part is stored), since the matrix, in its normal form, It stores a large number of zeros, which are generally unnecessary, this seeks to improve both performance and memory usage.

The OpenFEM, a set of finite element tools, has been used to analyze a given problem and thus obtain the system of equations, i.e. the stiffness matrix, then the matrix has been compressed using one of the storage formats, then solve the system using the corresponding kernel. The kernels (functions that run on the GPU) have been written using the OpenCL interface in C ++, but executed for convenience in Julia, through the use of shared libraries. In order to work with the memory directly on the GPU, the ArrayFire library in C ++ was used, and the ArrayFire.jl package in Julia. The memory used and the execution time for different matrix orders have been measured, and finally, tables have been prepared to compare results.

Keywords: numerical techniques, graphic processor, GPU, system of linear equations, structural analysis.

INTRODUCCIÓN

Los programas de análisis estructural más usados en la actualidad, usan el método de elementos finitos (MEF), método que consiste en discretizar sistemas continuos en elementos más simples, que se conectan mediante nodos (puntos en común), al analizar un problema con este método, se genera un **sistema de ecuaciones lineales (SEL)** que tiene como incógnitas los desplazamientos de los nodos, resolver este sistema de ecuaciones podría volverse un problema cuando el número de ecuaciones o incógnitas es considerablemente grande.

Se han propuesto infinidad de métodos para resolver sistemas de ecuaciones lineales de la forma $\boldsymbol{Ax = b}$, donde **A** es la matriz de coeficientes del sistema, **b** es el vector de elementos constantes y **x** es el vector incógnita, el más general consiste en obtener la inversa de la matriz A, después del cual podemos determinar el vector incógnita mediante la expresión $x = A^{-1}b$**,** sin embargo, a medida que el orden de A crece, calcular su inversa se vuelve complicado. Es por esto, que el uso de métodos alternativos resulta más adecuado, la principal razón es que la gran mayoría de estos métodos utilizan mucho menos recursos computacionales que el que se utilizaría para invertir la matriz, sin embargo, no todos los métodos son aplicables a un sistema de ecuaciones en particular, ya que el tipo de sistema de ecuaciones varía según el tipo de problema que se resuelva, para problemas de análisis estructural que nos concierne, la matriz de coeficientes (matriz de rigidez) es dispersa, cuadrada, simétrica y definida positiva; estas tres últimas características la hacen adecuada a la aplicación de la **factorización de Cholesky** y el método de los **gradientes conjugados,** métodos que hemos considerado en este trabajo de investigación, además, está el método **LDL^{T}**, método que no exige que la matriz sea definida positiva, y adicionalmente, se ha usado el método de **Gauss Jordan**, método más flexible en cuanto a las exigencias para la matriz de coeficientes aplicada a la matriz en formato denso (sin comprimir).

Otra característica de la matriz de rigidez que se puede explotar es su forma dispersa, es decir, la gran cantidad de ceros que puede llegar a tener, esto, desde el punto de vista computacional, puede aprovecharse almacenando en memoria solo los elementos distintos de cero o por lo menos almacenando ceros en la menor medida posible, los formatos de almacenamiento considerados en este trabajo de investigación son: el formato denso (formato por defecto en donde todos los n^2 elementos de la matriz son almacenados), el formato **CSC** (*Compressed Sparse Column*) y el **SKS** (*Skyline Storage*); adaptando estos dos últimos para almacenar una matriz simétrica, es decir, solo la parte triangular inferior de la matriz es almacenada.

El desarrollo de la tecnología ha logrado simplificar significativamente el trabajo de resolver SEL's, sin embargo, paralelo a esto, los problemas también son cada vez más complejos, uno de los principales enfoques que se desarrollan en la actualidad es el uso de **procesadores gráficos** (GPU's) para propósitos generales, nuestro propósito en este trabajo de investigación, fue el de resolver SEL's generados de un análisis estructural, buscando principalmente aprovechar su paralelismo. Durante alguna etapa en la aplicación de los métodos para resolver SEL's citados anteriormente, nos topamos con ciertas operaciones que pueden realizarse independientemente, es decir, no dependen una de la otra, esto permite la posibilidad de ejecutar instrucciones en paralelo, para este fin, la programación del procesador gráfico resulta de gran ayuda, sin embargo, no todas las instrucciones pueden hacerse en paralelo, dependiendo del método, hay instrucciones que imperativamente deben ejecutarse una después de otra, es aquí en donde se incluye el concepto de heterogeneidad, que consiste básicamente en el uso combinado de la **GPU** y la **CPU**, la GPU principalmente cuando se requiera o se pueda ejecutar instrucciones en paralelo y la CPU cuando las instrucciones deban ejecutarse en serie.

En resumen, lo que se ha buscado en el presente trabajo de investigación, es mediante la programación del procesador gráfico (GPU), proponer kernels (funciones que se ejecutan usando la GPU) que resuelvan un sistema de ecuaciones lineales generado de un análisis estructural, usando los métodos y los formatos de almacenamiento citados anteriormente, y probar su funcionamiento. Para determinar cuál técnica es más efectiva, se ha medido la memoria utilizada en MB (Mega Bytes) y el tiempo de

ejecución en segundos que utilizan para resolver el sistema, y se han preparado tablas para contrastar resultados.

CAPÍTULO I

PROBLEMA

1.1. Descripción del problema

Dentro del análisis de estructuras, casi siempre nos encontrarnos con problemas que consisten en encontrar el valor de ciertas variables, un ejemplo típico es el sistema de ecuaciones lineales $\boldsymbol{Ku = F}$ derivados de un análisis por rigideces o elementos finitos, donde $\boldsymbol{K}$ es la matriz de rigidez del sistema, $\boldsymbol{F}$ el vector de fuerzas y $\boldsymbol{u}$ el vector de desplazamientos que en este caso es la incógnita que tenemos que encontrar.

La solución de un **Sistema de ecuaciones lineales (SEL)** depende en gran medida de la capacidad de los ordenadores, como también del método que se emplee. El método más directo sería invertir la matriz, con la que se obtendría la solución directamente con $\boldsymbol{u = K^{-1}F}$, para ordenes pequeñas de la matriz, calcular su inversa no es un problema y puede obtenerse relativamente fácil, el problema comienza cuando el orden de la matriz crece, ya que a la par con esto, los recursos computacionales necesarios también aumentan y los recursos disponibles podrían no ser suficientes.

Considerando inviable la opción de invertir la matriz **K** por el gran tamaño que puede llegar a tener, se usan métodos alternativos para obtener el vector **u** que satisface el sistema; básicamente estos métodos se dividen en: métodos directos (Gauss-Jordan, factorización LU, LDL^T, Cholesky, etc.) y métodos iterativos (Jacobi, Gauss-Seidel, relajación, minimización, etc.); algunos de estos métodos exigen que la matriz tenga características particulares, como ser simétrica, ser diagonalmente dominante, ser definida positiva, etc. Es decir, no todos se pueden aplicar a un problema de análisis estructural. La matriz de rigidez es cuadrada, dispersa, simétrica y definida positiva;

lo que la hace adecuada a la aplicación de la factorización de Cholesky y el método de los gradientes conjugados, el método LDL^T solo exige que la matriz sea simétrica y no necesariamente definida positiva, por tal también es adecuado para usarse en este tipo de problemas, de los métodos generales que no tienen limitaciones en las características de la matriz de rigidez, el método de Gauss Jordan podría ser el más idóneo para utilizarse. Los recursos computacionales necesarios que demanda la utilización de los métodos citados anteriormente son mucho menores a los que se necesitarían si quisiéramos invertir la matriz.

La matriz de rigidez también es dispersa, es decir contiene una gran cantidad de ceros; al almacenar y resolver la matriz sin tomar en cuenta esto también resulta en un problema, ya que mientras más sean los ceros, el uso de la memoria resulta deficiente y a la par con esto las operaciones realizadas sobre estos ceros también son un desperdicio; existen formatos de almacenamiento que hacen posible almacenar solo los elementos distintos de cero u otros en donde el almacenamiento de ceros es mínimo, formatos como el CSR (*Compressed Sparse Row*), CSC (*Compressed Sparse Column*), SKS (*Skyline Storage*), son ideales para este fin, además pueden adaptarse para almacenar matrices simétricas en donde solo una de las partes triangulares de la matriz es almacenada, así, el almacenamiento se reduciría considerablemente y las operaciones realizadas sobre la matriz serían más eficientes, sin embargo, el uso eficiente de memoria no necesariamente conlleva a una mejora en el rendimiento, esto nos sugiere que debería tenerse cuidado en almacenar la matriz en un formato que nos permita un acceso a memoria eficiente.

Al programar de forma tradicional, también nos encontramos con la imposibilidad de paralelizar instrucciones, o por lo menos, la imposibilidad de hacerlo a grandes escalas. Los métodos mencionados anteriormente, en su procedimiento, contienen una gran cantidad de operaciones que pueden realizarse independientemente, un ejemplo simple sería la suma de vectores, al sumar elemento a elemento, cada suma es independiente de la otra y de ser posible podrían hacerse en paralelo, un código estándar realizaría esta suma por medio de un bucle en donde las sumas se realizarían en serie. El desarrollo de la tecnología ha hecho posible paralelizar instrucciones, el

uso de procesadores gráficos nos ofrece esa posibilidad, aunque estos no hayan sido creados inicialmente para este fin.

En resumen, el problema de resolver sistemas de ecuaciones lineales se centra en, la búsqueda del método más idóneo, la utilización de un formato de almacenamiento que optimice mejor el uso de memoria y finalmente, la necesidad de paralelizar instrucciones; es a estos tres problemas a los que intentamos buscar solución en este trabajo de investigación.

1.2. Formulación del problema

1.2.1. Problema general

- ¿Cuáles son las características de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural?

1.2.2. Problemas específicos

- ¿Cuál es el tiempo de ejecución de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural?

- ¿Cuál es la memoria utilizada de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural?

1.3. Objetivos

1.3.1. Objetivo general

- Determinar las características de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural.

1.3.2. Objetivos específicos

- Determinar el tiempo de ejecución de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural.

- Determinar la memoria utilizada de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural.

1.4. Justificación

Un ingeniero civil se enfrenta en su día a día a la solución de problemas complejos que lo obligan a apoyarse en la tecnología, esto es necesario en casi todas las áreas de la ingeniería, un ejemplo típico es el cálculo de estructuras, que incluyen, análisis sísmicos, diseño de elementos, cálculo de cimentaciones, etc. Para todo esto siempre existe algún software especializado, sin embargo, la mayoría de programas informáticos que se utiliza a nivel local es extranjero, las desventajas, posiblemente la barrera de idioma, la complejidad en el uso y en muchos casos los precios poco accesibles. A pesar de lo dicho anteriormente, que casi para todo lo que quiera hacer un ingeniero existe algún software especializado, no debemos descartar el hecho de toparnos con problemas aislados, que no puedan ser abarcados o resueltos por los programas que se tenga a la mano, para estos casos, sería ideal tener a nuestro alcance herramientas que nos permitan desarrollar software especializado, anulando así, la necesidad de buscar ayuda externa. Cabe resaltar, que no sería descabellada la idea de desarrollar software a nivel local para resolver problemas típicos, las ventajas serían obvias, precios más cómodos, la eliminación de la barrera de idioma, el fácil manejo, el rápido acceso a soporte, entre otros.

En análisis estructural y en muchas otras ramas de la ingeniería, resolver sistemas de ecuaciones lineales es un paso obligado y probablemente el que más tiempo consume, este trabajo de investigación abarca a nivel computacional este tema, intentado explotar el paralelismo de una GPU, optimizar la memoria utilizada y mejorar el

rendimiento. A nivel local, las investigaciones que abarcan este tipo de temas son muy pocos, principalmente por la falta de interés, ya que de alguna manera se ven como algo secundario, algo un poco contradictorio con el hecho de que el uso de software es casi imperativo en todas las áreas de la ingeniería.

1.5. Limitaciones

- No se encontró bibliografía en nuestro idioma, de manuales o guías de programación en OpenCL, las guías de programación que vienen junto con el instalador están en inglés y son casi únicos, por lo menos durante la ejecución de la tesis, no se encontraron manuales o guías adicionales. Casos parecidos ocurrieron con la biblioteca de ArrayFire y la herramienta de elementos finitos OpenFEM.
- La incompatibilidad con la plataforma CUDA del dispositivo (GPU) usada para la tesis, que en principio era el entorno en donde se pensaba realizar la programación del procesador gráfico, obligó a la búsqueda de otras alternativas de programación, es así, que se elige la API de OpenCL para este propósito.
- La dificultad de programar el procesador gráfico y obtener resultados consistentes fue complicado en un principio, la obtención de errores en tiempo de ejecución, de compilación, de sintaxis y otros, fue algo recurrente al principio.
- Luego de superar las dificultades anteriores, pasamos a otro un poco más complicado de manejar, nos referimos al rendimiento, hablando en términos de memoria y tiempo de ejecución; a pesar de que se han mejorado los códigos paulatinamente, no se ha llegado a alcanzar el nivel de rendimiento deseado.
- Para órdenes de matrices considerablemente grandes, la escritura de archivos resultó un tanto lenta.

CAPÍTULO II

MARCO TEÓRICO

2.1. Antecedentes

- ✓ (Sánchez Meza, 2000) **Sánchez Meza, Roque Alberto.** Procedimientos de gradiente conjugada en el análisis estructural. [ed.] Universidad Nacional de Ingeniería. Programa Cybertesis PERÚ. 2000.

Conclusiones: (Las más relevantes, propias de la referencia).

- En el análisis de pórticos planos, los valores de tiempo para el programa PT (método de Eliminación Gaussiana), son menores que aquellos que utilizan cualquier variante de gradiente conjugada, pero conforme aumenta el número de grados de libertad, la diferencia, en tiempo, se hace notable a favor de los métodos de gradiente conjugada.

- En el análisis tridimensional, las comparaciones de tiempo entre los programas SFE (análisis tridimensional de pórticos utilizando el método de Eliminación Gaussiana) y los métodos: gradiente conjugada con energía potencial (G.C. E.P.) y gradiente conjugada GC, son sustancialmente diferentes lo que nos ratifica que los procedimientos de gradiente conjugada produce los mismos resultados y además en un tiempo mucho menor. En todos los casos analizados los porcentajes de tiempo utilizados por los algoritmos de gradiente están muy por debajo del usado por el programa SFE.

- Los ejemplos tridimensionales no hicieron más que confirmar lo que ya se había avisorado en los ejemplos bidimensionales, es decir que el procedimiento empleado lleva a tiempos de cómputo más efectivos y rápidos, pero sin descuidar la precisión de los cálculos.

  - ✓ (Juan Carlos Camacho Puello, 2012) **Juan Carlos Camacho Puello, Marlos de Jesús Romero Torres**. Análisis estructural con el método de elementos finitos asistido por computadora. Cartagena de Indias: Universidad Tecnológica de Bolivar, 2012.
  - ✓ (Félix, 2015) **Félix, Miguel Vargas.** Factorización Cholesky simbólica. [En línea] 2015.

  Conclusiones: La referencia (Juan Carlos Camacho Puello, 2012) se basa en los resultados de (Félix, 2015) aunque se detallan más en (Juan Carlos Camacho Puello, 2012) el cual no tiene conclusiones, estas son transcritas directamente de la referencia (Juan Carlos Camacho Puello, 2012)**.**

- La descomposición de Cholesky es muy aplicada en programas estructurales que aplican el método de los elementos finitos, dado que la matriz de coeficientes es una matriz simétrica y definida positiva. BABEL 2.0[1] utiliza esta descomposición por ser fácil de implementar y por consumir menos recursos que algunos de los métodos tradicionales (pág. 58).

- En una prueba que llevo a cabo por (Félix, 2015) se utilizaron varios métodos de solución de ecuaciones para resolver una estructura compleja. La estructura (Figura 1) fue dividida en 264,250 elementos y 326,228 nodos. El tamaño de la matriz de rigidez es 978,684. Se resolvió el problema variando el número de procesadores usados. Como comparación, los resultados obtenidos usando el método del gradiente conjugado en paralelo

[1] *Programa de análisis estructural producto de la tesis de referencia*

para resolver el problema. El método del gradiente conjugado fue usado con y sin pre condicionamiento, el precondicionador usado es Jacobi. La tolerancia en la norma del gradiente usado para solvers iterativos es 1×10^{-5}. En la Figura 2 y Figura 3, los valores entre paréntesis representan el número de procesadores usados en paralelo (pág. 64).

- En las imágenes se puede ver que la descomposición de Cholesky tarda menos tiempo en solucionar el problema que los otros métodos, pero es menos eficiente en términos de consumo de memoria. Cabe aclarar que el problema presenta una gran cantidad de variables que pueden generar que la descomposición de Cholesky parezca un método poco eficaz, pero debido a que BABEL 2.0 trabaja con un número de variables más reducido, el tiempo de resolución del sistema será mucho menor y habrá un consumo de memoria que no alterara el rendimiento normal del equipo (pág. 66).

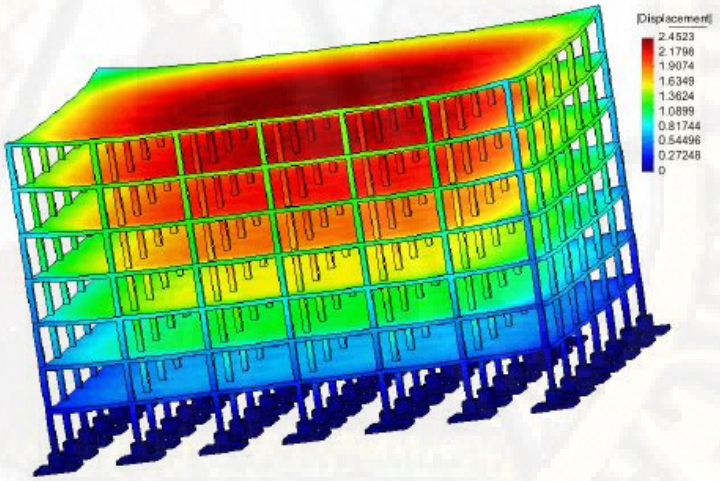


Figura 1. Ejemplo numérico de la deformación de una estructura (Félix, 2015)

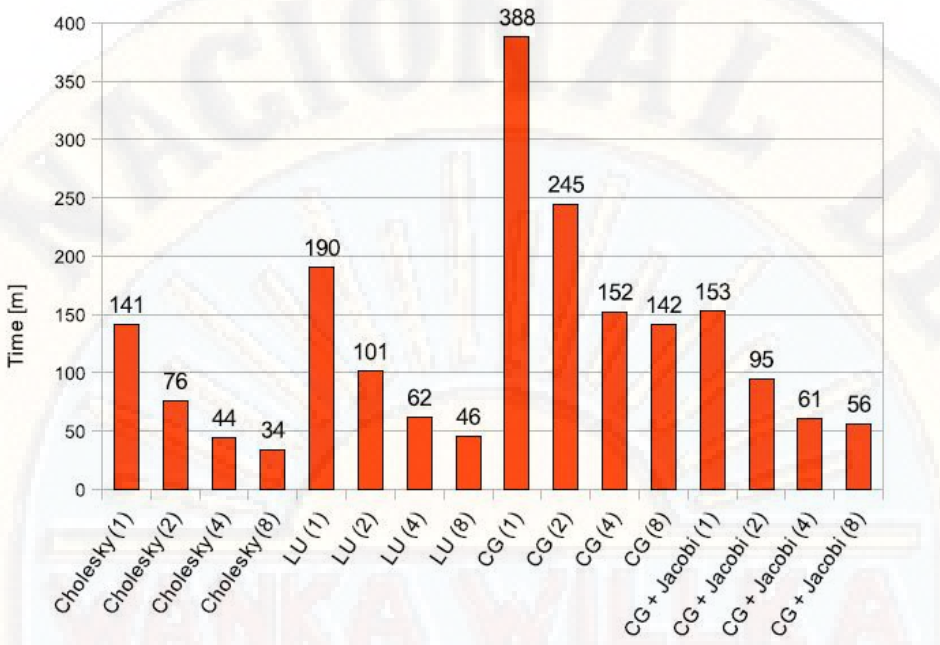


Figura 2. Comparación de los tiempos de solución de los diferentes solvers (Félix, 2015)

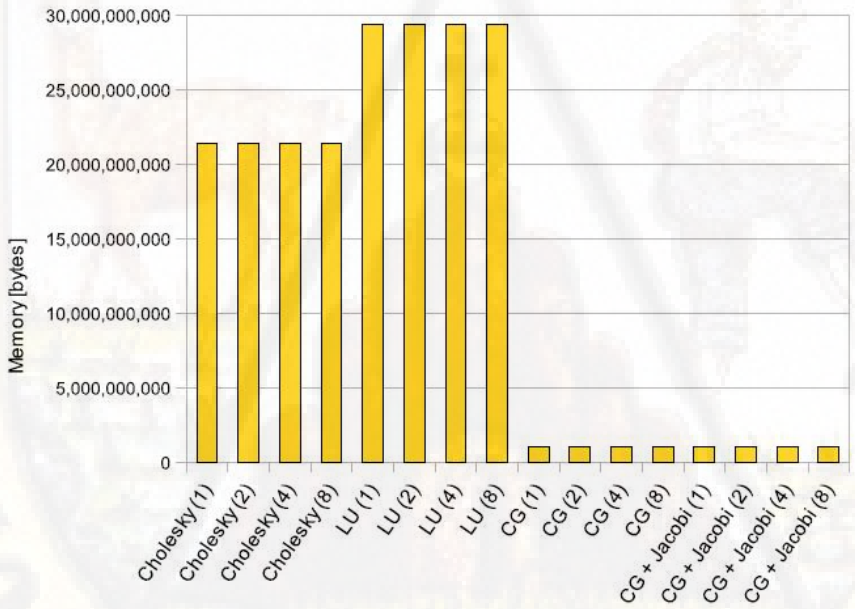


Figura 3. Comparación de la capacidad de memoria usada por los diferentes solvers paralelos (Félix, 2015)

2.2. Bases teóricas sobre el tema de investigación

2.2.1. Sistema de ecuaciones lineales (SEL)

Un sistema de ecuaciones lineales (SEL) es un conjunto de expresiones (ecuaciones) de primer grado que deben ser resueltas para un determinado número de variables.

El problema del SEL es uno de los más antiguos de la matemática y tiene una infinidad de aplicaciones, como en procesamiento digital de señales, análisis estructural, estimación, predicción y más generalmente en programación lineal así como en la aproximación de problemas no lineales de análisis numérico.[2]

En forma general un SEL tiene la siguiente forma:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2$$

$$\cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m$$

Donde x_1, x_2, x_3,…x_n, son incógnitas y a_{ij} junto con b_i {i=1, 2,3,…, m}, {j=1, 2,3,…, n} son constantes. Este conjunto de ecuaciones puede escribirse en forma matricial de la siguiente manera:

$$\begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} & x_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & x_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & x_n \end{matrix} = \begin{matrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{matrix}$$

O de forma más compacta: **Ax=b,** en donde A_{mxn} es la matriz de coeficientes del sistema, **x** un vector incógnita y **b** un vector con términos constantes.

[2] *Extraído de https://es.wikipedia.org/wiki/Sistema_de_ecuaciones_lineales el 06/03/2017*

2.2.1.1.Métodos de solución de sistemas de ecuaciones lineales

Existen diversidad de métodos para la solución de SEL's estos se pueden dividir básicamente en dos grupos, los métodos directos y los métodos iterativos; los primeros suelen consistir en una cantidad finita de operaciones ordenadas que deben seguirse hasta determinar los valores de las incógnitas, los segundos no tienen una cantidad determinada de operaciones ya que dependen del número de iteraciones seguidas hasta llegar a una solución exacta o por lo menos aceptable, de aquí en adelante se considerará que la matriz **A** es cuadrada de orden "n" y por tanto el número de ecuaciones es igual al número de incógnitas.

2.2.1.1.1. Método de Gauss-Jordan

En matemática, la eliminación de Gauss-Jordan, llamada así debido a **Carl Friedrich Gauss y Wilhelm Jordan**, es un algoritmo del álgebra lineal para determinar las soluciones de un sistema de ecuaciones lineales, encontrar matrices e inversas. Un sistema de ecuaciones se resuelve por el método de Gauss cuando se obtienen sus soluciones mediante la reducción del sistema dado a otro equivalente en el que cada ecuación tiene una incógnita menos que la anterior. El método de Gauss transforma la matriz de coeficientes en una matriz triangular superior. El método de Gauss-Jordan continúa el proceso de transformación hasta obtener una matriz diagonal[3].

El proceso generalizado consiste en n pasos (n es el orden de la matriz) y es el siguiente: considerar que en cada proceso el elemento de la diagonal principal es diferente de cero, si no es así podría intercambiarse las filas para cumplir esta condición.

Las operaciones se realizan sobre la matriz aumentada, es decir la matriz **A** unida al vector de elementos constantes, o sea:

[3] *Extraído de https://es.wikipedia.org/wiki/Eliminación_de_Gauss-Jordan el 06/03/17*

$$\begin{matrix} a_{11} & a_{12} & \dots & a_{1n} & b_1 \\ a_{21} & a_{22} & \dots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & b_n \end{matrix}$$

Para un paso i, la intención es convertir toda la columna i en ceros, excepto el elemento correspondiente a la diagonal, para lograr esto, a cada fila (a excepción de la fila i), se le suma la fila i multiplicada por un factor, tal que el i-ésimo elemento de la fila se vuelva cero.

Luego de realizar los n pasos, debe obtenerse una matriz diagonal, es decir la matriz aumentada tendría la siguiente forma:

$$\begin{matrix} a'_{11} & 0 & \dots & 0 & b'_1 \\ 0 & a'_{22} & \dots & 0 & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & a'_{nn} & b'_n \end{matrix}$$

Luego de esto, encontrar cada x_i se vuelve sencillo mediante las divisiones: $x_i = b'_i / a'_{ii}$ con $i = 1,2,3, \dots, n$

2.2.1.1.2. Factorización LDLT

Es un método de factorización aplicable a matrices simétricas que consiste en descomponer la matriz A en tres submatrices, dos triangulares unitarias L y L^T, tal que una es la transpuesta de la otra y otra diagonal D tal que:

$$A = LDL^T.$$

Con:

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} d_1 & 0 & 0 & \dots & 0 \\ 0 & d_2 & 0 & \dots & 0 \\ 0 & 0 & d_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & d_n \end{bmatrix}$$

Realizando la multiplicación obtenemos las siguientes fórmulas

$$d_1 = a_{11}$$

$$l_{i1} = a_{i1}/d_1$$

$$d_i = a_{ii} - \sum_{k=1}^{i-1} l_{ik}{}^2 \, d_k$$

$$l_{ij} = \quad a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \, d_k \qquad d_j$$

La última expresión solo es válida cuando i>j ya que por ser una matriz triangular inferior si j>i entonces $l_{ij} = 0$.

Luego de conseguir cada elemento de L y D usando las expresiones anteriores, se procede a resolver los sistemas equivalentes: **Lz=b, Dy=z** y finalmente $\mathbf{L^T x = y}$**,** que es simple ya que L es triangular y D es diagonal.

2.2.1.1.3. Factorización de Cholesky

En matemática, la **factorización o descomposición de Cholesky** toma su nombre del matemático **André-Louis Cholesky**, quien encontró que una **matriz simétrica definida positiva** puede ser descompuesta como el producto de una matriz triangular inferior y la traspuesta de la matriz triangular inferior. La matriz triangular inferior es el triángulo de Cholesky de la matriz original positiva definida. El resultado de Cholesky ha sido extendido a matrices con entradas complejas. Es una manera de resolver sistemas de ecuaciones matriciales y se deriva de la factorización LU con una pequeña variación[4].

Dado esto, si en el sistema **Ax=b,** la matriz A es simétrica y definida positiva puede descomponerse en la forma:

[4] *Extraída de https://es.wikipedia.org/wiki/Factorización_de_Cholesky el 06/03/2017*

$$A = LL^T$$

Donde la matriz L es triangular inferior, luego al igual como se hace con la factorización LU la solución del sistema Ax=b, se reduce a resolver los sistemas equivalentes **Ly=b, y L^Tx=y.**

La forma de la matriz L es la siguiente:

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{44} \end{bmatrix}$$

Al reemplazar en la expresión $A=LL^T$ obtenemos las siguientes fórmulas:

$$l_{11=\sqrt{a_{11}}}$$

$$l_{i1} = a_{i1}/l_{11}$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}$$

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}\right) \quad l_{jj}$$

La última expresión solo es válida cuando i>j ya que por ser una matriz triangular inferior si j>i entonces $l_{ij} = 0$.

2.2.1.1.4. Método de los gradientes conjugados

En matemática, el método de **los gradientes conjugados** es un algoritmo para resolver numéricamente los sistemas de ecuaciones lineales cuyas matrices son simétricas y definidas positivas. Es un método iterativo, así que se puede aplicar a los sistemas dispersos que son demasiado grandes para ser tratados por métodos directos como la descomposición de

Cholesky. Tales sistemas surgen frecuentemente cuando se resuelve numéricamente las ecuaciones en derivadas parciales[5].

Definimos la función $f: R^n \to R$ mediante:

$$f(x) = \frac{1}{2}x^T Ax - b^T x$$

Ambos términos a la derecha representan un producto interno, ya que el rango de la función son números reales.

Si A es simétrica se cumple que:

$$\nabla f(x) = Ax - b$$

Si $f(x)$ tiene un mínimo entonces este resultará de igualar su gradiente $\nabla f(x)$ a cero, es decir cuando $Ax = b$, por tanto resolver el sistema $Ax = b$ es equivalente a buscar el mínimo de la función f.

Puede demostrarse que para que el mínimo exista entonces A debe ser definida positiva.

Tomando como primera aproximación el vector x_0 la primera dirección a usar será el negativo de la gradiente en éste punto es decir $p_0 = r_0 = b - Ax_0$, a partir de estos valores puede procederse con la iteración:

Entonces, teniendo una aproximación anterior x_k y una nueva dirección de avance p_k el valor de α_k que hace mínima la función en ésta dirección es:

$$\alpha_k = \frac{r_k^T p_k}{p_k^T A p_k}$$

La nueva aproximación se determina entonces con:

$$x_{k+1} = x_k + \alpha_k p_k$$

[5] *Extraído de https://es.wikipedia.org/wiki/Método_del_gradiente_conjugado el 07/03/17*

Multiplicando por A y considerando que $r_k = b - Ax_k$ se llega a:

$$r_{k+1} = r_k - \alpha_k A p_k$$

La siguiente dirección se determinará con:

$$p_{k+1} = r_{k+1} + \beta_k p_k$$

El valor de β_k que hace que las direcciones sean conjugadas es decir que el producto $p_k^T A p_{k+1}$ sea cero es:

$$\beta_k = -\frac{p_k^T A r_{k+1}}{p_k^T A p_k}$$

Se vuelve a determinar α_{k+1} para la nueva dirección y se continua el proceso hasta acercarnos todo que queramos a la solución del sistema.

A partir de una matriz A de rango n solo se pueden definir n vectores A-conjugados, por lo tanto el algoritmo de los gradientes conjugados garantiza la obtención de una solución en un máximo de n iteraciones (Vargas-Félix, 2015).

2.2.2. Uso de procesadores gráficos (GPU) en propósitos generales

2.2.2.1.GPGPU (Wikipedia, 2013)

GPGPU o **General-Purpose Computing on Graphics Processing Units** es un concepto reciente dentro de informática que trata de estudiar y aprovechar las capacidades de cómputo de una GPU.

Una **GPU** es un procesador diseñado para los cómputos implicados en la generación de gráficos 3D interactivos. Algunas de sus características (bajo precio en relación a su potencia de cálculo, gran paralelismo, optimización para cálculos en coma flotante), se consideran atractivas para su uso en aplicaciones fuera de los gráficos por computadora, especialmente en el ámbito científico y de simulación. Así, se han desarrollado técnicas para la implementación de simulaciones de fluidos, bases de datos, algoritmos de clustering, etc.

2.2.2.1.1. Modelo de programación GPU

Debido a las diferencias fundamentales entre las arquitecturas de la **GPU y la CPU,** no cualquier problema se puede beneficiar de una implementación en la GPU. En concreto, el acceso a memoria plantea las mayores dificultades. Las CPU están diseñadas para el acceso aleatorio a memoria. Esto favorece la creación de estructuras de datos complejas, con punteros a posiciones arbitrarias en memoria. En cambio, en una GPU, el acceso a memoria está mucho más restringido. Por ejemplo, en un procesador de vértices (la parte de una GPU diseñada para transformar vértice en aplicaciones 3D), se favorece el modelo scatter, en el que el programa lee en una posición predeterminada de la memoria, pero escribe en una o varias posiciones arbitrarias. En cambio, un procesador de píxeles, o fragmentos, favorece el modelo gather, pudiendo el programa leer de varias posiciones arbitrarias, pero escribir en solo una posición predeterminada.

La tarea del diseñador de algoritmos GPGPU consiste principalmente en adaptar los accesos a memoria y las estructuras de datos a las características de la GPU. Generalmente, la forma de almacenar datos es en un buffer 2D, en lugar de lo que normalmente sería una textura. El acceso a esas estructuras de datos es el equivalente a una lectura o escritura de una posición en la textura. Puesto que generalmente no se puede leer y escribir en la misma textura, si esta operación es imprescindible para el desarrollo del algoritmo, este se debe dividir en varias pasadas.

Pese a que cualquier algoritmo que sea implementable en una CPU lo es también en una GPU, esas implementaciones no serán igual de eficientes en las dos arquitecturas. En concreto, los algoritmos con un alto grado de paralelismo, sin necesidad de estructuras de datos complejas y con una alta intensidad aritmética son los que mayores beneficios obtienen de su implementación en la GPU.

2.2.2.2.Programación de GPU en Julia (Juliacomputing.com, s.f.)[6]

Los problemas científicos se han resuelto tradicionalmente usando poderosos grupos de CPU's homogéneos, conectados en una variedad de topologías de red. Sin embargo, el número de superordenadores que emplean aceleradores ha ido en constante aumento, La última lista Top 500 publicada en SC'15 muestra que el número de supercomputadores que emplean aceleradores ha aumentado a 109.

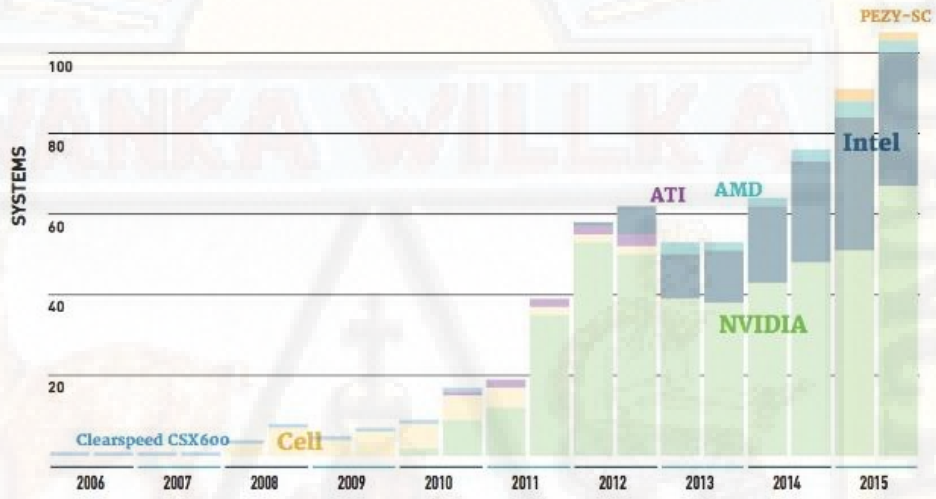


Figura 4. Número de supercomputadores que emplean aceleradores, extraída de (Juliacomputing.com, s.f.)

Los aceleradores que se emplean en la práctica son en su mayoría unidades de procesamiento gráfico (**GPU** por sus siglas en ingles), Xeon Phis y FPGAs. Estos aceleradores se aprovechan de las arquitecturas de núcleos múltiples que pueden utilizarse para explotar y cristalizar el paralelismo. Sin embargo el problema tradicional al usar las GPU's y otros aceleradores ha sido la facilidad (o falta de ello) a la hora de programarlos. Para este fin la corporación **NVIDIA** diseñó la actual Arquitectura Unificada de Dispositivos de Cómputo (CUDA por sus siglas en inglés) que incorpora en C una interface para la programación

[6] *La referencia originalmente en inglés, ha sido traducida para poder ser añadida en el marco teórico*

científica y de propósito general. Esta fue una considerable mejora sobre sistemas previos como DirectX u OpenGL que requerían habilidades avanzadas de programación gráfica. Sin embargo, **CUDA** todavía se caracteriza por una baja curva de productividad, para programadores que tienen que afinar sus aplicaciones para diferentes dispositivos y algoritmos. En este contexto, la programación interactiva sobre GPU's proporcionaría grandes beneficios a científicos y programadores que no solo desearían poner a prueba sus aplicaciones, sino implementarlas con pocos o ningún cambio en su código.

2.2.2.2.1. Julia sobre GPU's

Julia ofrece a los programadores la capacidad de codificar interactivamente en la GPU. Hay varias librerías incluidas en Julia, dando a los usuarios acceso a acelerados BLAS, FFT's, rutinas para matrices dispersas y solucionadores, y aprendizaje profundo. Con una combinación de estos paquetes, los programadores pueden desarrollar interactivamente Kernels (funciones) GPU personalizadas. Un ejemplo es el de los gradientes conjugados, el cual fue comparado a continuación.

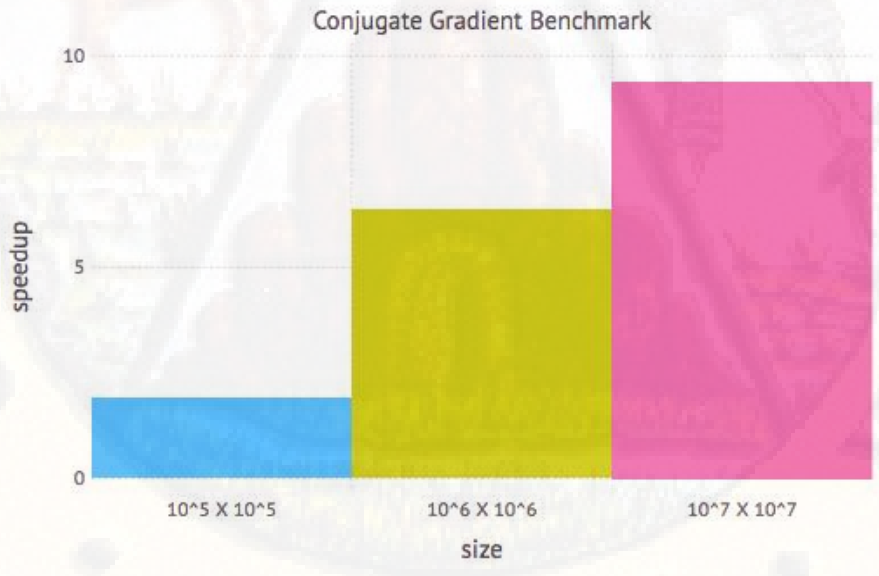


Figura 5. Comparación de la aceleración del gradiente conjugado en diferentes tamaños, extraído de (Juliacomputing.com, s.f.)

Sin embargo, se podría argumentar que las librerías de nivel inferior no aumentan en modo alguno la productividad del programador, ya que implican trabajar con interfaces de funciones oscuras. En tal caso, sería conveniente tener una interfaz limpia para matrices en la GPU con una librería estándar conveniente que pueda trabajar con matrices. Cada operación se sintonizaría con el dispositivo en cuestión para lograr un gran rendimiento. La gente de **ArrayFire** ha reunido una biblioteca de código abierto de alta calidad para trabajar en problemas científicos con GPU's.

2.2.2.2.2. ArrayFire.jl

ArrayFire.jl es un conjunto de enlaces de Julia a la biblioteca de ArrayFire, está diseñado para imitar la librería estándar de Julia en su versatilidad y facilidad de uso, proporcionando una interfaz fácil pero de potente formación que apunta hacia ubicaciones en la memoria de la GPU.

La capacidad de envío múltiple y programación genérica de Julia, hace posible que los usuarios escriban código matemático natural y aprovechen de manera transparente las GPU's para obtener rendimiento. Esto se hace definiendo un tipo **AFArray** como un subtipo de **AbstractArray. AFArray** que ahora actúa como una interfaz hacía una matriz en la memoria del dispositivo. Un conjunto de funciones se importan desde Base Julia y se distribuyen a través del nuevo tipo AFArray. Así los usuarios pueden ser capaces de escribir códigos en Julia que se ejecutan en la CPU, pudiendo ejecutarlos en la GPU con muy pocos cambios de código. Adicionalmente a las funciones que imitan la librería estándar de Julia, **ArrayFire.jl** proporciona poderosas funciones en procesamiento de imágenes y la visión por computador, entre otros.

2.3. Bases conceptuales

2.3.1. Programación de la GPU en OpenCL

Una función escrita para ejecutarse usando un dispositivo como un procesador gráfico (GPU), es comúnmente denominada como kernel, **OpenCL** brinda una interfaz fácil de usar para escribir estos kernels, un kernel es lanzado para una cierta cantidad de grupos de trabajo (work groups), cada una con una cantidad determinada de hilos (threads), estos hilos son los encargados de ejecutar una simple instrucción simultáneamente, la cantidad de grupos de trabajo y número de hilos por cada uno de estos se especifican antes de lanzar el kernel.

Existe un límite en el número de grupos de trabajo y de hilos según el dispositivo que se esté usando, estos están directamente relacionados con el número de multiprocesadores del dispositivo, es decir, mientras más multiprocesadores tenga un dispositivo, más instrucciones en paralelo podrá ejecutar y por lo tanto el tiempo de ejecución global será menor.

En la Figura 6, se muestra que tan rápido puede ejecutarse una instrucción con una GPU de cuatro multiprocesadores en comparación a una con solo dos multiprocesadores, el término escalabilidad automática, se refiere a que solo se especifica el número de grupos de trabajo (cada grupo de trabajo es un block en la figura) y estos son repartidos automáticamente de acuerdo al número de multiprocesadores disponibles.

Un kernel es definido mediante el uso del especificador *__kernel*, *c*ada hilo que ejecuta el kernel, tiene un único ID global y un único ID dentro del grupo al que pertenece, que son obtenidos usando las funciones *get_global_id* y *get_local_id* respectivamente. Del mismo modo cada grupo de trabajo tiene un único ID que es obtenido usando la función *get_group_id.*

Para especificar que un argumento de la función es un vector en la memoria del dispositivo, debe anteponerse el prefijo *__global* o *__local* antes del tipo dentro de la definición de la función, ya sea que esté en la memoria global o local

respectivamente, en la Figura 7 se muestra un ejemplo que suma dos vectores y guarda el resultado en un tercer vector dentro de la memoria del dispositivo

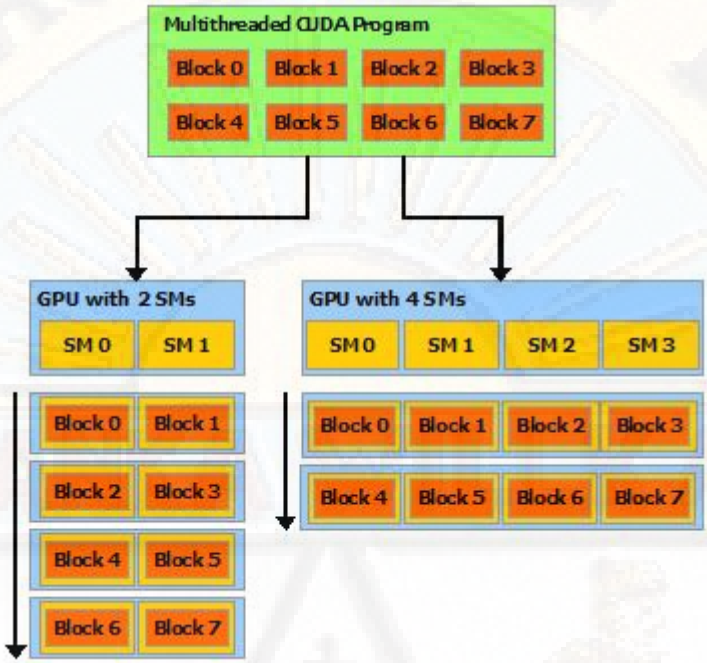


Figura 6. Escalabilidad automática, extraída de (NVIDIA Corporation)

```
1  __kernel void sum_kernel(__global float * gC, __global float
2  * gA, __global float * gB)
3  {
4          int id = get_global_id(0);
5          gC[id] = gA[id] + gB[id];
6  }
```

Figura 7. Ejemplo de un Kernel que suma dos vectores

2.3.1.1.Ejecutando un kernel usando la biblioteca de ArrayFire

ArrayFire nos permite ejecutar un kernel con la memoria directamente en la GPU, sin necesidad de trasferir datos entre la memoria central y el dispositivo y viceversa, el manual de **ArrayFire** (ArrayFire company, s.f.), en su sección *Interoperability with OpenCL* (interoperabilidad con OpenCL) nos muestra como:

Por defecto, ArrayFire administra su propio contexto, cola, memoria y crea ID's personalizados para los dispositivos, por tanto se deben seguir algunos pasos para poder integrar los kernels de OpenCL personalizados.

Si los kernels pueden operar en la misma cola que ArrayFire, deben seguirse los siguientes pasos:

1. Agregar un *include* para *af/opencl.h* en el proyecto.
2. Obtener el contexto, el dispositivo y la cola de OpenCL usado por ArrayFire.
3. Obtener las referencias *cl_mem* a los objetos *af::array.*
4. Cargar, construir y usar los kernels
5. Retornar el control de la memoria *af::array* a ArrayFire.

Notar que ArrayFire usa una cola ordenada, por tanto cuando ArrayFire y los kernels operan en la misma cola, no es necesario realizar ningún tipo de sincronización.

En la Figura 8, se muestra un ejemplo completo, donde se muestra todo éste proceso, notar que el kernel es escrito directamente en el cuerpo de la función *main*, esto no será conveniente cuando el cuerpo del kernel sea demasiado grande, por este motivo es común escribir los kernels en un archivo de extensión cl, para luego ser cargados en un objeto del tipo *char**.

2.3.1.2.Escribir funciones usando kernels personalizados para usarse en Julia usando el paquete ArrayFire.jl

ArrayFire tiene una biblioteca compartida con funciones que pueden llamarse desde otro lenguaje de programación como Julia, el paquete de Julia **ArrayFire.jl** cumple este cometido y nos permite usar las funciones de ArrayFire en Julia, de esta forma es también posible escribir funciones personalizadas que usan kernels OpenCL personalizados para usarlos en Julia.

```
#include <arrayfire.h>
// 1. Agregando el include para af/opencl.h en el proyecto
#include <af/opencl.h>
int main() {
        size_t length = 10;
        // creando vectores ArrayFire:
        af::array A = af::randu(length, f32);
        af::array B = af::constant(0, length, f32);
        // ... operaciones ArrayFire adicionales aquí
        // 2.  Obtener el contexto, el dispositivo y la cola de OpenCL
        // usado por ArrayFire
        static cl_context af_context = afcl::getContext();
        static cl_device_id af_device_id = afcl::getDeviceId();
        static cl_command_queue af_queue = afcl::getQueue();
        // 3. Obteniendo las referencias cl_mem a los objetos af::array
        cl_mem * d_A = A.device<cl_mem>();
        cl_mem * d_B = B.device<cl_mem>();
        // 4. Cargando, construyendo y usando los kernels.
        //por legibilidad se ha omitido la comprobación de errores
        int status = CL_SUCCESS;
        // Un kernel de copia simple, usando la sintaxis de C++11 para
        // cadenas en múltiples líneas
        const char * kernel_name = "copy_kernel";
        const char * source = R"(
         void __kernel
         copy_kernel(__global float * gA, __global float * gB)
         {
             int id = get_global_id(0);
             gB[id] = gA[id];
         }
    )";
        // Creando el programa, construyendo el ejecutable y extrayendo
        //el punto de entrada para el kernel.
        cl_program program = clCreateProgramWithSource(af_context,
                1, &source, NULL, &status);
        status = clBuildProgram(program, 1, &af_device_id, NULL,
                NULL, NULL);
        cl_kernel kernel = clCreateKernel(program, kernel_name, &status);
        // Estableciendo los argumentos y ejecutando el kernel
        clSetKernelArg(kernel, 0, sizeof(cl_mem), d_A);
        clSetKernelArg(kernel, 1, sizeof(cl_mem), d_B);
        clEnqueueNDRangeKernel(af_queue, kernel, 1, NULL, &length, NULL,
                0, NULL, NULL);
        // 5. Retornando el control de la memoria af::array a ArrayFire
        A.unlock();
        B.unlock();
        // ... continuar con las operaciones ArrayFire
        // Ya que los punteros del dispositivo, d_x y d_y, fueron
        // devueltas al control de ArrayFire mediante la función unlock,
        // no se necesita liberarlos usando clReleaseMemObject()
        return 0;
}
```

Figura 8. Ejemplo de la ejecución de un kernel usando la biblioteca de ArrayFire (ArrayFire company, s.f.)

En Julia una matriz ArrayFire es de un tipo *AFArray*, un objeto de este tipo tiene un campo *.arr* cuyo equivalente en ArrayFire es un objeto *af_array*, que es un puntero a una ubicación en la memoria del dispositivo.

Para llamar una función ArrayFire desde Julia deben conocerse los equivalentes de los tipos usados en la declaración de la función en ArrayFire con los tipos en Julia, estos tipos son principalmente dos:

Tipo en ArrayFire	Tipo en Julia (ArrayFire.jl)
af_array	af_array
af_array*	Ptr{af_array}

Tabla 1. Correspondencia de tipo entre objetos en ArrayFire y en Julia

Otro tipo de correspondencias son los existentes entre C y Julia, los cuales pueden verse en la sección *Calling C and Fortran Code* de la documentación de Julia (JuliaLang, s.f.).

La llamada se hace de la misma forma en la que se llaman funciones C, por ejemplo, la función de ArrayFire que suma dos vectores *af_array* cuya declaración se muestra en la Figura 9, es llamada por ArrayFire.jl en Julia usando la función mostrada en la Figura 10.

```
1  AFAPI af_err af_add   (af_array *out, const af_array lhs, const
2        af_array rhs, const bool batch);
```

Figura 9. Declaración de la función *af_add* de ArrayFire

Por lo anterior dicho, si queremos usar una función ArrayFire en Julia, es necesario escribir las funciones con argumentos del tipo *af_array* o *af_array**, combinándolos con cualquier otro tipo C que tenga su equivalente en Julia.

```
function add(lhs::AFArray{T1,N1},rhs::AFArray{T2,N2},
         batch::Bool) where {T1,N1,T2,N2}
    out = RefValue{af_array}(0)
    ccall((:af_add,af_lib),af_err,(Ptr{af_array},af_array,
                  af_array,Bool),out,lhs.arr,rhs.arr,batch)
    AFArray{typed{T1,T2},batched{N1,N2}}(out[])
end
```

Figura 10. Función *add* de ArrayFire.jl[7]

El objeto *af_array* de ArrayFire es similar al objeto *array*, puede obtenerse el objeto *af_array* a partir de un objeto *array* por medio de la función *get*, es decir, si A es una matriz del tipo *array*, obtendremos el tipo *af_array* con *A.get()*.

[7] *https://github.com/JuliaGPU/ArrayFire.jl/blob/master/src/wrap.jl*

La Figura 11, muestra un ejemplo, usando el kernel de suma de vectores, de una función con argumentos del tipo af_array, viable para ser llamada desde Julia, esta función es llamada en Julia mediante la función de la Figura 12.

2.3.2. Formato de almacenamiento CSC

El formato de compresión CSC es el acrónimo de *Compressed Sparse Column* (columna dispersa comprimida), también denominada como CCS, acrónimo de *Compressed Column Storage* (Almacenamiento de columna comprimida), es un formato de compresión por columnas de una matriz dispersa, en donde solo los elementos distintitos de cero de la matriz son almacenados en memoria como un vector, adicionalmente, para poder identificar la posición de estos elementos en la matriz original, se almacenan otros dos vectores que contienen la extensión de las columnas y los índices de fila de cada elemento. Este formato es análogo al formato CSR (Compressed Sparse Row) o CRS (Compressed Row Storage) que almacena la matriz por filas.

Sea la matriz:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 5 & 5 & 0 & 0 & 0 \\ 3 & 0 & 3 & 3 & 0 & 0 \\ 0 & 3 & 0 & 9 & 1 & 1 \\ 0 & 0 & 9 & 6 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

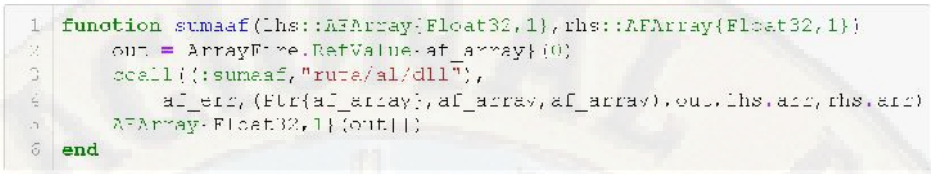
Esta matriz almacenada en formato CSC será representada por 3 vectores:

- Vector A de elementos diferentes de cero, se escriben solo los elementos distintos de cero columna a columna, es decir:

$$A = [1 \quad 2 \quad 3 \quad 5 \quad 3 \quad 5 \quad 3 \quad 9 \quad 3 \quad 9 \quad 6 \quad 1 \quad 5 \quad 1 \quad 8]$$

```
void sumaaf(af_array* out , af_array dA, af_array dB) {

        //para guardar el resultado
        af_array dC;
        af_copy_array(&dC, dA);

        // 2. Obteniendo el dispositivo, el contexto y la cola usada por ArrayFire
        static cl_context af_context = afcl::getContext();
        static cl_device_id af_device_id = afcl::getDeviceId();
        static cl_command_queue af_queue = afcl::getQueue();

        dim_t _order[AF_MAX_DIMS];
        af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], dA);
        size_t order = _order[0];

        int status = CL_SUCCESS;

        // 3. Obteniendo las referencias cl_mem a los objetos af_array
        cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
                CL_MEM_READ_ONLY, sizeof(float) * order,
                NULL, &status);
        af_get_device_ptr((void**)d_A, dA);

        cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
                CL_MEM_READ_ONLY, sizeof(float) * order,
                NULL, &status);
        af_get_device_ptr((void**)d_B, dB);

        cl_mem *d_C = (cl_mem*)clCreateBuffer(af_context,
                CL_MEM_WRITE_ONLY, sizeof(float) * order,
                NULL, &status);
        af_get_device_ptr((void**)d_C, dC);

        // 4. Cargando, construyendo y usando el kernel
        //    por legibilidad omitiremos la comprobación de errores
        // Escribiendo el kernel.
        const char * kernel_name = "sum_kernel";
        const char * source = R"(
        void __kernel
        sum_kernel(__global float * gC, __global float * gA, __global float * gB)
        {
            int id = get_global_id(0);
            gC[id] = gA[id]+gB[id];
        }
    )";
        // Creando el programa, construyendo el ejecutable y extrayendo el punto de entrada
        // para el kernel.
        cl_program program = clCreateProgramWithSource(af_context, 1, &source,
                NULL, &status);
        status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
        cl_kernel sumkernel = clCreateKernel(program, kernel_name, &status);
        // estableciendo los argumentos y ejecutando el kernel
        clSetKernelArg(sumkernel, 0, sizeof(cl_mem), d_C);
        clSetKernelArg(sumkernel, 1, sizeof(cl_mem), d_A);
        clSetKernelArg(sumkernel, 2, sizeof(cl_mem), d_B);
        clEnqueueNDRangeKernel(af_queue, sumkernel, 1, NULL, &order, NULL, 0,
                NULL, NULL);

        // 5. Retornando el control de la memoria af_array a ArrayFire
        af_unlock_array(dA);
        af_unlock_array(dB);
        af_unlock_array(dC);

        //copiando el resultado al objeto af_array de salida
        af_copy_array(out, dC);
}
```

Figura 11. Función con argumentos *af_array* que pueden ser llamados desde Julia



```
function sumaaf(lhs::AFArray{Float32,1},rhs::AFArray{Float32,1})
    out = ArrayFire.RefValue{af_array}(0)
    ccall((:sumaaf,"ruta/al/dll"),
        af_err,(Ptr{af_array},af_array,af_array),out,lhs.arr,rhs.arr)
    AFArray{Float32,1}(out[])
end
```

Figura 12. Código que llama en Julia a la función de la Figura 11

- Vector colA donde cada elemento indicará la ubicación en A del primer elemento distinto de cero en cada columna de M, es decir:

$$colA = [1 \quad 4 \quad 6 \quad 9 \quad 12 \quad 14 \quad 16]$$

- Vector rowA donde se guardará los índices de fila de todos los elementos, es decir:

$$rowA = [1 \quad 2 \quad 3 \quad 2 \quad 4 \quad 2 \quad 3 \quad 5 \quad 3 \quad 4 \quad 5 \quad 4 \quad 5 \quad 4 \quad 6]$$

Notar que el primer elemento de colA siempre será 1, y el último elemento representa la ubicación del primer elemento distinto de cero en A de la columna 7 de M (si existiera), este valor que a primera impresión no sirve, guarda la cantidad de elementos distintos de cero de M más uno, en algunos lenguajes de programación este valor necesita ser almacenado explícitamente. Si la indexación se basara en cero, este último elemento representaría exactamente la cantidad de elementos distintos de cero de M, puede obviarse este valor si n es necesario.

2.3.2.1.Formato de compresión CSC para matrices simétricas

Para matrices simétricas solo es necesario almacenar la parte triangular superior o inferior de la matriz, para este fin, puede usarse el formato de almacenamiento CSC asumiendo que una de las partes triangulares de la matriz tiene todos los elementos iguales a cero.

Si asumimos que los elementos de la diagonal de la matriz son todos diferentes de cero, puede obviarse el almacenamiento de sus índices en rowA por ser obvias.

Sea la matriz simétrica:

$$M = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 2 & 5 & 0 & 3 & 0 & 0 \\ 3 & 0 & 3 & 0 & 9 & 0 \\ 0 & 3 & 0 & 9 & 6 & 0 \\ 0 & 0 & 9 & 6 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Esta matriz almacenada en formato CSC será representada por 3 vectores, considerar que solo se guardará la parte triangular inferior, es decir, la parte triangular superior exceptuando la diagonal se asume que tiene todos los elementos iguales a cero:

- Vector A de elementos diferentes de cero, se escriben solo los elementos distintos de cero columna a columna de la parte triangular inferior, es decir:

$$A = [1 \quad 2 \quad 3 \quad 5 \quad 3 \quad 3 \quad 9 \quad 9 \quad 6 \quad 5 \quad 8]$$

- Vector colA donde cada elemento indicará la ubicación en A del primer elemento distinto de cero en cada columna de M correspondiente a la parte triangular inferior, es decir:

$$colA = [1 \quad 4 \quad 6 \quad 8 \quad 10 \quad 11]$$

- Vector rowA donde se guardará los índices de fila de todos los elementos en la parte triangular inferior, exceptuando los índices de fila de los elementos que coincidan con la diagonal, es decir:

$$rowA = [2 \quad 3 \quad 4 \quad 5 \quad 5]$$

2.3.3. Formato de almacenamiento SKS

El formato de almacenamiento SKS, acrónimo de *Skyline Storage* (Almacenamiento horizontal), es un formato de compresión, ideal para matrices dispersas en donde los elementos distintos de cero estén concentrados cerca de la

diagonal, en una matriz simétrica solo los elementos de la parte triangular superior o inferior son almacenados. Una forma de utilizar este formato sobre matrices simétricas es análogo al formato CSC visto anteriormente, sin embargo en este caso no será necesario un tercer vector de índices de fila ya que estos pueden obtenerse usando el segundo vector.

Sea la matriz M:

$$M = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 2 & 5 & 0 & 3 & 0 & 0 \\ 3 & 0 & 3 & 0 & 9 & 0 \\ 0 & 3 & 0 & 9 & 6 & 0 \\ 0 & 0 & 9 & 6 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Esta matriz almacenada en formato SKS será representada por 2 vectores, considerar que solo se guardará la parte triangular inferior columna a columna y que los elementos de la diagonal siempre serán guardados aunque alguno de ellos sean ceros:

- Vector A de elementos diferentes de cero, se busca en la parte triangular inferior, el último elemento distinto de cero en cada columna y se escriben los elementos desde la diagonal hasta este, esto significa que si en este tramo existen elementos iguales a cero, también deben escribirse, es decir:

  $$A = [1 \quad 2 \quad 3 \quad 5 \quad 0 \quad 3 \quad 3 \quad 0 \quad 9 \quad 9 \quad 6 \quad 5 \quad 8]$$

  Adicionalmente para cada columna, el último elemento distinto de cero debe tener índice de fila mayor o igual al índice de fila del elemento correspondiente a la columna anterior, esto para evitar realizar factorizaciones simbólicas en el caso de realizar una factorización de Cholesky o LDL^T, si esto no se cumple, debe almacenarse algunos ceros de esta columna para cumplir con lo mencionado.

- Vector idxA donde cada elemento indicará la ubicación en A del primer elemento distinto de cero en cada columna de M (es decir el elemento diagonal) correspondiente a la parte triangular inferior, es decir:

$$idxA = [1 \quad 4 \quad 7 \quad 10 \quad 12 \quad 13]$$

Hay otras formas de almacenar en este formato: está el análogo almacenamiento por filas, en donde podría guardarse solo la parte triangular superior; o en donde se almacena los elementos de la diagonal en un vector diferente como puede verse en (stuff, s.f.) y en (Watkins, 2002).

Como se dijo anteriormente este formato es ideal para cuando los elementos distintos de cero de la matriz se concentran cerca de la diagonal, de no ser este el caso, este formato podría no ser adecuado, ya que como se vio, se almacenan algunos ceros extras; o puede usarse algún método para reenumerar filas y columnas (sin perder la simetría), con tal de lograr que los elementos distintos de cero se concentren más cerca de la diagonal.

2.4. Definición de términos

ArrayFire: Biblioteca de software de alto rendimiento para computación paralela con una API fácil de usar. Su conjunto de funciones basado en matrices hace que la programación paralela sea más accesible.

ArrayFire.jl: Paquete de Julia que adecua la biblioteca de ArrayFire para usarse en este lenguaje.

Cola: traducción de *queue* en OpenCL, directamente relacionada con la cola de comandos (*command-queue*), que es una estructura de datos para coordinar la ejecución de los kernels en los dispositivos.

Contexto: traducción de *context* en OpenCL, se refiere a una definición que hace el procesador central para la ejecución de kernels.

Dispositivo: Se refiere a cualquier dispositivo de cómputo que pueda programarse con OpenCL, principalmente procesadores gráficos.

En sitio: Se refiere a que las operaciones de escritura se realizan sobre el mismo objeto de lectura, perdiéndose los datos anteriores.

Factorización de Cholesky: Método de descomposición de una matriz cuadrada simétrica y definida positiva en dos submatrices triangulares tal que una es la traspuesta de la otra.

Factorización LDL^{T}: Método de descomposición de una matriz cuadrada simétrica **A** en tres factores: dos matrices triangulares **L** y **L^{T}** tal que una es la transpuesta de la otra y una matriz diagonal **D** tal que $A = LDL^T$

Factorización simbólica: proceso simbólico de factorización para obtener la cantidad de memoria máxima necesaria al realizar la factorización de Cholesky o LDL^{T}.

Función atómica: función en OpenCL usada para que un hilo pueda escribir en la misma ubicación en memoria que otros hilos ejecutando una instrucción en paralelo.

Función de dispositivo: Función secundaria escrita en OpenCL para ejecutar instrucciones en paralelo usando el procesador gráfico y que es llamada desde un kernel principal.

GPU: Acrónimo de Graphics Processing Unit (en español unidad de procesamiento gráfico)

Grupo de trabajo: Traducción de *work group*, denominación de un bloque de hilos en OpenCL

Hilos: Traducción de *threads* en OpenCL, forma parte de un grupo de trabajo (*work group*), se refiere básicamente a un hilo de ejecución, es decir ejecutan una instrucción en paralelo a la par con otros hilos dentro del grupo.

Julia: Lenguaje de programación de alto nivel.

Kernel: Función escrita en OpenCL para ejecutar instrucciones en paralelo usando el procesador gráfico.

Memoria global: Traducción de *global memory* en OpenCL, se refiere a la memoria que reside en la memoria del dispositivo, que permite leer y escribir a todos los hilos de todos los grupos.

Memoria local: Traducción de *local memory* en OpenCL, se refiere a la memoria que reside en la memoria del dispositivo, pero asignado solo a un grupo de trabajo.

Método de Gauss-Jordan: Llamada también eliminación de Gauss-Jordan, es un método de solución de sistemas de ecuaciones lineales, que consiste en convertir la matriz por medio de operaciones entre sus filas en una matriz diagonal.

Método de los gradientes conjugados: Método iterativo de solución de sistemas de ecuaciones lineales que consiste en buscar el mínimo de la función $f: R^n \rightarrow R$: $f(x) = \frac{1}{2}x^T Ax - b^T x$

OpenCL: Acrónimo de Open Computing Language (en español lenguaje de computación abierto), interfaz de programación del procesador gráfico que se usa con C++.

OpenFEM: Conjunto de herramientas de elementos finitos de distribución libre.

SEL: Acrónimo de sistema de ecuaciones lineales

Sistema de ecuaciones lineales: conjunto de expresiones (ecuaciones) de primer grado que deben ser resueltas para un determinado número de variables.

Sistema triangular: Un sistema de ecuaciones lineales cuya matriz de coeficientes es triangular superior o inferior.

2.5. Variables

2.5.1. Variable independiente

Técnicas numéricas implementadas sobre procesador gráfico (GPU): las técnicas numéricas consideradas en el presente estudio son implementadas sobre procesador gráfico, entiéndase técnica como la manera de usar un método en particular, en nuestro caso, los métodos serían: el método de Gauss Jordan, la factorización de Cholesky, la factorización LDL^T y el método de los gradientes conjugados; y la forma sería la aplicación de estos métodos sobre matrices densas, en formato CSC y en formato SKS.

2.5.2. Variable dependiente

Solución de sistemas de ecuaciones lineales en análisis estructural: las técnicas numéricas (variable independiente) se utilizan para resolver el sistema de ecuaciones, la solución está directamente relacionada con la memoria utilizada y el tiempo de procesamiento que toman las técnicas para resolver el sistema, son estas variables las que nos interesa medir.

2.6. Operacionalización de variables

La Operacionalización de variables se describe en la siguiente tabla:

Tabla 2. Operacionalización de variables

Variables	Dimensiones	Metodología
V. Independiente • Técnicas numéricas sobre procesador gráfico.	Método (formato de almacenamiento) • Gauss Jordan (Matriz densa) • Factorización de Cholesky (matriz densa) • Factorización de Cholesky (matriz CSC) • Factorización de Cholesky (matriz SKS) • Factorización LDLT (Matriz densa). • Factorización LDLT (Matriz CSC). • Factorización LDLT (matriz SKS) • Gradientes conjugados (Matriz densa). • Gradientes conjugados (matriz CSC). • Gradientes conjugados (matriz SKS).	• Revisión bibliográfica • Escritura de funciones en Julia y ArrayFire. • Escritura de kernels en OpenCL. • Ejecución de funciones y/o kernels
V. Dependiente • Solución de sistemas de ecuaciones lineales en análisis estructural	• Tiempo de ejecución (s) • Memoria utilizada (MB)	**Obtención del sistema de ecuaciones:** OpenFEM, herramienta de elementos finitos **Memoria:** medición manual de acuerdo al número de elementos de la matriz y la precisión de los elementos. **Tiempo:** Macro @time de Julia Macro @elapsed de Julia

CAPÍTULO III

METODOLOGÍA DE INVESTIGACIÓN

3.1. Tipo de investigación

El tipo de investigación es **aplicada**, se busca aplicar conocimientos o procedimientos ya establecidos en la solución de problemas prácticos (Niño Rojas, 2011).

3.2. Nivel de investigación

El nivel de investigación es: **descriptivo,** se busca especificar propiedades, características y rasgos importantes del fenómeno que se analiza.
(Hernández Sampieri, Fernández Collado, & Baptista Lucio, 2010), cada una de las técnicas consideradas para resolver el sistema tendrá diferentes comportamientos computacionalmente hablando, las características principales son tiempo de ejecución y memoria utilizada.

El método que se usará en la realización del proyecto de investigación: **método deductivo**, de lo general a lo particular, lo general sería los conocimientos que se tienen sobre el método de solución de sistemas de ecuaciones lineales y lo particular sería la aplicación del método a problemas de análisis estructural.

3.3. Población, muestra y muestreo

3.1.1 Población:

- Modelos estructurales

3.1.2 Muestra

- Viga en voladizo
- Pilar de puente

3.4. Técnicas e instrumentos de obtención de datos

Cada sistema de ecuaciones lineales, fue generado al resolver problemas típicos de análisis estructural, se ha usado el **OpenFEM**, una herramienta de elementos finitos de libre distribución, al analizar una estructura por elementos finitos tenemos la ventaja de generar una matriz de rigidez del orden que queramos, basta con hacer los elementos cada vez más pequeños.

3.5. Procedimiento de obtención de datos

Este procedimiento comprenderá:

- ❖ Revisión de documentos: revisar la información existente y de artículos que involucren temas relacionados con la investigación, es decir en nuestro caso documentación, artículos, manuales o guías de programación de los lenguajes a usar.
- ❖ Analizar cada estructura usando el método adecuado.
- ❖ Cuando se use el método de elementos finitos, primero se decidirá en cuantos sub-elementos se dividirá cierto elemento estructural considerando una cantidad de nodos aceptable que nos genere una cantidad de incógnitas relativamente grande.

3.6. Técnicas de procesamiento y análisis de datos

Las técnicas para el procesamiento y análisis de los datos serán:

- ❖ **Cuantitativas:** Los resultados (tiempo y memoria utilizada) son cuantificables.

La programación del procesador gráfico se ha realizado usando la interfaz de **OpenCL** en C++, para ejecutar los kernels escritos se ha usado la librería de **ArrayFire**, escribiendo funciones que serán incluidas en otra biblioteca compartida con la finalidad de llamarlas desde Julia.

Por comodidad estas funciones serán puestas a prueba en **Julia**, usando el paquete **ArrayFire.jl** que hará posible usar las funciones con la memoria directamente en el dispositivo.

Todo el software mencionado anteriormente es de libre distribución y se ha usado la **versión comunity** de **Visual Studio**.

La medición del tiempo de procesamiento se obtendrán usando las macros **@time y/o @elapsed** de **Julia,** la medición de la memoria será directa, usando el orden de la matriz en el caso de una matriz densa, y las longitudes de los vectores que representan la matriz en el caso de una matriz dispersa, estos serán multiplicados por la memoria en (MB) MegaBytes que utiliza cada elemento en las componentes de la matriz.

- **Cualitativas:** se empleará para la interpretación de resultados.

CAPÍTULO IV

PRESENTACIÓN DE RESULTADOS

4.1. Análisis de información

4.1.1. Etapas

La investigación se compuso principalmente de siete etapas, partiendo de la escritura de los kernels y finalizando en la prueba de estos para resolver los sistemas de ecuaciones lineales midiendo el tiempo y la memoria utilizada, estas etapas se resumen a continuación:

Tabla 3. Primera etapa de la investigación

ETAPA I **Escritura de los kernels**	
Instrumentos	• API de OpenCL en C++ • Visual Studio comunity
Metodología	• Uso de manuales y guías de programación en C++ y OpenCL
Resultado	Kernels para resolver el sistema de ecuaciones lineales, con las siguientes combinaciones: método-formato de almacenamiento: • Gauss Jordan-denso • Cholesky-denso • Cholesky-CSC • Cholesky-SKS • LDL^T-denso • LDL^T -CSC • LDL^T –SKS • Gradientes conjugados-denso • Gradientes conjugados -CSC • Gradientes conjugados -SKS

Tabla 4. Segunda etapa de la investigación

ETAPA II **Escritura de funciones que utilizan los kernels y que se exportan en una biblioteca dinámica (dll)**	
Instrumentos	• Biblioteca de ArrayFire • Visual Studio comunity
Metodología	• Uso de manuales y guías de programación en ArrayFire y C++.
Resultado	• Archivo de extensión dll que contiene las funciones que utilizan los kernels para ser llamados desde otro lenguaje de programación.

Tabla 5. Tercera etapa de la investigación

ETAPA III **Escritura de funciones en Julia que llaman a las funciones exportadas**	
Instrumentos	• Julia • Paquete ArrayFire.jl de Julia
Metodología	• Uso de manuales y guías de programación en Julia.
Resultado	• Funciones *Julia* listas para recibir como argumentos la matriz de rigidez (en uno de los formatos de almacenamiento) y el vector de fuerzas, y devolver el resultado de resolver éste sistema de ecuaciones lineales. • Adicionalmente se escribieron funciones *Julia* que devuelven el tiempo de ejecución de resolver el sistema de ecuaciones lineales.

Tabla 6. Cuarta etapa de la investigación

ETAPA IV **Modelamiento de la estructura y obtención del sistema de ecuaciones lineales**	
Instrumentos	• OpenFEM • MatLab
Metodología	• Uso de manuales y guías de programación en MatLab y OpenFEM. • Afinamiento de la malla de elementos finitos para obtener diferentes órdenes de la matriz.
Resultado	• Cinco archivos de texto por cada orden de la matriz de rigidez de extensiones .nz, .row, .col, .fzs y .def; los tres primeros representan la matriz en formato COO, el cuarto es el vector de fuerzas y el último es el vector de deformaciones o vector resultado arrojado por el solver del OpenFEM que se exporta exclusivamente con fines de validación.

Tabla 7. Quinta etapa de la investigación

ETAPA V **Importación del sistema de ecuaciones lineales en Julia y construcción de la matriz con el formato de almacenamiento correspondiente**	
Instrumentos	• Julia • Paquete ArrayFire.jl de Julia
Metodología	• Uso de manuales y guías de programación de Julia
Resultado	• Matrices en formato denso, CSC o SKS, y vector de fuerzas, listas para ser resueltas usando las funciones escritas en la etapa 3.

Tabla 8. Sexta etapa de la investigación

ETAPA VI Resultados	
Instrumentos	• Julia • Paquete IJulia de Julia
Metodología	• Uso de manuales y guías de programación de Julia • Uso de las funciones para medir el tiempo de ejecución de la etapa 3. • Medición manual de la memoria utilizada en función al tamaño de los elementos.
Resultado	• Tiempo de ejecución en segundos que toma resolver el sistema de ecuaciones lineales. • Memoria utilizada en MB.

Tabla 9. Séptima etapa de la investigación

ETAPA VII Conclusiones	
Instrumentos	• Excel
Metodología	• Elaboración de gráficos y tablas con los resultados de la etapa 6. • Comparación con los resultados arrojados en tiempo de ejecución y memoria por las mismas funciones resolviendo el mismo sistema de ecuaciones pero ejecutadas usando otro dispositivo (GPU). • Comparación con los resultados arrojados en tiempo de ejecución y memoria por el solver del OpenFEM.
Resultado	• Entre las técnicas utilizadas encontrar la que tiene un mejor balance en tiempo de ejecución y memoria utilizada. • Resultado numérico de las diferencias que se obtienen al usar las mismas funciones, resolviendo el mismo sistema de ecuaciones, en diferentes dispositivos (GPU's). • Determinar la efectividad de estas técnicas en comparación al resultado arrojado por el solver de OpenFEM.

4.1.2. Compatibilidad en la indexación

La indexación en C++ y por lo tanto en OpenCL, está basada en cero, es decir el primer elemento en un vector tendrá índice cero. En Julia, la indexación está basada en uno, esta diferencia fue manejada mediante la disminución previa de los índices de los vectores en Julia antes de usarse en funciones C++, esto es solo necesario cuando se han almacenado índices o indicadores de posición en algún tipo de vector que será usado como argumento, por ejemplo, los vectores de índices creados al almacenar matrices dispersas en formato CSC o SKS, cada elemento de estos vectores se han reducido en uno antes de establecerlos como argumentos en alguna de las funciones compartidas de C++ que usan los kernels OpenCL.

4.1.3. Orden de almacenamiento preferencial

Los elementos de una matriz no son almacenadas de forma bidimensional como podría concebirse a priori, si no como un vector, en ArrayFire los elementos de una matriz (objetos del tipo *array* o *af_array*) se almacenan en un vector columna por columna, es decir, una matriz de *mxn* es almacenada en memoria como un vector de longitud *mxn.*

Por ejemplo sea la matriz de 6 x 6 siguiente:

$$\begin{bmatrix} 1 & 1 & 3 & 5 & 8 & 4 \\ 9 & 3 & 8 & 4 & 3 & 5 \\ 2 & 9 & 7 & 1 & 8 & 9 \\ 5 & 4 & 6 & 3 & 4 & 2 \\ 6 & 8 & 9 & 7 & 4 & 5 \\ 9 & 7 & 3 & 4 & 6 & 9 \end{bmatrix}$$

Se almacena en memoria como:

$$\begin{bmatrix} 1 \\ 9 \\ 2 \\ 5 \\ 6 \\ 9 \\ 1 \\ 3 \\ \vdots \\ 6 \\ 4 \\ 5 \\ 9 \\ 2 \\ 5 \\ 9 \end{bmatrix}$$

En memoria solo existe el vector y no la matriz en sí, acceder a los elementos de este vector es simple y solo requiere un id unidimensional, estos id en este caso son valores entre 0 y 35 (recordar que la indexación es basada en cero) que cubrirá los 36 elementos de esta matriz, ahora dado un id bidimensional (row, col), podemos acceder al elemento correspondiente en el vector partiendo de estos subíndices usando la siguiente expresión:

$$x = row + h_A col$$

h_A es la altura de la matriz, *row* y *col* son los índices de fila y columna respectivamente de un elemento.

Un método que usamos recurrentemente para acceder a un elemento en OpenCL, es el de declarar un apuntador hacia la memoria del dispositivo moviéndonos cierta cantidad de elementos, en OpenCL este movimiento es realizado mediante la siguiente declaración:

```
__global double* Colb = A + hA * col;
```

A es el apuntador normal a la memoria del dispositivo con los elementos de la matriz, *hA* es la altura de la matriz y *col* es el índice de columna, como resultado *Colb* apuntará a la memoria del dispositivo con un movimiento de $h_A col$ elementos hacia adelante, es decir, el elemento con id 0 de *Colb* equivaldrá al

elemento $h_A col$ de *A.* así para acceder al elemento *(row, col)* de *A* bastaría con acceder al elemento *row* de $Colb$ es decir:

$$A_{row,col} = Colb_{row}$$

4.1.4. Alistando los kernels

Los kernels fueron escritos en archivos de extensión cl, para cargarlos se ha creado una variable de entorno CL_FILES con valor igual a la ruta a la carpeta que contiene los archivos cl, creamos una función que obtiene el valor de una variable de entorno dado su nombre y creamos otra función con dos argumentos del tipo *char** que contienen el nombre de un archivo de encabezado y el nombre del archivo cl, esta última función utiliza la función anterior para obtener el valor de la variable CL_FILES, es decir, la ruta a la carpeta con los archivos cl, luego busca los archivos con los nombres especificados por lo argumentos, une su contenido y lo guarda en un objeto del tipo *char** listo para ser cargado. El archivo de encabezado mencionado es una archivo .h ubicada en la misma carpeta, contiene definiciones y otras líneas comunes a todos los archivos cl, en nuestro caso el archivo es llamado *Common.h* cuyo contenido se muestra en la Figura 13. Los archivos cl contienen exclusivamente los kernels y las funciones de dispositivo.

```
1  //tamaño de bloque para ser usado en funciones normales y Kernels
2  #define BLOCK_SIZE 32
3
4  //activando doble precisión (double) para kernels
5  #pragma OPENCL EXTENSION cl_khr_fp64 : enable
6
7  //activando funciones atómicas para 32 y 64 bits
8  #pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable
9  #pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable
```

Figura 13. Contenido de Common.h

4.1.5. Kernels sobre matrices densas

Se presentan los kernels elaborados para trabajar sobre matrices densas. Ya que la intención es trabajar con matrices de orden relativamente grande, se trabajó con precisión doble (tipo *double*), ya que en matrices con precisión simple (tipo *float*), se encontró pérdida de precisión en la solución a medida que aumentaba el orden de la matriz, cabe mencionar que existen métodos para trabajar en simple

precisión conservando una precisión aceptable, sin embargo, la aplicación de dichos métodos quedan fuera del alcance de esta tesis.

4.1.5.1.Gauss Jordan

El kernel se muestra en la Figura 14, como puede observarse el kernel llama a la función de dispositivo *Gss_Jrd_c* que se muestra en la Figura 15, el argumento *A* de esta función, es el apuntador hacia la memoria del dispositivo que contiene los elementos de la matriz, *wA* es el ancho de la matriz y *hA* es el alto de la matriz, los demás argumentos son usados por la función de la siguiente manera:

```
1  __kernel void
2  Gauss_Jordan_c(__global double* A,
3          int hA, int row)
4  {
5          //la iteración se realizará en el host, en aquí
6          //solo se realizará una operación sobre cada columna
7          //encargada a cada hilo
8          Gss_Jrd_c(A, hA + 1, hA, row, row, 0, row + 1);
9   }
```

Figura 14. Gauss Jordan: Kernel para trabajar sobre una matriz densa

La función resta a cada fila de *A* la fila *row* multiplicada por un factor (calculada en la línea 26), tal que el elemento ubicado en la posición *col* de cada fila sea cero luego de realizada esta operación, *fcol* indica la posición en cada fila desde donde se realizarán las operaciones, si es para la fila completa el valor correspondiente de *fcol* es cero, del mismo modo debe especificarse una fila (*frow*) para obviar las filas anteriores en las operaciones, para todas las filas este valor debe ser cero.

Como puede observarse esta función es generalizada para matrices no necesariamente cuadradas, en nuestro caso el argumento *A* deberá ser la matriz aumentada, es decir, con el vector de constantes añadido a la derecha de la matriz de coeficientes original, al final debe tenerse una matriz de *n x n+1*, siendo *n* el orden de la matriz. Cada llamada al kernel realizará una de las *n* iteraciones que deben hacerse para terminar con la eliminación de Gauss Jordan, esta iteración será realizada en el *host*. El término *host* en este caso se refiere a la entorno de ArrayFire.

```
1   void Gss_Jrd_c(__global double* A, int wA, int hA,
2           int row, int col, int frow, int fcol)
3   {
4           // fcol + índice local de grupo
5           int bx = fcol + get_group_id(0);
6
7           //índice del primer elemento en una columna
8           int tx = frow + get_local_id(0);
9
10          //factor de multiplicación
11          double fm;
12
13          //columna base, la que se vuelve cero, excepto el
14          //valor en la posición frow
15          __global double* Colb = A + hA * col;
16
17
18          //en este caso, cada grupo se encargará de las operaciones
19          //sobre todos los elementos de una columna
20          double keyEl = Colb[row];
21
22          for (int x = tx; x < hA; x += get_local_size(0)) {
23
24                  if (x != row && keyEl != 0) {
25
26                          fm = Colb[x] / keyEl;
27                          for (int y = bx; y < wA; y += get_num_groups(0)) {
28
29                                  __global double* ColEl = A + y * hA;
30                                  ColEl[x] -= fm * ColEl[row];
31                          }
32                  }
33          }
34  }
```

Figura 15. Función de dispositivo llamada por le kernel de la Figura 14

Notar que el kernel solo tiene tres argumentos, los argumentos de *Gss_Jrd_c* son establecidos en función a estos. El número de iteración que se realice, será igual al índice de la columna que se hace cero en esa iteración, es decir si estamos en la quinta iteración, al finalizar esta, la quinta columna tendrá que ser cero (exceptuando obviamente el quinto elemento de esta columna).

El código de *Gss_Jrd_c* está escrito para trabajar de la siguiente manera:

Cada grupo de trabajo y sus hilos se encargará de una columna, ya que la función está diseñada para trabajar partiendo de ciertos índices dados por *frow* y *fcol*, en las líneas 5 y 8 se suma a los índices de grupo y de hilo *fcol* y *frow* respectivamente, esto para hacer posible que cada grupo empiece a trabajar a partir del índice *fcol* y cada hilo a partir del índice *frow*.

La línea 11 declara una variable del tipo double para el factor de multiplicación que se calcula en la línea 26. La línea 15 declara un apuntador *Colb* hacía la memoria del dispositivo con inicio a una distancia *hA*col* elementos del inicio

de *A*, esto nos lleva a la columna base, la que se trabaja para convertir *n-1* de sus elementos en cero. En la línea 20 se obtiene el elemento *row* de *Colb* correspondiente a la fila base, la que multiplicaremos por el factor *fm* para restarla de las otras filas, en el kernel de Figura 14, equivaldría a obtener el elemento diagonal de A.

La iteración de la línea 22 a 33 se realiza con la finalidad de que el número de hilos en un grupo de trabajo cubra todos los elementos de la columna a trabajar, esto es necesario ya que el número de hilos por grupo, establecido al llamar el kernel, no depende del tamaño de la matriz. Si el método para llamar al kernel estableciera un número de hilos igual al número de filas que debe trabajarse, esta iteración no sería necesaria. De la misma manera ocurre con la iteración interior (líneas 27 a 31), que se hace con la finalidad de que los grupos disponibles cubran todas las columnas.

Un detalle más a tener en cuenta es el valor asignado a *fcol* en el kernel, este es igual a *row+1*, lo que significa que no se está trabajando la columna *row* (la que debe volverse cero), se puede omitir esta operación ya que no necesitamos el resultado final en esta columna por que no se utilizan en ninguna otra operación, estas serán implícitamente cero.

Al terminar con la iteración en el *host*, la solución del sistema estará dada por la división de los elementos de la última columna de la matriz aumentada (que corresponde a los elementos del vector de constantes del sistema) y los elementos de la diagonal principal.

4.1.5.2.Factorización de Cholesky

Para resolver un sistema de ecuaciones usando la factorización de Cholesky, son necesarias dos etapas, uno de la factorización misma y otro que consiste en resolver los sistemas triangulares generados después de la factorización.

El proceso de factorización fue dividido en tres partes, un paso en el que modificamos una columna *j* tal que:

$$A_{n,j} = \frac{A_{n,j}}{\sqrt{A_{j,j}}}, n > j$$

La función de dispositivo que realiza esta operación se muestra en la Figura 16, el trabajo de modificar los elementos de la columna se le asigna a todos los hilos de todos los grupos, es por eso que en vez de pedir el id local de hilo se pide el id global (línea 5), la columna a trabajar es la misma determinada por el argumento *step*.

```
1  void Chol_1_c(__global double* A, int order, int step)
2  {
3
4          //step + 1 + índice global de hilo
5          int tx = get_global_id(0);
6
7          //número total de hilos
8          int ntx = get_num_groups(0)*get_local_size(0);
9
10         //columna base
11         __global double* colB = A + step * order;
12
13         //raíz cuadrada del elemento A(step, step)
14         double _r = sqrt(colB[step]);
15
16         //cada hilo se encargará de modificar un elemento
17         //de la columna
18         for (int x = tx + step + 1; x < order; x += ntx) {
19                 colB[x] = colB[x] / _r;
20         }
21 }
```

Figura 16. Cholesky: función de dispositivo *Chol_1_c*

Otro paso consiste en modificar las columnas siguientes a la columna *j* modificada en el paso anterior tal que:

$$A_{n,s} = A_{n,s} - A_{s,j}A_{n,j}, s > j, n \geq s$$

La función de dispositivo que realiza esta tarea se muestra en la Figura 17, en este caso, cada grupo y sus hilos se encargarán de modificar una columna.

Estos dos primeros pasos deben realizarse uno después de otro durante *n* iteraciones, un tercer paso es necesario para terminar la factorización, este modifica los elementos de la diagonal tal que:

$$A_{j,j} = \sqrt{A_{j,j}}$$

```
1   void Chol_2_c(__global double* A, int order, int step)
2   {
3           //índice local de grupo
4           int bx = get_group_id(0);
5
6           //step + índice local de hilo
7           int tx = get_local_id(0);
8
9           //columna base
10          __global double* colB = A + step * order;
11
12          //parámetros de apoyo
13          double _k;
14
15          //en este caso, cada grupo se encargará de las
16          //operaciones sobre todos los elementos de una
17          //columna
18          for (int y = bx + step + 1; y < order; y += get_num_groups(0)) {
19                  _k = colB[y];
20                  __global double* colI = A + y * order;
21
22                  for (int x = tx + y; x < order;
23                          x += get_local_size(0)) {
24
25                          colI[x] -= _k * colB[x];
26                  }
27          }
28  }
```

Figura 17. Cholesky: función de dispositivo Chol_2_c

La función de dispositivo que realiza esta tarea se muestra en la Figura 18, todos los hilos de todos los grupos se encargan cada uno de modificar un elemento de la diagonal, este paso solo se realiza una vez luego de realizada las *n* iteraciones con los dos pasos vistos previamente.

Terminada la factorización se procede a solucionar los sistemas equivalentes, es decir si $Ax = b$ es el sistema original, luego de la factorización quedará:

$$LL^T x = b$$

Haciendo $L^T x = y$ entonces $Ly = b$. Primero se resuelve $Ly = b$ y finalmente $L^T x = y$, cuya solución en x es la solución del sistema $Ax = b$.

Como puede observarse, la factorización se realiza en sitio, es decir los elementos de L son almacenados en la misma matriz A, luego de la factorización, A contendrá en su parte triangular inferior, los elementos de L, asumiendo implícitamente que los elementos de la parte triangular superior (sin contar la diagonal) son ceros.

```
1  void Chol_3_c(__global double* A, int order)
2  {
3
4          //índice global de hilo
5          int tx = get_global_id(0);
6
7          //número total de hilos
8          int ntx = get_num_groups(0)*get_local_size(0);
9
10         //parámetros de apoyo
11         int diag;
12
13         for (int x = tx; x < order; x += ntx) {
14                 //ubicación del elemento diagonal en memoria
15                 diag = x * order + x;
16                 A[diag] = sqrt(A[diag]);
17         }
18 }
```

Figura 18. Cholesky: función de dispositivo *Chol_3_c*

Para resolver el sistema $Ly = b$, usaremos la eliminación típica de Gauss Jordan aplicada a una matriz triangular inferior, en cada iteración los elementos de cada columna fuera de la diagonal van volviéndose cero progresivamente, hasta llegar a una matriz diagonal, en ningún momento debe hacerse uso de la parte triangular superior (son implícitamente ceros, aunque en memoria guardan la parte triangular superior de la matriz original *A*), las operaciones sobre L pueden obviarse y dejarla intacta para poder usarla al resolver el siguiente sistema $L^T x = y$, es decir solo se modificará el vector de constantes *b* tal que para la iteración *j*:

$$b_n = b_n - b_j \frac{L_{n,j}}{L_{j,j}}, n > j$$

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 19, el número de iteración y por la tanto el índice de la columna implicada está dada por el valor del argumento *step*, esta función debe ser llamada *n* veces para terminar con la eliminación, luego será necesario dividir los elementos de *b* entre los elementos de la diagonal (función de dispositivo de la Figura 20), este vector será la solución del sistema $Ly = b$, que será usado como argumento al resolver el siguiente sistema triangular.

```
void Chol_4_c(__global double* L, __global double* b, int order,
        int step)
{

        //índice global de hilo
        int tx = get_global_id(0);

        //número total de hilos
        int ntx = get_num_groups(0)*get_local_size(0);

        //columna de L implicada
        __global double* colB = L + order * step;

        //elemento diagonal
        double diag = colB[step];

        double k = b[step] / diag;

        for (int x = tx + step + 1; x < order; x += ntx) {
                b[x] -= k * colB[x];
        }
 }
```

Figura 19. Cholesky: función de dispositivo *Chol_4_c*

Para resolver el sistema $L^T x = y$, se usará la misma matriz que almacena L, el artificio es asumir que está ordenada por filas y no por columnas, hecho esto la transposición de la matriz ya no será necesaria. L^T es una matriz triangular superior, se resolverá el sistema obteniendo el valor de las variables una a continuación de otra desde abajo, el valor de la variable en la posición *i* sería:

$$x_i = \frac{y_i - \sum_{j=i+1}^{n-1} x_j L_{i,j}}{L_{i,i}}$$

```
void Chol_5_c(__global double* A, __global double* b, int order)
{

        //índice global de hilo
        int tx = get_global_id(0);

        //número total de hilos
        int ntx = get_num_groups(0)*get_local_size(0);

        //parámetros de apoyo
        int diag;

        for (int x = tx; x < order; x += ntx) {
                //ubicación del elemento diagonal en memoria
                diag = x * order + x;
                b[x] = b[x] / A[diag];
        }
 }
```

Figura 20. Cholesky: función de dispositivo *Chol_5_c*

Se usa el mismo vector *y* para almacenar la solución desde abajo. La función de dispositivo que resuelve el sistema $L^T x = y$ se muestra en la Figura 21, cada llamada a esta función calculará una variable. Ya que el sistema se resuelve

desde abajo, para un paso *j* se calculará el valor de la variable *i* tal que *i=n-j-1*, la fila de *L* usada para calcular esta variable tiene el mismo índice (se obtiene en la línea 14). Esta función de dispositivo usará memoria local para calcular la sumatoria $\sum_{j=i+1}^{n-1} x_j L_{i,j}$ por reducción en paralelo, el tamaño de esta memoria local será el mismo que el tamaño de un grupo de trabajo, por esta razón, en este caso solo se utilizará un grupo de trabajo al usar esta función (en la línea 26 se especifica que el grupo de trabajo utilizado es únicamente el con id cero).

```
1   void Chol_6_c(__global double* L, __global double* y, int order,
2           int step, __local double* partialSum)
3   {
4           //índice de bloque
5           int bx = get_group_id(0);
6
7           //índice global de hilo
8           int tx = get_local_id(0);
9
10          //número total de hilos
11          int ntx = get_local_size(0);
12
13          //índice de la fila de L implicada
14          int fL = order - step - 1;
15
16          //fila de L implicada, data
17          __global double* filB = L + fL * order;
18
19          //elemento diagonal
20          double diag = filB[fL];
21
22          //valor donde se guardará una suma parcial
23          double sum = 0.0;
24
25          //solo se usará un bloque
26          if (bx == 0) {
27                  for (int x = tx + fL + 1; x < order; x += ntx) {
28                          sum += filB[x] * y[x];
29                  }
30                  partialSum[tx] = sum;
31
32                  //reducción en paralelo pare determinar la suma de
33                  //todos los elementos de partialSum
34                  for (int stride = ntx / 2; stride > 0; stride /= 2) {
35                          barrier(CLK_LOCAL_MEM_FENCE);
36                          if (tx < stride) {
37                                  partialSum[tx] += partialSum[tx + stride];
38                          }
39                  }
40                  if (tx == 0)
41                          y[fL] = (y[fL] - partialSum[0]) / diag;
42          }
43  }
```

Figura 21. Cholesky: función de dispositivo *Chol_6_c*

En la línea 27 a 29 se calcula una suma parcial almacenándola en la variable *sum* (declarada en la línea 23), luego este valor es almacenado en memoria local (línea 30), los elementos de *partialsum* se suman en paralelo en el bloque

for de las líneas 34 a 39, finalmente se calcula la variable correspondiente en la línea 41, esta tarea solo la realiza un hilo (en la línea 40 se especifica el id del hilo que realiza la operación). Esta función de dispositivo debe ser llamada también *n* veces para calcular las *n* variables.

```
1   __kernel void
2   Cholesky_c(__global double* A, __global double* b,
3           __local double* partialSum, int order, int step, int sstep)
4   {
5           //la iteración principal se realiza en el host
6
7           //se modificará una sola columna
8           if (sstep == 0) {
9                   Chol_1_c(A, order, step);
10          }
11          //se modificará las columnas siguientes a la columna
12          //implicada
13          else if (sstep == 1) {
14                  Chol_2_c(A, order, step);
15          }
16
17          //una vez usada las funciones anteriores todavía es
18          //necesario modificar la diagonal D[i]=sqrt(D[i]) para
19          //concluir la factorización
20          else if (sstep == 2) {
21                  Chol_3_c(A, order);
22          }
23
24          ///hasta aquí la factorización está concluida, la parte
25          //triangular inferior de A contendrá la factorización de
26          //Cholesky es decir L
27          //desde aquí se procederá a solucionar el sistema
28          //L*transpose(L)x=b
29
30          //se resolverá el sistema Ly=b, y=transpose(L)*x
31          else if (sstep == 3) {
32                  Chol_4_c(A, b, order, step);
33          }
34          else if (sstep == 4) {
35                  Chol_5_c(A, b, order);
36          }
37
38          //se resolverá el sistema transpose(L)*x=y
39          //"y" ya debe haberse obtenido usando la función anterior
40          else {
41                  Chol_6_c(A, b, order, step, partialSum);
42          }
43  }
```

Figura 22. Cholesky: Kernel para trabajar sobre una matriz densa

El kernel principal se muestra en la Figura 22, como puede observarse el kernel llama a las funciones de dispositivo mencionadas anteriormente, dependiendo del valor del argumento *sstep*, los valores 0, 1 y 2 indican que nos encontramos en el proceso de factorización y los valores 3, 4 y 5 que estamos en el proceso de solución de los sistemas triangulares.

El kernel es llamado *n* veces para *sstep* igual a 0 y 1 (con *step* de 0 a *n-1*), solo una vez para *sstep=2*, *n* veces para *sstep=3* (con *step* de 0 a *n-1*), una vez para *sstep=4* y finalmente *n* veces para *sstep=5*(con *step* de 0 a *n-1*).

4.1.5.3.Factorización LDL^T

El proceso para resolver un sistema usando la factorización LDL^T es similar al que utilizamos en la factorización de Cholesky.

Para resolver un sistema de ecuaciones usando la factorización LDL^T, son necesarias dos etapas, uno de la factorización misma y otro que consiste en resolver los sistemas triangulares generados después de la factorización.

El proceso de factorización fue dividido en dos partes, un paso en el que modificamos una columna *j* tal que:

$$A_{n,j} = \frac{A_{n,j}}{A_{j,j}}, n > j$$

```
1   void ldlt_1_c(__global double* A, int order, int step)
2   {
3
4           //índice global de hilo
5           int tx = get_global_id(0);
6
7           //número total de hilos
8           int ntx = get_num_groups(0)*get_local_size(0);
9
10          //columna base
11          __global double* colB = A + step * order;
12
13          //elemento A(step, step)
14          double _r = colB[step];
15
16          //cada hilo se encargará de modificar un elemento
17          //de la columna
18          for (int x = tx + step + 1; x < order; x += ntx) {
19                  colB[x] = colB[x] / _r;
20          }
21  }
```

Figura 23. LDL^T: función de dispositivo *ldlt_1_c*

La función de dispositivo que realiza esta operación se muestra en la Figura 23, el trabajo de modificar los elementos de la columna se le asigna a todos los hilos de todos los grupos, es por eso que en vez de pedir el id local de hilo se pide el id global (línea 5), la columna a trabajar es la misma determinada por el argumento *step*.

Otro paso consiste en modificar las columnas siguientes a la columna j modificada en el paso anterior tal que:

$$A_{n,s} = A_{n,s} - A_{s,j}A_{n,j}A_{j,j}, s > j, n \geq s$$

La función de dispositivo que realiza esta tarea se muestra en la Figura 24, en este caso, cada grupo y sus hilos se encargarán de modificar una columna.

```
1   void ldlt_2_c(__global double* A, int order, int step)
2   {
3           //índice de grupo
4           int bx = get_group_id(0);
5
6           //índice local de hilo
7           int tx = get_local_id(0);
8
9           //columna base
10          __global double* colB = A + step * order;
11
12          //elemento A(step, step)
13          double _r = colB[step];
14
15          //parámetros de apoyo
16          double _k;
17
18          //en este caso, cada grupo se encargará de las
19          //operaciones sobre todos los elementos de una
20          //columna
21          for (int y = bx + step + 1; y < order; y += get_num_groups(0)) {
22                  _k = _r * colB[y];
23                  __global double* colI = A + y * order;
24
25                  for (int x = tx + y; x < order;
26                          x += get_local_size(0)) {
27
28                          colI[x] -= _k * colB[x];
29                  }
30          }
31  }
```

Figura 24. LDL^T: función de dispositivo *ldlt_2_c*

Estos dos primeros pasos deben realizarse uno después de otro durante n iteraciones para terminar con la factorización. Terminada la factorización en sitio, A contendrá en su parte triangular inferior la factorización LDL^T, L en su parte estrictamente triangular inferior y D en su diagonal, debe recordarse que L por si sola tiene sus elementos diagonales iguales a 1, esto es implícito y debe tenerse en cuenta cuando llegue el momento de resolver el sistema.

Terminada la factorización se procede a solucionar los sistemas equivalentes, es decir si $Ax = b$ es el sistema original, luego de la factorización quedará:

$$LDL^T x = b$$

Haciendo $DL^Tx = y$ entonces $Ly = b$, es el primer sistema a resolverse para y ,luego haciendo $L^Tx = z$ entonces $Dz = y$ es el segundo sistema a resolver en z, que es simple ya que D es una matriz diagonal, finalmente se resuelve $L^Tx = z$, cuya solución en x es la solución del sistema $Ax = b$.

Para resolver el sistema $Ly = b$, usaremos la eliminación típica de Gauss Jordan aplicada a una matriz triangular inferior, en cada iteración los elementos de cada columna fuera de la diagonal van volviéndose cero progresivamente, hasta llegar a una matriz diagonal, en ningún momento debe hacerse uso de la parte triangular superior (son implícitamente ceros, aunque en memoria guardan la parte triangular superior de la matriz original *A*), resolver un sistema triangular nos permite obviar las operaciones sobre L y dejarla intacta para poder usarla al resolver el sistema $L^Tx = z$, solo se modificará el vector de constantes *b* tal que para la iteración *j*:

$$b_n = b_n - b_j L_{n,j}, n > j$$

```
1  void ldlt_3_c(__global double* L, __global double* b, int order,
2          int step)
3  {
4
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         //columna de L implicada
12         __global double* colB = L + order * step;
13
14         for (int x = tx + step + 1; x < order; x += ntx) {
15                 b[x] -= b[step] * colB[x];
16         }
17 }
```

Figura 25. LDL^T: función de dispositivo *ldlt_3_c*

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 25, el número de iteración y por la tanto el índice de la columna implicada está dada por el valor del argumento *step*, esta función debe ser llamada *n* veces para terminar con la eliminación, al final *b* contendrá la solución del sistema $Ly = b$, que será usado como argumento al resolver el siguiente sistema.

Luego se procede a resolver el sistema $Dz = y$, ya que D es diagonal, la solución se reduce a realizar la operación:

$$b_j = \frac{b_j}{D_{j,j}}$$

La función de dispositivo que realiza esta operación se muestra en la Figura 26, el kernel se ejecuta una sola vez.

```
1  void ldlt_4_c(__global double* D, __global double* b, int order)
2  {
3
4          //índice global de hilo
5          int tx = get_global_id(0);
6
7          //número total de hilos
8          int ntx = get_num_groups(0)*get_local_size(0);
9
10         //parámetros de apoyo
11         int diag;
12
13         for (int x = tx; x < order; x += ntx) {
14                 //ubicación del elemento diagonal en memoria
15                 diag = x * order + x;
16                 b[x] = b[x] / D[diag];
17         }
18 }
```

Figura 26. LDL^T: función de dispositivo *ldlt_4_c*

Para resolver el sistema $L^T x = z$, se usará la misma matriz que almacena L, el artificio es asumir que está ordenada por filas y no por columnas, con esto la transposición de la matriz ya no será necesaria. L^T es una matriz triangular superior, se resolverá el sistema obteniendo el valor de las variables una a continuación de otra desde abajo, recordar que los elementos de la diagonal de L son 1 aunque no estén almacenados, así, el valor de la variable en la posición *i* sería:

$$x_i = z_i - \sum_{j=i+1}^{n-1} z_j L_{i,j}$$

```
1   void ldlt_5_c(__global double* L, __global double* z, int order,
2           int step, __local double* partialSum)
3   {
4           //índice de bloque
5           int bx = get_group_id(0);
6
7           //índice global de hilo
8           int tx = get_local_id(0);
9
10          //número total de hilos
11          int ntx = get_local_size(0);
12
13          //índice de la fila de L implicada
14          int fL = order - step - 1;
15
16          //fila de L implicada, data
17          __global double* filB = L + fL * order;
18
19          //valor donde se guardará una suma parcial
20          double sum = 0.0;
21
22          //solo se usará un bloque
23          if (bx == 0) {
24                  for (int x = tx + fL + 1; x < order; x += ntx) {
25                          sum += filB[x] * z[x];
26                  }
27                  partialSum[tx] = sum;
28
29                  //reducción en paralelo pare determinar la suma de
30                  //todos los elementos de partialSum
31                  for (int stride = ntx / 2; stride > 0; stride /= 2) {
32                          barrier(CLK_LOCAL_MEM_FENCE);
33                          if (tx < stride) {
34                                  partialSum[tx] += partialSum[tx + stride];
35                          }
36                  }
37                  if (tx == 0)
38                          z[fL] -= partialSum[0];
39          }
40  }
```

Figura 27. LDL^T: función de dispositivo *ldlt_5_c*

Se usa el mismo vector *z* para almacenar la solución desde abajo. La función de dispositivo que resuelve el sistema $L^T x = z$ se muestra en la Figura 27, cada llamada a esta función calculará una variable. Ya que el sistema se resuelve desde abajo, para un paso *j* se calculará el valor de la variable *i* tal que *i=n-j-1*, la fila de *L* usada para calcular esta variable tiene el mismo índice (se obtiene en la línea 14). Esta función de dispositivo usará memoria local para calcular la sumatoria $\sum_{j=i+1}^{n-1} z_j L_{i,j}$ por reducción en paralelo, el tamaño de esta memoria local será el mismo que el tamaño de un grupo de trabajo, por esta razón, en este caso solo se utilizará un grupo de trabajo al usar esta función (en la línea 23 se especifica que el grupo de trabajo utilizado es únicamente el con id cero).

En la línea 24 a 26 se calcula una suma parcial almacenándola en la variable *sum* (declarada en la línea 20), luego este valor es almacenado en memoria

local (línea 27), los elementos de *partialsum* se suman en paralelo en el bloque *for* de las líneas 31 a 36, finalmente se calcula la variable correspondiente en la línea 38, esta tarea solo la realiza un hilo (en la línea 37 se especifica el id del hilo que realiza la operación). Esta función de dispositivo debe ser llamada también *n* veces para calcular las *n* variables.

```
1  __kernel void
2  ldlt_c(__global double* A, __global double* b,
3          __local double* partialSum, int order, int step, int sstep)
4  {
5          //la iteración principal se realiza en el host
6
7          //se modificará una sola columna
8          if (sstep == 0) {
9                  ldlt_1_c(A, order, step);
10         }
11         //se modificará las columnas siguientes a la columna
12         //implicada
13         else if (sstep == 1) {
14                 ldlt_2_c(A, order, step);
15         }
16
17         ///hasta aquí la factorización está concluida, la parte
18         //triangular inferior de A contendrá la factorización LDLT
19         //es decir L y D
20         //desde aquí se procederá a solucionar el sistema
21         //L*D*transpose(L)x=b
22
23         //se resolverá el sistema Ly=b, y=D*transpose(L)*x
24         else if (sstep == 2) {
25                 ldlt_3_c(A, b, order, step);
26         }
27
28         //se resolverá el sistema Dz=y, z=transpose(L)*x
29         else if (sstep == 3) {
30                 ldlt_4_c(A, b, order);
31         }
32
33         //se resolverá el sistema transpose(L)*x=z
34         //"z" ya debe haberse obtenido usando la función anterior
35         else {
36                 ldlt_5_c(A, b, order, step, partialSum);
37         }
38 }
```

Figura 28. LDL^T: Kernel para trabajar sobre una matriz densa

El kernel principal se muestra en la Figura 28, como puede observarse el kernel llama a las funciones de dispositivo mencionadas anteriormente, dependiendo del valor del argumento *sstep*, los valores 0 y 1 indican que nos encontramos en el proceso de factorización y los valores 2, 3 y 4 que estamos en el proceso de solución de los sistemas triangulares.

El kernel es llamado *n* veces para *sstep* igual a 0 y 1 (con *step* de 0 a *n-1*), *n* veces para *sstep=2* (con *step* de 0 a *n-1*), una vez para *sstep=3* y finalmente *n* veces para *sstep=4*(con *step* de 0 a *n-1*).

4.1.5.4.Gradientes conjugados

La ventaja de este método es que solo requiere de operaciones sencillas en cada iteración, estas operaciones puede hacerse haciendo uso de las funciones incorporadas en ArrayFire, por lo que solo necesitamos escribir el código en esta interface, sin necesidad de escribir un kernel.

En la Figura 29 se muestra la función que se exporta para llamarla desde Julia, *C* es el objeto *af_array** donde se guardará la solución del sistema, *A* es la matriz de coeficientes del sistema, *b* es el vector de constantes del sistema e *Ierr* es el error máximo para aceptar una solución. En las líneas 4 a 7 se extrae el orden de la matriz, las líneas 9 a 10 se extrae el tipo (*float o double*, cuyos equivalentes en ArrayFire son *f32* y *f64* respectivamente), en las líneas 12 a 26 se declaran objetos que se usarán durante las iteraciones, en las líneas 28 a 57 se establecen valores iniciales antes de empezar la iteración, y finalmente en las líneas 58 a 93 se procede con la iteración. Notar que la iteración termina una vez que la norma del vector *r* sea menor al valor de *Ierr* introducido como argumento (en las líneas 66 a 67 se hace este chequeo en cada iteración).

```
1  void AFire::SEL_gc(af_array* C, af_array A, af_array b,
2          double Ierr) {
3
4          dim_t _order[AF_MAX_DIMS];
5          af_get_dims(&_order[0], &_order[1], &_order[2],
6                  &_order[3], A);
7          size_t order = _order[0];
8
9          af_dtype typef;
10         af_get_type(&typef, A);
11
12         double normr;
13         //af_array de ayuda
14         af_array zero;
15         dim_t d_order[] = { 1 };
16         af_constant(&zero, 0, 1, d_order, typef);
17         af_array Ax0;
18         af_array rtxp;
19         af_array ptxz;
20         af_array axp;
21         af_array B;
22         af_array axz;
23         af_array copyr;
24         af_array rtxz;
```

```
25          af_array rsp;
26          af_array Bxp;
27
28          //x0=b
29          af_array x0;
30          af_copy_array(&x0, b);
31
32          //r=b-A*x0
33          af_array r;
34          af_matmul(&Ax0, A, x0, AF_MAT_NONE, AF_MAT_NONE);
35          af_sub(&r, b, Ax0, false);
36
37          //p = r
38          af_array p;
39          af_copy_array(&p, r);
40
41          //z=A*p
42          af_array z;
43          af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
44
45          //a = (r'*p)/(p'*z)
46          af_array a;
47          af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
48          af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
49          af_div(&a, rtxp, ptxz, false);
50
51          //x = x0 + a.*p
52          af_array x;
53          af_mul(&axp, a, p, true);
54          af_add(&x, x0, axp, false);
55
56          int contin;
57          int I = 0;
58          for (int i = 0; i < order; i++)
59          {
60                  //r -= a.*z
61                  af_copy_array(&copyr, r);
62                  af_mul(&axz, a, z, true);
63                  af_sub(&r, copyr, axz, false);
64
65                  af_norm(&normr, r, AF_NORM_EUCLID, 1, 1);
66                  if (normr <= Ierr)
67                          break;
68
69                  //B = -(r'*z)/(p'*z)
70                  af_matmul(&rtxz, r, z, AF_MAT_TRANS, AF_MAT_NONE);
71                  af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
72                  af_div(&rsp, rtxz, ptxz, false);
73                  af_sub(&B, zero, rsp, false);
74
75                  //p = r + B.*p
76                  af_mul(&Bxp, B, p, true);
77                  af_add(&p, r, Bxp, false);
78
79                  //z = A*p
80                  af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
81
82                  //a = (r'*p)/(p'*z)
83                  af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
84                  af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
85                  af_div(&a, rtxp, ptxz, false);
86
87                  //x += a.*p
88                  af_mul(&axp, a, p, true);
89                  af_copy_array(&x0, x);
90                  af_add(&x, x0, axp, false);
91
92                  I++;
93          }
94
```

```
95          //copiando el resultado en el argumento de salida
96          af_copy_array(C, x);
97
98          //liberando objetos af_array usados
99          af_release_array(zero);
100         af_release_array(Ax0);
101         af_release_array(rtxp);
102         af_release_array(ptxz);
203         af_release_array(axp);
104         af_release_array(B);
105         af_release_array(axz);
106         af_release_array(copyr);
107         af_release_array(rtxz);
108         af_release_array(rsp);
109         af_release_array(Bxp);
110         af_release_array(x0);
111         af_release_array(r);
112         af_release_array(p);
113         af_release_array(z);
114         af_release_array(a);
115         af_release_array(x);
116 }
```

Figura 29. Gradientes conjugados: función principal a exportar

4.1.6. Kernels sobre matrices en formato CSC

Se presentan los kernels elaborados para trabajar sobre matrices en formato CSC. Se ha definido una estructura *SparseMS* para los objetos en este formato cuya definición en Julia se muestra en la Figura 30, los campos son: *elements* que guardan todos los elementos distintos de cero, *colIdx* donde cada elemento indicará la ubicación en *elements* del primer elemento distinto de cero en cada columna de la matriz original y *rowIdx* que guarda los índices de fila de todos los elementos. De la misma manera definimos la estructura *af_SparseMS* (Figura 31) análogo a *SparseMS* que guardará la matriz dispersa en formato CSC con los campos del tipo *AFArray*, que indica que son objetos en la memoria del dispositivo.

```
1 struct SparseMS{T,N}
2     elements::Array{T,1}
3     colIdx::Array{N,1}
4     rowIdx::Array{N,1}
5 end
```

Figura 30. Estructura para objetos que almacenan la matriz en formato CSC

```
struct af_SparseMS{T,N}
    elements::AFArray{T,1}
    colIdx::AFArray{N,1}
    rowIdx::AFArray{N,1}
end
```

Figura 31. Estructura para objetos que almacenan la matriz en formato CSC con la memoria del dispositivo

```
function sparse_ms(A::Array{T,2}) where {T<:Real}
    colIdx=Array{Int32,1}(UndefInitializer(),0);
    rowIdx=Array{Int32,1}(UndefInitializer(),0);
    elements=Array{T,1}(UndefInitializer(),0);
    cont=1;
    for j in 1:size(A)[1]
        push!(colIdx,cont);
        for i in j:size(A)[1]
            if i==j
                #el elemento diagonal siempre será guardado, pero su indice será
                #implicito (no será almacenado en rowIdx)
                push!(elements,A[i,j]);
                cont+=1;
            else
                if A[i,j]!=0
                    push!(elements,A[i,j]);
                    push!(rowIdx,i);
                    cont+=1;
                end
            end
        end
    end
    SparseMS(elements,colIdx,rowIdx)
end
```

Figura 32. Conversión de una matriz densa a una matriz dispersa en formato CSC

La Figura 32 muestra el código en Julia para convertir una matriz simétrica en formato denso en una matriz en formato CSC, para poder usar las funciones exportadas todavía es necesario tener la memoria en el dispositivo, para esto, hacemos uso de la función incorporada de ArrayFire *AFArray*, que hace una copia en la memoria del dispositivo de un objeto del tipo *Array*, se ha sobrecargado la función para poder hacer una copia de un objeto del tipo *SparseMS* en la memoria del dispositivo, en otras palabras convertir del tipo *SparseMS* a *af_SparseMS*, esta función se muestra en la Figura 33.

```
function AFArray(A::SparseMS{T,N}) where {T<:Real,N<:Real}
    Ael=AFArray(A.elements);
    AcIdx=AFArray(A.colIdx);
    ArIdx=AFArray(A.rowIdx);
    af_SparseMS(Ael,AcIdx,ArIdx)
end
```

Figura 33. Función para convertir un objeto del tipo *SparseMS* a af*_SparseMS*

4.1.6.1.Factorización de Cholesky

Para resolver un sistema de ecuaciones usando la factorización de Cholesky, son necesarias dos etapas, uno de la factorización misma y otro que consiste en resolver los sistemas triangulares generados después de la factorización.

El proceso es similar al usado sobre matrices densas, sin embargo para empezar con la factorización es necesario realizar una factorización simbólica, esto con el fin de obtener la cantidad de memoria máxima requerida para almacenar los elementos de la factorización, en la Figura 34 se muestra el código en Julia propuesto, esta función obtiene un objeto del tipo *SparseMS* cuyos campos *colIdx* y *rowIdx* son los finales, sin embargo el campo *elements* solo contiene ceros, pero la cantidad de elementos es la necesaria para guardar la factorización.

Sea *A* la matriz original en formato CSC y *L* la matriz que contendrá la factorización de Cholesky también en formato CSC. Un paso previo es necesario antes de proceder con la factorización, este paso consiste en llenar los espacios del campo *elements* de L, con los correspondientes elementos del campo *elements* de *A*, la función de dispositivo que realiza esta tarea se muestra la Figura 35.

```
1  function symbolic_chol_fac(A::SparseMS{T,N}) where {T<:Real,N<:Real}
2      Col=A.colIdx;
3      Row=A.rowIdx;
4      orden=length(Col);
5
6      #construyendo un vector de vectores, donde se separará por columna los
7      #indices almacenados en Row
8      Rows=Array{Array{N,1},1}(UndefInitializer(),0);
9      cont=1;
10     for i in 1:orden-1
11         nEl=Col[i+1]-Col[i]-1;
12         if nEl>0
13             push!(Rows,Row[cont:cont+nEl-1]);
14             cont+=nEl;
15         else
16             #significa que no hay elementos fuera de la diagonal
17             push!(Rows,[]);
18         end
19     end
20
21   for i in 1:orden-1
22         nEl=length(Rows[i]);
23         #debe haber un minimo de 2 elementos
24         if nEl>1
25             for j in 1:nEl-1
26                 union!(Rows[Rows[i][j]],Rows[i][j+1:end]);
27                 sort!(Rows[Rows[i][j]]);
28             end
29         end
30   end
31     #generando los campos del objeto de salida
32     colId=ones(N,orden);
33     rowId=zeros(N,0);
34     for i in 1:orden-1
35         append!(rowId,Rows[1]);
36         colId[i+1]=colId[i]+length(Rows[1])+1;
37         popfirst!(Rows);
38     end
39     Elm=zeros(T,colId[end]);
40     SparseMS(Elm,colId,rowId)
41 end
```

Figura 34. Función que realiza la factorización de Cholesky simbólica

La función acepta seis argumentos, los campos de *A* y *L* en formato CSC (*elmA, colA, rowA* y *elmL, colL, rowL* respectivamente) y el tamaño del campo de *elmA.* En la línea 6 se obtiene el índice global del hilo, la línea 10 calcula el número total de hilos como la multiplicación del número de grupos y el tamaño de grupo, el bloque *for* de la línea 11 a 51 realiza la copia de elementos de *elmA* a *elmL*, cada hilo se encargará de realizar la copia de un elemento de *elmA* en el lugar correspondiente de *elmL*; el bloque *while* de las líneas 19 a 31 realiza la búsqueda de los índices de fila y columna del elemento actual en *elmA*, teniendo estos índices se calcula la ubicación del elemento en *elmL*, el bloque

if de las líneas 38 a 47 se encarga de esta tarea, finalmente en la línea 50 se ubica el elemento en el lugar correspondiente. Esta función es llamada una sola vez para rellenar los espacios de *elmL* con los elementos de *elmA* correspondientes, luego de esto la factorización se realizará en sitio sobre *elmL*.

```
1  void sparse_fill(__global double* elmA, __global int* colA,
2          __global int* rowA, __global double* elmL, __global int* colL,
3          __global int* rowL, int size_elmA)
4  {
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         for (int x = tx; x < size_elmA; x += ntx) {
12                 //variable donde se guardará el índice de columna y de fila
13                 //respectivamente del elemento elmA[tx]
14                 int col = 0;
15                 int row = 0;
16                 //variable que controlará la salida del bucle.
17                 int key = 0;
18
19                 while (key == 0) {
20                         if (colA[col] > x) {
21                                 col -= 1;
22                                 row = rowA[x - col - 1];
23                                 key = 1;
24                         }
25                         else if (colA[col] == x) {
26                                 row = col;
27                                 key = 1;
28                         }
29                         else
30                                 col += 1;
31                 }
32                 //hasta aquí para un elemento de ElmA, ya se
33                 //se cuenta con el índice de fila y de columna (row y col)
34                 //ahora necesitamos ubicar este elemento en elmL
35
36                 int posf;//posición final en elmL
37
38                 if (col != row) {
39                         int pos = colL[col] - col;
40                         int cont = 0;
41                         key = 0;
42                         while (key == 0)
43                                 rowL[pos + cont] == row ? key = 1 : cont++;
44                         posf = colL[col] + cont + 1;
45                 }
46                 else
47                         posf = colL[col];
48                 //escribiendo elementos de elmA en el lugar
49                 //correspondiente de elmL
50                 elmL[posf] = elmA[x];
51         }
52 }
```

Figura 35. Cholesky: función de dispositivo *sparse_fill*

Similar al proceso sobre matrices densas, la factorización fue dividida en tres partes, un paso en el que modificamos una columna *j* tal que:

$$A_{n,j} = \frac{A_{r,j}}{\sqrt{A_{j,j}}}, n > j$$

```
1  void Chol_sparse_1(__global double* elmL, __global int* colL,
2          int order, int step)
3  {
4          //es procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //obteniendo la ubicación del primer elemento
11                 //de la columna determinada por step
12                 int pos = colL[step];
13
14                 //posición siguiente
15                 int pos_nxt = colL[step + 1];
16
17                 //índice global de hilo
18                 int tx = get_global_id(0);
19
20                 //número total de hilos
21                 int ntx = get_num_groups(0)*get_local_size(0);
22
23                 //raíz cuadrada del elemento diagonal. El primer
24                 //elemento guardado en elmA para cada columna es el
25                 //elemento diagonal
26                 double _r = sqrt(elmL[pos]);
27
28                 //cada hilo se encargará de modificar un elemento
29                 //de la columna
30                 for (int x = tx + pos + 1; x < pos_nxt; x += ntx) {
31                         elmL[x] = elmL[x] / _r;
32                 }
33         }
34 }
```

Figura 36. Cholesky: función de dispositivo *Chol_sparse_1*

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 36, como puede observarse para realizar esta operación solo es necesario introducir como argumentos *elmL* y *colL*, la división se realiza sobre todos los elementos de una columna sin importar el índice de fila que tengan, cada hilo se encargará de realizar la modificación de uno de los elementos de la columna, en la línea 12 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna (la columna es determinada con el argumento *step*), de manera similar en la línea 15 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna siguiente, los elementos en *elmL* con índices entre estos dos valores son los que se tienen que modificar. En las líneas 18 y 21 se obtienen el índice global de hilo y el número total de hilos respectivamente, en la línea 26 se obtiene la raíz cuadrada del elemento diagonal y finalmente en la línea 31 se modifican los elementos de la columna.

Otro paso consiste en modificar las columnas siguientes a la columna *j* modificada en el paso anterior tal que:

$$A_{n,s} = A_{n,s} - A_{s,j}A_{n,j}, s > j, n \geq s$$

La función de dispositivo que realiza esta tarea se muestra en la Figura 37, en este caso, cada grupo y sus hilos se encargarán de modificar una columna. En las líneas 11 y 14 se obtiene el índice de grupo y el índice local de hilo respectivamente, la línea 17 obtiene la ubicación en *elmL* del primer elemento de la columna (determinada por *step*), en la línea 20 se calcula la ubicación en *rowL* de primer índice de fila almacenado para la columna, en la línea 23 se calcula el número de elementos en *rowL* que le corresponden a la columna, debe recordarse que no se almacena el índice de fila del elemento diagonal por ser obvio, el bloque *if* de las líneas 25 a 61 índica que el bloque *for* anidado (líneas 27 a 60) solo se utiliza cuando el número de elementos en *rowL* para la columna es por lo menos uno.

El bloque *for* de las líneas 27 a 60 realizará las modificaciones de todas las columnas siguientes a la columna actual (argumento *step*).

Estos dos primeros pasos deben realizarse uno después de otro *n-1* veces.

Un tercer paso es necesario para terminar la factorización, este modifica los elementos de la diagonal tal que:

$$A_{j,j} = \overline{\overline{A_{j,j}}}$$

La función de dispositivo que realiza esta tarea se muestra en la Figura 38, todos los hilos de todos los grupos se encargan cada uno de modificar un elemento de la diagonal, este paso solo se realiza una vez luego de realizada las *n-1* iteraciones con los dos pasos vistos previamente.

```
1  void Chol_sparse_2(__global double* elmL, __global int* colL,
2          __global int* rowL, int order, int step)
3  {
4          //es procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //índice local de grupo
11                 int bx = get_group_id(0);
12
13                 //índice local de hilo
14                 int tx = get_local_id(0);
15
16                 //posición del primer elemento de la columna de elmL
17                 int pos_elmL = colL[step];
18
19                 //posición del primer elemento de la columna en rowL
20                 int pos_rowL = pos_elmL - step;
21
22                 //número de elementos en rowL para la columna
23                 int nEl = colL[step + 1] - pos_elmL - 1;
24
25                 if (nEl > 0) {
26
27                         for (int b = bx; b < nEl; b += get_num_groups(0)) {
28                                 //índice de la columna a modificar
29                                 int colId = rowL[pos_rowL + b];
30
31                                 //primer factor
32                                 double k1 = elmL[pos_elmL + b + 1];
33
34                                 for (int x = b + tx; x < nEl;
35                                         x += get_local_size(0)) {
36
37                                         //índice de la fila en la columna a
38                                         //modificar
39                                         int rowId = rowL[pos_rowL + x];
40
41                                         //segundo factor
42                                         double k2 = elmL[pos_elmL + x + 1];
43
44                                         int posf;//posición final en elmL
45
46                                         if (colId != rowId) {
47                                                 int pos = colL[colId] - colId;
48                                                 int cont = 0;
49                                                 int key = 0;
50                                                 while (key == 0)
51                                                         rowL[pos + cont] ==
52                                                         rowId ? key = 1 : cont++;
53                                                 posf = colL[colId] + cont + 1;
54                                         }
55                                         else
56                                                 posf = colL[colId];
57
58                                         elmL[posf] -= k1 * k2;
59                                 }
60                         }
61                 }
62         }
63 }
```

Figura 37. Cholesky: función de dispositivo *Chol_sparse_2*

```
1  void Chol_sparse_3(__global double* elmL,
2          __global int* colL, int order)
3  {
4
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         for (int x = tx; x < order; x += ntx) {
12                 int diag = colL[x];
13                 elmL[diag] = sqrt(elmL[diag]);
14         }
15  }
```

Figura 38. Cholesky: función de dispositivo *Chol_sparse_3*

Terminada la factorización se procede a solucionar los sistemas equivalentes, es decir si $Ax = b$ es el sistema original, luego de la factorización quedará:

$$LL^T x = b$$

Haciendo $L^T x = y$ entonces $Ly = b$. Primero se resuelve $Ly = b$ y finalmente $L^T x = y$, cuya solución en x es la solución del sistema $Ax = b$.

Para resolver el sistema $Ly = b$, usaremos la eliminación típica de Gauss Jordan aplicada a una matriz triangular inferior, en cada iteración los elementos de cada columna fuera de la diagonal van volviéndose cero progresivamente, hasta llegar a una matriz diagonal, las operaciones sobre L pueden obviarse y dejarla intacta para poder usarla al resolver el siguiente sistema $L^T x = y$, es decir solo se modificará el vector de constantes *b* tal que para la iteración *j*:

$$b_n = b_n - b_j \frac{L_{n,j}}{L_{j,j}}, n > j$$

El kernel que realiza esta operación se muestra en la Figura 39, el número de iteración y por la tanto el índice de la columna implicada está dada por el valor del argumento *step*, esta función debe ser llamada *n-1* veces para terminar con la eliminación, luego será necesario dividir los elementos de *b* entre los elementos de la diagonal (función de dispositivo de la Figura 40), este vector será la solución del sistema $Ly = b$, que será usado como argumento al resolver el siguiente sistema triangular.

```
1  void Chol_sparse_4(__global double* elmL,
2          __global int* colL, __global int* rowL, __global double* b,
3          int order, int step)
4  {
5          if (step < order - 1) {
6                  //índice global de hilo
7                  int tx = get_global_id(0);
8
9                  //número total de hilos
10                 int ntx = get_num_groups(0)*get_local_size(0);
11
12                 //posición del elemento diagonal
13                 int diag = colL[step];
14
15                 //número de elementos para una columna
16                 int nEl = colL[step + 1] - diag;
17
18                 //deben haber elementos aparte del elemento diagonal
19                 if (nEl > 1) {
20
21                         //elemento diagonal y elemento correspondiente
22                         //en b
23                         double diag_elm = elmL[diag];
24                         double elmb = b[step];
25
26                         //posición en rowL del primer elemento fuera de
27                         //la diagonal (índice)
28                         int pos_rowL = diag - step;
29                         double k = elmb / diag_elm;
30
31                         for (int x = tx; x < nEl - 1; x += ntx) {
32                                 int rowId = rowL[pos_rowL + x];
33                                 b[rowId] -= k * elmL[diag + x + 1];
34                         }
35                 }
36         }
37 }
```

Figura 39. Cholesky: función de dispositivo *Chol_sparse_4*

```
1  void Chol_sparse_5(__global double* elmL,
2          __global int* colL, __global double* b, int order)
3  {
4
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         for (int x = tx; x < order; x += ntx) {
12                 //ubicación del elemento diagonal en elmL
13                 int diag = colL[x];
14                 b[x] = b[x]/ elmL[diag];
15         }
16 }
```

Figura 40. Cholesky: función de dispositivo *Chol_sparse_5*

Para resolver el sistema $L^T x = y$, se usará la misma matriz que almacena L, el artificio es asumir que está ordenada por filas y no por columnas, hecho esto la transposición de la matriz ya no será necesaria. L^T es una matriz triangular

superior, se resolverá el sistema obteniendo el valor de las variables una a continuación de otra desde abajo, el valor de la variable en la posición *i* sería:

$$x_i = \frac{y_i - \sum_{j=i+1}^{n-1} x_j L_{i,j}}{L_{i,i}}$$

```
1   void Chol_sparse_6(__global double* elmL,
2           __global int* colL, __global int* rowL, __global double* y,
3           int order, int step, __local double* partialSum)
4   {
5           //índice de bloque
6           int bx = get_group_id(0);
7
8           //índice global de hilo
9           int tx = get_local_id(0);
10
11          //número total de hilos
12          int ntx = get_local_size(0);
13
14          //índice de la fila de L implicada
15          int fL = order - step - 1;
16
17          //ubicación de elemento diagonal de la fila en elmL
18          int diagId = colL[fL];
19
20          //número de elementos en la fila
21          int nEl = (fL == order - 1 ? 1 : colL[fL + 1] - diagId);
22
23          //ubicación del índice del primer elemento fuera de la
24          //diagonal en rowL
25          int pos_rowL = diagId - fL;
26
27          //elemento diagonal
28          double diag = elmL[diagId];
29
30          //valor donde se guardará una suma parcial
31          double sum = 0.0;
32
33          //solo se usará un bloque
34          if (bx == 0) {
35                  for (int x = tx; x < nEl - 1; x += ntx) {
36                          int rowId = rowL[pos_rowL + x];
37                          sum += elmL[diagId + x + 1] * y[rowId];
38                  }
39                  partialSum[tx] = sum;
40
41                  //reducción en paralelo pare determinar la suma de
42                  //todos los elementos de partialSum
43                  for (int stride = ntx / 2; stride > 0; stride /= 2) {
44                          barrier(CLK_LOCAL_MEM_FENCE);
45                          if (tx < stride) {
46                                  partialSum[tx] += partialSum[tx + stride];
47                          }
48                  }
49                  if (tx == 0)
50                          y[fL] = (y[fL] - partialSum[0]) / diag;
51          }
52  }
```

Figura 41. Cholesky: función de dispositivo *Chol_sparse_6*

Se usa el mismo vector *y* para almacenar la solución desde abajo. La función de dispositivo que resuelve el sistema $L^T x = y$ se muestra en la Figura 41, cada llamada a esta función calculará una variable. Ya que el sistema se resuelve desde abajo, para un paso *j* se calculará el valor de la variable *i* tal que *i=n-j-1*, la fila de *L* usada para calcular esta variable tiene el mismo índice (se obtiene en la línea 15). Esta función de dispositivo usará memoria local para calcular la sumatoria $\sum_{j=i+1}^{n-1} x_j L_{i,j}$ por reducción en paralelo, el tamaño de esta memoria local será el mismo que el tamaño de un grupo de trabajo, por esta razón, en este caso solo se utilizará un grupo de trabajo al usar esta función (en la línea 34 se especifica que el grupo de trabajo utilizado es únicamente el con id cero).

```
1   __kernel void
2   Cholesky_sparse_c(__global double* elmA, __global int* colA,
3           __global int* rowA, __global double* elmL, __global int* colL,
4           __global int* rowL, int size_elmA, int size_colA,
5           __global double* b, __local double* partialSum, int step,
6           int sstep)
7   {
8
9           if (sstep == -1)
10                  sparse_fill(elmA, colA, rowA, elmL, colL,
11                          rowL, size_elmA);
12          //la iteración principal se realiza en el host
13
14                  //se modificará una sola columna
15          else if (sstep == 0)
16                  Chol_sparse_1(elmL, colL, size_colA, step);
17
18          //se modificará las columnas siguientes a la columna
19          //implicada
20          else if (sstep == 1)
21                  Chol_sparse_2(elmL, colL, rowL, size_colA, step);
22
23          //una vez usada las funciones anteriores todavía es
24          //necesario modificar la diagonal D[i]=sqrt(D[i]) para
25          //concluir la factorización
26          else if (sstep == 2)
27                  Chol_sparse_3(elmL, colL, size_colA);
28          //hasta aquí la factorización está concluida, la parte
29          //triangular inferior de A contendrá la factorización de
30          //Cholesky es decir L
31          //desde aquí se procederá a solucionar el sistema
32          //L*transpose(L)x=b
33
34          //se resolverá el sistema Ly=b, y=transpose(L)*x
35          else if (sstep == 3)
36                  Chol_sparse_4(elmL, colL, rowL, b, size_colA, step);
37          else if (sstep == 4)
38                  Chol_sparse_5(elmL, colL, b, size_colA);
39
40          //se resolverá el sistema transpose(L)*x=y
41          //"y" ya debe haberse obtenido usando la función anterior
42          else
43                  Chol_sparse_6(elmL, colL, rowL, b, size_colA, step,
44                          partialSum);
45  }
```

Figura 42. Cholesky: Kernel para trabajar sobre una matriz en formato CSC

En la línea 35 a 37 se calcula una suma parcial almacenándola en la variable *sum* (declarada en la línea 31), luego este valor es almacenado en memoria local (línea 39), los elementos de *partialsum* se suman en paralelo en el bloque *for* de las líneas 43 a 48, finalmente se calcula la variable correspondiente en la línea 50, esta tarea solo la realiza un hilo (en la línea 49 se especifica el id del hilo que realiza la operación). Esta función de dispositivo debe ser llamada *n* veces para calcular las *n* variables.

El kernel principal se muestra en la Figura 42, como puede observarse el kernel llama a las funciones de dispositivo mencionadas anteriormente, dependiendo del valor del argumento *sstep*, un valor igual a -1 corresponde al proceso de copia de los elementos de *elmA* en el lugar correspondiente de *elmL*, los valores 0, 1 y 2 indican que nos encontramos en el proceso de factorización y los valores 3, 4 y 5 que estamos en el proceso de solución de los sistemas triangulares.

El kernel es llamado 1 vez para *sstep=-1*, *n-1* veces para *sstep* igual a 0 y 1 (con *step* de 0 a *n-2*), solo una vez para *sstep=2*, *n-1* veces para *sstep=3* (con *step* de 0 a *n-2*), una vez para *sstep=4* y finalmente *n* veces para *sstep=5*(con *step* de 0 a *n-1*).

4.1.6.2.Factorización LDL^T

El proceso para resolver un sistema usando la factorización LDL^T es similar al que utilizamos en la factorización de Cholesky.

Para resolver un sistema de ecuaciones usando la factorización LDL^T, son necesarias dos etapas, uno de la factorización misma y otro que consiste en resolver los sistemas triangulares generados después de la factorización.

El proceso es similar al usado sobre matrices densas, sin embargo para empezar con la factorización es necesario realizar una factorización simbólica, esto con el fin de obtener la cantidad de memoria máxima requerida para almacenar los elementos de la factorización, el espacio requerido para almacenar el campo *elements* de una factorización LDL^T es el mismo requerido para una

factorización de Cholesky, por tal la misma función usada para la factorización de Cholesky simbólica es usada para la factorización LDL^T simbólica (función Julia de la Figura 34 pág. 86), esta función obtiene un objeto del tipo *SparseMS* cuyos campos *colIdx* y *rowIdx* son los finales, sin embargo el campo *elements* solo contiene ceros, pero la cantidad de elementos es la necesaria para guardar la factorización.

```
1  void sparse_fill_c(__global double* elmA, __global int* colA,
2          __global int* rowA, __global double* elmL, __global int* colL,
3          __global int* rowL, int size_elmA)
4  {
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         for (int x = tx; x < size_elmA; x += ntx) {
12                 //variable donde se guardará el índice de columna y de fila
13                 //respectivamente del elemento elmA[tx]
14                 int col = 0;
15                 int row = 0;
16                 //variable que controlará la salida del bucle.
17                 int key = 0;
18
19                 while (key == 0) {
20                         if (colA[col] > x) {
21                                 col -= 1;
22                                 row = rowA[x - col - 1];
23                                 key = 1;
24                         }
25                         else if (colA[col] == x) {
26                                 row = col;
27                                 key = 1;
28                         }
29                         else
30                                 col += 1;
31                 }
32                 //hasta aquí para un elemento de ElmA, ya se
33                 //se cuenta con el índice de fila y de columna (row y col)
34                 //ahora necesitamos ubicar este elemento en elmL
35
36                 int posf;//posición final en elmL
37
38                 if (col != row) {
39                         int pos = colL[col] - col;
40                         int cont = 0;
41                         key = 0;
42                         while (key == 0)
43                                 rowL[pos + cont] == row ? key = 1 : cont++;
44                         posf = colL[col] + cont + 1;
45                 }
46                 else
47                         posf = colL[col];
48                 //escribiendo elementos de elmA en el lugar
49                 //correspondiente de elmL
50                 elmL[posf] = elmA[x];
51         }
52 }
```

Figura 43. LDL^T: función de dispositivo *sparse_fill_c*

Sea *A* la matriz original en formato CSC y *L* la matriz que contendrá la factorización LDL^T también en formato CSC. Un paso previo es necesario antes de proceder con la factorización, este paso consiste en copiar los elementos del campo *elements* de *A*, en el lugar correspondiente del campo *elements* de L para poder realizar la factorización en sitio sobre este último, la función de dispositivo que realiza esta tarea se muestra la Figura 43.

La función acepta seis argumentos, los campos de *A* y *L* en formato CSC (*elmA, colA, rowA* y *elmL, colL, rowL* respectivamente) y el tamaño de *elmA*. En la línea 6 se obtiene el índice global del hilo, la línea 10 calcula el número total de hilos como la multiplicación del número de grupos y el tamaño de grupo, el bloque *for* de la línea 11 a 51 realiza la copia de elementos de *elmA* a *elmL*, cada hilo se encargará de realizar la copia de un elemento de *elmA* en el lugar correspondiente de *elmL*; el bloque *while* de las líneas 19 a 31 realiza la búsqueda de los índices de fila y columna del elemento actual en *elmA*, teniendo estos índices se calcula la ubicación del elemento en *elmL*, el bloque *if* de las líneas 38 a 47 se encarga de esta tarea, finalmente en la línea 50 se ubica el elemento en el lugar correspondiente. Esta función es llamada una sola vez para rellenar los espacios de *elmL* con los elementos de *elmA* correspondientes, luego de esto la factorización se realizará en sitio sobre *elmL*.

El proceso de factorización fue dividido en dos partes, un paso en el que modificamos una columna *j* tal que:

$$A_{n,j} = \frac{A_{n,j}}{A_{j,j}}, n > j$$

La función de dispositivo que realiza esta operación se muestra en la Figura 44, como puede observarse para realizar esta operación solo es necesario introducir como argumentos *elmL* y *colL*, la división se realiza sobre todos los elementos de una columna sin importar el índice de fila que tengan, cada hilo se encargará de realizar la modificación de uno de los elementos de la columna, en la línea 12 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna (la columna es determinada con el argumento *step*), de manera similar en la

línea 15 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna siguiente, los elementos en *elmL* con índices entre estos dos valores son los que se tienen que modificar. En las líneas 18 y 21 se obtienen el índice global de hilo y el número total de hilos respectivamente, en la línea 26 se obtiene el elemento diagonal y finalmente en la línea 31 se modifican los elementos de la columna.

```
1  void ldlt_sparse_1_c(__global double* elmL, __global int* colL,
2          int order, int step)
3  {
4          //es procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //obteniendo la ubicación del primer elemento
11                 //de la columna determinada por step
12                 int pos = colL[step];
13
14                 //posición siguiente
15                 int pos_nxt = colL[step + 1];
16
17                 //índice global de hilo
18                 int tx = get_global_id(0);
19
20                 //número total de hilos
21                 int ntx = get_num_groups(0)*get_local_size(0);
22
23                 //elemento diagonal. El primer
24                 //elemento guardado en elmA para cada columna es el
25                 //elemento diagonal
26                 double _r = elmL[pos];
27
28                 //cada hilo se encargará de modificar un elemento
29                 //de la columna
30                 for (int x = tx + pos + 1; x < pos_nxt; x += ntx) {
31                         elmL[x] /= _r;
32                 }
33         }
34 }
```

Figura 44. LDL^T: función de dispositivo *ldlt_sparse_1_c*

Otro paso consiste en modificar las columnas siguientes a la columna *j* modificada en el paso anterior tal que:

$$A_{n,s} = A_{n,s} - A_{s,j}A_{n,j}A_{j,j}, s > j, n \geq s$$

```
1  void ldlt_sparse_2_c(__global double* elmL, __global int* colL,
2          __global int* rowL, int order, int step)
3  {
4          //es procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //índice local de grupo
11                 int bx = get_group_id(0);
12
13                 //índice local de hilo
14                 int tx = get_local_id(0);
15
16                 //posición del primer elemento de la columna de elmL
17                 int pos_elmL = colL[step];
18
19                 //elemento diagonal
20                 double diag = elmL[pos_elmL];
21
22                 //posición del primer elemento de la columna en rowL
23                 int pos_rowL = pos_elmL - step;
24
25                 //número de elementos en rowL para la columna
26                 int nEl = colL[step + 1] - pos_elmL - 1;
27
28                 if (nEl > 0) {
29
30                         for (int b = bx; b < nEl; b += get_num_groups(0)) {
31                                 //índice de la columna a modificar
32                                 int colId = rowL[pos_rowL + b];
33
34                                 //primer factor
35                                 double k1 = elmL[pos_elmL + b + 1];
36
37                                 for (int x = b + tx; x < nEl;
38                                         x += get_local_size(0)) {
39
40                                         //índice de la fila en la columna
41                                         //a modificar
42                                         int rowId = rowL[pos_rowL + x];
43
44                                         //segundo factor
45                                         double k2 = elmL[pos_elmL + x + 1];
46
47                                         int posf;//posición final en elmL
48
49                                         if (colId != rowId) {
50                                                 int pos = colL[colId] - colId;
51                                                 int cont = 0;
52                                                 int key = 0;
53                                                 while (key == 0)
54                                                         rowL[pos + cont] ==
55                                                         rowId ? key = 1 : cont++;
56                                                 posf = colL[colId] + cont + 1;
57                                         }
58                                         else
59                                                 posf = colL[colId];
60
61                                         elmL[posf] -= diag * k1 * k2;
62                                 }
63                         }
64                 }
65         }
66 }
```

Figura 45. LDL^T: función de dispositivo *ldlt_sparse_2_c*

La función de dispositivo propuesta que realiza esta tarea se muestra en la Figura 45, en este caso, cada grupo y sus hilos se encargarán de modificar una columna. En las líneas 11 y 14 se obtiene el índice de grupo y el índice local de hilo respectivamente, la línea 17 obtiene la ubicación en *elmL* del primer elemento de la columna actual (determinada por *step*), en la línea 20 se obtiene el elemento diagonal, en la línea 23 se calcula la ubicación en *rowL* de primer índice de fila almacenado para la columna, en la línea 26 se calcula el número de elementos en *rowL* que le corresponden a la columna, debe recordarse que no se almacena el índice de fila del elemento diagonal por ser obvio, el bloque *if* de las líneas 28 a 64 índica que el bloque *for* anidado (líneas 30 a 63) solo se utiliza cuando el número de elementos en *rowL* para la columna es por lo menos uno.

El bloque *for* de las líneas 30 a 63 realizará las modificaciones de todas las columnas siguientes a la columna actual (argumento *step*).

Estos dos primeros pasos deben realizarse uno después de otro *n-1* veces para terminar con la factorización. Terminada la factorización en sitio, el campo *elements* de *L* contendrá los elementos de la factorización LDL^T, este incluye los elementos de la diagonal *D*, almacenados como si fueran la diagonal de *L*, es decir, en formato CSC, *L* almacena una matriz triangular inferior, debe recordarse que L por si sola tiene sus elementos diagonales iguales a 1, esto es implícito y debe tenerse en cuenta cuando llegue el momento de resolver el sistema.

Terminada la factorización se procede a solucionar los sistemas equivalentes, es decir si $Ax = b$ es el sistema original, luego de la factorización quedará:

$$LDL^T x = b$$

Haciendo $DL^T x = y$ entonces $Ly = b$, es el primer sistema a resolverse para y ,luego haciendo $L^T x = z$ entonces $Dz = y$ es el segundo sistema a resolver en z, que es simple ya que D es una matriz diagonal, finalmente se resuelve $L^T x = z$, cuya solución en x es la solución del sistema $Ax = b$.

Para resolver el sistema $Ly = b$, usaremos la eliminación típica de Gauss Jordan aplicada a una matriz triangular inferior, en cada iteración los elementos de cada columna fuera de la diagonal van volviéndose cero progresivamente, hasta llegar a una matriz diagonal, resolver un sistema triangular nos permite obviar las operaciones sobre L y dejarla intacta para poder usarla al resolver el sistema $L^T x = z$, solo se modificará el vector de constantes b tal que para la iteración j:

$$b_n = b_n - b_j L_{n,j}, n > j$$

```
1   void ldlt_sparse_3_c(__global double* elmL,
2           __global int* colL, __global int* rowL, __global double* b,
3           int order, int step)
4   {
5           if (step < order - 1) {
6                   //índice global de hilo
7                   int tx = get_global_id(0);
8
9                   //número total de hilos
10                  int ntx = get_num_groups(0)*get_local_size(0);
11
12                  //posición del elemento diagonal
13                  int diag = colL[step];
14
15                  //número de elementos para una columna
16                  int nEl = colL[step + 1] - diag;
17
18                  //deben haber elementos aparte del elemento diagonal
19                  if (nEl > 1) {
20
21                          //elemento diagonal y elemento correspondiente en b
22                          double diag_elm = elmL[diag];
23                          double elmb = b[step];
24
25                          //posición en rowL del primer elemento fuera de la
26                          //diagonal (índice)
27                          int pos_rowL = diag - step;
28
29                          for (int x = tx; x < nEl - 1; x += ntx) {
30                                  int rowId = rowL[pos_rowL + x];
31                                  b[rowId] -= elmb * elmL[diag + x + 1];
32                          }
33                  }
34          }
35  }
```

Figura 46. LDL^T: función de dispositivo *ldlt_sparse_3_c*

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 46, el índice de la columna actual está dada por el valor del argumento *step*, esta función debe ser llamada *n-1* veces para terminar con la eliminación, al final b contendrá la solución del sistema $Ly = b$, que será usado como argumento al resolver el siguiente sistema.

Luego se procede a resolver el sistema $Dz = y$, ya que D es diagonal, la solución se reduce a realizar la operación:

$$b_j = \frac{b_j}{D_{j,j}}$$

La función de dispositivo que realiza esta operación se muestra en la Figura 47, esta función se ejecuta una sola vez.

```
1   void ldlt_sparse_4_c(__global double* elmL,
2           __global int* colL, __global double* b, int order)
3   {
4
5           //índice global de hilo
6           int tx = get_global_id(0);
7
8           //número total de hilos
9           int ntx = get_num_groups(0)*get_local_size(0);
10
11          for (int x = tx; x < order; x += ntx) {
12                  //ubicación del elemento diagonal en elmL
13                  int diag = colL[x];
14                  b[x] /= elmL[diag];
15          }
16  }
```

Figura 47. LDL^T: función de dispositivo *ldlt_sparse_4_c*

Para resolver el sistema $L^T x = z$, se usará la misma matriz que almacena L, el artificio es asumir que está ordenada por filas y no por columnas, con esto la transposición de la matriz ya no será necesaria. L^T es una matriz triangular superior, se resolverá el sistema obteniendo el valor de las variables una a continuación de otra desde abajo, recordar que los elementos de la diagonal de *L* son 1 aunque no estén almacenados, así, el valor de la variable en la posición *i* sería:

$$x_i = z_i - \sum_{j=i+1}^{n-1} z_j L_{i,j}$$

Se usa el mismo vector *z* para almacenar la solución desde abajo. La función de dispositivo propuesta que resuelve el sistema $L^T x = z$ se muestra en la Figura 48, cada llamada a esta función calculará una variable. Ya que el sistema se resuelve desde abajo, para un paso *j* se calculará el valor de la variable *i* tal que *i=n-j-1*, la fila de *L* usada para calcular esta variable tiene el mismo índice (se obtiene en la línea 16). Esta función de dispositivo usará memoria local

para calcular la sumatoria $\sum_{j=i+1}^{n-1} z_j L_{i,j}$ por reducción en paralelo, el tamaño de esta memoria local será el mismo que el tamaño de un grupo de trabajo, por esta razón, en este caso solo se utilizará un grupo de trabajo al usar esta función (en la línea 32 se especifica que el grupo de trabajo utilizado es únicamente el con id cero).

```
1  void ldlt_sparse_5_c(__global double* elmL,
2          __global int* colL, __global int* rowL,
3          __global double* y, int order, int step,
4          __local double* partialSum)
5  {
6          //índice de bloque
7          int bx = get_group_id(0);
8
9          //índice global de hilo
10         int tx = get_local_id(0);
11
12         //número total de hilos
13         int ntx = get_local_size(0);
14
15         //índice de la fila de L implicada
16         int fL = order - step - 1;
17
18         //ubicación de elemento diagonal de la fila en elmL
19         int diagId = colL[fL];
20
21         //número de elementos en la fila
22         int nEl = (fL == order - 1 ? 1 : colL[fL + 1] - diagId);
23
24         //ubicación del índice del primer elemento fuera de la
25         //diagonal en rowL
26         int pos_rowL = diagId - fL;
27
28         //valor donde se guardará una suma parcial
29         double sum = 0.0;
30
31         //solo se usará un bloque
32         if (bx == 0) {
33                 for (int x = tx; x < nEl - 1; x += ntx) {
34                         int rowId = rowL[pos_rowL + x];
35                         sum += elmL[diagId + x + 1] * y[rowId];
36                 }
37                 partialSum[tx] = sum;
38
39                 //reducción en paralelo pare determinar la suma de
40                 //todos los elementos de partialSum
41                 for (int stride = ntx / 2; stride > 0; stride /= 2) {
42                         barrier(CLK_LOCAL_MEM_FENCE);
43                         if (tx < stride) {
44                                 partialSum[tx] += partialSum[tx + stride];
45                         }
46                 }
47                 if (tx == 0)
48                         y[fL] -= partialSum[0];
49         }
50 }
```

Figura 48. LDL^T: función de dispositivo *ldlt_sparse_5_c*

En la línea 33 a 36 se calcula una suma parcial almacenándola en la variable *sum* (declarada en la línea 29), luego este valor es almacenado en memoria

local (línea 37), los elementos de *partialsum* se suman en paralelo en el bloque *for* de las líneas 41 a 46, finalmente se calcula la variable correspondiente en la línea 48, esta tarea solo la realiza un hilo (en la línea 47 se especifica el id del hilo que realiza la operación). Esta función de dispositivo debe ser llamada *n* veces para calcular las *n* variables.

```
1   __kernel void
2   ldlt_sparse_c(__global double* elmA, __global int* colA,
3           __global int* rowA, __global double* elmL,
4           __global int* colL, __global int* rowL, int size_elmA,
5           int size_colA, __global double* b,
6           __local double* partialSum, int step, int sstep)
7   {
8           if (sstep == -1)
9                   sparse_fill_c(elmA, colA, rowA, elmL, colL,
10                          rowL, size_elmA);
11          //la iteración principal se realiza en el host
12
13          //se modificará una sola columna
14          else if (sstep == 0)
15                  ldlt_sparse_1_c(elmL, colL, size_colA, step);
16
17          //se modificará las columnas siguientes a la columna
18          //implicada
19          else if (sstep == 1)
20                  ldlt_sparse_2_c(elmL, colL, rowL, size_colA, step);
21
22          //hasta aquí la factorización está concluida, la parte
23          //triangular inferior de A contendrá la factorización
24          //LDLt es decir L y D
25          //desde aquí se procederá a solucionar el sistema
26          //L*D*transpose(L)x=b
27
28          //se resolverá el sistema Ly=b, y=D*transpose(L)*x
29          else if (sstep == 2)
30                  ldlt_sparse_3_c(elmL, colL, rowL, b, size_colA, step);
31
32          //se resolverá el sistema Dz=y, z=transpose(L)*x
33          else if (sstep == 3)
34                  ldlt_sparse_4_c(elmL, colL, b, size_colA);
35
36          //se resolverá el sistema transpose(L)*x=z
37          //"z" ya debe haberse obtenido usando la función anterior
38          else
39                  ldlt_sparse_5_c(elmL, colL, rowL, b, size_colA, step,
40                          partialSum);
41  }
```

Figura 49. LDL^T: Kernel para trabajar sobre una matriz en formato CSC

El kernel principal se muestra en la Figura 49, como puede observarse el kernel llama a las funciones de dispositivo mencionadas anteriormente, dependiendo del valor del argumento *sstep*, para un valor de -1 copiará los elementos de *elmA* en el lugar correspondiente de *elmL*, *los* valores 0 y 1 indican que nos

encontramos en el proceso de factorización y los valores 2, 3 y 4 que estamos en el proceso de solución de los sistemas triangulares.

El kernel es llamado 1 vez para *sstep=-1*, *n-1* veces para *sstep* igual a 0 y 1 (con *step* de 0 a *n-2*), *n-1* veces para *sstep=2* (con *step* de 0 a *n-2*), una vez para *sstep=3* y finalmente *n* veces para *sstep=4*(con *step* de 0 a *n-1*).

4.1.6.3.Gradientes conjugados

La ventaja de este método es que solo requiere de operaciones sencillas en cada iteración, casi todas estas operaciones puede hacerse haciendo uso de las funciones incorporadas en ArrayFire, la única operación que necesita una función de dispositivo es la que se encargará de multiplicar matriz por vector, con la matriz almacenada en formato CSC.

La matriz en su forma densa puede descomponerse en tres términos: una matriz estrictamente triangular inferior, una matriz estrictamente triangular superior y una matriz diagonal, entonces se cumple que:

$$L + D + U = A$$

Siendo L la matriz estrictamente triangular inferior con los elementos de la parte estrictamente triangular inferior de A, U la matriz estrictamente triangular superior con los elementos de la parte estrictamente triangular superior de A y D la matriz diagonal con los elementos de la diagonal de A.

Ya que se trata de una matriz simétrica entonces $U = L^T$ por lo que:

$$L + D + L^T = A$$

Multiplicando por un vector b:

$$Ab = Lb + Db + L^T b$$

La matriz en formato CSC almacena L y D, para L^T puede usarse la misma L asumiendo que está ordenada por filas como se hizo al resolver los sistemas triangulares de la factorización de Cholesky o LDL^T.

```
1  void sparse_mat_vec_mul1(__global double* elmA,
2          __global int* colA, __global int* rowA, __global double* b,
3          __global double* c, int order, __local double* partialSum)
4  {
5          //índice de bloque
6          int bx = get_group_id(0);
7
8          //índice local de hilo
9          int tx = get_local_id(0);
10
11         for (int row = bx; row < order; row += get_num_groups(0)) {
12
13                 //ubicación de elemento diagonal de la fila en elmL
14                 int dId = colA[row];
15
16                 //número de elementos en la fila
17                 int nEl = (row == order - 1 ? 1 : colA[row + 1] - dId);
18
19                 //ubicación del índice del primer elemento fuera de la
20                 //diagonal en rowL
21                 int baseId = dId - row;
22
23                 //valor donde se guardará una suma parcial
24                 double sum = 0.0;
25
26                 for (int x = tx; x < nEl - 1; x += get_local_size(0)) {
27                         int cId = rowA[baseId + x];
28                         sum += elmA[dId + x + 1] * b[cId];
29                 }
30                 partialSum[tx] = sum;
31
32                 //reducción en paralelo pare determinar la suma de
33                 //todos los elementos de partialSum
34                 for (int stride = get_local_size(0) / 2;
35                         stride > 0; stride /= 2) {
36                         barrier(CLK_LOCAL_MEM_FENCE);
37                         if (tx < stride) {
38                                 partialSum[tx] += partialSum[tx + stride];
39                         }
40                 }
41                 if (tx == 0)
42                         c[row] = partialSum[0];
43         }
44 }
```

Figura 50. Gradientes conjugados: función de dispositivo *sparse_mat_vec_mul1*

La función de dispositivo propuesta para obtener $L^T b$ se muestra en la Figura 50, *elmA*, *colA* y *rowA* son los componentes de la matriz en formato CSC, *b* es el vector por el que se desea multiplicar, *c* es el vector donde se almacenará el resultado, *order* es el orden de la matriz y *partialsum* es la memoria local que se utilizará para acumular sumas parciales, en el bloque *for* de las líneas 26 a 29 se calcula una suma parcial y en la línea 30 se almacena está suma parcial en memoria local, posteriormente se suman todos los elementos de la memoria local por reducción en paralelo (bloque *for* de las líneas 34 a 40), esta suma total viene a ser un elemento del vector resultado que es almacenada en la línea 42 por solo uno de los hilos.

```
1   void sparse_mat_vec_mul2(__global double* elmA,
2           __global int* colA, __global int* rowA, __global double* b,
3           __global double* c, int order)
4   {
5           //índice de grupo
6           int bx = get_group_id(0);
7
8           //índice local de hilo
9           int tx = get_local_id(0);
10
11          for (int col = bx; col < order; col += get_num_groups(0)) {
12                  //posición del elemento diagonal
13                  int dId = colA[col];
14
15                  //número de elementos para una columna
16                  int nEl = (col == order - 1 ? 1 : colA[col + 1] - dId);
17
18                  //posición en rowL del primer elemento fuera de la
19                  //diagonal (índice)
20                  int baseId = dId - col;
21
22                  //elemento en b, por el que se multiplica
23                  double mul = b[col];
24
25                  for (int x = tx; x < nEl - 1; x += get_local_size(0)) {
26                          int rId = rowA[baseId + x];
27                          double sum = mul * elmA[dId + x + 1];
28                          atom_add_double(&(c[rId]), sum);
29                  }
30                  barrier(CLK_GLOBAL_MEM_FENCE);
31          }
32  }
```

Figura 51. Gradientes conjugados: función de dispositivo *sparse_mat_vec_mul2*

La función de dispositivo propuesta para obtener Lb se muestra en la Figura 51, esta función tiene los mismos argumentos que la función anterior sin la memoria local, el resultado de la operación se acumula directamente en el vector resultado de la función anterior, cada grupo se encarga de una columna y cada hilo de este grupo acumulará en el vector resultado el producto del elemento en la columna correspondiente al hilo y el elemento correspondiente en *b*, ya que es posible que dos hilos intenten escribir simultáneamente en la misma ubicación, se hace uso de la función atómica personalizada *atom_add_double*, análoga a la función incorporada de OpenCL *atom_add*, adaptada para trabajar sobre valores del tipo *double*, está función de dispositivo se muestra en la Figura 52, esta fue escrita adaptando la función CUDA *atomicAdd* para tipos *double* para dispositivos con capacidad computacional menor a 6.0, presentado en la sección *Programming Guide* del CUDA *toolkit documentation* (NVIDIA Corporation).

```
double atom_add_double(__global double* address, double val) {
        __global long* address_as_ull =
                (__global long*)address;
        long old = *address_as_ull;
        long assumed;

        do {
                assumed = old;
                old = atom_cmpxchg(address_as_ull, assumed,
                        as_long(val + as_double(assumed)));
                // Note: uses integer comparison to avoid hang in case
                //of NaN (since NaN != NaN)
        } while (assumed != old);

        return as_double(old);
 }
```

Figura 52. Gradientes conjugados: función de dispositivo *atom_add_double*

Finalmente la función de dispositivo propuesta para obtener Db se muestra en la Figura 53, al igual que en la función anterior, el resultado se acumula directamente en el vector resultado anterior, quedando después de usar esta última función, la multiplicación final matriz por vector. Esta función es básica y directa ya que se trata de multiplicar cada elemento de *b* por el elemento diagonal correspondiente y acumular este valor en el vector resultado, no es necesario introducir como argumento el vector de índices de fila *rowA*.

```
void sparse_mat_vec_mul3(__global double* elmA,
        __global int* colA, __global double* b, __global double* c,
        int order)
{

        //índice global de hilo
        int tx = get_global_id(0);

        //número total de hilos
        int ntx = get_num_groups(0)*get_local_size(0);

        for (int x = tx; x < order; x += ntx) {
                //ubicación del elemento diagonal en elmL
                int diag = colA[x];
                c[x] += b[x] * elmA[diag];
        }
 }
```

Figura 53. Gradientes conjugados: función de dispositivo *sparse_mat_vec_mul3*

El kernel encargado de realizar la tarea completa de multiplicación se muestra en Figura 54, como puede observarse se hace uso de las funciones de dispositivo anteriores dependiendo del valor del argumento *step*, para el valor de *step=0* calcula $L^T b$, para *step=1* calcula Lb acumulando el resultado en el mismo vector resultado de la operación anterior, y finalmente para *step=2*

calcula Db acumulando de la misma manera el resultado en el vector resultado anterior.

```
__kernel void
sparse_mat_vec_mul(__global double* elmA, __global int* colA,
        __global int* rowA, __global double* b, __global double* c,
        __local double* partialSum, int order, int step)
{
        //multiplicación de matriz por vector

        //transpose(L)*b, los elementos de c serán reemplazados
        //c[i]=valor
        if (step == 0)
                sparse_mat_vec_mul1(elmA, colA, rowA, b, c, order,
                        partialSum);
        //L*b, los elementos de c se acumularán a partir de su
        //valor actual
        //c[i]+=valor o valores
        else if (step == 1)
                sparse_mat_vec_mul2(elmA, colA, rowA, b, c, order);

        //D*b, los elementos de c se acumularán a partir de su
        //valor actual
        //c[i]+=valor
        else if (step == 2)
                sparse_mat_vec_mul3(elmA, colA, b, c, order);
        else;
 }
```

Figura 54. Gradientes conjugados: Kernel para multiplicar matriz en formato CSC por vector

La función que se exporta es similar a la presentada para matrices densas (Figura 29 pág. 83), cambiando la parte donde se calcula la multiplicación matriz por vector, por la función personalizada que se escribe usando el kernel presentado en esta sección.

4.1.7. Kernels sobre matrices en formato SKS

Se presentan los kernels elaborados para trabajar sobre matrices en formato SKS. Se ha definido una estructura *sparse_sks_ms* para matrices en este formato cuya definición en Julia se muestra en la Figura 55, los campos son: *elements* que guardan todos los elementos distintos de cero e *index* que guarda índices que indican la ubicación en *elements* del primer elemento distinto de cero para cada columna de la matriz original. De la misma manera definimos la estructura *af_sparse_sks_ms* (Figura 56) análogo a *sparse_sks_ms* que guardará la matriz dispersa en formato SKS con los campos del tipo *AFArray*, que indica que son objetos con memoria en el dispositivo.

```
struct sparse_sks_ms{T,N}
    elements::Array{T,1}
    index::Array{N,1}
end
```

Figura 55. Estructura para una matriz en formato SKS

```
struct af_sparse_sks_ms{T,N}
    elements::AFArray{T,1}
    index::AFArray{N,1}
end
```

Figura 56. Estructura para una matriz en formato SKS con memoria en el dispositivo

La Figura 57 muestra el código en Julia para convertir una matriz simétrica en formato denso en una matriz en formato SKS, para poder usar las funciones exportadas todavía es necesario tener la memoria en el dispositivo, para esto, hacemos uso de la función incorporada de ArrayFire *AFArray*, que hace una copia en la memoria del dispositivo de un objeto del tipo *Array*, se ha sobrecargado la función para poder hacer una copia de un objeto del tipo *sparse_sks_ms* en la memoria del dispositivo, en otras palabras convertir del tipo *sparse_sks_ms* a *af_sparse_sks_ms*, esta función se muestra en la Figura 58.

```
function sparse_sks_ms(A::Array{T,2}) where {T<:Real}
    orden=size(A)[1];
    index=ones(Int32,orden);
    elements=Array{T,1}(UndefInitializer(),0);
    key=1;
    for j in 1:orden
        index[j]+=length(elements);
        skl=key;
        key=orden;
        while A[key,j]==0 && key>j && key>skl
            key-=1;
        end
        append!(elements,A[j:key,j])
    end
    sparse_sks_ms(elements,index)
end
```

Figura 57. Conversión de una matriz densa a una matriz en formato SKS

```
function AFArray(A::sparse_sks_ms{T,N}) where {T<:Real,N<:Real}
    Ael=AFArray(A.elements);
    AIdx=AFArray(A.index);
    af_sparse_sks_ms(Ael,AIdx)
end
```

Figura 58. Función para convertir un objeto del tipo *sparse_sks_ms* a *af_sparse_sks_ms*

4.1.7.1.Factorización de Cholesky

Para resolver un sistema de ecuaciones usando la factorización de Cholesky, son necesarias dos etapas, uno de la factorización misma y otro que consiste en resolver los sistemas triangulares generados después de la factorización.

El proceso es similar al usado sobre matrices densas, la ventaja de almacenar en este formato es que no es necesaria una factorización simbólica, cosa que si fue necesaria al almacenar en formato CSC. En otras palabras, la memoria necesaria para almacenar la factorización es la misma que utiliza la matriz almacenada en formato SKS, siempre y cuando se siga el procedimiento descrito en la sección 2.3.3. Formato de almacenamiento SKS pág. 50.

Sea *A* la matriz original en formato SKS y *L* la matriz que contendrá la factorización de Cholesky también en formato SKS, ya que no hay diferencia en la cantidad de memoria usada para la factorización y para la matriz misma en formato SKS, la factorización puede ser realizada en sitio, es decir, si realizamos la factorización en sitio, *A* contendrá la final en su campo *elements* los elementos de *L*.

Similar al proceso sobre matrices densas, la factorización fue dividida en tres partes, un paso en el que modificamos una columna *j* tal que:

$$A_{n,j} = \frac{A_{n,j}}{\sqrt{A_{j,j}}}, n > j$$

```
1  void chol_sparse_1_sks(__global double* elmL,
2          __global int* idxL, int order, int step)
3  {
4          //es procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //obteniendo la ubicación del primer elemento
11                 //de la columna determinada por step
12                 int pos = idxL[step];
13
14                 //posición siguiente
15                 int pos_nxt = idxL[step + 1];
16
17                 //índice global de hilo
18                 int tx = get_global_id(0);
19
20                 //número total de hilos
21                 int ntx = get_num_groups(0)*get_local_size(0);
22
23                 //elemento diagonal. El primer
24                 //elemento guardado en elmA para cada columna es el
25                 //elemento diagonal
26                 double _r = sqrt(elmL[pos]);
27
28                 //cada hilo se encargará de modificar un elemento
29                 //de la columna
30                 for (int x = tx + pos + 1; x < pos_nxt; x += ntx) {
31                         elmL[x] /= _r;
32                 }
33         }
34 }
```

Figura 59. Cholesky: función de dispositivo *chol_sparse_1_sks*

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 59, *elmL* e *idxL* son los campos que representan a la matriz en formato SKS, cada hilo se encargará de realizar la modificación de uno de los elementos de la columna, en la línea 12 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna (la columna es determinada con el argumento *step*), de manera similar en la línea 15 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna siguiente, los elementos en *elmL* con índices entre estos dos valores son los que se tienen que modificar. En las líneas 18 y 21 se obtienen el índice global de hilo y el número total de hilos respectivamente, en la línea 26 se obtiene la raíz cuadrada del elemento diagonal y finalmente en la línea 31 se modifican los elementos de la columna.

Otro paso consiste en modificar las columnas siguientes a la columna *j* modificada en el paso anterior tal que:

$$A_{n,s} = A_{n,s} - A_{s,j}A_{n,j}, s > j, n \geq s$$

```
1  void chol_sparse_2_sks(__global double* elmL,
2          __global int* idxL, int order, int step)
3  {
4          //este procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //índice local de grupo
11                 int bx = get_group_id(0);
12
13                 //índice local de hilo
14                 int tx = get_local_id(0);
15
16                 //posición del primer elemento de la columna en elmL
17                 int pos_elmL = idxL[step];
18
19                 //número de elementos en elmL para la columna
20                 //sin contar la diagonal
21                 int nEl = idxL[step + 1] - pos_elmL - 1;
22
23                 if (nEl > 0) {
24                         for (int b = bx; b < nEl; b += get_num_groups(0)) {
25                                 //índice de la columna a modificar
26                                 int colId = step + b + 1;
27
28                                 //primer factor
29                                 double k1 = elmL[pos_elmL + b + 1];
30
31                                 for (int x = b + tx; x < nEl;
32                                         x += get_local_size(0)) {
33
34                                 //segundo factor
35                                 double k2 = elmL[pos_elmL + x + 1];
36
37                                 //posición final en elmL
38                                 int     posf = idxL[colId] + x - b;
39
40                                 elmL[posf] -= k1 * k2;
41                                 }
42                         }
43                 }
44         }
45 }
```

Figura 60. Cholesky: función de dispositivo *chol_sparse_2_sks*

La función de dispositivo propuesta que realiza esta tarea se muestra en la Figura 60, en este caso, cada grupo y sus hilos se encargarán de modificar una columna. En las líneas 11 y 14 se obtiene el índice de grupo y el índice local de hilo respectivamente, la línea 17 obtiene la ubicación en *elmL* del primer elemento de la columna (determinada por *step*), en la línea 21 se calcula el número de elementos en *elmL* para la columna descontando el elemento diagonal, el bloque *if* de las líneas 23 a 43 índica que el bloque *for* anidado

(líneas 24 a 42) solo se utiliza cuando el número de elementos para la columna fuera de la diagonal es por lo menos uno.

El bloque *for* de las líneas 24 a 42 realizará las modificaciones de todas las columnas siguientes a la columna actual (argumento *step*).

Estos dos primeros pasos deben realizarse uno después de otro *n-1* veces.

Un tercer paso es necesario para terminar la factorización, este modifica los elementos de la diagonal tal que:

$$A_{j,j} = \overline{\overline{A_{j,j}}}$$

La función de dispositivo propuesta que realiza esta tarea se muestra en la Figura 61, todos los hilos de todos los grupos se encargan cada uno de modificar un elemento de la diagonal, este paso se realiza una sola vez luego de realizada las *n-1* iteraciones con los dos pasos vistos previamente.

Terminada la factorización se procede a solucionar los sistemas equivalentes, es decir si $Ax = b$ es el sistema original, luego de la factorización quedará:

$$LL^T x = b$$

```
1  void chol_sparse_3_sks(__global double* elmL,
2          __global int* idxL, int order)
3  {
4
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         for (int x = tx; x < order; x += ntx) {
12                 int diag = idxL[x];
13                 elmL[diag] = sqrt(elmL[diag]);
14         }
15 }
```

Figura 61. Cholesky: función de dispositivo *chol_sparse_3_sks*

Haciendo $L^T x = y$ entonces $Ly = b$. Primero se resuelve $Ly = b$ y finalmente $L^T x = y$, cuya solución en x es la solución del sistema $Ax = b$.

Para resolver el sistema $Ly = b$, usaremos la eliminación típica de Gauss Jordan aplicada a una matriz triangular inferior, en cada paso los elementos de

cada columna fuera de la diagonal van volviéndose cero progresivamente, hasta llegar a una matriz diagonal, sin embargo, al resolver un sistema triangular, las operaciones sobre L pueden hacerse implícitamente y dejarla intacta para poder usarla al resolver el siguiente sistema $L^T x = y$, con esto solo sería necesario modificar el vector de constantes b tal que para el paso j:

$$b_n = b_n - b_j \frac{L_{n,j}}{L_{j,j}}, n > j$$

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 62, el número de iteración y por la tanto el índice de la columna actual está dada por el valor del argumento *step*, esta función debe ser llamada *n-1* veces para terminar con la eliminación, luego será necesario dividir los elementos de *b* entre los elementos de la diagonal (función de dispositivo de la Figura 63), este vector será la solución del sistema $Ly = b$, que será usado como argumento al resolver el siguiente sistema triangular.

```
1  void chol_sparse_4_sks(__global double* elmL,
2          __global int* idxL, __global double* b, int order,
3          int step)
4  {
5          if (step < order - 1) {
6                  //índice global de hilo
7                  int tx = get_global_id(0);
8
9                  //número total de hilos
10                 int ntx = get_num_groups(0)*get_local_size(0);
11
12                 //posición del elemento diagonal
13                 int diag = idxL[step];
14
15                 //número de elementos para una columna
16                 int nEl = idxL[step + 1] - diag;
17
18                 //deben haber elementos aparte del elemento diagonal
19                 if (nEl > 1) {
20
21                         //elemento diagonal y elemento correspondiente en b
22                         double k = b[step] / elmL[diag];
23
24                         for (int x = tx; x < nEl - 1; x += ntx) {
25                                 int rowId = step + x + 1;
26                                 b[rowId] -= k * elmL[diag + x + 1];
27                         }
28                 }
29         }
30 }
```

Figura 62. Cholesky: función de dispositivo *chol_sparse_4_sks*

```
1  void chol_sparse_5_sks(__global double* elmL,
2          __global int* idxL, __global double* b, int order)
3  {
4
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         for (int x = tx; x < order; x += ntx) {
12                 //ubicación del elemento diagonal en elmL
13                 int diag = idxL[x];
14                 b[x] /= elmL[diag];
15         }
16  }
```

Figura 63. Cholesky: función de dispositivo *chol_sparse_5_sks*

Para resolver el sistema $L^T x = y$, se usará la misma matriz que almacena L asumiendo que está ordenada por filas y que se trata de una matriz triangular superior, con esto ya no sería necesaria la transposición explícita de la matriz. Se resolverá el sistema obteniendo el valor de las variables una a continuación de otra desde abajo, el valor de la variable en la posición *i* sería:

$$x_i = \frac{y_i - \sum_{j=i+1}^{n-1} x_j L_{i,j}}{L_{i,i}}$$

Se usa el mismo vector *y* para almacenar la solución. La función de dispositivo propuesta que resuelve el sistema $L^T x = y$ se muestra en la Figura 64, cada llamada a esta función calculará una variable. Ya que el sistema se resuelve desde abajo, para un paso *j* se calculará el valor de la variable *i* tal que *i=n-j-1*, la fila de *L* usada para calcular esta variable tiene el mismo índice (se obtiene en la línea 15). Esta función de dispositivo usará memoria local para calcular la sumatoria $\sum_{j=i+1}^{n-1} x_j L_{i,j}$ por reducción en paralelo, el tamaño de esta memoria local será el mismo que el tamaño de un grupo de trabajo, por esta razón, en este caso solo se utilizará un grupo de trabajo al usar esta función (en la línea 30 se especifica que el grupo de trabajo utilizado es únicamente el con id cero).

En la línea 31 a 34 se calcula una suma parcial almacenándola en la variable *sum* (declarada en la línea 27), luego este valor es almacenado en memoria local (línea 35), los elementos de *partialsum* se suman en paralelo en el bloque *for* de las líneas 39 a 44, finalmente se calcula la variable correspondiente en la línea 46, esta tarea solo la realiza un hilo (en la línea 45 se especifica el id

del hilo que realiza la operación). Esta función de dispositivo debe ser llamada *n* veces para calcular las *n* variables.

```
1   void chol_sparse_6_sks(__global double* elmL,
2           __global int* idxL, __global double* y,
3           int order, int step, __local double* partialSum)
4   {
5           //índice de bloque
6           int bx = get_group_id(0);
7
8           //índice global de hilo
9           int tx = get_local_id(0);
10
11          //número total de hilos
12          int ntx = get_local_size(0);
13
14          //índice de la fila de L implicada
15          int fL = order - step - 1;
16
17          //ubicación del elemento diagonal de la fila en elmL
18          int diagId = idxL[fL];
19
20          //número de elementos en la fila
21          int nEl = (fL == order - 1 ? 1 : idxL[fL + 1] - diagId);
22
23          //elemento diagonal
24          double diag = elmL[diagId];
25
26          //valor donde se guardará una suma parcial
27          double sum = 0.0;
28
29          //solo se usará un bloque
30          if (bx == 0) {
31                  for (int x = tx; x < nEl - 1; x += ntx) {
32                          int rowId = fL + x + 1;
33                          sum += elmL[diagId + x + 1] * y[rowId];
34                  }
35                  partialSum[tx] = sum;
36
37                  //reducción en paralelo pare determinar la suma de
38                  //todos los elementos de partialSum
39                  for (int stride = ntx / 2; stride > 0; stride /= 2) {
40                          barrier(CLK_LOCAL_MEM_FENCE);
41                          if (tx < stride) {
42                                  partialSum[tx] += partialSum[tx + stride];
43                          }
44                  }
45                  if (tx == 0)
46                          y[fL] = (y[fL] - partialSum[0]) / diag;
47          }
48   }
```

Figura 64. Cholesky: función de dispositivo *chol_sparse_6_sks*

El kernel principal se muestra en la Figura 65, como puede observarse el kernel llama a las funciones de dispositivo mencionadas anteriormente, dependiendo del valor del argumento *sstep*, los valores 0, 1 y 2 indican que nos encontramos en el proceso de factorización y los valores 3, 4 y 5 que estamos en el proceso de solución de los sistemas triangulares.

```
__kernel void
chol_sparse_sks(__global double* elmL,
        __global int* idxL, int size_elmL, int size_idxL,
        __global double* b, __local double* partialSum, int step,
        int sstep)
{
        //la iteración principal se realiza en el host

        //se modificará una sola columna
        if (sstep == 0)
                chol_sparse_1_sks(elmL, idxL, size_idxL, step);

        //se modificará las columnas siguientes a la columna
        //implicada
        else if (sstep == 1)
                chol_sparse_2_sks(elmL, idxL, size_idxL, step);

        //una vez usada las funciones anteriores todavía es
        //necesario modificar la diagonal D[i]=sqrt(D[i]) para
        //concluir la factorización
        else if (sstep == 2)
                chol_sparse_3_sks(elmL, idxL, size_idxL);

        //hasta aquí la factorización está concluida, la parte
        //triangular inferior de A contendrá la factorización
        //de Cholesky
        //desde aquí se procederá a solucionar el sistema
        //L*transpose(L)x=b

        //se resolverá el sistema Ly=b, y=transpose(L)*x
        else if (sstep == 3)
                chol_sparse_4_sks(elmL, idxL, b, size_idxL,
                        step);
        else if (sstep == 4)
                chol_sparse_5_sks(elmL, idxL, b, size_idxL);

        //se resolverá el sistema transpose(L)*x=y
        //"y" ya debe haberse obtenido usando la función anterior
        else
                chol_sparse_6_sks(elmL, idxL, b, size_idxL,
                        step, partialSum);
 }
```

Figura 65. Cholesky: Kernel para trabajar sobre una matriz en formato SKS

El kernel es llamado *n-1* veces para *sstep* igual a 0 y 1 (con *step* de 0 a *n-2*), solo una vez para *sstep=2*, *n-1* veces para *sstep=3* (con *step* de 0 a *n-2*), una vez para *sstep=4* y finalmente *n* veces para *sstep=5*(con *step* de 0 a *n-1*).

4.1.7.2.Factorización LDL^T

El proceso para resolver un sistema usando la factorización LDL^T es similar al que utilizamos en la factorización de Cholesky.

Para resolver un sistema de ecuaciones usando la factorización LDL^T, son necesarias dos etapas, uno de la factorización misma y otro que consiste en resolver los sistemas triangulares generados después de la factorización.

El proceso es similar al usado sobre matrices densas, la ventaja de usar este formato de almacenamiento es que ya no es necesaria una factorización simbólica, cosa que si fue necesaria al almacenar en formato CSC, si para almacenar la matriz en formato SKS se usó el procedimiento descrito en la sección 2.3.3. (Formato de almacenamiento SKS pág. 50), el tamaño de memoria usado para el campo *elements* de la matriz, tiene el mismo tamaño que la memoria necesaria para guardar el campo *elements* de su factorización.

Sea *A* la matriz original en formato SKS y *L* la matriz que contendrá la factorización LDL^T también en formato SKS, como se dijo, el campo *elements* de la matriz tiene el mismo tamaño que el campo *elements* de su factorización, esto nos permite realizar una factorización en sitio sobre el mismo campo *elements* de *A*, si hacemos la factorización en sitio, el campo *elements* de A contendrá al final los elementos correspondientes del campo *elements* de *L*, es decir *A* se convertirá en *L*.

```
1  void ldlt_sparse_1_sks(__global double* elmL,
2          __global int* idxL, int order, int step)
3  {
4          //es procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //obteniendo la ubicación del primer elemento
11                 //de la columna determinada por step
12                 int pos = idxL[step];
13
14                 //posición siguiente
15                 int pos_nxt = idxL[step + 1];
16
17                 //índice global de hilo
18                 int tx = get_global_id(0);
19
20                 //número total de hilos
21                 int ntx = get_num_groups(0)*get_local_size(0);
22
23                 //elemento diagonal. El primer
24                 //elemento guardado en elmA para cada columna es el
25                 //elemento diagonal
26                 double _r = elmL[pos];
27
28                 //cada hilo se encargará de modificar un elemento
29                 //de la columna
30                 for (int x = tx + pos + 1; x < pos_nxt; x += ntx) {
31                         elmL[x] /= _r;
32                 }
33         }
34 }
```

Figura 66. LDL^T: función de dispositivo *ldlt_sparse_1_sks*

El proceso de factorización fue dividido en dos partes, un paso en el que modificamos una columna *j* tal que:

$$A_{n,j} = \frac{A_{n,j}}{A_{j,j}}, n > j$$

```
1  void ldlt_sparse_2_sks(__global double* elmL,
2          __global int* idxL, int order, int step)
3  {
4          //este procedimiento solo deberá realizarse hasta
5          //la columna order-2
6          //Siendo order el orden de la matriz, la última
7          //columna tendrá índice order-1, donde solo contiene
8          //el elemento diagonal
9          if (step < order - 1) {
10                 //índice local de grupo
11                 int bx = get_group_id(0);
12
13                 //índice local de hilo
14                 int tx = get_local_id(0);
15
16                 //posición del primer elemento de la columna de elmL
17                 int pos_elmL = idxL[step];
18
19                 //elemento diagonal
20                 double diag = elmL[pos_elmL];
21
22                 //número de elementos en elmL para la columna
23                 //sin contar la diagonal
24                 int nEl = idxL[step + 1] - pos_elmL - 1;
25
26                 if (nEl > 0) {
27                         for (int b = bx; b < nEl; b += get_num_groups(0)) {
28                                 //índice de la columna a modificar
29                                 int colId = step + b + 1;
30
31                                 //primer factor
32                                 double k1 = elmL[pos_elmL + b + 1];
33
34                                 for (int x = b + tx; x < nEl;
35                                         x += get_local_size(0)) {
36
37                                         //segundo factor
38                                         double k2 = elmL[pos_elmL + x + 1];
39
40                                         //posición final en elmL
41                                         int     posf = idxL[colId] + x - b;
42
43                                         elmL[posf] -= diag * k1 * k2;
44                                 }
45                         }
46                 }
47         }
48 }
```

Figura 67. LDL^T: función de dispositivo *ldlt_sparse_2_sks*

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 66, cada hilo se encargará de modificar un elemento de la columna, en la línea 12 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna (la columna es determinada con el argumento *step*), de manera similar

en la línea 15 se obtiene el índice de ubicación en *elmL* del primer elemento de la columna siguiente, los elementos en *elmL* con índices entre estos dos valores son los que se tienen que modificar. En las líneas 18 y 21 se obtienen el índice global de hilo y el número total de hilos respectivamente, en la línea 26 se obtiene el elemento diagonal y finalmente en la línea 31 se calcula y se escribe el valor final en *elmL*.

Otro paso consiste en modificar las columnas siguientes a la columna *j* modificada en el paso anterior tal que:

$$A_{n,s} = A_{n,s} - A_{s,j}A_{n,j}A_{j,j}, s > j, n \geq s$$

La función de dispositivo propuesta que realiza esta tarea se muestra en la Figura 67, en este caso, cada grupo y sus hilos se encargarán de modificar una columna. En las líneas 11 y 14 se obtiene el índice de grupo y el índice local de hilo respectivamente, la línea 17 obtiene la ubicación en *elmL* del primer elemento de la columna actual (determinada por *step*), en la línea 20 se obtiene el elemento diagonal, en la línea 24 se calcula el número de elementos en *elmL* fuera de la diagonal que le corresponden a la columna, el bloque *if* de las líneas 26 a 46 índica que el bloque *for* anidado (líneas 27 a 45) solo se utiliza cuando el valor calculado en la línea 24 es por lo menos uno.

El bloque *for* de las líneas 27 a 45 realizará las modificaciones de todas las columnas siguientes a la columna actual (argumento *step*).

Estos dos pasos deben realizarse uno después de otro *n-1* veces para terminar con la factorización. Terminada la factorización, el campo *elements* de *L* contendrá los elementos de la factorización LDL^T, este incluye los elementos de la diagonal *D* almacenados como si fueran la diagonal de *L*, debe recordarse que *L* por si sola tiene elementos diagonales iguales a 1, esto es implícito y debe tenerse en cuenta al momento de resolver el sistema.

Terminada la factorización se procede a solucionar los sistemas equivalentes, es decir si $Ax = b$ es el sistema original, luego de la factorización quedará:

$$LDL^T x = b$$

Haciendo $DL^T x = y$ entonces $Ly = b$, es el primer sistema a resolverse para y ,luego haciendo $L^T x = z$ entonces $Dz = y$ es el segundo sistema a resolver en z, que es simple ya que D es una matriz diagonal, finalmente se resuelve $L^T x = z$, cuya solución en x es la solución del sistema $Ax = b$.

Para resolver el sistema $Ly = b$, usaremos la eliminación típica de Gauss Jordan aplicada a una matriz triangular inferior, en cada paso, los elementos de cada columna fuera de la diagonal van volviéndose cero progresivamente, hasta llegar a una matriz diagonal. Al resolver un sistema triangular las operaciones sobre L pueden hacer implícitamente dejándola intacta para resolver los siguientes sistemas, con esto, solo sería necesario modificar el vector de constantes b tal que para el paso j:

$$b_n = b_n - b_j L_{n,j}, n > j$$

```
1   void ldlt_sparse_3_sks(__global double* elmL,
2           __global int* idxL, __global double* b, int order,
3           int step)
4   {
5           if (step < order - 1) {
6                   //índice global de hilo
7                   int tx = get_global_id(0);
8
9                   //número total de hilos
10                  int ntx = get_num_groups(0)*get_local_size(0);
11
12                  //posición del elemento diagonal
13                  int diag = idxL[step];
14
15                  //número de elementos para una columna
16                  int nEl = idxL[step + 1] - diag;
17
18                  //deben haber elementos aparte del elemento diagonal
19                  if (nEl > 1) {
20
21                          //elemento diagonal y elemento correspondiente en b
22                          double diag_elm = elmL[diag];
23                          double elmb = b[step];
24
25                          for (int x = tx; x < nEl - 1; x += ntx) {
26                                  int rowId = step + x + 1;
27                                  b[rowId] -= elmb * elmL[diag + x + 1];
28                          }
29                  }
30          }
31  }
```

Figura 68. LDL^T: función de dispositivo *ldlt_sparse_3_sks*

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 68, el índice de la columna actual es el mismo determinado por el argumento *step*, esta función debe ser llamada *n-1* veces para terminar con la eliminación, al final *b* contendrá la solución del sistema $Ly = b$, que será usado como argumento al resolver el siguiente sistema.

Luego se procede a resolver el sistema $Dz = y$, ya que D es diagonal, la solución se reduce a realizar la operación:

$$b_j = \frac{b_j}{D_{j,j}}$$

La función de dispositivo propuesta que realiza esta operación se muestra en la Figura 69, esta función se ejecuta una sola vez.

```
1  void ldlt_sparse_4_sks(__global double* elmL,
2          __global int* idxL, __global double* b, int order)
3  {
4
5          //índice global de hilo
6          int tx = get_global_id(0);
7
8          //número total de hilos
9          int ntx = get_num_groups(0)*get_local_size(0);
10
11         for (int x = tx; x < order; x += ntx) {
12                 //ubicación del elemento diagonal en elmL
13                 int diag = idxL[x];
14                 b[x] /= elmL[diag];
15         }
16 }
```

Figura 69. LDL^T: función de dispositivo *ldlt_sparse_4_sks*

Para resolver el sistema $L^T x = z$, se usará la misma matriz que almacena L, asumiendo que está ordenada por filas y que se trata de una matriz triangular superior, con esto, no será necesaria la transposición explícita de la matriz, se resolverá el sistema obteniendo el valor de las variables una a continuación de otra desde abajo, recordar que los elementos de la diagonal de *L* son 1 aunque no estén almacenados, así, el valor de la variable en la posición *i* sería:

$$x_i = z_i - \sum_{j=i+1}^{n-1} z_j L_{i,j}$$

Se usa el mismo vector *z* para almacenar la solución. La función de dispositivo propuesta que resuelve el sistema $L^T x = z$ se muestra en la Figura 70, cada llamada a esta función calculará una variable. Ya que el sistema se resuelve desde abajo, para un paso *j* se calculará el valor de la variable *i* tal que *i=n-j-1*, la fila de *L* usada para calcular esta variable tiene el mismo índice (se obtiene en la línea 15). Esta función de dispositivo usará memoria local para calcular la sumatoria $\sum_{j=i+1}^{n-1} z_j L_{i,j}$ por reducción en paralelo, el tamaño de esta memoria local será el mismo que el tamaño de un grupo de trabajo, por esta razón, en este caso solo se utilizará un grupo de trabajo al usar esta función (en la línea 27 se especifica que el grupo de trabajo utilizado es únicamente el con id cero).

```
1  void ldlt_sparse_5_sks(__global double* elmL,
2          __global int* idxL, __global double* y,
3          int order, int step, __local double* partialSum)
4  {
5          //índice de bloque
6          int bx = get_group_id(0);
7
8          //índice global de hilo
9          int tx = get_local_id(0);
10
11         //número total de hilos
12         int ntx = get_local_size(0);
13
14         //índice de la fila de L implicada
15         int fL = order - step - 1;
16
17         //ubicación del elemento diagonal de la fila en elmL
18         int diagId = idxL[fL];
19
20         //número de elementos en la fila
21         int nEl = (fL == order - 1 ? 1 : idxL[fL + 1] - diagId);
22
23         //valor donde se guardará una suma parcial
24         double sum = 0.0;
25
26         //solo se usará un bloque
27         if (bx == 0) {
28                 for (int x = tx; x < nEl - 1; x += ntx) {
29                         int rowId = fL + x + 1;
30                         sum += elmL[diagId + x + 1] * y[rowId];
31                 }
32                 partialSum[tx] = sum;
33
34                 //reducción en paralelo pare determinar la suma de
35                 //todos los elementos de partialSum
36                 for (int stride = ntx / 2; stride > 0; stride /= 2) {
37                         barrier(CLK_LOCAL_MEM_FENCE);
38                         if (tx < stride) {
39                                 partialSum[tx] += partialSum[tx + stride];
40                         }
41                 }
42                 if (tx == 0)
43                         y[fL] -= partialSum[0];
44         }
45 }
```

Figura 70. LDL^T: función de dispositivo *ldlt_sparse_5_sks*

En la línea 28 a 31 se calcula una suma parcial almacenándola en la variable *sum* (declarada en la línea 24), luego este valor es almacenado en memoria local (línea 32), los elementos de *partialsum* se suman en paralelo en el bloque *for* de las líneas 36 a 41, finalmente se escribe el valor final de la variable correspondiente en la línea 43, esta tarea solo la realiza un hilo (en la línea 42 se especifica el id del hilo que realiza la operación). Esta función de dispositivo debe ser llamada *n* veces para calcular las *n* variables.

```
1  __kernel void
2  ldlt_sparse_sks(__global double* elmL,
3          __global int* idxL, int size_elmL, int size_idxL,
4          __global double* b, __local double* partialSum, int step,
5          int sstep)
6  {
7          //la iteración principal se realiza en el host
8
9          //se modificará una sola columna
10         if (sstep == 0)
11                 ldlt_sparse_1_sks(elmL, idxL, size_idxL, step);
12
13         //se modificará las columnas siguientes a la columna
14         //implicada
15         else if (sstep == 1)
16                 ldlt_sparse_2_sks(elmL, idxL, size_idxL, step);
17
18         //hasta aquí la factorización está concluida, la parte
19         //triangular inferior de A contendrá la factorización
20         //LDLt es decir L y D
21         //desde aquí se procederá a solucionar el sistema
22         //L*D*transpose(L)x=b
23
24         //se resolverá el sistema Ly=b, y=D*transpose(L)*x
25         else if (sstep == 2)
26                 ldlt_sparse_3_sks(elmL, idxL, b, size_idxL,
27                         step);
28
29         //se resolverá el sistema Dz=y, z=transpose(L)*x
30         else if (sstep == 3)
31                 ldlt_sparse_4_sks(elmL, idxL, b, size_idxL);
32
33         //se resolverá el sistema transpose(L)*x=z
34         //"z" ya debe haberse obtenido usando la función anterior
35         else
36                 ldlt_sparse_5_sks(elmL, idxL, b, size_idxL,
37                         step, partialSum);
38 }
```

Figura 71. LDL^T: kernel para trabajar sobre una matriz en formato SKS

El kernel principal se muestra en la Figura 71, como puede observarse el kernel llama a las funciones de dispositivo mencionadas anteriormente, dependiendo del valor del argumento *sstep*, los valores 0 y 1 indican que nos encontramos en el proceso de factorización y los valores 2, 3 y 4 que estamos en el proceso de solución de los sistemas triangulares.

El kernel es llamado *n-1* veces para *sstep* igual a 0 y 1 (con *step* de 0 a *n-2*), *n-1* veces para *sstep=2* (con *step* de 0 a *n-2*), una vez para *sstep=3* y finalmente *n* veces para *sstep=4*(con *step* de 0 a *n-1*).

4.1.7.3.Gradientes conjugados

La ventaja de este método es que solo requiere de operaciones sencillas en cada iteración, casi todas estas operaciones puede hacerse haciendo uso de las funciones incorporadas en ArrayFire, la única operación que necesita una función de dispositivo es la que se encargará de multiplicar matriz por vector, con la matriz almacenada en formato SKS.

La matriz en su forma densa puede descomponerse en dos términos: una matriz estrictamente triangular inferior y una matriz triangular superior, entonces se cumplirá:

$$L + U = A$$

Siendo L la matriz estrictamente triangular inferior con los elementos de la parte estrictamente triangular inferior de A y U la matriz triangular superior con los elementos de la parte triangular superior de A, tomar en cuenta que ya que se trata de una matriz simétrica entonces la parte estrictamente superior de U contiene la transpuesta de L.

Multiplicando por un vector b:

$$Ab = Lb + Ub$$

La matriz en formato SKS almacena L y la diagonal de A, es decir, la transpuesta de U. Para multiplicar por U puede usarse la misma matriz A en formato SKS, asumiendo que está ordenada por filas y que se trata de una matriz triangular superior, así, la trasposición explícita de la matriz ya no es necesaria.

```
1   void sparse_mat_vec_mul1_sks(__global double* elmA,
2           __global int* idxA, __global double* b,
3           __global double* c, int order,
4           __local double* partialSum)
5   {
6           //índice de bloque
7           int bx = get_group_id(0);
8
9           //índice local de hilo
10          int tx = get_local_id(0);
11
12          for (int row = bx; row < order; row += get_num_groups(0)) {
13
14                  //ubicación del elemento diagonal de la fila en elmA
15                  int dId = idxA[row];
16
17                  //número de elementos en la fila
18                  int nEl = (row == order - 1 ? 1 : idxA[row + 1] - dId);
19
20                  //valor donde se guardará una suma parcial
21                  double sum = 0.0;
22
23                  for (int x = tx; x < nEl; x += get_local_size(0)) {
24                          int cId = row + x;
25                          sum += elmA[dId + x] * b[cId];
26                  }
27                  partialSum[tx] = sum;
28
29                  //reducción en paralelo pare determinar la suma de
30                  //todos los elementos de partialSum
31                  for (int stride = get_local_size(0) / 2;
32                          stride > 0; stride /= 2) {
33                          barrier(CLK_LOCAL_MEM_FENCE);
34                          if (tx < stride) {
35                                  partialSum[tx] += partialSum[tx + stride];
36                          }
37                  }
38                  if (tx == 0)
39                          c[row] = partialSum[0];
40          }
41  }
```

Figura 72. Gradientes conjugados: función de dispositivo *sparse_mat_vec_mul1_sks*

La función de dispositivo propuesta para obtener Ub se muestra en la Figura 72, *elmA* e *idxA* son los componentes de la matriz en formato SKS, *b* es el vector por el que se desea multiplicar, *c* es el vector donde se almacenará el resultado, *order* es el orden de la matriz y *partialsum* es la memoria local que se utilizará para acumular sumas parciales, en el bloque *for* de las líneas 23 a 26 se calcula una suma parcial y en la línea 27 se almacena esta suma parcial en memoria local, posteriormente se suman todos los elementos de esta por reducción en paralelo (bloque *for* de las líneas 31 a 37), esta suma total viene a ser un elemento del vector resultado que es almacenada en la línea 39 por solo uno de los hilos.

```
1  void sparse_mat_vec_mul2_sks(__global double* elmA,
2          __global int* idxA, __global double* b,
3          __global double* c, int order)
4  {
5          //índice de grupo
6          int bx = get_group_id(0);
7
8          //índice local de hilo
9          int tx = get_local_id(0);
10
11         for (int col = bx; col < order; col += get_num_groups(0)) {
12                 //posición del elemento diagonal
13                 int dId = idxA[col];
14
15                 //número de elementos para una columna
16                 int nEl = (col == order - 1 ? 1 : idxA[col + 1] - dId);
17
18                 //elemento en b, por el que se multiplica
19                 double mul = b[col];
20
21                 for (int x = tx; x < nEl - 1; x += get_local_size(0)) {
22                         int rId = col + x + 1;
23                         double sum = mul * elmA[dId + x + 1];
24                         atom_add_double(&(c[rId]), sum);
25                 }
26                 barrier(CLK_GLOBAL_MEM_FENCE);
27         }
28  }
```

Figura 73. Gradientes conjugados: función de dispositivo *sparse_mat_vec_mul2_sks*

La función de dispositivo propuesta para obtener Lb se muestra en la Figura 73, esta función tiene los mismos argumentos que la función anterior sin la memoria local, el resultado de la operación se acumula directamente en el vector resultado de la función anterior, cada grupo se encarga de una columna y cada hilo de este grupo acumulará en el vector resultado el producto del elemento en la columna correspondiente al hilo y el elemento correspondiente en *b*, ya que es posible que dos hilos intenten escribir simultáneamente en la misma ubicación, se hace uso de la función atómica personalizada *atom_add_double*, análoga a la función incorporada de OpenCL *atom_add*, adaptada para trabajar sobre valores del tipo *double*, está función de dispositivo se muestra en la Figura 74, esta fue escrita adaptando la función CUDA *atomicAdd* para tipos *double* para dispositivos con capacidad computacional menor a 6.0, presentado en la sección *Programming Guide* del CUDA toolkit documentation (NVIDIA Corporation).

```
double atom_add_double(__global double* address, double val) {
        __global long* address_as_ull =
                (__global long*)address;
        long old = *address_as_ull;
        long assumed;

        do {
                assumed = old;
                old = atom_cmpxchg(address_as_ull, assumed,
                        as_long(val + as_double(assumed)));
                // Note: uses integer comparison to avoid hang in case
                //of NaN (since NaN != NaN)
        } while (assumed != old);

        return as_double(old);
 }
```

Figura 74. Gradientes conjugados: función de dispositivo *atom_add_double*

El kernel encargado de realizar la tarea completa de multiplicación se muestra en Figura 75, como puede observarse se hace uso de las funciones de dispositivo anteriores dependiendo del valor del argumento *step*, para el valor de *step=0* calcula Ub y para *step=1* calcula Lb acumulando el resultado en el mismo vector resultado de la operación anterior.

```
__kernel void
sparse_mat_vec_mul_sks(__global double* elmA,
        __global int* idxA, __global double* b,
        __global double* c, __local double* partialSum,
        int order, int step)
{
        //multiplicación de matriz por vector

        //U*b, los elementos de c serán reemplazados
        //c[i]=valor
        if (step == 0)
                sparse_mat_vec_mul1_sks(elmA, idxA, b, c,
                        order, partialSum);
        //L*b, los elementos de c se acumularán a partir de su
        //valor actual
        //c[i]+=valor o valores
        else if (step == 1)
                sparse_mat_vec_mul2_sks(elmA, idxA, b, c,
                        order);
        else;
 }
```

Figura 75. Gradientes conjugados: kernel para multiplicar matriz en formato SKS por vector

La función que se exporta es similar a la presentada para matrices densas (Figura 29 pág. 83), cambiando la parte donde se calcula la multiplicación matriz por vector, por la función personalizada que usa el kernel presentado en esta sección.

4.1.8. Medición del tiempo y memoria utilizada

En general, el tiempo de ejecución de un kernel consiste de tres componentes: tiempo de preparación del kernel, tiempo de cómputo y tiempo de sincronización, el cuál es representado como:

$$T = \sum_{i=1}^{M} (t_O^{(i)} + t_C^{(i)} + t_S^{(i)})$$

Donde M es el número de lanzamientos del kernel, $t_O^{(i)}$ es el tiempo de preparación del kernel, $t_C^{(i)}$ es el tiempo de cómputo sobre una GPU y $t_S^{(i)}$ es el tiempo de sincronización para el i-ésimo lanzamiento del kernel, cada uno de estos componentes es influenciado por algunos factores, por ejemplo, el tiempo de preparación del kernel depende de la tasa de transferencia de datos del procesador central al dispositivo así como del tamaño del código y los parámetros del kernel. Para el tiempo de cómputo, este es afectado por los métodos de accesos a memoria, la organización de los hilos (número de hilos por grupo y número de grupos) en el kernel, etc. (Shucai Xiao, 2009).

El tiempo de lanzamiento del kernel es muy pequeño en comparación a los otros dos (Shucai Xiao, 2009), los resultados de tiempo de ejecución en el presente trabajo, incluyen estos tres componentes.

Respecto a la memoria utilizada, en matrices densas, ya que no se utiliza ningún formato de almacenamiento que ignore los ceros y considerando que en los métodos con factorización se realizan dichas factorizaciones en sitio, entonces, la cantidad de memoria en MB (Mega bytes) utilizada estará dada por (recordar que hacemos uso de precisión doble):

$$M_{MB} = \frac{8n(n+1)}{1x10^6}$$

Siendo *n* el orden de la matriz, la expresión anterior considera la matriz de rigidez (matriz de coeficientes del sistema) y el vector de fuerzas (vector de constantes).

Para matrices almacenadas en formato CSC se suma la memoria utilizada por los tres componentes de la matriz en este formato y la memoria utilizada por el vector de fuerzas, considerando que los vectores que almacenan los índices lo hacen como enteros de 32 bits (4 bytes), la memoria total utilizada sería:

$$M_{MB} = \frac{4(3l_e + 2n)}{1x10^6}$$

Siendo l_e el tamaño de los campos *elements* de la matriz en formato CSC y *n* es el orden de la matriz (o el tamaño del vector de fuerzas). También se está considerando que la matriz en formato CSC original es sustituida por la matriz *L* (también en formato CSC), en los métodos que usan factorización.

Finalmente, Para matrices almacenadas en formato SKS se suma la memoria utilizada por los dos componentes de la matriz en este formato y la memoria utilizada por el vector de fuerzas, considerando que el vector que almacenan los índices lo hace como enteros de 32 bits (4 bytes), la memoria total utilizada sería:

$$M_{MB} = \frac{4(2l_e + 3n)}{1x10^6}$$

Siendo l_e el tamaño del campo *elements* de la matriz en formato SKS y *n* el orden de la matriz (o el tamaño del vector de fuerzas). También se está considerando una factorización en sitio para los métodos que usan factorización.

4.1.9. Características del dispositivo

Las pruebas fueron realizadas en una **GeForce GT 630M,** cuyas características principales son**:** número de multiprocesadores 2, número de núcleos CUDA 96, con capacidad de cálculo (compute capability) 2.1, cantidad total de memoria global 2048 MB, cantidad máxima de hilos (o work ítems) en un grupo de trabajo 1024, cantidad total de memoria local por grupo de trabajo 48 KB. En general, al usarse otros dispositivos, el rendimiento mejorará según aumenten estos valores.

4.1.10. Validación de resultados

El vector de desplazamientos (solución del sistema de ecuaciones lineales) arrojado por las funciones, se validaron comparándolos con el vector de

desplazamientos arrojados por el solver incorporado en OpenFEM, el error máximo encontrado para la diferencia entre elementos correspondientes no excedió de $1x10^{-9}$, y de todo el vector (medido mediante la obtención de la norma de la diferencia de vectores) no excedió de $1x10^{-5}$.

4.1.11. Modelo 1: Viga en voladizo

Se ha comenzado analizando una viga simple de concreto en voladizo de 210 Kg/cm^2, con una sección de 0.30 x 0.50 m, longitud de 3.00 m y sometida a una carga distribuida de 2 tn/m (19613.3 N/m). Analizando la viga como un sólido, se hace el mallado correspondiente del elemento para analizarla usando el método de los elementos finitos, en la Figura 76 se muestra el mallado de la viga, se ha hecho el mallado dividiendo la sección y el largo de la viga en partes variables con tal de obtener una cantidad de elementos más densa, el gráfico es generado usando el complemento **Medit** de OpenFEM.

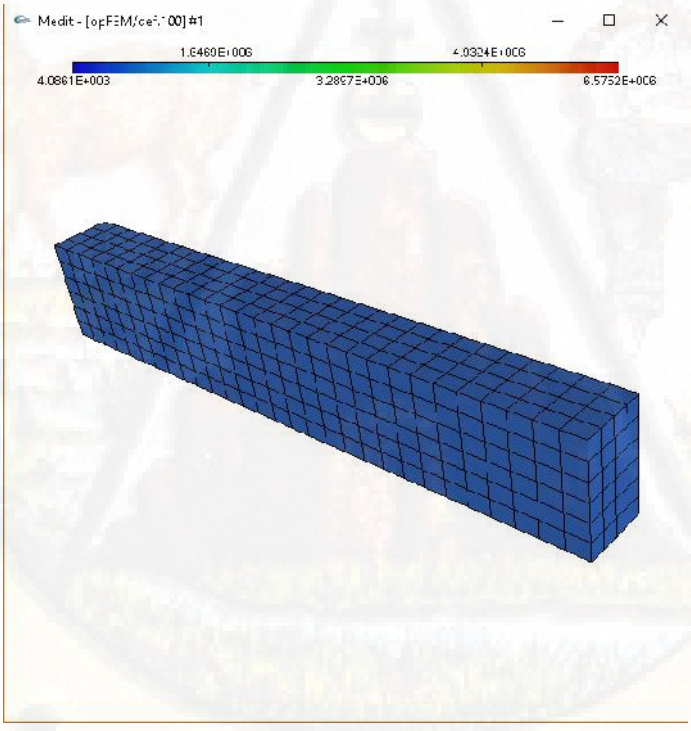


Figura 76. malla de elementos finitos: viga en voladizo

Para este modelo se han generado sistemas de ecuaciones con matrices que van del orden de 1080 a 10044, se ha considerado la matriz tal y como la devuelve la

herramienta de elementos finitos (OpenFEM), es decir sin ningún pivote previo, para analizar el comportamiento de las funciones y formatos de almacenamiento.

4.1.11.1. Memoria utilizada

Los resultados de memoria utilizada para los técnicas consideradas se muestran en la Tabla 10, puede notarse que para el formato denso (sin comprimir), la memoria es la misma para los cuatro métodos (resultado trivial), sin embargo, para el formato CSC existen diferencias entre la memoria utilizada por los métodos con factorización y los métodos sin factorización (método de los gradientes conjugados), algo que se esperaba, ya que los métodos con factorización requieren mayor cantidad de memoria que la original para almacenar su factorización, es por eso, que antes de realizar la factorización en este formato de almacenamiento, es necesaria una factorización simbólica previa. Lo anterior no pasa al almacenar en formato SKS ya que el almacenamiento en este formato se hace de tal manera que la cantidad de memoria necesaria para almacenar la matriz original es suficiente también para almacenar su factorización, es una ventaja del formato SKS sobre el CSC.

Tabla 10. Modelo 1: memoria utilizada en MB

Memoria utilizada (MB)										
orden	Denso				CSC			SKS		
	Gauss Jordan	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados
1080	9.34	9.34	9.34	9.34	3.36	3.36	**0.35**	2.72	2.72	2.72
2016	32.53	32.53	32.53	32.53	10.15	10.15	**0.72**	8.33	8.33	8.33
3045	74.20	74.20	74.20	74.20	18.74	18.74	**1.15**	13.88	13.88	13.88
4032	130.09	130.09	130.09	130.09	28.47	28.47	**1.59**	20.67	20.67	20.67
5040	203.25	203.25	203.25	203.25	39.72	39.72	**2.03**	29.82	29.82	29.82
6048	292.67	292.67	292.67	292.67	59.23	59.23	**2.46**	45.70	45.70	45.70
7161	410.30	410.30	410.30	410.30	82.28	82.28	**2.96**	63.74	63.74	63.74
8184	535.89	535.89	535.89	535.89	91.57	91.57	**3.42**	65.02	65.02	65.02
9216	679.55	679.55	679.55	679.55	115.93	115.93	**3.88**	82.74	82.74	82.74
10044	807.14	807.14	807.14	807.14	126.58	126.58	**4.27**	97.26	97.26	97.26

Un aspecto importante a resaltar es la diferencia considerable en el uso de memoria entre el formato CSC sin factorizar y el mismo formato con

factorización, en estas condiciones, para este modelo estructural, podemos apreciar que el método de los gradientes conjugados aplicada sobre la matriz en formato CSC es la que utiliza la memoria más eficientemente.

Esta diferencia puede notarse de forma más evidente en el siguiente gráfico donde *CSC_fact* es la memoria que utiliza los métodos con factorización (Cholesky y LDL^T). La memoria que utiliza el formato SKS y el formato CSC con factorización no se diferencian considerablemente.

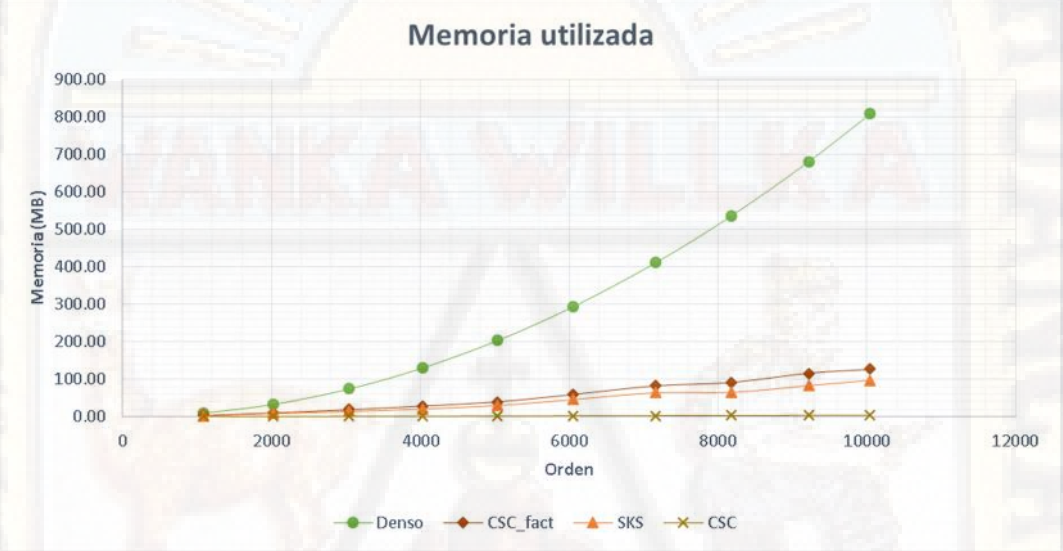


Gráfico 1: Modelo 1: memoria utilizada en MB

4.1.11.2. Tiempo de ejecución

La Tabla 11 muestra el tiempo de ejecución para los diferentes formatos de almacenamiento y métodos de solución. Puede observarse que ambos métodos de factorización aplicadas a la matriz en formato SKS arrojan los mejores resultados, los tiempos para los demás métodos y formatos de almacenamiento se alejan considerablemente de estos resultados.

El Gráfico 2 muestra los tiempos de ejecución en segundos para todas las técnicas, el Gráfico 3, muestra los tiempos de ejecución para las mejores cinco técnicas. Puede notarse las diferencias significativas de los resultados en tiempo de ejecución de los métodos con factorización sobre la matriz en

formato SKS con los demás tiempos arrojados, algo que se notó también analizando el otro modelo estructural.

Tabla 11. Modelo 1: tiempo de ejecución en segundos

Tiempo de ejecución (s)										
	Denso				CSC			SKS		
orden	Gauss Jordan	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados
1080	0.61	0.39	0.88	0.36	1.90	1.89	3.72	**0.30**	**0.25**	4.77
2016	2.70	1.53	1.83	1.47	8.51	8.49	6.57	**0.71**	**0.66**	10.32
3045	9.32	4.33	4.33	4.25	16.94	17.04	11.05	**1.15**	**1.07**	18.80
4032	29.10	10.32	10.45	9.99	30.29	29.99	20.00	**1.72**	**1.62**	28.77
5040	42.67	17.48	17.48	11.89	45.87	45.77	24.89	**2.53**	**2.52**	44.58
6048	71.26	30.18	30.25	15.91	92.85	93.24	29.38	**4.46**	**4.30**	61.87
7161	134.42	55.91	56.01	25.90	159.43	160.01	34.12	**6.41**	**6.28**	83.83
8184	197.91	77.18	78.16	36.09	156.71	157.16	41.44	**6.34**	**6.15**	106.85
9216	290.66	115.33	115.46	50.10	238.10	238.93	49.01	**9.18**	**8.99**	123.64
10044	369.93	146.98	149.13	64.05	307.67	308.91	61.15	**10.00**	**9.33**	160.84

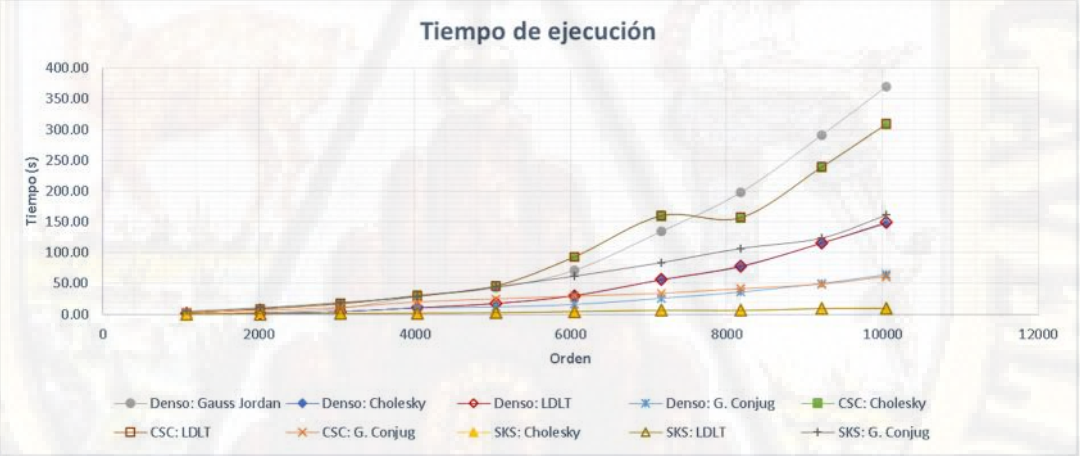


Gráfico 2: Modelo 1: tiempo de ejecución en segundos

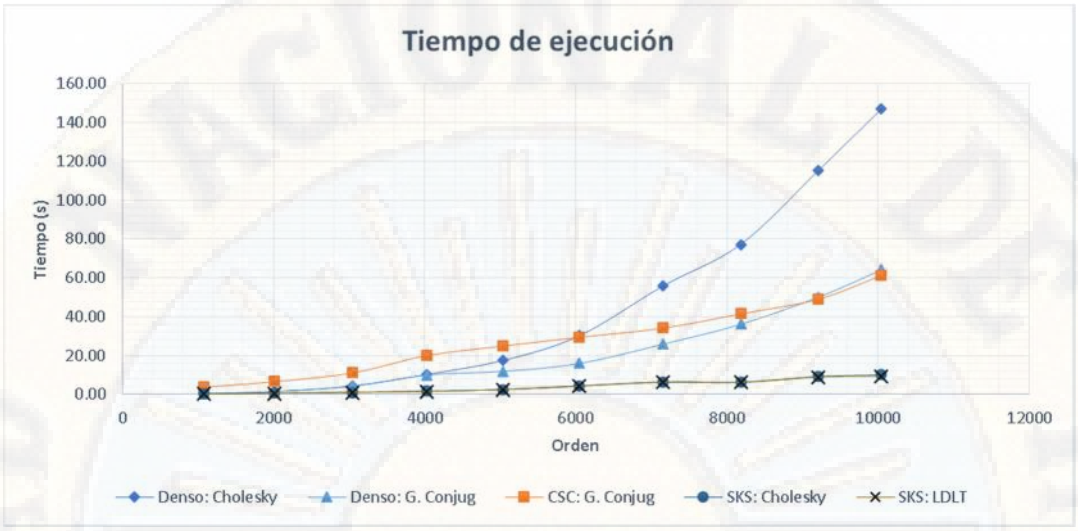


Gráfico 3: Modelo 1: tiempo de ejecución en segundos

4.1.12. Modelo 2: Pilar de puente

Se ha modelado pilar de puente con columnas circulares, viga de sección variable y zapatas cuadradas (Ver Figura 77), las columnas tienen un diámetro de 1 m y altura de 10 m, la viga tiene un ancho de 1.04 m y un peralte variable de 1 m en el centro de la luz y final de las alas, y de 1.5 m en los extremos; la distancia entre ejes de columna es de 10.04 m, la longitud total de la viga es de 15.08 m, el material es de concreto de 280 Kg/cm^2 y la carga distribuida sobre la viga de 50 tn/m (490332.5 N/m), se hizo el mallado correspondiente del elemento para analizarla usando el método de los elementos finitos, en la Figura 77 se muestra el mallado de la estructura, se ha hecho el mallado dividiendo los elementos en partes cada vez más grandes con tal de obtener una cantidad de nodos más grande, el gráfico es generado usando el complemento **Medit** de OpenFEM.

Para este modelo se han generado sistemas de ecuaciones con matrices que van del orden de 1572 hasta 10449 para todos los métodos y hasta 205602 para pruebas adicionales con los más sobresalientes, se ha considerado la matriz con pivote previo de filas y columnas, usando el reordenamiento de Cuthill–McKee, con esto buscamos concentrar elementos distintos de cero más cerca de la diagonal y por tanto mejorar el rendimiento y optimizar aún más el uso de memoria.

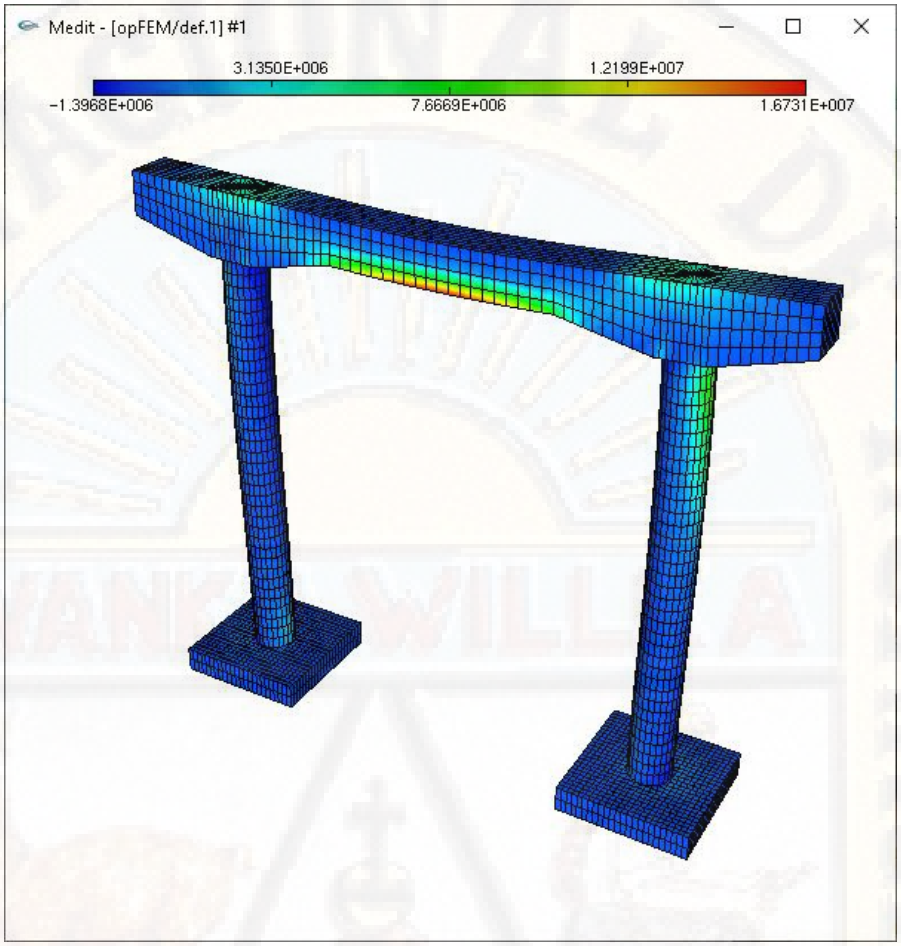


Figura 77. Malla de elementos finitos: soporte central de puente

4.1.12.1. Memoria utilizada

Los resultados de memoria utilizada para todos los métodos considerados se muestran en la Tabla 12, puede notarse que para el formato denso (sin comprimir), la memoria es la misma para los cuatro métodos, sin embargo, como ya se vio en los resultados de memoria para el modelo anterior, para el formato CSC existen diferencias entre la memoria utilizada por los métodos con factorización y los métodos sin factorización (método de los gradientes conjugados), algo que se esperaba, ya que los métodos con factorización requieren mayor cantidad de memoria que la original para almacenar la matriz factorizada, es por eso, que antes de realizar la factorización en este formato de almacenamiento, es necesaria una factorización simbólica previa. Lo anterior no pasa al almacenar en formato SKS ya que el almacenamiento en este formato

se hace de tal manera que la cantidad de memoria necesaria para almacenar la matriz original es suficiente también para almacenar su factorización, es una ventaja del formato SKS sobre el CSC.

Tabla 12. Modelo 2: memoria utilizada en MB

Memoria utilizada (MB)										
	Denso				CSC			SKS		
Orden	Gauss Jordan	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados
1572	19.78	19.78	19.78	19.78	1.19	1.19	**0.33**	1.43	1.43	1.43
3534	99.94	99.94	99.94	99.94	3.21	3.21	**1.01**	3.26	3.26	3.26
5409	234.10	234.10	234.10	234.10	7.54	7.54	**1.81**	7.80	7.80	7.80
8331	555.31	555.31	555.31	555.31	13.29	13.29	**2.51**	12.94	12.94	12.94
10449	873.54	873.54	873.54	873.54	24.55	24.55	**3.51**	24.89	24.89	24.89
15129	1831.21	1831.21	1831.21	1831.21	30.86	30.86	**5.16**	30.48	30.48	30.48
21987	3867.60	3867.60	3867.60	3867.60	67.99	67.99	**7.54**	65.13	65.13	65.13
28773	6623.31	6623.31	6623.31	6623.31	125.73	125.73	**10.44**	118.88	118.88	118.88
39051	12200.16	12200.16	12200.16	12200.16	246.77	246.77	**14.29**	226.55	226.55	226.55
51156	20935.90	20935.90	20935.90	20935.90	322.65	322.65	**19.78**	290.52	290.52	290.52

Un aspecto importante a resaltar es la diferencia considerable en el uso de memoria entre el formato CSC sin factorizar y el mismo formato factorizado, en estas condiciones, para este modelo estructural, podemos apreciar que el método de los gradientes conjugados aplicada sobre la matriz en formato CSC es la que utiliza la memoria más eficientemente, algo que se notó también en el modelo estructural anterior.

En el Gráfico 4 puede notarse la diferencia significativa que existe al usar la matriz densa (sin comprimir) y los otros formatos de almacenamiento, en este gráfico, aparentemente, no existe diferencia significativa entre los formatos CSC (con factorización o sin factorización) y SKS, sin embargo, al quitar la línea de la matriz densa pueden apreciarse las diferencias (ver Gráfico 5). La memoria que utiliza el formato SKS y el formato CSC con factorización no se diferencian considerablemente, como se vio también en el modelo estructural anterior.

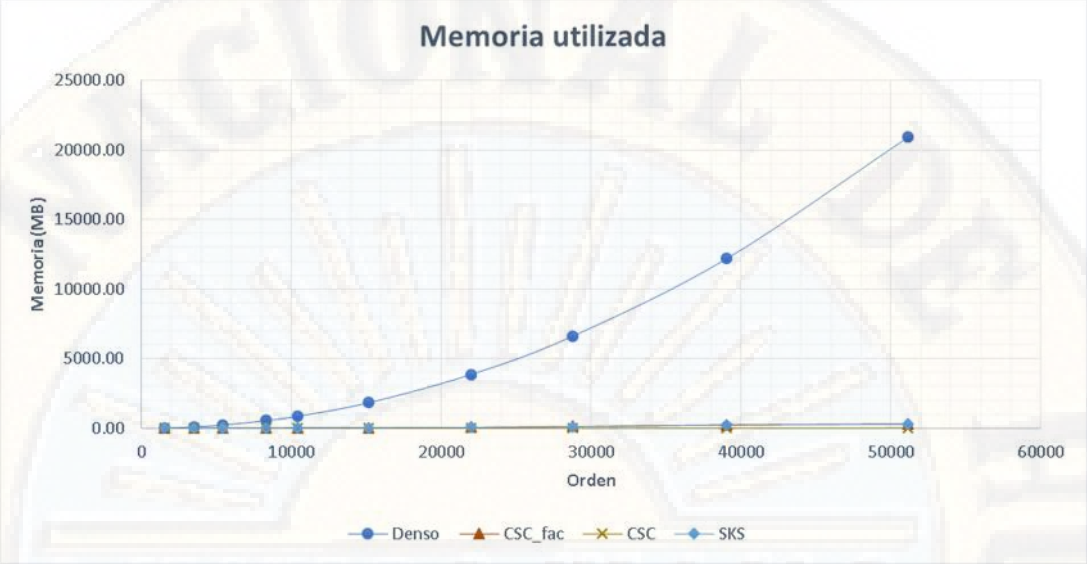


Gráfico 4: Modelo 2: memoria utilizada en MB

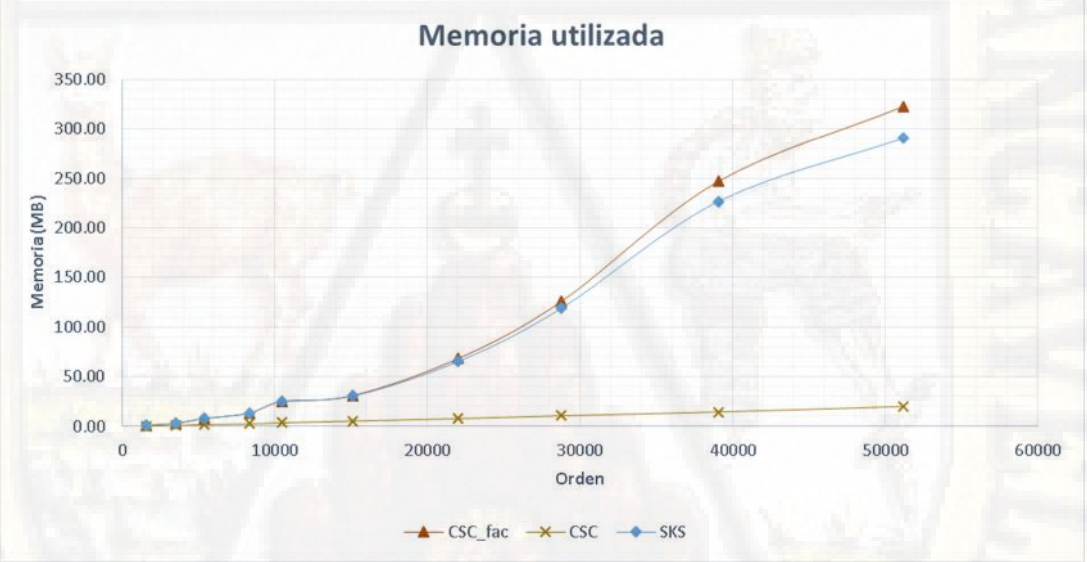


Gráfico 5: Modelo 2: memoria utilizada en MB

4.1.12.2. Tiempo de ejecución

La Tabla 13 muestra el tiempo de ejecución para los diferentes formatos de almacenamiento y métodos de solución. Puede observarse que ambos métodos de factorización aplicadas a la matriz en formato SKS arrojan los mejores resultados, los tiempos para los demás métodos y formatos de almacenamiento se alejan considerablemente de estos. Hay aspectos bastante importantes a

resaltar con el modelo anterior, y es que, aunque las técnicas que arrojan los mejores resultados se mantienen, para ordenes similares, el tiempo de ejecución para este modelo es mucho menor (10.00 s para el orden de 10044 para el modelo 1 y 2.61 s para el orden de 10449 para el modelo 2), esta diferencia es muy probablemente debido al reordenamiento previo que se hizo a la matriz para este modelo.

Otra observación importante es el deterioro del método de los gradientes conjugados con la matriz en formato CSC, en comparación a los resultados arrojados para el modelo estructural anterior y, por el contrario, la mejora significativa que tuvieron los métodos con factorización.

Tabla 13. Modelo 2: tiempo de ejecución en segundos

Tiempo de ejecución (s)										
orden	Denso				CSC			SKS		
	Gauss Jordan	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados	Cholesky	LDL^T	Gradientes conjugados
1572	1.47	0.88	0.81	4.13	0.47	0.42	13.68	**0.25**	**0.20**	14.45
3534	15.74	6.52	6.43	18.11	1.20	1.12	34.75	**0.52**	**0.43**	39.98
5409	57.01	21.64	21.56	38.77	3.28	3.14	61.69	**1.00**	**0.96**	73.68
8331	216.29	78.58	79.38	90.77	5.85	5.61	130.11	**1.61**	**1.35**	103.33
10449	300.88	125.99	127.66	139.14	14.61	14.44	281.13	**2.61**	**2.27**	175.44

Tabla 14. Modelo 2: tiempo de ejecución en segundos

Tiempo de ejecución (s)				
Orden	CSC		SKS	
	Cholesky	LDL^T	Cholesky	LDL^T
1572	0.47	0.42	**0.25**	**0.20**
3534	1.20	1.12	**0.52**	**0.43**
5409	3.28	3.14	**1.00**	**0.96**
8331	5.85	5.61	**1.61**	**1.35**
10449	14.61	14.44	**2.61**	**2.27**
15129	17.37	16.90	**3.39**	**2.94**
21987	56.19	55.72	**6.69**	**6.06**
28773	176.26	176.27	**12.47**	**11.73**

La razón por la que solo se probaron todas las técnicas hasta el orden de 10449 en el caso de los métodos sobre la matriz densa, es el límite de memoria del dispositivo (Ver sección 4.1.9. Características del dispositivo pág. 131), los resultados de memoria de la Tabla 12 (pág. 138) fueron calculadas manualmente, observar que el límite de dispositivo se alcanza apenas en el orden 15129, y que para el mayor orden en ésta tabla se necesitaría un dispositivo de más de 20 000 MB (20 GB) si se quisiera trabajar con la matriz sin comprimir.

Descartando las técnicas con los resultados más desfavorables, se hicieron pruebas adicionales (hasta el orden de 28773), para los cuatro técnicas con mejores tiempos, la Tabla 14 y el Gráfico 6 muestras los resultados. Puede observarse la similitud entre los tiempos que arrojan los métodos con factorización para un mismo formato de almacenamiento.

Gráfico 6: Modelo 2: tiempo de ejecución en segundos

4.1.13. Balance entre memoria y tiempo de ejecución

Para el formato CSC, la memoria utilizada es mucho menor al usar el método de los gradientes conjugados (ver Tabla 10 y Tabla 12, pág. 133 y 138 respectivamente) ya que la matriz original en formato CSC no requiere ningún tipo de factorización, sin embargo, la contraparte es que al usar el método de los gradientes conjugados con la matriz en formato CSC los tiempos de ejecución

resultantes son mucho mayores (ver Tabla 11 y Tabla 13, pág. 135 y 140 respectivamente).

Moviéndonos hacia los métodos que usan, en segundo lugar, más eficientemente la memoria, llegamos a los métodos que usan la matriz en formato SKS, podemos observar que para los métodos con factorización (Cholesky y LDL^T) en este formato, los tiempos son bastante mejores, no es difícil concluir entonces, que estos métodos en éste formato de almacenamiento, tienen el mejor balance de memoria y tiempo de ejecución. De aquí en adelante, para el modelo 2, se harán pruebas adicionales usando este formato de almacenamiento y estos métodos, ya que para órdenes más grandes, la tendencia en tiempo de ejecución para el formato CSC es obtener valores mucho más grandes, superior a los 170 segundos, lo cual es exagerado (ver Tabla 14, pág. 140).

Tabla 15. Modelo 2: memoria utilizada en MB para el formato SKS

Memoria utilizada (MB)				
Orden	**Longitud del campo *elements***	**Memoria utilizada M_u**	**Memoria en formato denso M_d**	**M_u / M_d**
1572	176955	1.43	19.78	0.073
3534	402647	3.26	99.94	0.033
5409	966955	7.80	234.10	0.033
8331	1605480	12.94	555.31	0.023
10449	3096149	24.89	873.54	0.028
15129	3787050	30.48	1831.21	0.017
21987	8107764	65.13	3867.60	0.017
28773	14816912	118.88	6623.31	0.018
39051	28260478	226.55	12200.16	0.019
51156	36238341	290.52	20935.90	0.014
64836	59114778	473.70	33630.17	0.014
72570	89764866	718.99	42131.82	0.017
89733	108759760	871.15	64416.81	0.014
102630	121760423	975.31	84264.16	0.012
117390	124626462	998.42	110244.24	0.009
142422	185245727	1483.67	162273.35	0.009

Se han realizado pruebas adicionales sobre órdenes de la matriz de rigidez más grandes, llegando al orden 142422, afinando más la malla de elementos finitos. Se ha utilizado el formato de almacenamiento SKS y los métodos de la factorización de Cholesky y LDL^T. La memoria utilizada por ambos métodos es la misma, los

resultados se muestran en la Tabla 15, donde puede observarse que a medida que el orden aumenta, el índice de dispersión (relación M_u/M_d) disminuye, para el orden máximo se alcanzó un valor de 0.009 lo que significa que la memoria utilizada es de un 0.9% de la memoria que se utilizaría con la matriz en formato denso (ver Gráfico 7).

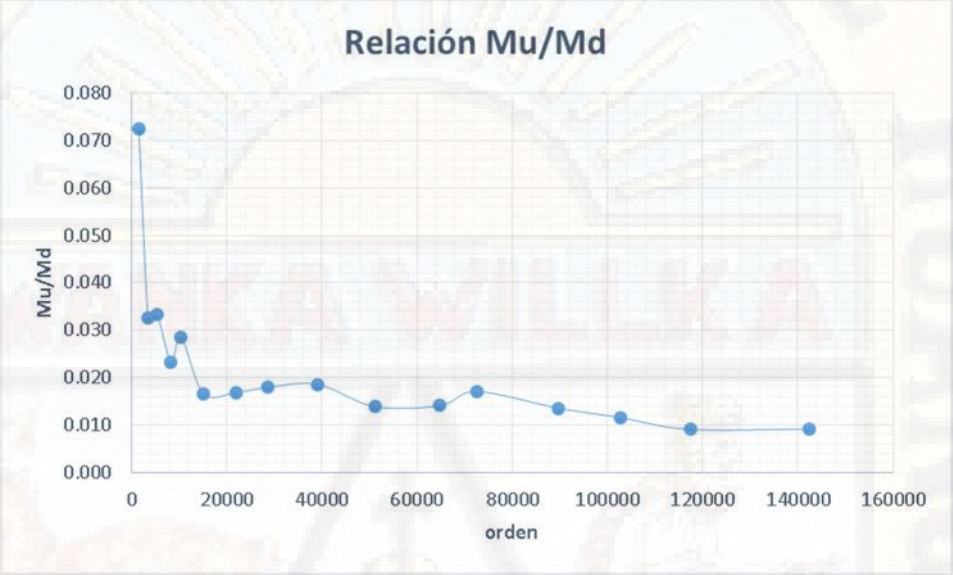


Gráfico 7: índice de dispersión para el formato SKS

Tabla 16. Modelo 2: tiempo de ejecución en matrices en formato SKS

Tiempo de ejecución (s)		
orden	**Cholesky**	**LDL^T**
1572	0.25	0.20
3534	0.52	0.43
5409	1.00	0.96
8331	1.61	1.35
10449	2.61	2.27
15129	3.39	2.94
21987	6.69	6.06
28773	12.47	11.73
39051	24.28	23.30
51156	29.57	28.47
64836	52.98	50.48
72570	94.96	93.66
89733	121.05	119.09
102630	132.26	129.67
117390	135.51	132.64
142422	225.44	223.96

Los resultados de tiempo de ejecución se muestran en la Tabla 16 y Gráfico 8, puede observarse que los tiempos arrojados por ambos métodos son muy parecidos, el máximo tiempo obtenido para el orden 142422 es de 225.44 segundos para la factorización de Cholesky y de 223.96 segundos para la factorización LDL^T.

Gráfico 8: tiempo de ejecución en matrices en formato SKS

4.1.14. Pruebas adicionales

4.1.14.1. GT 630M vs GTX 1070

Para mostrar el impacto que tiene la utilización de los mismas técnicas al resolver el mismo sistema de ecuaciones lineales pero en otro dispositivo (otra GPU), se han realizado pruebas adicionales en otro dispositivo con mejores características, las características de ambos dispositivos (arrojados por la aplicación ocldevicequery.exe de OpenCL) se muestran en la Tabla 17, la GT 630M es el dispositivo que se usó en esta investigación; en esta tabla puede observarse las ventajas que tiene la GTX 1070 sobre la GT 630M, la diferencia más notable está en el número de núcleos CUDA para ambos dispositivos. Estas diferencias se reflejaron también significativamente al realizar las

pruebas, usando las mismas funciones (y por lo tanto los mismos kernels) y resolviendo los mismos sistemas de ecuaciones lineales para el modelo 2 (pilar de puente), estos resultados se muestran en la Tabla 18 y el Gráfico 9.

Tabla 17. Diferencias en las características de los dispositivos

	GT 630M	**GTX 1070**
Número de multiprocesadores	2	16
Número de núcleos CUDA	96	4294967280
Capacidad de cálculo (compute capability)	2.1	6.1
Cantidad total de memoria global	2048 MB	8192 MB
Cantidad máxima de hilos (o work ítems) en un grupo de trabajo	1024	1024
Cantidad total de memoria local por grupo de trabajo (work group)	48 KB	48 KB

Tabla 18. Modelo 2: tiempo de ejecución en diferentes dispositivos

Tiempo de ejecución (s), para matrices en formato SKS usando diferentes dispositivos				
	GT 630M		**GTX 1070**	
orden	**Cholesky**	**LDL^T**	**Cholesky**	**LDL^T**
1572	0.25	0.20	0.06	0.06
3534	0.52	0.43	0.12	0.12
5409	1.00	0.96	0.2	0.19
8331	1.61	1.35	0.29	0.28
10449	2.61	2.27	0.37	0.39
15129	3.39	2.94	0.52	0.51
21987	6.69	6.06	0.91	0.9
28773	12.47	11.73	1.5	1.48
39051	24.28	23.30	2.86	2.88
51156	29.57	28.47	3.41	3.4
64836	52.98	50.48	5.95	6.02
72570	94.96	93.66	10.92	11.11
89733	121.05	119.09	12.34	12.58
102630	132.26	129.67	13.34	13.57
117390	135.51	132.64	13.85	14.23
142422	225.44	223.96	23.11	23.86

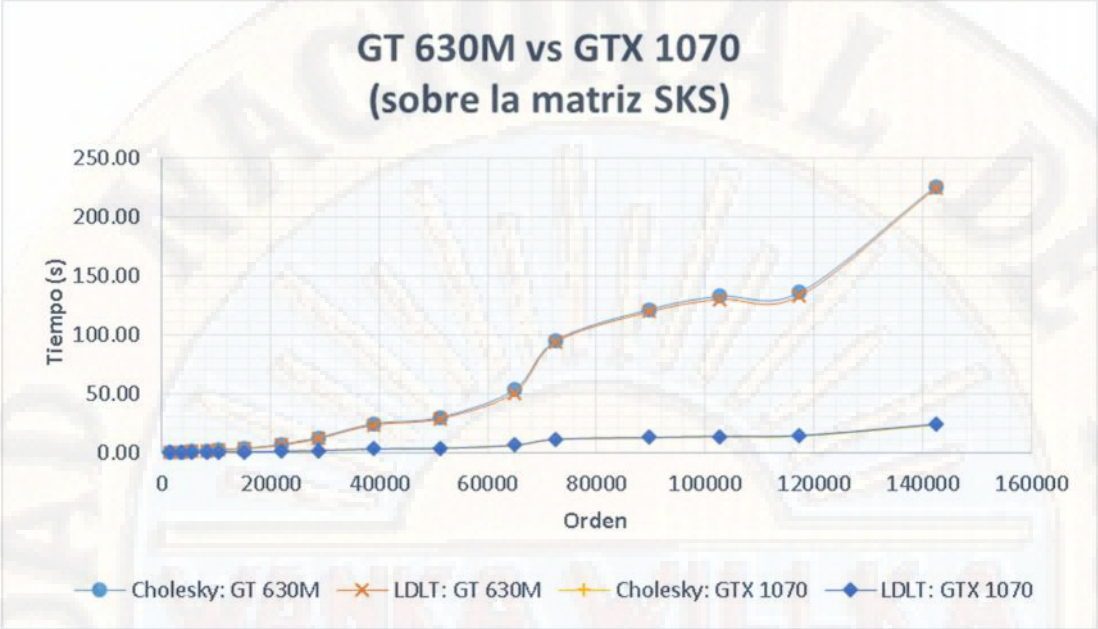


Gráfico 9: tiempo de ejecución en diferentes dispositivos

4.1.14.2. Comparación con el solver incorporado de OpenFEM

Como referencia, también se ha hecho la comparación con el solver incorporado de OpenFEM que usa la factorización LDL^T sobre la matriz en formato COO, los resultados no son tan alentadores para la GT 630M (ver Gráfico 10), sin embargo, a comparación con la GTX 1070 se observó una mejora significativa (ver Gráfico 11), para esta última prueba se llegó al orden 205609.

Estos últimos resultados son solo referenciales, ya que, incluso sabiendo el método y formato de almacenamiento que usa el solver de OpenFEM, se desconoce el tipo de dispositivo que usa para resolver el sistema.

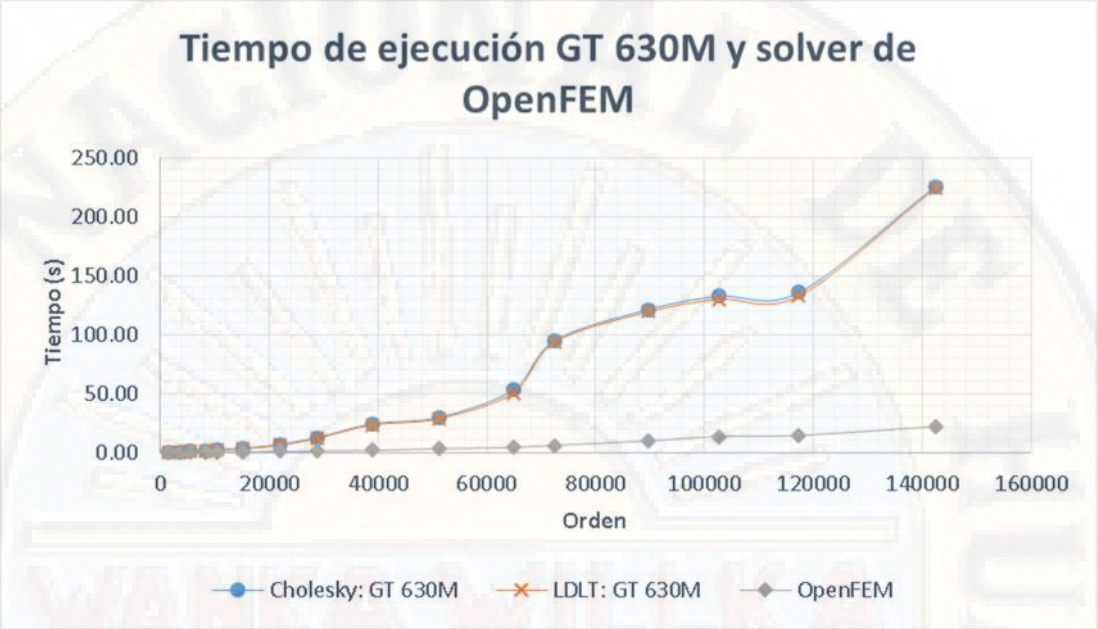


Gráfico 10: tiempo de ejecución GT 630M y solver de OpenFEM

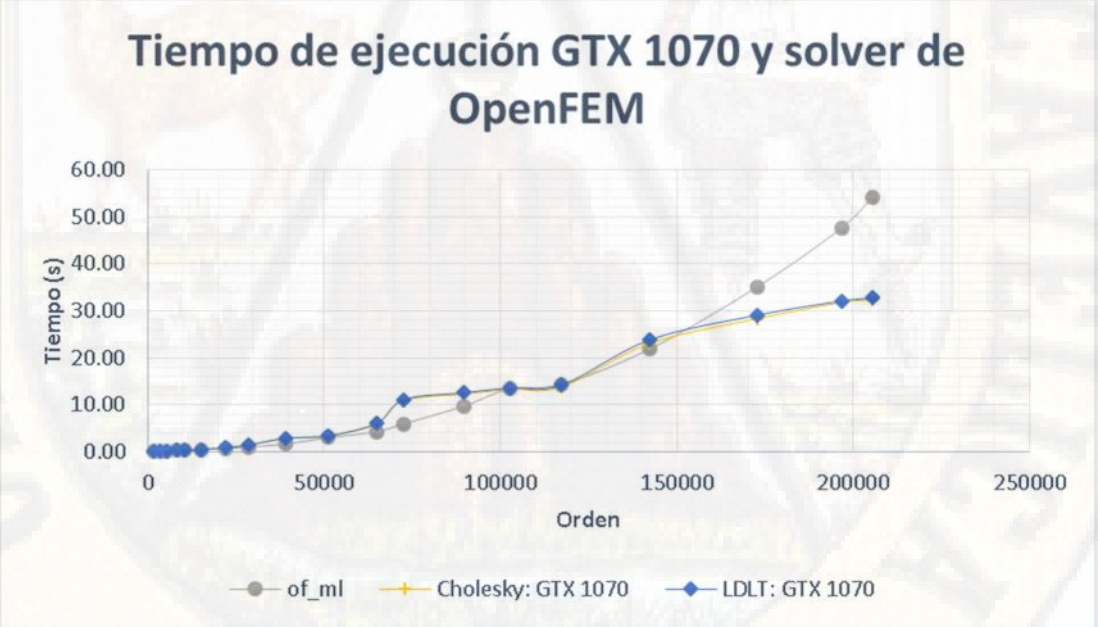


Gráfico 11: tiempo de ejecución GTX 1070 y solver de OpenFEM

4.1. Discusión de resultados

Según los visto en la sección anterior, entre las técnicas usadas en ésta tesis, al aplicar los métodos de la factorización de Cholesky y la factorización LDL^T en matrices en

formato SKS se obtienen los mejores balances de memoria y tiempo de ejecución; uno de los principales motivos por el que la aplicación de estos métodos sobre matrices en formato CSC tienen muy bajo rendimiento es por los accesos a memoria, los hilos en un grupo de trabajo podrían estar apuntado a ubicaciones en memoria que no son continuos, como dice el *OpenCL Programming for the CUDA Architecture*: "asegurándonos que los accesos a memoria global se fusionen tan seguido como sea posible es una de las más importantes optimizaciones para la arquitectura CUDA" (NVIDIA).

El problema con la memoria fusionada (o *memory coalesced*) al realizar la factorización de Cholesky en matrices en formato CSC se explica a continuación:

Sea una matriz simétrica donde los elementos de la parte triangular inferior se distribuyen como sigue:

$$A = \begin{bmatrix} \mathbf{1} & * & * & * & * & * & * \\ \mathbf{2} & \mathbf{2} & * & * & * & * & * \\ 0 & \mathbf{5} & \mathbf{4} & * & * & * & * \\ 0 & \mathbf{6} & \mathbf{0} & \mathbf{3} & * & * & * \\ \mathbf{3} & \mathbf{7} & \mathbf{0} & \mathbf{0} & \mathbf{7} & * & * \\ 0 & 0 & \mathbf{4} & \mathbf{6} & \mathbf{2} & \mathbf{9} & * \\ 0 & 0 & 0 & 0 & \mathbf{9} & \mathbf{6} & \mathbf{9} \end{bmatrix}$$

Luego de realizada la compresión a formato CSC y realizada la factorización simbólica, los elementos que compondrán el campo *elements* de la matriz en este formato son los que aparecen en negrita, recordar que los ceros que se agregan al campo dejarán de ser ceros en algún momento de la factorización, en memoria el campo *elements* sería:

[1 2 3 2 5 6 7 4 0 0 4 3 0 6 7 2 9 9 6 9]

Durante la factorización, al usar la función *Chol_sparse_2* (Ver función en la Figura 37 pág. 90), en el primer paso, se modifican las columnas siguientes a la primera columna, en este caso se modifican las columnas con índices del 1 hasta el 6 (recordar que la indexación es con base en cero, es decir, la primera columna tiene índice 0), cada grupo de trabajo con sus hilos se encarga de una columna, así, el grupo de trabajo

con índice 0 se encargará de modificar la segunda columna, es decir, la columna con índice 1, pero en base a los elementos de la primera columna, esto según la expresión:

$$A_{n,1} = A_{n,1} - A_{1,0}A_{n,0}, n \geq 1$$

En este simple ejemplo el grupo de trabajo encargado de modificar la segunda columna solo modificará los elementos correspondientes a los elementos 2 y 3 de la primera columna, es decir 2 y 7, estos elementos se muestran en negrita en la matriz siguiente en donde el símbolo de asterisco simboliza los elementos que no están almacenados en memoria.

$$A = \begin{bmatrix} \mathbf{1} & * & \square & \square & \square & \square & \square \\ \mathbf{2} & \mathbf{2} & \square & \square & \square & \square & \square \\ * & 5 & 4 & \square & \square & \square & \square \\ * & 6 & 0 & 3 & \square & \square & \square \\ \mathbf{3} & \mathbf{7} & 0 & 0 & 7 & \square & \square \\ * & * & 4 & 6 & 2 & 9 & \square \\ \square & \square & \square & \square & 9 & 6 & 9 \end{bmatrix}$$

En el campo *elements* sería:

[1 **2** **3** **2** 5 6 **7** 4 0 0 4 3 0 6 7 2 9 9 6 9]

Ya que solo son dos elementos a modificar entonces solo los hilos con ID's 0 y 1 realizarán la operación en la siguiente forma:

$$Hilo\ 0 \rightarrow A_{1,1} = A_{1,1} - A_{1,0} \square A_{1,0} = 2 - 2 \square 2 = -2$$

$$Hilo\ 1 \rightarrow A_{4,1} = A_{4,1} - A_{1,0} \square A_{4,0} = 7 - 2 \square 3 = 1$$

La matriz modificada quedaría:

$$A = \begin{bmatrix} \mathbf{1} & \square & \square & \square & \square & \square & \square \\ \mathbf{2} & \mathbf{-2} & \square & \square & \square & \square & \square \\ * & 5 & 4 & \square & \square & \square & \square \\ * & 6 & 0 & 3 & \square & \square & \square \\ \mathbf{3} & \mathbf{1} & 0 & 0 & 7 & \square & \square \\ * & * & 4 & 6 & 2 & 9 & \square \\ \square & \square & \square & \square & 9 & 6 & 9 \end{bmatrix}$$

Y el campo *elements* quedaría:

[1 **2** **3** **−2** 5 6 **1** 4 0 0 4 3 0 6 7 2 9 9 6 9]

Como se puede ver, dos hilos consecutivos han escrito en ubicaciones en memoria no consecutivas ya que se saltaron los elementos 5 y 6, es esta falta de continuidad a la hora de escribir en memoria la que perjudica considerablemente el rendimiento, en este ejemplo, la distancia entre estos dos elementos es solamente dos, pero trabajando en matrices de ordenes más grandes, esta distancia puede ser mucho mayor, y no solo entre dos hilos consecutivos, sino entre más hilos de varios grupos de trabajo, que en conjunto conllevarían una considerable baja en el rendimiento.

Este comportamiento no ocurre al almacenar en formato SKS ya que los ceros entre los elementos 2 y 3 de la primera columna son almacenados en este formato, con estos ceros es posible mantener la memoria fusionada a la hora de escribir en memoria.

Otro aspecto a tomar en cuenta es el ensamblado de la matriz de rigidez, para hacer más efectivo el uso del formato SKS, la matriz de rigidez debería ser ensamblada tal que los elementos distintos de cero estén más concentrados cerca de la diagonal, esto está relacionado con la numeración de los nudos; en estructuras complejas, hacer esto podría no ser tan fácil, para estos casos el reordenamiento de filas y columnas por el algoritmo de Cuthill–McKee podría ser beneficioso.

CONCLUSIONES

- Entre las técnicas consideradas en esta tesis, las que arrojan mejores resultados (mejor balance en memoria y tiempo de ejecución), son los métodos con factorización (factorización de Cholesky y la factorización LDL^T) aplicadas sobre la matriz en formato SKS, no existiendo diferencias significativas entre estas y, en cambio, con diferencias significativas en comparación con las demás.

- La matriz sin comprimir alcanza rápidamente el límite de memoria del dispositivo (GPU). Para el dispositivo usado en esta tesis (GT 630M), el máximo orden alcanzado fue de 142422 (Ver Tabla 15 pág. 142), la memoria utilizada usando el formato SKS fue de 1483.67 MB (casi 1.5 GB); la memoria que se utilizaría en formato denso llegaría a 162273.35 MB (más de 160 GB!), capacidad que no ha sido alcanzada hasta la fecha por ninguna tarjeta gráfica, un problema de estas dimensiones ha sido resuelto por una tarjeta gráfica de apenas 2 GB de memoria, gracias a la compresión de la matriz.

- El formato de almacenamiento CSC, resulta más efectivo que el SKS para una matriz determinada, esto porque en el formato CSC se ignora la totalidad de ceros en la matriz, cosa que en el formato SKS no pasa, ya que dependiendo de la posición de los elementos distintos de cero, podría ser necesario almacenar algunos ceros, sin embargo, para los métodos con factorización, almacenar la matriz en formato CSC conlleva un problema, y es que la memoria que se necesita para almacenar la factorización es mayor a la memoria utilizada en la matriz original en formato CSC. Según los resultados, aplicar una factorización aumenta la memoria utilizada de la matriz en formato CSC a un valor incluso mayor que la que utiliza el formato SKS, la ventaja de este último es que no requiere memoria adicional para almacenar la

factorización. Al usar el método de los gradientes conjugados no se requiere ninguna factorización, por lo que la matriz original en formato CSC se mantiene, y es por tanto el método que mejor optimiza la memoria.

- Aunque el método de los gradientes conjugados, según la conclusión anterior, aplicada sobre la matriz en formato CSC, es la que mejor optimiza la memoria, el problema está en el tiempo de ejecución (Ver Tabla 11 pág. 135 y Tabla 13 pág. 140), es por eso que los métodos con factorización, aplicadas sobre la matriz en formato SKS, por tener el mejor balance de memoria y tiempo de ejecución, son las que sobresalen de las demás.

- El índice de dispersión para el formato SKS para el soporte central de puente analizado, disminuye cuando el orden aumenta; el índice de dispersión para el orden cercano a 142422 es de 0.009, lo que significa un 0.09% de la memoria que se utilizaría si almacenáramos en formato denso (Ver Gráfico 7 pág. 143).

- Las pruebas fueron realizadas en una **GeForce GT 630M,** cuyas características principales son**:** número de multiprocesadores 2, número de núcleos CUDA 96, con capacidad de cálculo (*compute capability*) 2.1, cantidad total de memoria global 2048 MB, cantidad máxima de hilos (o work ítems) en un grupo de trabajo 1024, cantidad total de memoria local por grupo de trabajo 48 KB. En general, al usarse otros dispositivos, el rendimiento mejorará según aumenten estos valores.

- Adicionalmente para notar el impacto que tiene usar otro dispositivo con mejores características se ha usado la tarjeta gráfica GTX 1070, cuyas características en contraste con la GT 630M puede observarse en la Tabla 17 (pág. 145). La diferencia encontrada al usar otro dispositivo, usando las mismas funciones (es decir los mismos kernels), resolviendo el mismo sistema de ecuaciones, es bastante significativa, para el orden de 142422 la tarjeta gráfica GT 630M arrojó un tiempo de 225.44 segundos para la factorización

de Cholesky sobre la matriz SKS, mientras que la GTX 1070, para el mismo método y formato arrojó un tiempo de 23.11 segundos.

- Visto los resultados en tiempo de ejecución con los arrojados por el solver incorporado del OpenFEM, concluimos que la codificación aún está sujeta a mejoras, tanto para los métodos con factorización sobre la matriz SKS, y en mayor grado para las demás técnicas.

- Los tiempos de ejecución para los métodos con factorización sobre la matriz CSC, no incluyen el tiempo necesario para realizar la factorización simbólica, es decir, si incluyéramos este tiempo, los resultados para éste formato de almacenamiento serían algo mayores.

- Para el método de los gradientes conjugados, lo que al principio tomamos como una ventaja, en la que éste método solo requería de operaciones sencillas en cada iteración, y que podían hacerse haciendo uso de las funciones incorporadas en ArrayFire, sin necesidad de escribir un kernel; resultaron teniendo inconvenientes en el tiempo de ejecución, podemos conjeturar que esto es debido a las demasiadas llamadas, construcciones y ejecuciones de kernels que se hacen para cada función incorporada utilizada, en nuestro caso, estos podrían haberse hecho dentro de una misma llamada a un kernel único (dentro de lo que fuera posible).

RECOMENDACIONES

- Limitándonos a las técnicas aplicadas en ésta tesis, y a los códigos correspondientes a cada una de estas, se recomienda el uso de los métodos con factorización (factorización de Cholesky y factorización LDL^T), sobre la matriz en formato SKS, ya que son los que arrojan resultados más balanceados en memoria y tiempo de ejecución.

- Para hacer más efectivo el uso del formato SKS e incluso mejorar el uso del formato CSC, la matriz de rigidez debería ser ensamblada tal que los elementos distintos de cero estén más concentrados cerca de la diagonal, esto está relacionado con la numeración de los nudos; en estructuras complejas hacer esto podría no ser tan fácil, para estos casos se recomienda el uso del algoritmo de Cuthill–McKee para reenumerar filas y columnas, sin perder la simetría, y conseguir una matriz con elementos distintos de cero concentrados más cerca de la diagonal, obviamente el vector de fuerzas también debe pivotarse consistentemente con el pivotado de la matriz de rigidez; luego de resolver el sistema puede hacerse un pivote inverso para conseguir la solución del sistema original.

- Trabajar las lecturas o escrituras sobre memoria fusionada es uno de los temas más importantes a la hora de escribir kernels, la gran diferencia en el rendimiento entre la aplicación de los métodos sobre matrices en formato SKS y CSC recae justamente en este aspecto, básicamente se trata de que hilos consecutivos lean o escriban en ubicaciones en memoria consecutiva, como se vio en la sección de discusión de resultados, cuando este no es el caso, el rendimiento decae considerablemente.

- Usar un dispositivo con más multiprocesadores mejora el rendimiento, en general, mientras más multiprocesadores tenga un dispositivo, más instrucciones en paralelo podrán realizarse y por lo tanto el trabajo terminará más rápido, algo que se evidenció al comparar los resultados arrojadas por las tarjetas gráficas GT 630M y GTX 1070 en esta tesis.

Bibliografía

ArrayFire company. (s.f.). *ArrayFire documentation*. Obtenido de http://arrayfire.org/docs/index.htm

B.P.Demidovich, I. (1977). *Cálculo numérico fundamental.* (M. M. Calvo, Trad.) Madrid: PARANINFO.

Celigüeta Lizarza , J. T. (2008). *Método de los Elementos Finitos para análisis estructural* (Tercera ed.). Navarra: Tecnun.

Félix, M. V. (2015). *Factorización Cholesky simbólica*.

Hernández Sampieri, R., Fernández Collado, C., & Baptista Lucio, P. (2010). *Metodología de la investigación* (Quinta ed.). México: The McGraw-Hill.

Juan Carlos Camacho Puello, M. d. (2012). Análisis estructural con el método de elementos finitos asistido por computadora. Cartagena de Indias: Universidad Tecnológica de Bolivar.

Juliacomputing.com. (s.f.). *GPU Programming in Julia*. Obtenido de https://juliacomputing.com/blog/2016/06/09/julia-gpu.html

JuliaLang. (s.f.). *Julia 1.1 Documentation*. Obtenido de https://docs.julialang.org/en/v1.1/

Niño Rojas, V. M. (2011). *Metodología de la investigación.* Bogotá: Ediciones de la U.

NVIDIA Corporation. (n.d.). CUDA C Programming Guide. Retrieved from https://docs.nvidia.com/cuda/archive/8.0/cuda-c-programming-guide/index.html

NVIDIA. (s.f.). OpenCL Programming for the CUDA Architecture.

O'Connor, J. L. (1997). *Técnicas de cálculo para sistemas de ecuaciones, programación lineal y programación entera.* Madrid.

Sánchez Meza, R. A. (2000). Procedimientos de gradiente conjugada en el análisis estructural. (U. N. PERÚ, Ed.) Obtenido de http://cybertesis.uni.edu.pe/handle/uni/465

Santamaría, M. J. (1999). *Factorización de Cholesky modificada de matrices dispersas sobre multiprocesadores.* Universidad de Santiago de Compostela, Facultad de física, Departamento de Electrónica y Computación.

Shucai Xiao, W.-c. F. (2009). Inter-Block GPU Communication via Fast Barrier Synchronization.

Steven C. Rennich, D. S. (2016). Accelerating sparse Cholesky factorization on GPUs.

stuff, W. (s.f.). *Skyline Storage Format for sparse matrices*. Obtenido de https://234satwant.wordpress.com/2014/08/29/skyline-method-for-sparse-matrices/

Valdivia, F. A. (2014). *Procedimiento para el análisis dinámico de estructuras usando el método de los elementos finitos.* Lima: Pontifica Universidad Católica del Perú.

Vargas-Félix, M. (2015). *Gradiente conjugado*.

Watkins, D. S. (2002). *Fundamentals of matrix computations* (Second Edition ed.). (I. John Wiley & Sons, Ed.) New York: Wiley-Interscience. Obtenido de https://books.google.com.pe

Wikipedia. (26 de Octubre de 2013). *GPGPU*. Obtenido de https://es.wikipedia.org/wiki/GPGPU

Wikipedia. (s.f.). *Sparse matrix*. Obtenido de https://en.wikipedia.org/wiki/Sparse_matrix

ANEXOS

Matriz de consistencia y operacionalidad de variables

MATRIZ DE CONSISTENCIA

"TÉCNICAS NUMÉRICAS SOBRE PROCESADOR GRÁFICO, PARA LA SOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES EN ANÁLISIS ESTRUCTURAL"

<table>
<tr>
<th>Problema</th>
<th>Objetivos</th>
<th>Marco Teórico</th>
<th>Variables</th>
<th>Dimensiones</th>
<th>Metodología</th>
</tr>
<tr>
<td>Problema general:
¿Cuáles son las características de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural?

Problemas específicos:
1. ¿Cuál es el tiempo de ejecución de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural?

2. ¿Cuál es la memoria utilizada de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural?</td>
<td>Objetivo general:
Determinar las características de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural.

Objetivos específicos:
1. Determinar el tiempo de ejecución de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural.

2. Determinar la memoria utilizada de las técnicas numéricas sobre procesador gráfico, al resolver sistemas de ecuaciones lineales en problemas de análisis estructural.</td>
<td>Antecedentes:

1. Internacional
✓ Juan Carlos Camacho Puello, Marlos de Jesús Romero Torres. Análisis estructural con el método de elementos finitos asistido por computadora. Cartagena de Indias: Universidad Tecnológica de Bolivar, 2012.
✓ Félix, Miguel Vargas. Factorización Cholesky simbólica. [En línea] 2015.

✓ Accelerating sparse Cholesky factorization on GPUs. Steven C. Rennich, Darko Stosic, Timothy A. Davis. United States: ScienceDirect, 2016.

2. Nacionales
✓ Sánchez Meza, Roque Alberto. Procedimientos de gradiente conjugada en el análisis estructural. [ed.] Universidad Nacional de Ingeniería. Programa Cybertesis PERÚ. 2000.

Definiciones de términos:
✓ Sistemas de Ecuaciones Lineales
✓ Métodos de solución de Sistemas de Ecuaciones Lineales: Gauss-Jordan, factorización de Cholesky, factorización LDL^T, Gradiente conjugado
✓ Uso de procesador gráfico
-GPGPU
-Programación de GPU en Julia</td>
<td>.

V. Independiente

➢ Técnicas numéricas sobre procesador gráfico

V. Dependiente

➢ Solución de sistemas de ecuaciones lineales en análisis estructural</td>
<td>Método (formato de almacenamiento)
- Gauss Jordan (Matriz densa)
- Factorización de Cholesky (matriz densa)
- Factorización de Cholesky (matriz CSC)
- Factorización de Cholesky (matriz SKS)
- Factorización LDL^T (Matriz densa).
- Factorización LDL^T (Matriz CSC).
- Factorización LDL^T (matriz SKS)
- Gradientes conjugados (Matriz densa).
- Gradientes conjugados (matriz CSC).
- Gradientes conjugados (matriz SKS).

- Tiempo de procesamiento (s)
- Memoria utilizada (MB)</td>
<td>Tipo de investigación: Aplicada.

Nivel de investigación: Descriptivo

Diseño de investigación: Transversal

Población y Muestra

Población:
- Modelos estructurales

Muestra:

- Viga en voladizo
- Pilar de puente

Técnicas e Instrumentos de obtención de datos
Para obtener el SEL se analizará cada estructura usando el OpenFEM, herramienta de elementos finitos de libre distribución.

Técnicas de procesamiento y análisis de datos
• Cuantitativas: Usaremos como herramienta de programación el OpenCL, C++, la biblioteca de ArrayFire, el paquete ArrayFire.jl y el lenguaje Julia, los resultados (tiempo de procesamiento se medirán usando la función @time y/o @elapsed de Julia. La memoria se medirá manualmente de acuerdo al número de elementos de la matriz y la precisión de los elementos.
• Cualitativas: se empleará para la interpretación de resultados.</td>
</tr>
</table>

OPERACIONALIDAD DE VARIABLES

Variables	Dimensiones	Metodología
V. Independiente • Técnicas numéricas sobre procesador gráfico.	Método (formato de almacenamiento) • Gauss Jordan (Matriz densa) • Factorización de Cholesky (matriz densa) • Factorización de Cholesky (matriz CSC) • Factorización de Cholesky (matriz SKS) • Factorización LDLT (Matriz densa). • Factorización LDLT (Matriz CSC). • Factorización LDLT (matriz SKS) • Gradientes conjugados (Matriz densa). • Gradientes conjugados (matriz CSC). • Gradientes conjugados (matriz SKS).	• Revisión bibliográfica • Escritura de funciones en Julia y ArrayFire. • Escritura de kernels en OpenCL. • Ejecución de funciones y/o kernels
V. Dependiente • Solución de sistemas de ecuaciones lineales en análisis estructural	• Tiempo de ejecución (s) • Memoria utilizada (MB)	**Obtención del sistema de ecuaciones:** OpenFEM, herramienta de elementos finitos **Memoria:** medición manual de acuerdo al número de elementos de la matriz y la precisión de los elementos. **Tiempo:** Macro @time de Julia Macro @elapsed de Julia

Características del dispositivo

Salida del ejecutable ocldevicequery.exe de OpenCL

```
G:\oclDeviceQuery.exe Starting...

OpenCL SW Info:

 CL_PLATFORM_NAME:    NVIDIA CUDA
 CL_PLATFORM_VERSION:      OpenCL 1.2 CUDA 9.1.84
 OpenCL SDK Revision:      7027912


OpenCL Device Info:

 1 devices found supporting OpenCL:

 ---------------------------------
 Device GeForce GT 630M
 ---------------------------------
  CL_DEVICE_NAME:                GeForce GT 630M
  CL_DEVICE_VENDOR:              NVIDIA Corporation
  CL_DRIVER_VERSION:             391.35
  CL_DEVICE_VERSION:             OpenCL 1.1 CUDA
  CL_DEVICE_OPENCL_C_VERSION:          OpenCL C 1.1
  CL_DEVICE_TYPE:                CL_DEVICE_TYPE_GPU
  CL_DEVICE_MAX_COMPUTE_UNITS:         2
  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: 3
  CL_DEVICE_MAX_WORK_ITEM_SIZES: 1024 / 1024 / 64
  CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024
  CL_DEVICE_MAX_CLOCK_FREQUENCY: 950 MHz
  CL_DEVICE_ADDRESS_BITS:        64
  CL_DEVICE_MAX_MEM_ALLOC_SIZE:        512 MByte
  CL_DEVICE_GLOBAL_MEM_SIZE:           2048 MByte
  CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
  CL_DEVICE_LOCAL_MEM_TYPE:      local
  CL_DEVICE_LOCAL_MEM_SIZE:      48 KByte
  CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
  CL_DEVICE_QUEUE_PROPERTIES:
      CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
  CL_DEVICE_QUEUE_PROPERTIES:
      CL_QUEUE_PROFILING_ENABLE
  CL_DEVICE_IMAGE_SUPPORT:       1
  CL_DEVICE_MAX_READ_IMAGE_ARGS: 128
  CL_DEVICE_MAX_WRITE_IMAGE_ARGS:      8
  CL_DEVICE_SINGLE_FP_CONFIG:          denorms INF-quietNaNs
round-to-nearest round-to-zero round-to-inf fma

  CL_DEVICE_IMAGE <dim>                2D_MAX_WIDTH      16384
                             2D_MAX_HEIGHT     16384
                             3D_MAX_WIDTH      2048
                             3D_MAX_HEIGHT     2048
                             3D_MAX_DEPTH      2048

  CL_DEVICE_EXTENSIONS:                cl_khr_global_int32
_base_atomics
                             cl_khr_global_int32
_extended_atomics
                             cl_khr_local_int32_base_atomics
                             cl_khr_local_int32_extended_atomics
                             cl_khr_fp64
```

```
                                cl_khr_byte_addressable_store
                                cl_khr_icd
                                cl_khr_gl_sharing
                                cl_nv_compiler_options
                                cl_nv_device_attribute_query
                                cl_nv_pragma_unroll
                                cl_nv_d3d10_sharing
                                cl_khr_d3d10_sharing
                                cl_nv_d3d11_sharing
                                cl_nv_copy_opts


  CL_DEVICE_COMPUTE_CAPABILITY_NV:    2.1
  NUMBER OF MULTIPROCESSORS:          2
  NUMBER OF CUDA CORES:               96
  CL_DEVICE_REGISTERS_PER_BLOCK_NV:   32768
  CL_DEVICE_WARP_SIZE_NV:             32
  CL_DEVICE_GPU_OVERLAP_NV:           CL_TRUE
  CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV:   CL_TRUE
  CL_DEVICE_INTEGRATED_MEMORY_NV:     CL_FALSE
  CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>CHAR 1, SHORT 1, INT 1,
LONG 1, FLOAT 1, DOUBLE 1


  ---------------------------------
  2D Image Formats Supported (71)
  ---------------------------------
  #     Channel Order   Channel Type

  1     CL_R            CL_FLOAT
  2     CL_R            CL_HALF_FLOAT
  3     CL_R            CL_UNORM_INT8
  4     CL_R            CL_UNORM_INT16
  5     CL_R            CL_SNORM_INT16
  6     CL_R            CL_SIGNED_INT8
  7     CL_R            CL_SIGNED_INT16
  8     CL_R            CL_SIGNED_INT32
  9     CL_R            CL_UNSIGNED_INT8
  10    CL_R            CL_UNSIGNED_INT16
  11    CL_R            CL_UNSIGNED_INT32
  12    CL_A            CL_FLOAT
  13    CL_A            CL_HALF_FLOAT
  14    CL_A            CL_UNORM_INT8
  15    CL_A            CL_UNORM_INT16
  16    CL_A            CL_SNORM_INT16
  17    CL_A            CL_SIGNED_INT8
  18    CL_A            CL_SIGNED_INT16
  19    CL_A            CL_SIGNED_INT32
  20    CL_A            CL_UNSIGNED_INT8
  21    CL_A            CL_UNSIGNED_INT16
  22    CL_A            CL_UNSIGNED_INT32
  23    CL_RG           CL_FLOAT
  24    CL_RG           CL_HALF_FLOAT
  25    CL_RG           CL_UNORM_INT8
  26    CL_RG           CL_UNORM_INT16
  27    CL_RG           CL_SNORM_INT16
  28    CL_RG           CL_SIGNED_INT8
```

```
29    CL_RG           CL_SIGNED_INT16
30    CL_RG           CL_SIGNED_INT32
31    CL_RG           CL_UNSIGNED_INT8
32    CL_RG           CL_UNSIGNED_INT16
33    CL_RG           CL_UNSIGNED_INT32
34    CL_RA           CL_FLOAT
35    CL_RA           CL_HALF_FLOAT
36    CL_RA           CL_UNORM_INT8
37    CL_RA           CL_UNORM_INT16
38    CL_RA           CL_SNORM_INT16
39    CL_RA           CL_SIGNED_INT8
40    CL_RA           CL_SIGNED_INT16
41    CL_RA           CL_SIGNED_INT32
42    CL_RA           CL_UNSIGNED_INT8
43    CL_RA           CL_UNSIGNED_INT16
44    CL_RA           CL_UNSIGNED_INT32
45    CL_RGBA         CL_FLOAT
46    CL_RGBA         CL_HALF_FLOAT
47    CL_RGBA         CL_UNORM_INT8
48    CL_RGBA         CL_UNORM_INT16
49    CL_RGBA         CL_SNORM_INT16
50    CL_RGBA         CL_SIGNED_INT8
51    CL_RGBA         CL_SIGNED_INT16
52    CL_RGBA         CL_SIGNED_INT32
53    CL_RGBA         CL_UNSIGNED_INT8
54    CL_RGBA         CL_UNSIGNED_INT16
55    CL_RGBA         CL_UNSIGNED_INT32
56    CL_BGRA         CL_UNORM_INT8
57    CL_BGRA         CL_SIGNED_INT8
58    CL_BGRA         CL_UNSIGNED_INT8
59    CL_ARGB         CL_UNORM_INT8
60    CL_ARGB         CL_SIGNED_INT8
61    CL_ARGB         CL_UNSIGNED_INT8
62    CL_INTENSITY    CL_FLOAT
63    CL_INTENSITY    CL_HALF_FLOAT
64    CL_INTENSITY    CL_UNORM_INT8
65    CL_INTENSITY    CL_UNORM_INT16
66    CL_INTENSITY    CL_SNORM_INT16
67    CL_LUMINANCE    CL_FLOAT
68    CL_LUMINANCE    CL_HALF_FLOAT
69    CL_LUMINANCE    CL_UNORM_INT8
70    CL_LUMINANCE    CL_UNORM_INT16
71    CL_LUMINANCE    CL_SNORM_INT16

---------------------------------
3D Image Formats Supported (71)
---------------------------------
#     Channel Order   Channel Type

1     CL_R            CL_FLOAT
2     CL_R            CL_HALF_FLOAT
3     CL_R            CL_UNORM_INT8
4     CL_R            CL_UNORM_INT16
5     CL_R            CL_SNORM_INT16
6     CL_R            CL_SIGNED_INT8
7     CL_R            CL_SIGNED_INT16
8     CL_R            CL_SIGNED_INT32
```

```
9     CL_R            CL_UNSIGNED_INT8
10    CL_R            CL_UNSIGNED_INT16
11    CL_R            CL_UNSIGNED_INT32
12    CL_A            CL_FLOAT
13    CL_A            CL_HALF_FLOAT
14    CL_A            CL_UNORM_INT8
15    CL_A            CL_UNORM_INT16
16    CL_A            CL_SNORM_INT16
17    CL_A            CL_SIGNED_INT8
18    CL_A            CL_SIGNED_INT16
19    CL_A            CL_SIGNED_INT32
20    CL_A            CL_UNSIGNED_INT8
21    CL_A            CL_UNSIGNED_INT16
22    CL_A            CL_UNSIGNED_INT32
23    CL_RG           CL_FLOAT
24    CL_RG           CL_HALF_FLOAT
25    CL_RG           CL_UNORM_INT8
26    CL_RG           CL_UNORM_INT16
27    CL_RG           CL_SNORM_INT16
28    CL_RG           CL_SIGNED_INT8
29    CL_RG           CL_SIGNED_INT16
30    CL_RG           CL_SIGNED_INT32
31    CL_RG           CL_UNSIGNED_INT8
32    CL_RG           CL_UNSIGNED_INT16
33    CL_RG           CL_UNSIGNED_INT32
34    CL_RA           CL_FLOAT
35    CL_RA           CL_HALF_FLOAT
36    CL_RA           CL_UNORM_INT8
37    CL_RA           CL_UNORM_INT16
38    CL_RA           CL_SNORM_INT16
39    CL_RA           CL_SIGNED_INT8
40    CL_RA           CL_SIGNED_INT16
41    CL_RA           CL_SIGNED_INT32
42    CL_RA           CL_UNSIGNED_INT8
43    CL_RA           CL_UNSIGNED_INT16
44    CL_RA           CL_UNSIGNED_INT32
45    CL_RGBA         CL_FLOAT
46    CL_RGBA         CL_HALF_FLOAT
47    CL_RGBA         CL_UNORM_INT8
48    CL_RGBA         CL_UNORM_INT16
49    CL_RGBA         CL_SNORM_INT16
50    CL_RGBA         CL_SIGNED_INT8
51    CL_RGBA         CL_SIGNED_INT16
52    CL_RGBA         CL_SIGNED_INT32
53    CL_RGBA         CL_UNSIGNED_INT8
54    CL_RGBA         CL_UNSIGNED_INT16
55    CL_RGBA         CL_UNSIGNED_INT32
56    CL_BGRA         CL_UNORM_INT8
57    CL_BGRA         CL_SIGNED_INT8
58    CL_BGRA         CL_UNSIGNED_INT8
59    CL_ARGB         CL_UNORM_INT8
60    CL_ARGB         CL_SIGNED_INT8
61    CL_ARGB         CL_UNSIGNED_INT8
62    CL_INTENSITY    CL_FLOAT
63    CL_INTENSITY    CL_HALF_FLOAT
64    CL_INTENSITY    CL_UNORM_INT8
65    CL_INTENSITY    CL_UNORM_INT16
```

```
  66    CL_INTENSITY    CL_SNORM_INT16
  67    CL_LUMINANCE    CL_FLOAT
  68    CL_LUMINANCE    CL_HALF_FLOAT
  69    CL_LUMINANCE    CL_UNORM_INT8
  70    CL_LUMINANCE    CL_UNORM_INT16
  71    CL_LUMINANCE    CL_SNORM_INT16

oclDeviceQuery, Platform Name = NVIDIA CUDA, Platform Version
= OpenCL 1.2 CUDA 9.1.84, SDK Revision = 7027912, NumDevs = 1,
Device = GeForce GT 630M

System Info:

 Local Time/Date = 0:37:29, 12/11/2019
 CPU Arch: 9
 CPU Level: 6
 # of CPU processors: 8
 Windows Build: 9200
 Windows Ver: 6.2 (Windows Vista / Windows 7)
```

```
oclDeviceQuery Starting...

OpenCL SW Info:

 CL_PLATFORM_NAME:    NVIDIA CUDA
 CL_PLATFORM_VERSION:      OpenCL 1.2 CUDA 9.2.239
 OpenCL SDK Revision:      7027912


OpenCL Device Info:

 1 devices found supporting OpenCL:

 ---------------------------------
 Device GeForce GTX 1070
 ---------------------------------
  CL_DEVICE_NAME:                GeForce GTX 1070
  CL_DEVICE_VENDOR:              NVIDIA Corporation
  CL_DRIVER_VERSION:             399.31
  CL_DEVICE_VERSION:             OpenCL 1.2 CUDA
  CL_DEVICE_OPENCL_C_VERSION:          OpenCL C 1.2
  CL_DEVICE_TYPE:                CL_DEVICE_TYPE_GPU
  CL_DEVICE_MAX_COMPUTE_UNITS:         16
  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS: 3
  CL_DEVICE_MAX_WORK_ITEM_SIZES: 1024 / 1024 / 64
  CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024
  CL_DEVICE_MAX_CLOCK_FREQUENCY: 1645 MHz
  CL_DEVICE_ADDRESS_BITS:        64
  CL_DEVICE_MAX_MEM_ALLOC_SIZE:        2048 MByte
  CL_DEVICE_GLOBAL_MEM_SIZE:           8192 MByte
  CL_DEVICE_ERROR_CORRECTION_SUPPORT: no
  CL_DEVICE_LOCAL_MEM_TYPE:      local
  CL_DEVICE_LOCAL_MEM_SIZE:      48 KByte
  CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KByte
  CL_DEVICE_QUEUE_PROPERTIES:
      CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
  CL_DEVICE_QUEUE_PROPERTIES:
      CL_QUEUE_PROFILING_ENABLE
  CL_DEVICE_IMAGE_SUPPORT:       1
  CL_DEVICE_MAX_READ_IMAGE_ARGS: 256
  CL_DEVICE_MAX_WRITE_IMAGE_ARGS:      16
  CL_DEVICE_SINGLE_FP_CONFIG:          denorms INF-quietNaNs
round-to-nearest round-to-zero round-to-inf fma

  CL_DEVICE_IMAGE <dim>                2D_MAX_WIDTH      16384
                             2D_MAX_HEIGHT     32768
                             3D_MAX_WIDTH      16384
                             3D_MAX_HEIGHT     16384
                             3D_MAX_DEPTH      16384

  CL_DEVICE_EXTENSIONS:                cl_khr_global_int32
_base_atomics
                             cl_khr_global_int32
_extended_atomics
                             cl_khr_local_int32_base_atomics
                             cl_khr_local_int32_extended_atomics
                             cl_khr_fp64
```

```
                                cl_khr_byte_addressable_store
                                cl_khr_icd
                                cl_khr_gl_sharing
                                cl_nv_compiler_options
                                cl_nv_device_attribute_query
                                cl_nv_pragma_unroll
                                cl_nv_d3d10_sharing
                                cl_khr_d3d10_sharing
                                cl_nv_d3d11_sharing
                                cl_nv_copy_opts


  CL_DEVICE_COMPUTE_CAPABILITY_NV:    6.1
  NUMBER OF MULTIPROCESSORS:          16
  NUMBER OF CUDA CORES:               4294967280
  CL_DEVICE_REGISTERS_PER_BLOCK_NV:   65536
  CL_DEVICE_WARP_SIZE_NV:        32
  CL_DEVICE_GPU_OVERLAP_NV:      CL_TRUE
  CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV:   CL_TRUE
  CL_DEVICE_INTEGRATED_MEMORY_NV:     CL_FALSE
  CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>CHAR 1, SHORT 1, INT 1,
LONG 1, FLOAT 1, DOUBLE 1


  ---------------------------------
  2D Image Formats Supported (75)
  ---------------------------------
  #     Channel Order   Channel Type

  1     CL_R            CL_FLOAT
  2     CL_R            CL_HALF_FLOAT
  3     CL_R            CL_UNORM_INT8
  4     CL_R            CL_UNORM_INT16
  5     CL_R            CL_SNORM_INT16
  6     CL_R            CL_SIGNED_INT8
  7     CL_R            CL_SIGNED_INT16
  8     CL_R            CL_SIGNED_INT32
  9     CL_R            CL_UNSIGNED_INT8
  10    CL_R            CL_UNSIGNED_INT16
  11    CL_R            CL_UNSIGNED_INT32
  12    CL_A            CL_FLOAT
  13    CL_A            CL_HALF_FLOAT
  14    CL_A            CL_UNORM_INT8
  15    CL_A            CL_UNORM_INT16
  16    CL_A            CL_SNORM_INT16
  17    CL_A            CL_SIGNED_INT8
  18    CL_A            CL_SIGNED_INT16
  19    CL_A            CL_SIGNED_INT32
  20    CL_A            CL_UNSIGNED_INT8
  21    CL_A            CL_UNSIGNED_INT16
  22    CL_A            CL_UNSIGNED_INT32
  23    CL_RG           CL_FLOAT
  24    CL_RG           CL_HALF_FLOAT
  25    CL_RG           CL_UNORM_INT8
  26    CL_RG           CL_UNORM_INT16
  27    CL_RG           CL_SNORM_INT16
  28    CL_RG           CL_SIGNED_INT8
```

```
29    CL_RG           CL_SIGNED_INT16
30    CL_RG           CL_SIGNED_INT32
31    CL_RG           CL_UNSIGNED_INT8
32    CL_RG           CL_UNSIGNED_INT16
33    CL_RG           CL_UNSIGNED_INT32
34    CL_RA           CL_FLOAT
35    CL_RA           CL_HALF_FLOAT
36    CL_RA           CL_UNORM_INT8
37    CL_RA           CL_UNORM_INT16
38    CL_RA           CL_SNORM_INT16
39    CL_RA           CL_SIGNED_INT8
40    CL_RA           CL_SIGNED_INT16
41    CL_RA           CL_SIGNED_INT32
42    CL_RA           CL_UNSIGNED_INT8
43    CL_RA           CL_UNSIGNED_INT16
44    CL_RA           CL_UNSIGNED_INT32
45    CL_RGBA         CL_FLOAT
46    CL_RGBA         CL_HALF_FLOAT
47    CL_RGBA         CL_UNORM_INT8
48    CL_RGBA         CL_UNORM_INT16
49    CL_RGBA         CL_SNORM_INT16
50    CL_RGBA         CL_SIGNED_INT8
51    CL_RGBA         CL_SIGNED_INT16
52    CL_RGBA         CL_SIGNED_INT32
53    CL_RGBA         CL_UNSIGNED_INT8
54    CL_RGBA         CL_UNSIGNED_INT16
55    CL_RGBA         CL_UNSIGNED_INT32
56    CL_BGRA         CL_UNORM_INT8
57    CL_BGRA         CL_SIGNED_INT8
58    CL_BGRA         CL_UNSIGNED_INT8
59    CL_ARGB         CL_UNORM_INT8
60    CL_ARGB         CL_SIGNED_INT8
61    CL_ARGB         CL_UNSIGNED_INT8
62    CL_INTENSITY    CL_FLOAT
63    CL_INTENSITY    CL_HALF_FLOAT
64    CL_INTENSITY    CL_UNORM_INT8
65    CL_INTENSITY    CL_UNORM_INT16
66    CL_INTENSITY    CL_SNORM_INT16
67    CL_LUMINANCE    CL_FLOAT
68    CL_LUMINANCE    CL_HALF_FLOAT
69    CL_LUMINANCE    CL_UNORM_INT8
70    CL_LUMINANCE    CL_UNORM_INT16
71    CL_LUMINANCE    CL_SNORM_INT16
72    CL_BGRA         CL_SNORM_INT8
73    CL_BGRA         CL_SNORM_INT16
74    CL_ARGB         CL_SNORM_INT8
75    CL_ARGB         CL_SNORM_INT16

---------------------------------
3D Image Formats Supported (75)
---------------------------------
#     Channel Order   Channel Type

1     CL_R            CL_FLOAT
2     CL_R            CL_HALF_FLOAT
3     CL_R            CL_UNORM_INT8
4     CL_R            CL_UNORM_INT16
```

5	CL_R	CL_SNORM_INT16
6	CL_R	CL_SIGNED_INT8
7	CL_R	CL_SIGNED_INT16
8	CL_R	CL_SIGNED_INT32
9	CL_R	CL_UNSIGNED_INT8
10	CL_R	CL_UNSIGNED_INT16
11	CL_R	CL_UNSIGNED_INT32
12	CL_A	CL_FLOAT
13	CL_A	CL_HALF_FLOAT
14	CL_A	CL_UNORM_INT8
15	CL_A	CL_UNORM_INT16
16	CL_A	CL_SNORM_INT16
17	CL_A	CL_SIGNED_INT8
18	CL_A	CL_SIGNED_INT16
19	CL_A	CL_SIGNED_INT32
20	CL_A	CL_UNSIGNED_INT8
21	CL_A	CL_UNSIGNED_INT16
22	CL_A	CL_UNSIGNED_INT32
23	CL_RG	CL_FLOAT
24	CL_RG	CL_HALF_FLOAT
25	CL_RG	CL_UNORM_INT8
26	CL_RG	CL_UNORM_INT16
27	CL_RG	CL_SNORM_INT16
28	CL_RG	CL_SIGNED_INT8
29	CL_RG	CL_SIGNED_INT16
30	CL_RG	CL_SIGNED_INT32
31	CL_RG	CL_UNSIGNED_INT8
32	CL_RG	CL_UNSIGNED_INT16
33	CL_RG	CL_UNSIGNED_INT32
34	CL_RA	CL_FLOAT
35	CL_RA	CL_HALF_FLOAT
36	CL_RA	CL_UNORM_INT8
37	CL_RA	CL_UNORM_INT16
38	CL_RA	CL_SNORM_INT16
39	CL_RA	CL_SIGNED_INT8
40	CL_RA	CL_SIGNED_INT16
41	CL_RA	CL_SIGNED_INT32
42	CL_RA	CL_UNSIGNED_INT8
43	CL_RA	CL_UNSIGNED_INT16
44	CL_RA	CL_UNSIGNED_INT32
45	CL_RGBA	CL_FLOAT
46	CL_RGBA	CL_HALF_FLOAT
47	CL_RGBA	CL_UNORM_INT8
48	CL_RGBA	CL_UNORM_INT16
49	CL_RGBA	CL_SNORM_INT16
50	CL_RGBA	CL_SIGNED_INT8
51	CL_RGBA	CL_SIGNED_INT16
52	CL_RGBA	CL_SIGNED_INT32
53	CL_RGBA	CL_UNSIGNED_INT8
54	CL_RGBA	CL_UNSIGNED_INT16
55	CL_RGBA	CL_UNSIGNED_INT32
56	CL_BGRA	CL_UNORM_INT8
57	CL_BGRA	CL_SIGNED_INT8
58	CL_BGRA	CL_UNSIGNED_INT8
59	CL_ARGB	CL_UNORM_INT8
60	CL_ARGB	CL_SIGNED_INT8
61	CL_ARGB	CL_UNSIGNED_INT8

```
  62    CL_INTENSITY    CL_FLOAT
  63    CL_INTENSITY    CL_HALF_FLOAT
  64    CL_INTENSITY    CL_UNORM_INT8
  65    CL_INTENSITY    CL_UNORM_INT16
  66    CL_INTENSITY    CL_SNORM_INT16
  67    CL_LUMINANCE    CL_FLOAT
  68    CL_LUMINANCE    CL_HALF_FLOAT
  69    CL_LUMINANCE    CL_UNORM_INT8
  70    CL_LUMINANCE    CL_UNORM_INT16
  71    CL_LUMINANCE    CL_SNORM_INT16
  72    CL_BGRA         CL_SNORM_INT8
  73    CL_BGRA         CL_SNORM_INT16
  74    CL_ARGB         CL_SNORM_INT8
  75    CL_ARGB         CL_SNORM_INT16

oclDeviceQuery, Platform Name = NVIDIA CUDA, Platform Version
= OpenCL 1.2 CUDA 9.2.239, SDK Revision = 7027912, NumDevs =
1, Device = GeForce GTX 1070

System Info:

 Local Time/Date = 15:55:36, 11/23/2019
 CPU Arch: 9
 CPU Level: 6
 # of CPU processors: 8
 Windows Build: 9200
 Windows Ver: 6.2 (Windows Vista / Windows 7)
```

Código fuente

Se muestra el código fuente del archivo donde están escritas las funciones usando la biblioteca de ArrayFire, que llaman a los kernels OpenCL, según el siguiente detalle:

Formato de la matriz	Método	Kernel	función	línea
Denso	Gauss Jordan	Gauss_Jordan_c	SELgj_c	807
	Factorización de Cholesky	Cholesky_c	SELchol_c, fac_chol_c	1681, 1828
	Factorización LDL^T	ldlt_c	SELldlt_c, fac_ldlt_c	1930, 2071
	Gradientes conjugados		SEL_gc	1289
CSC	Factorización de Cholesky	Cholesky_sparse_c	fac_sparse_chol_c, SELchol_sparse_c	2165, 2549
	Factorización LDL^T	ldlt_sparse_c	fac_sparse_ldlt_c, SELldlt_sparse_c	2920, 3090
	Gradientes conjugados		SELgc_sparse	1406
SKS	Factorización de Cholesky	chol_sparse_sk	fac_sparse_chol_sks , SELchol_sparse_sks	2692, 2798
	Factorización LDL^T	ldlt_sparse_sks	fac_sparse_ldlt_sks , SELldlt_sparse_sks	3235, 3335
	Gradientes conjugados		SELgc_sparse_sks	1527

Nota: Los códigos (kernels y funciones) de la presente tesis están almacenadas en el repositorio: https://github.com/4lrdyD/oclLinealSol, los números de línea de la tabla podrían no mantenerse debido a posibles actualizaciones de código.

```
1  //=========================================
2  //revisión 0.9.6 01-12-2019, 18:10 VS 2017
3  //=========================================
4
5  #include "Header.h"
6  #include "Common.h"
7  char * matrixMul_source=AFire::kernel_src("Common.h",
8      "matrixMul.cl");
9  char * GJordan_source = AFire::kernel_src("Common.h",
10     "Gauss_Jordan.cl");
11 char * Cholesky_source = AFire::kernel_src("Common.h",
12     "Cholesky.cl");
13 char * ldlt_source = AFire::kernel_src("Common.h", "ldlt.cl");
14 char * gc_source = AFire::kernel_src("Common.h",
15     "Gradiente_conjugado.cl");
16 char * util_src = AFire::kernel_src("Common.h", "util.cl");
17 using namespace af;
18
19 //----------
20 //utilidades
21 //----------
22 void AFire::copy(const af::array &A, af::array &B,size_t length) {
23
24     // 2. Obtain the device, context, and queue used by ArrayFire
25     static cl_context af_context = afcl::getContext();
26     static cl_device_id af_device_id = afcl::getDeviceId();
27     static cl_command_queue af_queue = afcl::getQueue();
28     // 3. Obtain cl_mem references to af::array objects
29     cl_mem * d_A = A.device<cl_mem>();
30     cl_mem * d_B = B.device<cl_mem>();
31
32     // 4. Load, build, and use your kernels.
33     //    For the sake of readability, we have omitted error checking.
34     int status = CL_SUCCESS;
35     // A simple copy kernel, uses C++11 syntax for multi-line strings.
36     const char * kernel_name = "copy_kernel";
37     const char * source = R"(
38         void __kernel
39         copy_kernel(__global float * gA, __global float * gB)
40         {
41             int id = get_global_id(0);
42             gB[id] = gA[id];
43         }
44     )";
45     // Create the program, build the executable, and extract the entry point
46     // for the kernel.
47     cl_program program = clCreateProgramWithSource(af_context, 1, &source,
         NULL, &status);
48     status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
49     cl_kernel kernel = clCreateKernel(program, kernel_name, &status);
50     // Set arguments and launch your kernels
51     clSetKernelArg(kernel, 0, sizeof(cl_mem), d_A);
52     clSetKernelArg(kernel, 1, sizeof(cl_mem), d_B);
53     clEnqueueNDRangeKernel(af_queue, kernel, 1, NULL, &length, NULL, 0, NULL,
         NULL);
54     // 5. Return control of af::array memory to ArrayFire
```

```
55      A.unlock();
56      B.unlock();
57      // ... resume ArrayFire operations
58      // Because the device pointers, d_x and d_y, were returned to ArrayFire's
59      // control by the unlock function, there is no need to free them using
60      // clReleaseMemObject()
61  }
62
63  void AFire::sumaaf(af_array* out , af_array dA, af_array dB) {
64
65      //to store the result
66      af_array dC;
67      af_copy_array(&dC, dA);
68
69      // 2. Obtain the device, context, and queue used by ArrayFire
70      static cl_context af_context = afcl::getContext();
71      static cl_device_id af_device_id = afcl::getDeviceId();
72      static cl_command_queue af_queue = afcl::getQueue();
73
74      dim_t _order[AF_MAX_DIMS];
75      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], dA);
76      size_t order = _order[0];
77
78      int status = CL_SUCCESS;
79
80      // 3. Obtain cl_mem references to af_array objects
81      cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
82          CL_MEM_READ_ONLY, sizeof(float) * order,
83          NULL, &status);
84      af_get_device_ptr((void**)d_A, dA);
85
86      cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
87          CL_MEM_READ_ONLY, sizeof(float) * order,
88          NULL, &status);
89      af_get_device_ptr((void**)d_B, dB);
90
91      cl_mem *d_C = (cl_mem*)clCreateBuffer(af_context,
92          CL_MEM_WRITE_ONLY, sizeof(float) * order,
93          NULL, &status);
94      af_get_device_ptr((void**)d_C, dC);
95
96      // 4. Load, build, and use your kernels.
97      //    For the sake of readability, we have omitted error checking.
98      // A simple sum kernel, uses C++11 syntax for multi-line strings.
99      const char * kernel_name = "sum_kernel";
100     const char * source = R"(
101         void __kernel
102         sum_kernel(__global float * gC, __global float * gA, __global float *
            gB)
103         {
104             int id = get_global_id(0);
105             gC[id] = gA[id]+gB[id];
106         }
107     )";
108     // Create the program, build the executable, and extract the entry point
109     // for the kernel.
```

```
110     cl_program program = clCreateProgramWithSource(af_context, 1, &source,
          NULL, &status);
111     status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
112     cl_kernel sumkernel = clCreateKernel(program, kernel_name, &status);
113     // Set arguments and launch your kernels
114     clSetKernelArg(sumkernel, 0, sizeof(cl_mem), d_C);
115     clSetKernelArg(sumkernel, 1, sizeof(cl_mem), d_A);
116     clSetKernelArg(sumkernel, 2, sizeof(cl_mem), d_B);
117     clEnqueueNDRangeKernel(af_queue, sumkernel, 1, NULL, &order, NULL, 0,
          NULL, NULL);
118
119     // 5. Return control of af::array memory to ArrayFire
120     af_unlock_array(dA);
121     af_unlock_array(dB);
122     af_unlock_array(dC);
123
124     //copy results to output argument
125     af_copy_array(out, dC);
126
127     // ... resume ArrayFire operations
128     // Because the device pointers, d_x and d_y, were returned to ArrayFire's
129     // control by the unlock function, there is no need to free them using
130     // clReleaseMemObject()
131 }
132
133 void AFire::printh(float *A,size_t length)
134 {
135
136     for (int i = 0; i < length; i++)
137     {
138         std::cout << A[i] << std::endl;
139     }
140     std::cout << std::endl;
141 }
142
143 void AFire::mulaf(af_array* dC, const af_array dA,
144     const af_array dB) {
145     //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
146     static cl_context af_context = afcl::getContext();
147     static cl_device_id af_device_id = afcl::getDeviceId();
148     static cl_command_queue af_queue = afcl::getQueue();
149
150     //-------------------------------------
151     //detalles importantes sobre el kernel
152     //-------------------------------------
153     /*suponiendo que las matrices son compatibles para la multiplicación:
154     El detalle aquí es el uso adecuado del Kernel en matrixMul.cl
155     el kernel está escrito para multiplicar matrices cuyos datos están
          ordenados por filas,
156     sin embargo las matrices de tipo af::array son ordenadas por columnas, si
          establecemos los argumentos
157     en el kernel sin tomar en cuenta este detalle, el resultado será erroneo.
158     el Kernel acepta ocho argumentos: suponiendo C=A*B
159     1.vector que guarda los datos de C
160     2.vector que guarda los datos de A
161     3.vector que guarda los datos de B
```

```
162      4.cero con el tamaño necesario de memoria para un grupo en A
163      5.cero con el tamaño necesario de memoria para un grupo en B
164      6.ancho de A
165      7.ancho de B
166      8.alto de A
167
168      sean 2 matrices del tipo af::array
169
170   A=
171    0  4   8  12  16  20  24  28
172    1  5   9  13  17  21  25  29
173    2  6  10  14  18  22  26  30
174    3  7  11  15  19  23  27  31
175
176  B=
177    1   9  17  25  33  41
178    2  10  18  26  34  42
179    3  11  19  27  35  43
180    4  12  20  28  36  44
181    5  13  21  29  37  45
182    6  14  22  30  38  46
183    7  15  23  31  39  47
184    8  16  24  32  40  48
185
186      si queremos multiplicar estas 2 matrices usando este kernel, los datos ↵
           serían:
187      dA={0,1,2,...,31}
188      dB={1,2,3,...,48}
189      así son extraídos con: A.device<cl_mem>();
190      como podemos comprobar los están ordenados por columnas al construir las ↵
           matrices correspondientes
191
192      así como estan definidas las matrices las dimensiones serían:
193
194      ancho de A=8
195      ancho de B=6
196      alto de A=4
197
198      sin embargo si usamos el kernel en matrixMul.cl, con los argumentos así ↵
           como están, en realidad se
199      estarían multiplicando estas matrices:
200
201  A=
202    0   1   2   3   4   5   6   7
203    8   9  10  11  12  13  14  15
204   16  17  18  19  20  21  22  23
205   24  25  26  27  28  29  30  31
206
207  B=
208    1   2   3   4   5   6
209    7   8   9  10  11  12
210   13  14  15  16  17  18
211   19  20  21  22  23  24
212   25  26  27  28  29  30
213   31  32  33  34  35  36
214   37  38  39  40  41  42
```

```
215   43  44  45  46  47  48
216
217   que son diferentes a las originales, ya que como se dijo al comienzo el
        kernel asume que los valores estan
218   ordenados por filas, si se realiza la multiplicación el resultado será
        erroneo.
219
220   Sin embargo es posible usar el mismo kernel para obtener un resultado
        válido, y es invirtiendo el orden de
221   los argumentos de entrada, invirtiendo también para cada unos sus
        dimensiones, es decir colocando B de 6x8 como
222   primer argumento y A de 8x4 como segundo argumento, como se podrá comprobar
        el resultado final es el esperado.
223
224   B=
225    1   2   3   4   5   6   7   8
226    9  10  11  12  13  14  15  16
227   17  18  19  20  21  22  23  24
228   25  26  27  28  29  30  31  32
229   33  34  35  36  37  38  39  40
230   41  42  43  44  45  46  47  48
231
232  A=
233    0   1   2   3
234    4   5   6   7
235    8   9  10  11
236   12  13  14  15
237   16  17  18  19
238   20  21  22  23
239   24  25  26  27
240   28  29  30  31
241
242   B*A=
243   672   708   744   780
244   1568  1668  1768  1868
245   2464  2628  2792  2956
246   3360  3588  3816  4044
247   4256  4548  4840  5132
248   5152  5508  5864  6220
249
250   A*B= (A y B originales)
251    672  1568  2464  3360  4256  5152
252   708  1668  2628  3588  4548  5508
253   744  1768  2792  3816  4840  5864
254   780  1868  2956  4044  5132  6220
255
256   no es necesario transponer B*A (cambiados) para obtener A*B (originales) ya
        que el resultado saldrá
257   en forma de vector, estos se ordenarán automáticamente por columna una vez
        vuelvan a ser parte del campo de C
258   o matriz resultado.
259
260   */
261
262      dim_t Bdims[AF_MAX_DIMS];
263      af_get_dims(&Bdims[0], &Bdims[1], &Bdims[2], &Bdims[3], dB);
```

```
264     size_t wB = Bdims[1];
265     size_t hB = Bdims[0];
266
267     dim_t Adims[AF_MAX_DIMS];
268     af_get_dims(&Adims[0], &Adims[1], &Adims[2], &Adims[3], dA);
269     size_t hA = Adims[0];
270
271     af_dtype typef;
272     af_get_type(&typef, dA);
273
274     int msize = 0;
275     if (typef == f64)
276         msize = sizeof(double);
277     else if (typef == f32)
278         msize = sizeof(float);
279     else;
280
281     //matriz resultado
282     af_array Rs;
283     unsigned ndims = 2;
284     dim_t dim[] = { Adims[0],Bdims[1] };
285     af_randu(&Rs, ndims, dim, typef);
286
287
288     size_t program_length = strlen(matrixMul_source);
289     int status = CL_SUCCESS;
290
291     //obteniendo las referencias cl_mem de los objetos af::array
292     cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
293         CL_MEM_READ_ONLY, msize * Adims[0] * Adims[1],
294         NULL, &status);
295     af_get_device_ptr((void**)d_A, dA);
296
297     cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
298         CL_MEM_READ_ONLY, msize * Bdims[0] * Bdims[1],
299         NULL, &status);
300     af_get_device_ptr((void**)d_B, dB);
301
302     cl_mem *d_C = (cl_mem*)clCreateBuffer(af_context,
303         CL_MEM_WRITE_ONLY, msize * Adims[0] * Bdims[1],
304         NULL, &status);
305     af_get_device_ptr((void**)d_C, Rs);
306
307     //creando el programa, construyendo el ejecutable y extrayendo el punto ↵
          de entrada
308     // para el Kernel
309     cl_program program = clCreateProgramWithSource(
310         af_context, 1, (const char **)&matrixMul_source,
311         &program_length, &status);
312
313     status = clBuildProgram(program, 1, &af_device_id,
314         NULL, NULL, NULL);
315
316     char* kernelName;
317     if (typef == f64)
318         kernelName = "matrixMul";
```

```
319     else if (typef == f32)
320         kernelName = "matrixMul_sp";
321     else;
322
323     cl_kernel mulkernel = clCreateKernel(program,
324         kernelName, &status);
325
326     // estableciendo los argumentos, ver detalles importantes sobre el kernel
        (más arriba)
327     int i = 0;
328     clSetKernelArg(mulkernel, i++, sizeof(cl_mem), d_C);
329     clSetKernelArg(mulkernel, i++, sizeof(cl_mem), d_B);
330     clSetKernelArg(mulkernel, i++, sizeof(cl_mem), d_A);
331     clSetKernelArg(mulkernel, i++,
332         sizeof(double) * BLOCK_SIZE *BLOCK_SIZE, 0);
333     clSetKernelArg(mulkernel, i++,
334         sizeof(double) * BLOCK_SIZE *BLOCK_SIZE, 0);
335     clSetKernelArg(mulkernel, i++, sizeof(cl_int), &hB);
336     clSetKernelArg(mulkernel, i++, sizeof(cl_int), &hA);
337     clSetKernelArg(mulkernel, i++, sizeof(cl_int), &wB);
338
339     size_t localWorkSize[] = { BLOCK_SIZE, BLOCK_SIZE };
340     size_t globalWorkSize[] = { shrRoundUp(BLOCK_SIZE, hA),
341         shrRoundUp(BLOCK_SIZE, wB) };
342
343     //ejecutando el Kernel
344     clEnqueueNDRangeKernel(af_queue, mulkernel, 2, 0,
345         globalWorkSize, localWorkSize, 0, NULL, NULL);
346
347     //devolviendo el control de memoria af::array a ArrayFire
348     af_unlock_array(dA);
349     af_unlock_array(dB);
350     af_unlock_array(Rs);
351
352     // ... reanudando las operaciones ArrayFire
353 //copiando el resultado en el argumento de salida
354     af_copy_array(dC, Rs);
355     af_release_array(Rs);
356 }
357
358 char * AFire::find_kernel_test(const char* file)
359 {
360     char * cl_files = AFire::get_env_path(CL_VAR_ENT);
361     std::string cd_files_path = cl_files;
362     std::string file_name = file;
363     std::string path_to_file = cd_files_path + "/" + file_name;
364     size_t program_length;
365     char *header = oclLoadProgSource(path_to_file.c_str(), "",
        &program_length);
366     return header;
367 }
368
369 char * AFire::helpk(const char * string)
370 {
371     char * str = (char *)malloc(10 * sizeof(char));
372
```

```
373     strcpy_s(str, sizeof(char) * 10, "hola ");
374     strcat_s(str, sizeof(char) * 10, "will");
375     return str;
376
377     //return (char*)string;
378 }
379
380 //Obtiene la ruta explicita de una variable de
381 //entorno, si no se encuentra la variable
382 //devolverá "./"
383 char * AFire::get_env_path(const char * env)
384 {
385     std::string env_s = env;
386     env_s += "=";
387     int cont = 0;
388     bool bool_s = 0;
389     size_t pos_s;
390     std::string path_s;
391
392     extern char ** environ;
393     while (environ[cont] != NULL && !bool_s)
394     {
395         path_s = std::string(environ[cont]);
396         pos_s = path_s.find(env_s);
397         if (pos_s != std::string::npos)
398         {
399             path_s.replace(pos_s, env_s.length(), "");
400             bool_s = 1;
401         }
402         cont += 1;
403     }
404
405     if (bool_s)
406     {
407         char* file_path = (char*)malloc(path_s.length() + 1);
408         strcpy_s(file_path, path_s.length() + 1, path_s.c_str());
409         return file_path;
410     }
411     else
412     {
413         return "./";
414     }
415 }
416
417 /*Obtiene el código fuente de un kernel (OpenCL) y lo concatena
418 con el correspondiente encabezado, el programa estará listo para
419 construirse con el código devuelto
420 */
421 char * AFire::kernel_src(const char* hfile, const char * clfile)
422 {
423     size_t program_length;
424     //obteniendo el valor de la variable de entorno
425     char * cl_files = AFire::get_env_path(CL_VAR_ENT);
426     std::string cl_files_path = cl_files;
427
428     //encabezado comun a los Kernels
```

```
429     std::string file_name = hfile;
430     std::string header_path = cl_files_path + "/" + file_name;
431     char *header = oclLoadProgSource(header_path.c_str(), "",
          &program_length);
432
433     //kernel
434     file_name = clfile;
435     std::string source_path = cl_files_path + "/" + file_name;
436     char * source = oclLoadProgSource(source_path.c_str(), header,
          &program_length);
437     return source;
438 }
439
440 void AFire::sparse_mat_vec_mul(af_array* dC, af_array elmA,
441     af_array colA, af_array rowA, af_array dB) {
442     //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
443     //cl_context af_context;
444     static cl_context af_context = afcl::getContext();
445     static cl_device_id af_device_id = afcl::getDeviceId();
446     static cl_command_queue af_queue = afcl::getQueue();
447
448     //creando copia de dB
449     af_array c;
450     af_copy_array(&c, dB);
451
452     //2. Obteniendo parámetros necesarios
453
454     //longitud de los vectores
455     dim_t _order[AF_MAX_DIMS];
456     af_get_dims(&_order[0], &_order[1], &_order[2],
457         &_order[3], elmA);
458     cl_int size_elmA = _order[0];
459
460     af_get_dims(&_order[0], &_order[1], &_order[2],
461         &_order[3], colA);
462     cl_int size_colA = _order[0];
463
464     af_get_dims(&_order[0], &_order[1], &_order[2],
465         &_order[3], rowA);
466     cl_int size_rowA = _order[0];
467
468     size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
469     size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
470
471     int status = CL_SUCCESS;
472
473     af_dtype typef;
474     af_get_type(&typef, elmA);
475
476     int msize = 0;
477     if (typef == f64)
478         msize = sizeof(double);
479     else if (typef == f32)
480         msize = sizeof(float);
481     else;
482
```

```
483     //3.obteniendo las referencias cl_mem de los objetos af::array
484     cl_mem *d_elmA = (cl_mem*)clCreateBuffer(af_context,
485         CL_MEM_READ_ONLY, msize*size_elmA,
486         NULL, &status);
487     af_get_device_ptr((void**)d_elmA, elmA);
488
489     cl_mem *d_colA = (cl_mem*)clCreateBuffer(af_context,
490         CL_MEM_READ_ONLY, sizeof(int)*size_colA,
491         NULL, &status);
492     af_get_device_ptr((void**)d_colA, colA);
493
494     cl_mem *d_rowA = (cl_mem*)clCreateBuffer(af_context,
495         CL_MEM_READ_ONLY, sizeof(int)*size_rowA,
496         NULL, &status);
497     af_get_device_ptr((void**)d_rowA, rowA);
498
499     cl_mem *d_b = (cl_mem*)clCreateBuffer(af_context,
500         CL_MEM_READ_ONLY, msize*size_colA,
501         NULL, &status);
502     af_get_device_ptr((void**)d_b, dB);
503
504     cl_mem *d_c = (cl_mem*)clCreateBuffer(af_context,
505         CL_MEM_READ_WRITE, msize*size_colA,
506         NULL, &status);
507     af_get_device_ptr((void**)d_c, c);
508
509     size_t program_length = strlen(gc_source);
510
511     //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
        de entrada
512     // para el Kernel
513     cl_program program = clCreateProgramWithSource(af_context,
514         1, (const char **)&gc_source, &program_length,
515         &status);
516     status = clBuildProgram(program, 1, &af_device_id,
517         NULL, NULL, NULL);
518
519     char* kernelName;
520     if (typef == f64)
521         kernelName = "sparse_mat_vec_mul";
522     else if (typef == f32)
523         kernelName = "sparse_mat_vec_mul_sp";
524     else;
525     cl_kernel kernel = clCreateKernel(program, kernelName,
526         &status);
527
528     cl_int step = 0;
529     // 5.estableciendo los argumentos
530     int i = 0;
531     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmA);
532     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colA);
533     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowA);
534     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_b);
535     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_c);
536     clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
537     clSetKernelArg(kernel, i++, sizeof(cl_int), &size_colA);
```

```
538     clSetKernelArg(kernel, i++, sizeof(cl_int), &step);
539
540     //6. ejecutando el kernel
541
542     //transpose(L)*b
543     clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
544         &globalWorkSize, &localWorkSize, 0, NULL,
545         NULL);
546
547     //L*b
548     step++;
549     clSetKernelArg(kernel, 7, sizeof(cl_int), &step);
550     clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
551         &globalWorkSize, &localWorkSize, 0, NULL,
552         NULL);
553
554     //D*b
555     step++;
556     clSetKernelArg(kernel, 7, sizeof(cl_int), &step);
557     clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
558         &globalWorkSize, &localWorkSize, 0, NULL,
559         NULL);
560
561     //7. devolviendo el control de memoria af::array a ArrayFire
562     af_unlock_array(elmA);
563     af_unlock_array(colA);
564     af_unlock_array(rowA);
565     af_unlock_array(dB);
566     af_unlock_array(c);
567
568     //copiando al argumento de salida
569     af_copy_array(dC, c);
570
571     af_release_array(c);
572 }
573
574 void AFire::sparse_sks_mat_vec_mul(af_array* dC,
575     af_array elmA, af_array idxA, af_array dB) {
576     //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
577     //cl_context af_context;
578     static cl_context af_context = afcl::getContext();
579     static cl_device_id af_device_id = afcl::getDeviceId();
580     static cl_command_queue af_queue = afcl::getQueue();
581
582     //creando copia de dB
583     af_array c;
584     af_copy_array(&c, dB);
585
586     //2. Obteniendo parámetros necesarios
587
588     //longitud de los vectores
589     dim_t _order[AF_MAX_DIMS];
590     af_get_dims(&_order[0], &_order[1], &_order[2],
591         &_order[3], elmA);
592     cl_int size_elmA = _order[0];
593
```

```
594     af_get_dims(&_order[0], &_order[1], &_order[2],
595         &_order[3], idxA);
596     cl_int size_idxA = _order[0];
597
598     size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
599     size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
600
601     int status = CL_SUCCESS;
602
603     af_dtype typef;
604     af_get_type(&typef, elmA);
605
606     int msize = 0;
607     if (typef == f64)
608         msize = sizeof(double);
609     else if (typef == f32)
610         msize = sizeof(float);
611     else;
612
613     //3.obteniendo las referencias cl_mem de los objetos af::array
614     cl_mem *d_elmA = (cl_mem*)clCreateBuffer(af_context,
615         CL_MEM_READ_ONLY, msize*size_elmA,
616         NULL, &status);
617     af_get_device_ptr((void**)d_elmA, elmA);
618
619     cl_mem *d_idxA = (cl_mem*)clCreateBuffer(af_context,
620         CL_MEM_READ_ONLY, sizeof(int)*size_idxA,
621         NULL, &status);
622     af_get_device_ptr((void**)d_idxA, idxA);
623
624     cl_mem *d_b = (cl_mem*)clCreateBuffer(af_context,
625         CL_MEM_READ_ONLY, msize*size_idxA,
626         NULL, &status);
627     af_get_device_ptr((void**)d_b, dB);
628
629     cl_mem *d_c = (cl_mem*)clCreateBuffer(af_context,
630         CL_MEM_READ_WRITE, msize*size_idxA,
631         NULL, &status);
632     af_get_device_ptr((void**)d_c, c);
633
634     size_t program_length = strlen(gc_source);
635
636     //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
        de entrada
637     // para el Kernel
638     cl_program program = clCreateProgramWithSource(af_context,
639         1, (const char **)&gc_source, &program_length,
640         &status);
641     status = clBuildProgram(program, 1, &af_device_id,
642         NULL, NULL, NULL);
643
644     char* kernelName;
645     if (typef == f64)
646         kernelName = "sparse_mat_vec_mul_sks";
647     else if (typef == f32)
648         kernelName = "sparse_mat_vec_mul_sks_sp";
```

```
649         else;
650         cl_kernel kernel = clCreateKernel(program, kernelName,
651             &status);
652 
653         cl_int step = 0;
654         // 5.estableciendo los argumentos
655         int i = 0;
656         clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmA);
657         clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idxA);
658         clSetKernelArg(kernel, i++, sizeof(cl_mem), d_b);
659         clSetKernelArg(kernel, i++, sizeof(cl_mem), d_c);
660         clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
661         clSetKernelArg(kernel, i++, sizeof(cl_int), &size_idxA);
662         clSetKernelArg(kernel, i++, sizeof(cl_int), &step);
663 
664         //6. ejecutando el kernel
665 
666         //U*b
667         clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
668             &globalWorkSize, &localWorkSize, 0, NULL,
669             NULL);
670 
671         //L*b
672         step++;
673         clSetKernelArg(kernel, 6, sizeof(cl_int), &step);
674         clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
675             &globalWorkSize, &localWorkSize, 0, NULL,
676             NULL);
677 
678 
679         //7. devolviendo el control de memoria af::array a ArrayFire
680         af_unlock_array(elmA);
681         af_unlock_array(idxA);
682         af_unlock_array(dB);
683         af_unlock_array(c);
684 
685         //copiando al argumento de salida
686         af_copy_array(dC, c);
687 
688         af_release_array(c);
689     }
690     //--------------
691     //Algebra Lineal
692     //--------------
693 
694     void AFire::SELgj_f(af_array* dC, af_array dA, af_array dB) {
695         //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
696         //cl_context af_context;
697         static cl_context af_context = afcl::getContext();
698         static cl_device_id af_device_id = afcl::getDeviceId();
699         static cl_command_queue af_queue = afcl::getQueue();
700 
701         //acoplando A con B
702         af_array Ac;
703         af_join(&Ac, 1, dA, dB);
704 
```

```
705     //ya que el kernel asume la matriz como ordenada por filas
706     //debemos transponer la matriz af::array A para obtener la
707     //solución correcta, ya que ArrayFire ordena lo elementos
708     //por columna
709     af_array At;
710     af_transpose(&At, Ac, false);
711
712     dim_t _order[AF_MAX_DIMS];
713     af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], dA);
714     size_t order = _order[0];
715
716     size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
717     size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
718
719     int status = CL_SUCCESS;
720
721     af_dtype typef;
722     af_get_type(&typef, dA);
723
724     int msize = 0;
725     if (typef == f64)
726         msize = sizeof(double);
727     else if (typef == f32)
728         msize = sizeof(float);
729     else;
730
731     //obteniendo las referencias cl_mem de los objetos af::array
732     cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
733         CL_MEM_READ_WRITE, msize * order*(order + 1),
734         NULL, &status);
735     af_get_device_ptr((void**)d_A, At);
736
737     size_t program_length = strlen(GJordan_source);
738
739
740     //creando el programa, construyendo el ejecutable y extrayendo el punto ↵
          de entrada
741     // para el Kernel
742     cl_program program = clCreateProgramWithSource(af_context, 1, (const char ↵
          **)&GJordan_source, &program_length, &status);
743     status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
744
745     char* kernelName;
746     if (typef == f64)
747         kernelName = "Gauss_Jordan_f";
748     else if (typef == f32)
749         kernelName = "Gauss_Jordan_f_sp";
750     else;
751     cl_kernel kernel = clCreateKernel(program, kernelName,
752         &status);
753
754     // estableciendo los argumentos
755     int i = 0;
756     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
757     clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
758
```

```
759     for (int j = 0; j < order; j++)
760     {
761         clSetKernelArg(kernel, 2, sizeof(cl_int), &j);
762         //ejecutando el Kernel
763         clEnqueueNDRangeKernel(af_queue, kernel, 1, 0, &globalWorkSize,
              &localWorkSize,
764             0, NULL, NULL);
765     }
766
767     //devolviendo el control de memoria af::array a ArrayFire
768     af_unlock_array(At);
769
770
771     //hasta aqui At contiene en su última fila
772     //y en su diagonal principal, los valores
773     //finales que deben dividirse para obtener
774     //la solución final
775
776     //extrayendo la última fila
777     af_release_array(Ac);
778     af_index_t* indexers = 0;
779     af_create_indexers(&indexers);
780     af_set_seq_param_indexer(indexers, order, order, 1,
781         0, false);
782     af_set_seq_param_indexer(indexers, 0, order - 1, 1,
783         1, false);
784     af_index_gen(&Ac, At, 2, indexers);
785
786     //transponiendo
787     af_array Atr;
788     af_transpose(&Atr, Ac, false);
789
790     //extrayendo la diagonal
791     af_release_array(Ac);
792     af_diag_extract(&Ac, At, 0);
793
794     //dividiendo
795     af_release_array(At);
796     af_div(&At, Atr, Ac, false);
797
798     // copiando el resultado en dC
799     af_copy_array(dC, At);
800
801     af_release_array(Atr);
802     af_release_array(Ac);
803     af_release_array(At);
804     af_release_indexers(indexers);
805 }
806
807 void AFire::SELgj_c(af_array* dC, af_array dA, af_array dB) {
808     //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
809     //cl_context af_context;
810     static cl_context af_context = afcl::getContext();
811     static cl_device_id af_device_id = afcl::getDeviceId();
812     static cl_command_queue af_queue = afcl::getQueue();
813
```

```
814     //acoplando A con B
815     af_array Ac;
816     af_join(&Ac, 1, dA, dB);
817
818     dim_t _order[AF_MAX_DIMS];
819     af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], dA);
820     size_t order = _order[0];
821
822     size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
823     size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
824
825     int status = CL_SUCCESS;
826
827     af_dtype typef;
828     af_get_type(&typef, dA);
829
830     int msize = 0;
831     if (typef == f64)
832         msize = sizeof(double);
833     else if (typef == f32)
834         msize = sizeof(float);
835     else;
836
837     //obteniendo las referencias cl_mem de los objetos af::array
838     cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
839         CL_MEM_READ_WRITE, msize * order*(order + 1),
840         NULL, &status);
841     af_get_device_ptr((void**)d_A, Ac);
842
843     size_t program_length = strlen(GJordan_source);
844
845     //creando el programa, construyendo el ejecutable y extrayendo el punto ↵
          de entrada
846     // para el Kernel
847     cl_program program =
848         clCreateProgramWithSource(af_context, 1,
849         (const char **)&GJordan_source, &program_length,
850             &status);
851     status = clBuildProgram(program, 1, &af_device_id,
852         NULL, NULL, NULL);
853
854     char* kernelName;
855     if (typef == f64)
856         kernelName = "Gauss_Jordan_c";
857     else if (typef == f32)
858         kernelName = "Gauss_Jordan_c_sp";
859     else;
860     cl_kernel kernel = clCreateKernel(program, kernelName,
861         &status);
862
863     // estableciendo los argumentos
864     int i = 0;
865     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
866     clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
867
868     for (int j = 0; j < order; j++)
```

```
869     {
870         clSetKernelArg(kernel, 2, sizeof(cl_int), &j);
871         //ejecutando el Kernel
872         clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
873             &globalWorkSize, &localWorkSize, 0, NULL,
874             NULL);
875     }
876
877     //devolviendo el control de memoria af::array a ArrayFire
878     af_unlock_array(Ac);
879
880     //hasta aqui Ac contiene en su última columna
881     //y en su diagonal principal, los valores
882     //finales que deben dividirse para obtener
883     //la solución final
884
885     //extrayendo la última columna
886     af_array Au;
887     af_index_t* indexers = 0;
888     af_create_indexers(&indexers);
889     af_set_seq_param_indexer(indexers, 0, order - 1, 1,
890         0, false);
891     af_set_seq_param_indexer(indexers, order, order, 1,
892         1, false);
893     af_index_gen(&Au, Ac, 2, indexers);
894
895     //extrayendo la diagonal
896     af_array Ad;
897     af_diag_extract(&Ad, Ac, 0);
898
899     //dividiendo
900     af_release_array(Ac);
901     af_div(&Ac, Au, Ad, false);
902
903     // copiando el resultado en dC
904     af_copy_array(dC, Ac);
905
906     af_release_array(Au);
907     af_release_array(Ad);
908     af_release_array(Ac);
909     af_release_indexers(indexers);
910 }
911
912 void AFire::SELgj_fshr(af_array* dC, af_array dA,
913     af_array dB) {
914     //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
915     static cl_context af_context = afcl::getContext();
916     static cl_device_id af_device_id = afcl::getDeviceId();
917     static cl_command_queue af_queue = afcl::getQueue();
918
919     /*creando copia de B*/
920     af_array Bc;
921     af_copy_array(&Bc, dB);
922
923     /*ya que el kernel asume la matriz como ordenada por filas
924     debemos transponer la matriz af::array A para obtener la
```

```
925     solución correcta, ya que ArrayFire ordena lo elementos
926     por columna*/
927     af_array At;
928     af_transpose(&At, dA, false);
929
930     //para almacenar el resultado
931     af_array Rs;
932     af_copy_array(&Rs, dB);
933
934     dim_t Adims[AF_MAX_DIMS];
935     af_get_dims(&Adims[0], &Adims[1], &Adims[2], &Adims[3],
936         dA);
937     size_t order = Adims[0];
938
939     af_dtype typef;
940     af_get_type(&typef, dA);
941
942     int msize = 0;
943     if (typef == f64)
944         msize = sizeof(double);
945     else if (typef == f32)
946         msize = sizeof(float);
947     else;
948
949     size_t program_length = strlen(GJordan_source);
950     int status = CL_SUCCESS;
951
952     //obteniendo las referencias cl_mem de los objetos af::array
953     cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
954         CL_MEM_READ_WRITE, msize * order * order,
955         NULL, &status);
956     af_get_device_ptr((void**)d_A, At);
957
958     cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
959         CL_MEM_READ_WRITE, msize * order, NULL, &status);
960     af_get_device_ptr((void**)d_B, Bc);
961
962     cl_mem *d_C = (cl_mem*)clCreateBuffer(af_context,
963         CL_MEM_WRITE_ONLY, msize * order, NULL, &status);
964     af_get_device_ptr((void**)d_C, Rs);
965
966     //creando el programa, construyendo el ejecutable y extrayendo el punto
        de entrada
967     // para el Kernel
968     cl_program program = clCreateProgramWithSource(af_context, 1, (const char
        **)&GJordan_source, &program_length, &status);
969     status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
970
971     char* kernelName;
972     if (typef == f64)
973         kernelName = "Gauss_Jordan_fshr";
974     else if (typef == f32)
975         kernelName = "Gauss_Jordan_fshr_sp";
976     else;
977     cl_kernel kernel = clCreateKernel(program, kernelName,
978         &status);
```

```
979
980     // estableciendo los argumentos
981     int i = 0;
982     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_C);
983     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
984     clSetKernelArg(kernel, i++, sizeof(cl_mem), d_B);
985     clSetKernelArg(kernel, i++, sizeof(double)*BLOCK_SIZE*BLOCK_SIZE, 0);
986     clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
987
988     size_t localWorkSize[] = { BLOCK_SIZE,BLOCK_SIZE };
989     size_t globalWorkSize[] =
990     { shrRoundUp(localWorkSize[0],order),
991         shrRoundUp(localWorkSize[1], order) };
992
993     //ejecutando el Kernel
994     clEnqueueNDRangeKernel(af_queue, kernel, 2, 0,
995         globalWorkSize, localWorkSize,0, NULL, NULL);
996
997     //devolviendo el control de memoria af::array a ArrayFire
998     af_unlock_array(At);
999     af_unlock_array(Bc);
1000    af_unlock_array(Rs);
1001
1002    //copiando el resultado
1003    af_copy_array(dC, Rs);
1004
1005    af_release_array(At);
1006    af_release_array(Bc);
1007    af_release_array(Rs);
1008 }
1009
1010 void AFire::SELgj_f2d(af_array* dC, af_array dA,
1011     af_array dB) {
1012     //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
1013     static cl_context af_context = afcl::getContext();
1014     static cl_device_id af_device_id = afcl::getDeviceId();
1015     static cl_command_queue af_queue = afcl::getQueue();
1016
1017     /*creando copia de B*/
1018     af_array Bc;
1019     af_copy_array(&Bc, dB);
1020
1021     //para almacenar el resultado
1022     af_array Rs;
1023     af_copy_array(&Rs, dB);
1024
1025     /*ya que el kernel asume la matriz como ordenada por filas
1026     debemos transponer la matriz af::array A para obtener la
1027     solución correcta, ya que ArrayFire ordena lo elementos
1028     por columna*/
1029     af_array At;
1030     af_transpose(&At, dA, false);
1031
1032     dim_t Adims[AF_MAX_DIMS];
1033     af_get_dims(&Adims[0], &Adims[1], &Adims[2], &Adims[3],
1034         dA);
```

```
1035      size_t order = Adims[0];
1036
1037      af_dtype typef;
1038      af_get_type(&typef, dA);
1039
1040      int msize = 0;
1041      if (typef == f64)
1042          msize = sizeof(double);
1043      else if (typef == f32)
1044          msize = sizeof(float);
1045      else;
1046
1047      size_t program_length = strlen(GJordan_source);
1048      int status = CL_SUCCESS;
1049
1050      //obteniendo las referencias cl_mem de los objetos af::array
1051      cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
1052          CL_MEM_READ_WRITE, msize * order * order,
1053          NULL, &status);
1054      af_get_device_ptr((void**)d_A, At);
1055
1056      cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
1057          CL_MEM_READ_WRITE, msize * order, NULL, &status);
1058      af_get_device_ptr((void**)d_B, Bc);
1059
1060      cl_mem *d_C = (cl_mem*)clCreateBuffer(af_context,
1061          CL_MEM_WRITE_ONLY, msize * order, NULL, &status);
1062      af_get_device_ptr((void**)d_C, Rs);
1063
1064      //creando el programa, construyendo el ejecutable y extrayendo el punto ↵
            de entrada
1065      // para el Kernel
1066      cl_program program = clCreateProgramWithSource(af_context, 1, (const char ↵
            **)&GJordan_source, &program_length, &status);
1067      status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
1068
1069      char* kernelName;
1070      if (typef == f64)
1071          kernelName = "Gauss_Jordan_f2d";
1072      else if (typef == f32)
1073          kernelName = "Gauss_Jordan_f2d_sp";
1074      else;
1075      cl_kernel kernel = clCreateKernel(program, kernelName,
1076          &status);
1077
1078      // estableciendo los argumentos
1079      int i = 0;
1080      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_C);
1081      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
1082      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_B);
1083      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
1084
1085      size_t localWorkSize[] = { BLOCK_SIZE,BLOCK_SIZE };
1086      size_t globalWorkSize[] = { BLOCK_SIZE,BLOCK_SIZE };
1087
1088      //ejecutando el Kernel
```

```
1089      clEnqueueNDRangeKernel(af_queue, kernel, 2, 0,
1090          globalWorkSize, localWorkSize, 0, NULL, NULL);
1091
1092      //devolviendo el control de memoria af::array a ArrayFire
1093      af_unlock_array(At);
1094      af_unlock_array(Bc);
1095      af_unlock_array(Rs);
1096
1097      //copiando el resultado
1098      af_copy_array(dC, Rs);
1099
1100      af_release_array(At);
1101      af_release_array(Bc);
1102      af_release_array(Rs);
1103  }
1104
1105  void AFire::SELgj_c2d(af_array* dC, af_array dA,
1106      af_array dB) {
1107      //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
1108      static cl_context af_context = afcl::getContext();
1109      static cl_device_id af_device_id = afcl::getDeviceId();
1110      static cl_command_queue af_queue = afcl::getQueue();
1111
1112      /*creando copia de B*/
1113      af_array Bc;
1114      af_copy_array(&Bc, dB);
1115
1116      //para almacenar el resultado
1117      af_array Rs;
1118      af_copy_array(&Rs, dB);
1119
1120      /*creando copia de A*/
1121      af_array At;
1122      af_copy_array(&At, dA);
1123
1124      dim_t Adims[AF_MAX_DIMS];
1125      af_get_dims(&Adims[0], &Adims[1], &Adims[2], &Adims[3],
1126          dA);
1127      size_t order = Adims[0];
1128
1129      af_dtype typef;
1130      af_get_type(&typef, dA);
1131
1132      int msize = 0;
1133      if (typef == f64)
1134          msize = sizeof(double);
1135      else if (typef == f32)
1136          msize = sizeof(float);
1137      else;
1138
1139      size_t program_length = strlen(GJordan_source);
1140      int status = CL_SUCCESS;
1141
1142      //obteniendo las referencias cl_mem de los objetos af::array
1143      cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
1144          CL_MEM_READ_WRITE, msize * order * order,
```

```
1145            NULL, &status);
1146        af_get_device_ptr((void**)d_A, At);
1147
1148        cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
1149            CL_MEM_READ_WRITE, msize * order, NULL, &status);
1150        af_get_device_ptr((void**)d_B, Bc);
1151
1152        cl_mem *d_C = (cl_mem*)clCreateBuffer(af_context,
1153            CL_MEM_WRITE_ONLY, msize * order, NULL, &status);
1154        af_get_device_ptr((void**)d_C, Rs);
1155
1156        //creando el programa, construyendo el ejecutable y extrayendo el punto
              de entrada
1157        // para el Kernel
1158        cl_program program = clCreateProgramWithSource(af_context, 1, (const char
              **)&GJordan_source, &program_length, &status);
1159        status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
1160
1161        char* kernelName;
1162        if (typef == f64)
1163            kernelName = "Gauss_Jordan_c2d";
1164        else if (typef == f32)
1165            kernelName = "Gauss_Jordan_c2d_sp";
1166        else;
1167        cl_kernel kernel = clCreateKernel(program, kernelName,
1168            &status);
1169
1170        // estableciendo los argumentos
1171        int i = 0;
1172        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_C);
1173        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
1174        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_B);
1175        clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
1176
1177        size_t localWorkSize[] = { BLOCK_SIZE,BLOCK_SIZE };
1178        size_t globalWorkSize[] = { BLOCK_SIZE,BLOCK_SIZE };
1179
1180        //ejecutando el Kernel
1181        clEnqueueNDRangeKernel(af_queue, kernel, 2, 0,
1182            globalWorkSize, localWorkSize, 0, NULL, NULL);
1183
1184        //devolviendo el control de memoria af::array a ArrayFire
1185        af_unlock_array(At);
1186        af_unlock_array(Bc);
1187        af_unlock_array(Rs);
1188
1189        //copiando el resultado
1190        af_copy_array(dC, Rs);
1191
1192        af_release_array(At);
1193        af_release_array(Bc);
1194        af_release_array(Rs);
1195    }
1196
1197    void AFire::prueba_shr(af_array* dC, af_array dA,
1198        af_array dB) {
```

```
1199      //Obteniendo el dispositivo, contexto y la cola usada
1200      //por ArrayFire
1201      static cl_context af_context = afcl::getContext();
1202      static cl_device_id af_device_id = afcl::getDeviceId();
1203      static cl_command_queue af_queue = afcl::getQueue();
1204
1205      dim_t Adims[AF_MAX_DIMS];
1206      af_get_dims(&Adims[0], &Adims[1], &Adims[2], &Adims[3], dA);
1207      size_t order = Adims[0];
1208
1209      af_dtype typef;
1210      af_get_type(&typef, dA);
1211
1212      int msize = 0;
1213      if (typef == f64)
1214          msize = sizeof(double);
1215      else if (typef == f32)
1216          msize = sizeof(float);
1217      else;
1218
1219      size_t program_length = strlen(GJordan_source);
1220      int status = CL_SUCCESS;
1221
1222      //creando copia de dA y dB, para poder modificarlos
1223      af_array Bc;
1224      af_copy_array(&Bc, dB);
1225      af_array Ac;
1226      af_copy_array(&Ac, dA);
1227
1228      //para almacenar el resultado
1229      af_array Rs;
1230      af_copy_array(&Rs, dB);
1231
1232      //obteniendo las referencias cl_mem de los objetos af::array
1233      cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
1234          CL_MEM_READ_WRITE, msize * order * order,
1235          NULL, &status);
1236      af_get_device_ptr((void**)d_A, Ac);
1237
1238      cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
1239          CL_MEM_READ_WRITE, msize * order, NULL, &status);
1240      af_get_device_ptr((void**)d_B, Bc);
1241
1242      cl_mem *d_C = (cl_mem*)clCreateBuffer(af_context,
1243          CL_MEM_WRITE_ONLY, msize * order, NULL, &status);
1244      af_get_device_ptr((void**)d_C, Rs);
1245
1246
1247      //creando el programa, construyendo el ejecutable y extrayendo el punto
        de entrada
1248      // para el Kernel
1249      cl_program program = clCreateProgramWithSource(af_context, 1, (const char
        **)&GJordan_source, &program_length, &status);
1250      status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
1251
1252      char* kernelName;
```

```
1253      if (typef == f64)
1254          kernelName = "prueba_shr";
1255      else if (typef == f32)
1256          kernelName = "prueba_shr_sp";
1257      else;
1258      cl_kernel kernel = clCreateKernel(program, kernelName,
1259          &status);
1260
1261      // estableciendo los argumentos
1262      int i = 0;
1263      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_C);
1264      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
1265      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_B);
1266      clSetKernelArg(kernel, i++, sizeof(double)*BLOCK_SIZE*BLOCK_SIZE, 0);
1267      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
1268
1269      size_t localWorkSize[] = { BLOCK_SIZE, BLOCK_SIZE };
1270      size_t globalWorkSize[] =
1271          { shrRoundUp(localWorkSize[0], order),shrRoundUp(localWorkSize[1],
              order) };
1272      //ejecutando el Kernel
1273      clEnqueueNDRangeKernel(af_queue, kernel, 2, 0,
1274          globalWorkSize, localWorkSize, 0, NULL, NULL);
1275
1276      //devolviendo el control de memoria af::array a ArrayFire
1277      af_unlock_array(Ac);
1278      af_unlock_array(Bc);
1279      af_unlock_array(Rs);
1280
1281      //copiando el resultado
1282      af_copy_array(dC, Rs);
1283
1284      af_release_array(Ac);
1285      af_release_array(Bc);
1286      af_release_array(Rs);
1287  }
1288
1289  void AFire::SEL_gc(af_array* C, af_array A, af_array  ,
1290      double Ierr) {
1291
1292      dim_t _order[AF_MAX_DIMS];
1293      af_get_dims(&_order[0], &_order[1], &_order[2],
1294          &_order[3], A);
1295      size_t order = _order[0];
1296
1297      af_dtype typef;
1298      af_get_type(&typef, A);
1299
1300      double normr;
1301      //af_array de ayuda
1302      af_array zero;
1303      dim_t d_order[] = { 1 };
1304      af_constant(&zero, 0, 1, d_order, typef);
1305      af_array Ax0;
1306      af_array rtxp;
1307      af_array ptxz;
```

```
1308      af_array axp;
1309      af_array B;
1310      af_array axz;
1311      af_array copyr;
1312      af_array rtxz;
1313      af_array rsp;
1314      af_array Bxp;
1315
1316      //x0=b
1317      af_array x0;
1318      af_copy_array(&x0,  );
1319
1320      //r=b-A*x0
1321      af_array r;
1322      af_matmul(&Ax0, A, x0, AF_MAT_NONE, AF_MAT_NONE);
1323      af_sub(&r,  , Ax0, false);
1324
1325      //p = r
1326      af_array p;
1327      af_copy_array(&p, r);
1328
1329      //z=A*p
1330      af_array z;
1331      af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
1332
1333      //a = (r'*p)/(p'*z)
1334      af_array a;
1335      af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
1336      af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1337      af_div(&a, rtxp, ptxz, false);
1338
1339      //x = x0 + a.*p
1340      af_array x;
1341      af_mul(&axp, a, p, true);
1342      af_add(&x, x0, axp, false);
1343
1344      int contin;
1345      int I = 0;
1346      for (int i = 0; i < order; i++)
1347      {
1348          //r -= a.*z
1349          af_copy_array(&copyr, r);
1350          af_mul(&axz, a, z, true);
1351          af_sub(&r, copyr, axz, false);
1352
1353          af_norm(&normr, r, AF_NORM_EUCLID, 1, 1);
1354          if (normr <= Ierr)
1355              break;
1356
1357          //B = -(r'*z)/(p'*z)
1358          af_matmul(&rtxz, r, z, AF_MAT_TRANS, AF_MAT_NONE);
1359          af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1360          af_div(&rsp, rtxz, ptxz, false);
1361          af_sub(&B, zero, rsp, false);
1362
1363          //p = r + B.*p
```

```
1364            af_mul(&Bxp, B, p, true);
1365            af_add(&p, r, Bxp, false);
1366
1367            //z = A*p
1368            af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
1369
1370            //a = (r'*p)/(p'*z)
1371            af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
1372            af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1373            af_div(&a, rtxp, ptxz, false);
1374
1375            //x += a.*p
1376            af_mul(&axp, a, p, true);
1377            af_copy_array(&x0, x);
1378            af_add(&x, x0, axp, false);
1379
1380            I++;
1381        }
1382
1383        //copiando el resultado en el argumento de salida
1384        af_copy_array(C, x);
1385
1386        //liberando objetos af_array usados
1387        af_release_array(zero);
1388        af_release_array(Ax0);
1389        af_release_array(rtxp);
1390        af_release_array(ptxz);
1391        af_release_array(axp);
1392        af_release_array(B);
1393        af_release_array(axz);
1394        af_release_array(copyr);
1395        af_release_array(rtxz);
1396        af_release_array(rsp);
1397        af_release_array(Bxp);
1398        af_release_array(x0);
1399        af_release_array(r);
1400        af_release_array(p);
1401        af_release_array(z);
1402        af_release_array(a);
1403        af_release_array(x);
1404    }
1405
1406    void AFire::SELgc_sparse(af_array* C, af_array elmA,
1407        af_array colA, af_array rowA, af_array  ,
1408        double Ierr) {
1409
1410        dim_t _order[AF_MAX_DIMS];
1411        af_get_dims(&_order[0], &_order[1], &_order[2],
1412            &_order[3], colA);
1413        size_t order = _order[0];
1414
1415        af_dtype typef;
1416        af_get_type(&typef, elmA);
1417
1418        double normr;
1419        //af_array de ayuda
```

```
1420      af_array zero;
1421      dim_t d_order[] = { 1 };
1422      af_constant(&zero, 0, 1, d_order, typef);
1423      af_array Ax0;
1424      af_array rtxp;
1425      af_array ptxz;
1426      af_array axp;
1427      af_array B;
1428      af_array axz;
1429      af_array copyr;
1430      af_array rtxz;
1431      af_array rsp;
1432      af_array Bxp;
1433
1434      //x0=b
1435      af_array x0;
1436      af_copy_array(&x0,  );
1437
1438      //r=b-A*x0
1439      af_array r;
1440      //af_matmul(&Ax0, A, x0, AF_MAT_NONE, AF_MAT_NONE);
1441      AFire::sparse_mat_vec_mul(&Ax0, elmA, colA, rowA, x0);
1442      af_sub(&r,  , Ax0, false);
1443
1444      //p = r
1445      af_array p;
1446      af_copy_array(&p, r);
1447
1448      //z=A*p
1449      af_array z;
1450      //af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
1451      AFire::sparse_mat_vec_mul(&z, elmA, colA, rowA, p);
1452
1453      //a = (r'*p)/(p'*z)
1454      af_array a;
1455      af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
1456      af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1457      af_div(&a, rtxp, ptxz, false);
1458
1459      //x = x0 + a.*p
1460      af_array x;
1461      af_mul(&axp, a, p, true);
1462      af_add(&x, x0, axp, false);
1463
1464      int contin;
1465      int I = 0;
1466      for (int i = 0; i < order; i++)
1467      {
1468          //r -= a.*z
1469          af_copy_array(&copyr, r);
1470          af_mul(&axz, a, z, true);
1471          af_sub(&r, copyr, axz, false);
1472
1473          af_norm(&normr, r, AF_NORM_EUCLID, 1, 1);
1474          if (normr <= Ierr)
1475              break;
```

```
1476
1477            //B = -(r'*z)/(p'*z)
1478            af_matmul(&rtxz, r, z, AF_MAT_TRANS, AF_MAT_NONE);
1479            af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1480            af_div(&rsp, rtxz, ptxz, false);
1481            af_sub(&B, zero, rsp, false);
1482
1483            //p = r + B.*p
1484            af_mul(&Bxp, B, p, true);
1485            af_add(&p, r, Bxp, false);
1486
1487            //z = A*p
1488            //af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
1489            AFire::sparse_mat_vec_mul(&z, elmA, colA, rowA, p);
1490
1491            //a = (r'*p)/(p'*z)
1492            af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
1493            af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1494            af_div(&a, rtxp, ptxz, false);
1495
1496            //x += a.*p
1497            af_mul(&axp, a, p, true);
1498            af_copy_array(&x0, x);
1499            af_add(&x, x0, axp, false);
1500
1501            I++;
1502        }
1503
1504        //copiando el resultado en el argumento de salida
1505        af_copy_array(C, x);
1506
1507        //liberando objetos af_array usados
1508        af_release_array(zero);
1509        af_release_array(Ax0);
1510        af_release_array(rtxp);
1511        af_release_array(ptxz);
1512        af_release_array(axp);
1513        af_release_array(B);
1514        af_release_array(axz);
1515        af_release_array(copyr);
1516        af_release_array(rtxz);
1517        af_release_array(rsp);
1518        af_release_array(Bxp);
1519        af_release_array(x0);
1520        af_release_array(r);
1521        af_release_array(p);
1522        af_release_array(z);
1523        af_release_array(a);
1524        af_release_array(x);
1525    }
1526
1527    void AFire::SELgc_sparse_sks(af_array* C, af_array elmA,
1528        af_array idxA, af_array  , double Ierr) {
1529
1530        dim_t _order[AF_MAX_DIMS];
1531        af_get_dims(&_order[0], &_order[1], &_order[2],
```

```
1532          &_order[3], idxA);
1533      size_t order = _order[0];
1534
1535      af_dtype typef;
1536      af_get_type(&typef, elmA);
1537
1538      double normr;
1539      //af_array de ayuda
1540      af_array zero;
1541      dim_t d_order[] = { 1 };
1542      af_constant(&zero, 0, 1, d_order, typef);
1543      af_array Ax0;
1544      af_array rtxp;
1545      af_array ptxz;
1546      af_array axp;
1547      af_array B;
1548      af_array axz;
1549      af_array copyr;
1550      af_array rtxz;
1551      af_array rsp;
1552      af_array Bxp;
1553
1554      //x0=b
1555      af_array x0;
1556      af_copy_array(&x0,  );
1557
1558      //r=b-A*x0
1559      af_array r;
1560      //af_matmul(&Ax0, A, x0, AF_MAT_NONE, AF_MAT_NONE);
1561      AFire::sparse_sks_mat_vec_mul(&Ax0, elmA, idxA, x0);
1562      af_sub(&r,  , Ax0, false);
1563
1564      //p = r
1565      af_array p;
1566      af_copy_array(&p, r);
1567
1568      //z=A*p
1569      af_array z;
1570      //af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
1571      AFire::sparse_sks_mat_vec_mul(&z, elmA, idxA, p);
1572
1573      //a = (r'*p)/(p'*z)
1574      af_array a;
1575      af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
1576      af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1577      af_div(&a, rtxp, ptxz, false);
1578
1579      //x = x0 + a.*p
1580      af_array x;
1581      af_mul(&axp, a, p, true);
1582      af_add(&x, x0, axp, false);
1583
1584      int contin;
1585      int I = 0;
1586      for (int i = 0; i < order; i++)
1587      {
```

```
1588            //r -= a.*z
1589            af_copy_array(&copyr, r);
1590            af_mul(&axz, a, z, true);
1591            af_sub(&r, copyr, axz, false);
1592
1593            af_norm(&normr, r, AF_NORM_EUCLID, 1, 1);
1594            if (normr <= Ierr)
1595                break;
1596
1597            //B = -(r'*z)/(p'*z)
1598            af_matmul(&rtxz, r, z, AF_MAT_TRANS, AF_MAT_NONE);
1599            af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1600            af_div(&rsp, rtxz, ptxz, false);
1601            af_sub(&B, zero, rsp, false);
1602
1603            //p = r + B.*p
1604            af_mul(&Bxp, B, p, true);
1605            af_add(&p, r, Bxp, false);
1606
1607            //z = A*p
1608            //af_matmul(&z, A, p, AF_MAT_NONE, AF_MAT_NONE);
1609            AFire::sparse_sks_mat_vec_mul(&z, elmA, idxA, p);
1610
1611            //a = (r'*p)/(p'*z)
1612            af_matmul(&rtxp, r, p, AF_MAT_TRANS, AF_MAT_NONE);
1613            af_matmul(&ptxz, p, z, AF_MAT_TRANS, AF_MAT_NONE);
1614            af_div(&a, rtxp, ptxz, false);
1615
1616            //x += a.*p
1617            af_mul(&axp, a, p, true);
1618            af_copy_array(&x0, x);
1619            af_add(&x, x0, axp, false);
1620
1621            I++;
1622        }
1623
1624        //copiando el resultado en el argumento de salida
1625        af_copy_array(C, x);
1626
1627        //liberando objetos af_array usados
1628        af_release_array(zero);
1629        af_release_array(Ax0);
1630        af_release_array(rtxp);
1631        af_release_array(ptxz);
1632        af_release_array(axp);
1633        af_release_array(B);
1634        af_release_array(axz);
1635        af_release_array(copyr);
1636        af_release_array(rtxz);
1637        af_release_array(rsp);
1638        af_release_array(Bxp);
1639        af_release_array(x0);
1640        af_release_array(r);
1641        af_release_array(p);
1642        af_release_array(z);
1643        af_release_array(a);
```

```
1644      af_release_array(x);
1645  }
1646
1647  array AFire::SEL_gc(array A, array  , double Ierr){
1648
1649      bool gfor_stat = gforGet();
1650      gforSet(true);
1651      size_t order = A.dims(0);
1652
1653      array x0 =  ;
1654      array r = b - matmul(A,x0);
1655      array p = r;
1656      array z = matmul(A, p);
1657      array a = matmul(r, p, AF_MAT_TRANS) /
1658          matmul(p, z, AF_MAT_TRANS);
1659      array x = x0 + a * p;
1660      array B;
1661
1662      int I = 0;
1663      for (int i = 0; i < order; i++)
1664      {
1665          r -= a * z;
1666          if (norm(r) <= Ierr)
1667              break;
1668          B = -matmul(r, z, AF_MAT_TRANS) /
1669              matmul(p, z, AF_MAT_TRANS);
1670          p = r + B * p;
1671          z = matmul(A, p);
1672          a = matmul(r, p, AF_MAT_TRANS) /
1673              matmul(p, z, AF_MAT_TRANS);
1674          x += a * p;
1675          I++;
1676      }
1677      gforSet(gfor_stat);
1678      return x;
1679  }
1680
1681  void AFire::SELchol_c(af_array* dC, af_array dA, af_array dB) {
1682      //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
1683      //cl_context af_context;
1684      static cl_context af_context = afcl::getContext();
1685      static cl_device_id af_device_id = afcl::getDeviceId();
1686      static cl_command_queue af_queue = afcl::getQueue();
1687
1688      //copia de A y B
1689      af_array Ac;
1690      af_array Bc;
1691      af_copy_array(&Ac, dA);
1692      af_copy_array(&Bc, dB);
1693
1694      dim_t _order[AF_MAX_DIMS];
1695      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], dA);
1696      size_t order = _order[0];
1697
1698      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
1699      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
```

```
1700
1701        int status = CL_SUCCESS;
1702
1703        af_dtype typef;
1704        af_get_type(&typef, dA);
1705
1706        int msize = 0;
1707        if (typef == f64)
1708            msize = sizeof(double);
1709        else if (typef == f32)
1710            msize = sizeof(float);
1711        else;
1712
1713        //obteniendo las referencias cl_mem de los objetos af::array
1714        cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
1715            CL_MEM_READ_WRITE, msize*order*order,
1716            NULL, &status);
1717        af_get_device_ptr((void**)d_A, Ac);
1718
1719        cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
1720            CL_MEM_READ_WRITE, msize*order,
1721            NULL, &status);
1722        af_get_device_ptr((void**)d_B, Bc);
1723
1724        size_t program_length = strlen(Cholesky_source);
1725
1726
1727        //creando el programa, construyendo el ejecutable y extrayendo el punto
              de entrada
1728        // para el Kernel
1729        cl_program program = clCreateProgramWithSource(af_context, 1, (const char
              **)&Cholesky_source, &program_length, &status);
1730        status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
1731
1732        char* kernelName;
1733        if (typef == f64)
1734            kernelName = "Cholesky_c";
1735        else if (typef == f32)
1736            kernelName = "Cholesky_c_sp";
1737        else;
1738        cl_kernel kernel = clCreateKernel(program, kernelName,
1739            &status);
1740
1741        // estableciendo los argumentos
1742        int i = 0;
1743        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
1744        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_B);
1745        clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
1746        clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
1747
1748        for (int j = 0; j < order; j++)
1749        {
1750            //Se modifican los elementos de la columna j a partir
1751            //de la fila j + 1.
1752            //A(n, j) = A(n, j) / sqrt(A(j, j)) n > j
1753            int sstep = 0;
```

```
1754            clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
1755            clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
1756            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1757                &globalWorkSize, &localWorkSize, 0, NULL,
1758                NULL);
1759
1760            //para una columna s, s > j, se modifican las filas n >= s
1761            //A(n, s) = A(n, s) - A(s, j)*A(n, j) s > j, n >= s
1762            sstep++;
1763            clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
1764            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1765                &globalWorkSize, &localWorkSize, 0, NULL,
1766                NULL);
1767        }
1768
1769        //cada elemento diagonal es reemplazado por su raiz cuadrada
1770        //asi termina la factorización, la matriz Ac, tendra en su
1771        //parte triangular inferior, los elementos de la factorización
1772        //de cholesky
1773        int fstep = 2;
1774        clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
1775        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1776            &globalWorkSize, &localWorkSize, 0, NULL,
1777            NULL);
1778
1779        //luego de la factorización se tiene el sistema equivalente
1780        //L*transpose(L)x=b que debe resolverse separadamente para
1781        //Ly=b
1782        //y luego
1783        //transpose(L)x=y
1784
1785        //resolviendo Ly=b
1786        fstep++;
1787        clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
1788        for (int j = 0; j < order; j++)
1789        {
1790            //para un paso j, modificará el vector b tal que :
1791            //b[n] = b[n] - b[j] * L[n, j] / L[j, j] n > j
1792            clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
1793            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1794                &globalWorkSize, &localWorkSize, 0, NULL,
1795                NULL);
1796        }
1797
1798        fstep++;
1799        clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
1800        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1801            &globalWorkSize, &localWorkSize, 0, NULL,
1802            NULL);
1803
1804        //resolviendo transpose(L)x=y
1805        fstep++;
1806        clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
1807        for (int j = 0; j < order; j++)
1808        {
1809            //para un paso j, modifica b[order-j], este valor
```

```
1810            //es un componente final de la solución del sistema
1811            clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
1812            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1813                &globalWorkSize, &localWorkSize, 0, NULL,
1814                NULL);
1815        }
1816
1817        //devolviendo el control de memoria af::array a ArrayFire
1818        af_unlock_array(Ac);
1819        af_unlock_array(Bc);
1820
1821        // copiando el resultado en dC
1822        af_copy_array(dC, Bc);
1823
1824        af_release_array(Ac);
1825        af_release_array(Bc);
1826    }
1827
1828    void AFire::fac_chol_c(af_array* L, af_array A) {
1829        //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
1830        //cl_context af_context;
1831        static cl_context af_context = afcl::getContext();
1832        static cl_device_id af_device_id = afcl::getDeviceId();
1833        static cl_command_queue af_queue = afcl::getQueue();
1834
1835        //2. Obteniendo parámetros necesarios
1836        //copia de A
1837        af_array Ac;
1838        af_copy_array(&Ac, A);
1839
1840        dim_t _order[AF_MAX_DIMS];
1841        af_get_dims(&_order[0], &_order[1], &_order[2],
1842            &_order[3], A);
1843        size_t order = _order[0];
1844
1845        size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
1846        size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
1847
1848        int status = CL_SUCCESS;
1849
1850        af_dtype typef;
1851        af_get_type(&typef, A);
1852
1853        int msize = 0;
1854        if (typef == f64)
1855            msize = sizeof(double);
1856        else if (typef == f32)
1857            msize = sizeof(float);
1858        else;
1859
1860        //3.obteniendo las referencias cl_mem de los objetos af::array
1861        cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
1862            CL_MEM_READ_WRITE, msize*order*order,
1863            NULL, &status);
1864        af_get_device_ptr((void**)d_A, Ac);
1865
```

```
1866      size_t program_length = strlen(Cholesky_source);
1867
1868      //4.creando el programa, construyendo el ejecutable y extrayendo el punto
             de entrada
1869      // para el Kernel
1870      cl_program program = clCreateProgramWithSource(af_context, 1, (const char
             **)&Cholesky_source, &program_length, &status);
1871      status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
1872
1873      char* kernelName;
1874      if (typef == f64)
1875          kernelName = "Cholesky_c";
1876      else if (typef == f32)
1877          kernelName = "Cholesky_c_sp";
1878      else;
1879      cl_kernel kernel = clCreateKernel(program, kernelName,
1880          &status);
1881
1882      // 5.estableciendo los argumentos
1883      int i = 0;
1884      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
1885      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
1886      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
1887      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
1888
1889      //6.Ejecutando el kernel
1890      for (int j = 0; j < order; j++)
1891      {
1892          //Se modifican los elementos de la columna j a partir
1893          //de la fila j + 1.
1894          //A(n, j) = A(n, j) / sqrt(A(j, j)) n > j
1895          int sstep = 0;
1896          clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
1897          clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
1898          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1899              &globalWorkSize, &localWorkSize, 0, NULL,
1900              NULL);
1901
1902          //para una columna s, s > j, se modifican las filas n >= s
1903          //A(n, s) = A(n, s) - A(s, j)*A(n, j) s > j, n >= s
1904          sstep++;
1905          clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
1906          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1907              &globalWorkSize, &localWorkSize, 0, NULL,
1908              NULL);
1909      }
1910
1911      //cada elemento diagonal es reemplazado por su raiz cuadrada
1912      //asi termina la factorización, la matriz Ac, tendra en su
1913      //parte triangular inferior, los elementos de la factorización
1914      //de cholesky
1915      int fstep = 2;
1916      clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
1917      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
1918          &globalWorkSize, &localWorkSize, 0, NULL,
1919          NULL);
```

```
1920
1921        //7. devolviendo el control de memoria af::array a ArrayFire
1922        af_unlock_array(Ac);
1923
1924        // copiando el resultado en dC
1925        af_copy_array(L, Ac);
1926
1927        af_release_array(Ac);
1928    }
1929
1930    void AFire::SELldlt_c(af_array* dC, af_array dA, af_array dB) {
1931        //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
1932        //cl_context af_context;
1933        static cl_context af_context = afcl::getContext();
1934        static cl_device_id af_device_id = afcl::getDeviceId();
1935        static cl_command_queue af_queue = afcl::getQueue();
1936
1937        //copia de A y B
1938        af_array Ac;
1939        af_array Bc;
1940        af_copy_array(&Ac, dA);
1941        af_copy_array(&Bc, dB);
1942
1943        dim_t _order[AF_MAX_DIMS];
1944        af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], dA);
1945        size_t order = _order[0];
1946
1947        size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
1948        size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
1949
1950        int status = CL_SUCCESS;
1951
1952        af_dtype typef;
1953        af_get_type(&typef, dA);
1954
1955        int msize = 0;
1956        if (typef == f64)
1957            msize = sizeof(double);
1958        else if (typef == f32)
1959            msize = sizeof(float);
1960        else;
1961
1962        //obteniendo las referencias cl_mem de los objetos af::array
1963        cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
1964            CL_MEM_READ_WRITE, msize*order*order,
1965            NULL, &status);
1966        af_get_device_ptr((void**)d_A, Ac);
1967
1968        cl_mem *d_B = (cl_mem*)clCreateBuffer(af_context,
1969            CL_MEM_READ_WRITE, msize*order,
1970            NULL, &status);
1971        af_get_device_ptr((void**)d_B, Bc);
1972
1973        size_t program_length = strlen(ldlt_source);
1974
1975
```

```
1976      //creando el programa, construyendo el ejecutable y extrayendo el punto
            de entrada
1977      // para el Kernel
1978      cl_program program = clCreateProgramWithSource(af_context, 1, (const char
            **)&ldlt_source, &program_length, &status);
1979      status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
1980
1981      char* kernelName;
1982      if (typef == f64)
1983          kernelName = "ldlt_c";
1984      else if (typef == f32)
1985          kernelName = "ldlt_c_sp";
1986      else;
1987      cl_kernel kernel = clCreateKernel(program, kernelName,
1988          &status);
1989
1990      // estableciendo los argumentos
1991      int i = 0;
1992      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
1993      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_B);
1994      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
1995      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
1996
1997      for (int j = 0; j < order; j++)
1998      {
1999          //Se modifican los elementos de la columna j a partir
2000          //de la fila j + 1.
2001          //A(n,j)=A(n,j)/A(j,j) n>j
2002          int sstep = 0;
2003          clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
2004          clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
2005          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2006              &globalWorkSize, &localWorkSize, 0, NULL,
2007              NULL);
2008
2009          //para una columna s, s > j, se modifican las filas
2010          //n >= s
2011          //A(n, s) = A(n, s) - A(s, j)*A(n, j)*A(j, j)
2012          //s > j, n >= s
2013          sstep++;
2014          clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
2015          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2016              &globalWorkSize, &localWorkSize, 0, NULL,
2017              NULL);
2018      }
2019
2020      //luego de la factorización se tiene el sistema equivalente
2021      //L*D*transpose(L)x=b que debe resolverse separadamente para
2022      //Ly=b, luego para Dz=y y finalmente para transpose(L)x=z
2023
2024      //resolviendo Ly=b
2025      int fstep = 2;
2026      clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
2027      for (int j = 0; j < order; j++)
2028      {
2029          //para un paso j, modificará el vector b tal que :
```

```
2030            //b[n] = b[n] - b[j] * L[n, j] n > j
2031            clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
2032            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2033                &globalWorkSize, &localWorkSize, 0, NULL,
2034                NULL);
2035        }
2036
2037        //resolviendo Dz=y
2038        fstep++;
2039        clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
2040        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2041            &globalWorkSize, &localWorkSize, 0, NULL,
2042            NULL);
2043
2044        //resolviendo transpose(L)x=z
2045        fstep++;
2046        clSetKernelArg(kernel, 5, sizeof(cl_int), &fstep);
2047        for (int j = 0; j < order; j++)
2048        {
2049            //para un paso j modificará el elemento b[i]
2050            //tal que i = n - j:
2051            //b[i] = b[i] - sumatoria{ j = i + 1,n }(z[j] * L[j, i])
2052            //este valor será la solución final x[i] del
2053            //sistema Ax = b
2054            clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
2055            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2056                &globalWorkSize, &localWorkSize, 0, NULL,
2057                NULL);
2058        }
2059
2060        //devolviendo el control de memoria af::array a ArrayFire
2061        af_unlock_array(Ac);
2062        af_unlock_array(Bc);
2063
2064        // copiando el resultado en dC
2065        af_copy_array(dC, Bc);
2066
2067        af_release_array(Ac);
2068        af_release_array(Bc);
2069    }
2070
2071    void AFire::fac_ldlt_c(af_array* L, af_array A) {
2072        //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
2073        //cl_context af_context;
2074        static cl_context af_context = afcl::getContext();
2075        static cl_device_id af_device_id = afcl::getDeviceId();
2076        static cl_command_queue af_queue = afcl::getQueue();
2077
2078        //2. Obteniendo parámetros necesarios
2079        //copia de A
2080        af_array Ac;
2081        af_copy_array(&Ac, A);
2082
2083        dim_t _order[AF_MAX_DIMS];
2084        af_get_dims(&_order[0], &_order[1], &_order[2],
2085            &_order[3], A);
```

```
2086      size_t order = _order[0];
2087
2088      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
2089      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
2090
2091      int status = CL_SUCCESS;
2092
2093      af_dtype typef;
2094      af_get_type(&typef, A);
2095
2096      int msize = 0;
2097      if (typef == f64)
2098          msize = sizeof(double);
2099      else if (typef == f32)
2100          msize = sizeof(float);
2101      else;
2102
2103      //3.obteniendo las referencias cl_mem de los objetos af::array
2104      cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
2105          CL_MEM_READ_WRITE, msize*order*order,
2106          NULL, &status);
2107      af_get_device_ptr((void**)d_A, Ac);
2108
2109      size_t program_length = strlen(ldlt_source);
2110
2111      //4.creando el programa, construyendo el ejecutable y extrayendo el punto
          de entrada
2112      // para el Kernel
2113      cl_program program = clCreateProgramWithSource(af_context, 1, (const char
          **)&ldlt_source, &program_length, &status);
2114      status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
2115
2116      char* kernelName;
2117      if (typef == f64)
2118          kernelName = "ldlt_c";
2119      else if (typef == f32)
2120          kernelName = "ldlt_c_sp";
2121      else;
2122      cl_kernel kernel = clCreateKernel(program, kernelName,
2123          &status);
2124
2125      // 5.estableciendo los argumentos
2126      int i = 0;
2127      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
2128      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
2129      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
2130      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
2131
2132      //6.Ejecutando el kernel
2133      for (int j = 0; j < order; j++)
2134      {
2135          //Se modifican los elementos de la columna j a partir
2136          //de la fila j + 1.
2137          //A(n,j)=A(n,j)/A(j,j) n>j
2138          int sstep = 0;
2139          clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
```

```
2140            clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
2141            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2142                &globalWorkSize, &localWorkSize, 0, NULL,
2143                NULL);
2144
2145            //para una columna s, s > j, se modifican las filas
2146            //n >= s
2147            //A(n, s) = A(n, s) - A(s, j)*A(n, j)*A(j, j)
2148            //s > j, n >= s
2149            sstep++;
2150            clSetKernelArg(kernel, 5, sizeof(cl_int), &sstep);
2151            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2152                &globalWorkSize, &localWorkSize, 0, NULL,
2153                NULL);
2154        }
2155
2156        //7. devolviendo el control de memoria af::array a ArrayFire
2157        af_unlock_array(Ac);
2158
2159        // copiando el resultado en dC
2160        af_copy_array(L, Ac);
2161
2162        af_release_array(Ac);
2163    }
2164
2165    void AFire::fac_sparse_chol_c(af_array elmA, af_array colA,
2166        af_array rowA, af_array elmL, af_array colL,
2167        af_array rowL) {
2168        //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
2169        //cl_context af_context;
2170        static cl_context af_context = afcl::getContext();
2171        static cl_device_id af_device_id = afcl::getDeviceId();
2172        static cl_command_queue af_queue = afcl::getQueue();
2173
2174        //2. Obteniendo parámetros necesarios
2175
2176        //longitud de los vectores
2177        dim_t _order[AF_MAX_DIMS];
2178        af_get_dims(&_order[0], &_order[1], &_order[2],
2179            &_order[3], elmA);
2180        size_t size_elmA = _order[0];
2181
2182        af_get_dims(&_order[0], &_order[1], &_order[2],
2183            &_order[3], colA);
2184        size_t size_colA = _order[0];
2185
2186        af_get_dims(&_order[0], &_order[1], &_order[2],
2187            &_order[3], rowA);
2188        size_t size_rowA = _order[0];
2189
2190        af_get_dims(&_order[0], &_order[1], &_order[2],
2191            &_order[3], elmL);
2192        size_t size_elmL = _order[0];
2193
2194        af_get_dims(&_order[0], &_order[1], &_order[2],
2195            &_order[3], colL);
```

```
2196      size_t size_colL = _order[0];
2197
2198      af_get_dims(&_order[0], &_order[1], &_order[2],
2199          &_order[3], rowL);
2200      size_t size_rowL = _order[0];
2201
2202      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
2203      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
2204
2205      int status = CL_SUCCESS;
2206
2207      af_dtype typef;
2208      af_get_type(&typef, elmA);
2209
2210      int msize = 0;
2211      if (typef == f64)
2212          msize = sizeof(double);
2213      else if (typef == f32)
2214          msize = sizeof(float);
2215      else;
2216
2217      //3.obteniendo las referencias cl_mem de los objetos af::array
2218      cl_mem *d_elmA = (cl_mem*)clCreateBuffer(af_context,
2219          CL_MEM_READ_ONLY, msize*size_elmA,
2220          NULL, &status);
2221      af_get_device_ptr((void**)d_elmA, elmA);
2222
2223      cl_mem *d_colA = (cl_mem*)clCreateBuffer(af_context,
2224          CL_MEM_READ_ONLY, sizeof(int)*size_colA,
2225          NULL, &status);
2226      af_get_device_ptr((void**)d_colA, colA);
2227
2228      cl_mem *d_rowA = (cl_mem*)clCreateBuffer(af_context,
2229          CL_MEM_READ_ONLY, sizeof(int)*size_rowA,
2230          NULL, &status);
2231      af_get_device_ptr((void**)d_rowA, rowA);
2232
2233      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
2234          CL_MEM_READ_WRITE, msize*size_elmL,
2235          NULL, &status);
2236      af_get_device_ptr((void**)d_elmL, elmL);
2237
2238      cl_mem *d_colL = (cl_mem*)clCreateBuffer(af_context,
2239          CL_MEM_READ_WRITE, sizeof(int)*size_colL,
2240          NULL, &status);
2241      af_get_device_ptr((void**)d_colL, colL);
2242
2243      cl_mem *d_rowL = (cl_mem*)clCreateBuffer(af_context,
2244          CL_MEM_READ_WRITE, sizeof(int)*size_rowL,
2245          NULL, &status);
2246      af_get_device_ptr((void**)d_rowL, rowL);
2247
2248      size_t program_length = strlen(Cholesky_source);
2249
2250      //4.creando el programa, construyendo el ejecutable y extrayendo el punto
          de entrada
```

```
2251      // para el Kernel
2252      cl_program program = clCreateProgramWithSource(af_context, 1, (const char
             **)&Cholesky_source, &program_length, &status);
2253      status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
2254
2255      char* kernelName;
2256      if (typef == f64)
2257          kernelName = "Cholesky_sparse_c";
2258      else if (typef == f32)
2259          kernelName = "Cholesky_sparse_c_sp";
2260      else;
2261      cl_kernel kernel = clCreateKernel(program, kernelName,
2262          &status);
2263
2264      int step = 0;
2265      int sstep = -1;
2266
2267      // 5.estableciendo los argumentos
2268      int i = 0;
2269      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmA);
2270      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colA);
2271      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowA);
2272      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
2273      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colL);
2274      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowL);
2275      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_elmA);
2276      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_colA);
2277      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
2278      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
2279      clSetKernelArg(kernel, i++, sizeof(cl_int), &step);
2280      clSetKernelArg(kernel, i++, sizeof(cl_int), &sstep);
2281
2282      //6. ejecutando el kernel
2283
2284      //primera ejecución con sstep=-1 copiará los elementos
2285      //de elmA en el lugar correspondiente de elmL, tomar
2286      //en cuenta que los elementos iniciales del elmL son
2287      //todos cero.
2288      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2289          &globalWorkSize, &localWorkSize, 0, NULL,
2290          NULL);
2291
2292      //6.Ejecutando el kernel
2293      for (int j = 0; j < size_colA-1; j++)
2294      {
2295          //Se modifican los elementos de la columna j a partir
2296          //de la fila j + 1.
2297          //A(n, j) = A(n, j) / sqrt(A(j, j)) n > j
2298          int sstep = 0;
2299          clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
2300          clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2301          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2302              &globalWorkSize, &localWorkSize, 0, NULL,
2303              NULL);
2304
2305          //para una columna s, s > j, se modifican las filas n >= s
```

```
2306            //A(n, s) = A(n, s) - A(s, j)*A(n, j) s > j, n >= s
2307            sstep++;
2308            clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2309            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2310                &globalWorkSize, &localWorkSize, 0, NULL,
2311                NULL);
2312        }
2313
2314        //cada elemento diagonal es reemplazado por su raiz cuadrada
2315        //asi termina la factorización, la matriz Ac, tendra en su
2316        //parte triangular inferior, los elementos de la factorización
2317        //de cholesky
2318        int fstep = 2;
2319        clSetKernelArg(kernel, 11, sizeof(cl_int), &fstep);
2320        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2321            &globalWorkSize, &localWorkSize, 0, NULL,
2322            NULL);
2323
2324        //7. devolviendo el control de memoria af::array a ArrayFire
2325        af_unlock_array(elmA);
2326        af_unlock_array(colA);
2327        af_unlock_array(rowA);
2328        af_unlock_array(elmL);
2329        af_unlock_array(colL);
2330        af_unlock_array(rowL);
2331    }
2332
2333    void AFire::SELchol_sparse_c(af_array* dC, af_array elmA,
2334        af_array colA, af_array rowA, af_array elmL,
2335        af_array colL,af_array rowL, af_array dB) {
2336
2337        AFire::fac_sparse_chol_c(elmA, colA, rowA, elmL,
2338            colL, rowL);
2339        AFire::SELchol_sparse_c(dC, elmL, colL, rowL, dB);
2340        /*//1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
2341        //cl_context af_context;
2342        static cl_context af_context = afcl::getContext();
2343        static cl_device_id af_device_id = afcl::getDeviceId();
2344        static cl_command_queue af_queue = afcl::getQueue();
2345
2346        //creando copia de dB
2347        af_array b;
2348        af_copy_array(&b, dB);
2349
2350        //2. Obteniendo parámetros necesarios
2351
2352        //longitud de los vectores
2353        dim_t _order[AF_MAX_DIMS];
2354        af_get_dims(&_order[0], &_order[1], &_order[2],
2355            &_order[3], elmA);
2356        cl_int size_elmA = _order[0];
2357
2358        af_get_dims(&_order[0], &_order[1], &_order[2],
2359            &_order[3], colA);
2360        cl_int size_colA = _order[0];
2361
```

```
2362      af_get_dims(&_order[0], &_order[1], &_order[2],
2363          &_order[3], rowA);
2364      cl_int size_rowA = _order[0];
2365
2366      af_get_dims(&_order[0], &_order[1], &_order[2],
2367          &_order[3], elmL);
2368      cl_int size_elmL = _order[0];
2369
2370      af_get_dims(&_order[0], &_order[1], &_order[2],
2371          &_order[3], colL);
2372      cl_int size_colL = _order[0];
2373
2374      af_get_dims(&_order[0], &_order[1], &_order[2],
2375          &_order[3], rowL);
2376      cl_int size_rowL = _order[0];
2377
2378      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
2379      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
2380
2381      int status = CL_SUCCESS;
2382
2383      af_dtype typef;
2384      af_get_type(&typef, elmA);
2385
2386      int msize = 0;
2387      if (typef == f64)
2388          msize = sizeof(double);
2389      else if (typef == f32)
2390          msize = sizeof(float);
2391      else;
2392
2393      //3.obteniendo las referencias cl_mem de los objetos af::array
2394      cl_mem *d_elmA = (cl_mem*)clCreateBuffer(af_context,
2395          CL_MEM_READ_ONLY, msize*size_elmA,
2396          NULL, &status);
2397      af_get_device_ptr((void**)d_elmA, elmA);
2398
2399      cl_mem *d_colA = (cl_mem*)clCreateBuffer(af_context,
2400          CL_MEM_READ_ONLY, sizeof(int)*size_colA,
2401          NULL, &status);
2402      af_get_device_ptr((void**)d_colA, colA);
2403
2404      cl_mem *d_rowA = (cl_mem*)clCreateBuffer(af_context,
2405          CL_MEM_READ_ONLY, sizeof(int)*size_rowA,
2406          NULL, &status);
2407      af_get_device_ptr((void**)d_rowA, rowA);
2408
2409      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
2410          CL_MEM_READ_WRITE, msize*size_elmL,
2411          NULL, &status);
2412      af_get_device_ptr((void**)d_elmL, elmL);
2413
2414      cl_mem *d_colL = (cl_mem*)clCreateBuffer(af_context,
2415          CL_MEM_READ_WRITE, sizeof(int)*size_colL,
2416          NULL, &status);
2417      af_get_device_ptr((void**)d_colL, colL);
```

```
2418
2419      cl_mem *d_rowL = (cl_mem*)clCreateBuffer(af_context,
2420          CL_MEM_READ_WRITE, sizeof(int)*size_rowL,
2421          NULL, &status);
2422      af_get_device_ptr((void**)d_rowL, rowL);
2423
2424      cl_mem *d_b = (cl_mem*)clCreateBuffer(af_context,
2425          CL_MEM_READ_WRITE, msize*size_colA,
2426          NULL, &status);
2427      af_get_device_ptr((void**)d_b, b);
2428
2429      size_t program_length = strlen(Cholesky_source);
2430
2431      //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
            de entrada
2432      // para el Kernel
2433      cl_program program = clCreateProgramWithSource(af_context,
2434          1, (const char **)&Cholesky_source, &program_length,
2435          &status);
2436      status = clBuildProgram(program, 1, &af_device_id,
2437          NULL, NULL, NULL);
2438
2439      char* kernelName;
2440      if (typef == f64)
2441          kernelName = "Cholesky_sparse_c";
2442      else if (typef == f32)
2443          kernelName = "Cholesky_sparse_c_sp";
2444      else;
2445      cl_kernel kernel = clCreateKernel(program, kernelName,
2446          &status);
2447
2448      cl_int step = 0;
2449      cl_int sstep = -1;
2450
2451      // 5.estableciendo los argumentos
2452      int i = 0;
2453      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmA);
2454      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colA);
2455      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowA);
2456      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
2457      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colL);
2458      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowL);
2459      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_elmA);
2460      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_colA);
2461      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_b);
2462      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
2463      clSetKernelArg(kernel, i++, sizeof(cl_int), &step);
2464      clSetKernelArg(kernel, i++, sizeof(cl_int), &sstep);
2465
2466      //6. ejecutando el kernel
2467
2468      //primera ejecución con sstep=-1 copiará los elementos
2469      //de elmA en el lugar correspondiente de elmL, tomar
2470      //en cuenta que los elementos iniciales del elmL son
2471      //todos cero.
2472      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
```

```
2473            &globalWorkSize, &localWorkSize, 0, NULL,
2474            NULL);
2475
2476        for (cl_int j = 0; j < size_colA - 1; j++)
2477        {
2478            //Se modifican los elementos de la columna j a partir
2479            //de la fila j + 1.
2480            //A(n, j) = A(n, j) / sqrt(A(j, j)) n > j
2481            sstep = 0;
2482            clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
2483            clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2484            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2485                &globalWorkSize, &localWorkSize, 0, NULL,
2486                NULL);
2487
2488            //para una columna s, s > j, se modifican las filas n >= s
2489            //A(n, s) = A(n, s) - A(s, j)*A(n, j) s > j, n >= s
2490            sstep++;
2491            clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2492            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2493                &globalWorkSize, &localWorkSize, 0, NULL,
2494                NULL);
2495        }
2496
2497        //cada elemento diagonal es reemplazado por su raiz cuadrada
2498        //asi termina la factorización, la matriz Ac, tendra en su
2499        //parte triangular inferior, los elementos de la factorización
2500        //de cholesky
2501        sstep++;
2502        clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2503        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2504            &globalWorkSize, &localWorkSize, 0, NULL,
2505            NULL);
2506
2507        //termina la factorización
2508        //resolviendo el sistema Ly=b (y=transpose(L)*x)
2509        sstep++;
2510        clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2511        for (cl_int j = 0; j < size_colA - 1; j++) {
2512            clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
2513            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2514                &globalWorkSize, &localWorkSize, 0, NULL,
2515                NULL);
2516        }
2517        sstep++;
2518        clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2519        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2520            &globalWorkSize, &localWorkSize, 0, NULL,
2521            NULL);
2522
2523        //resolviendo el sistema transpose(L)*x=y
2524        sstep++;
2525        clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2526
2527        for (cl_int j = 0; j < size_colA; j++) {
2528            clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
```

```
2529            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2530                &globalWorkSize, &localWorkSize, 0, NULL,
2531                NULL);
2532        }
2533
2534        //7. devolviendo el control de memoria af::array a ArrayFire
2535        af_unlock_array(elmA);
2536        af_unlock_array(colA);
2537        af_unlock_array(rowA);
2538        af_unlock_array(elmL);
2539        af_unlock_array(colL);
2540        af_unlock_array(rowL);
2541        af_unlock_array(b);
2542
2543        //copiando al argumento de salida
2544        af_copy_array(dC, b);
2545
2546        af_release_array(b);*/
2547    }
2548
2549    void AFire::SELchol_sparse_c(af_array* dC, af_array elmL,
2550        af_array colL, af_array rowL, af_array dB) {
2551        //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
2552        //cl_context af_context;
2553        static cl_context af_context = afcl::getContext();
2554        static cl_device_id af_device_id = afcl::getDeviceId();
2555        static cl_command_queue af_queue = afcl::getQueue();
2556
2557        //creando copia de dB
2558        af_array b;
2559        af_copy_array(&b, dB);
2560
2561        //2. Obteniendo parámetros necesarios
2562
2563        //longitud de los vectores
2564        dim_t _order[AF_MAX_DIMS];
2565        af_get_dims(&_order[0], &_order[1], &_order[2],
2566            &_order[3], elmL);
2567        size_t size_elmL = _order[0];
2568
2569        af_get_dims(&_order[0], &_order[1], &_order[2],
2570            &_order[3], colL);
2571        size_t size_colL = _order[0];
2572
2573        af_get_dims(&_order[0], &_order[1], &_order[2],
2574            &_order[3], rowL);
2575        size_t size_rowL = _order[0];
2576
2577        size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
2578        size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
2579
2580        int status = CL_SUCCESS;
2581
2582        af_dtype typef;
2583        af_get_type(&typef, elmL);
2584
```

```
2585      int msize = 0;
2586      if (typef == f64)
2587          msize = sizeof(double);
2588      else if (typef == f32)
2589          msize = sizeof(float);
2590      else;
2591
2592      //3.obteniendo las referencias cl_mem de los objetos af::array
2593      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
2594          CL_MEM_READ_WRITE, msize*size_elmL,
2595          NULL, &status);
2596      af_get_device_ptr((void**)d_elmL, elmL);
2597
2598      cl_mem *d_colL = (cl_mem*)clCreateBuffer(af_context,
2599          CL_MEM_READ_WRITE, sizeof(int)*size_colL,
2600          NULL, &status);
2601      af_get_device_ptr((void**)d_colL, colL);
2602
2603      cl_mem *d_rowL = (cl_mem*)clCreateBuffer(af_context,
2604          CL_MEM_READ_WRITE, sizeof(int)*size_rowL,
2605          NULL, &status);
2606      af_get_device_ptr((void**)d_rowL, rowL);
2607
2608      cl_mem *d_b = (cl_mem*)clCreateBuffer(af_context,
2609          CL_MEM_READ_WRITE, msize*size_colL,
2610          NULL, &status);
2611      af_get_device_ptr((void**)d_b, b);
2612
2613      size_t program_length = strlen(Cholesky_source);
2614
2615      //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
             de entrada
2616      // para el Kernel
2617      cl_program program = clCreateProgramWithSource(af_context,
2618          1, (const char **)&Cholesky_source, &program_length,
2619          &status);
2620      status = clBuildProgram(program, 1, &af_device_id,
2621          NULL, NULL, NULL);
2622
2623      char* kernelName;
2624      if (typef == f64)
2625          kernelName = "Cholesky_sparse_c";
2626      else if (typef == f32)
2627          kernelName = "Cholesky_sparse_c_sp";
2628      else;
2629      cl_kernel kernel = clCreateKernel(program, kernelName,
2630          &status);
2631
2632      int step = 0;
2633      int sstep = 0;
2634      int kk = 0;
2635      // 5.estableciendo los argumentos
2636      int i = 0;
2637      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
2638      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
2639      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
```

```
2640      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
2641      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colL);
2642      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowL);
2643      clSetKernelArg(kernel, i++, sizeof(cl_int), &kk);
2644      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_colL);
2645      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_b);
2646      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
2647      clSetKernelArg(kernel, i++, sizeof(cl_int), &step);
2648      clSetKernelArg(kernel, i++, sizeof(cl_int), &sstep);
2649
2650      //6. ejecutando el kernel
2651
2652      //resolviendo el sistema Ly=b (y=transpose(L)*x)
2653      sstep=3;
2654      clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2655      for (int j = 0; j < size_colL - 1; j++) {
2656
2657          clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
2658          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2659              &globalWorkSize, &localWorkSize, 0, NULL,
2660              NULL);
2661      }
2662      sstep++;
2663      clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2664      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2665          &globalWorkSize, &localWorkSize, 0, NULL,
2666          NULL);
2667
2668      //resolviendo el sistema transpose(L)*x=y
2669      sstep++;
2670      clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
2671
2672      for (int j = 0; j < size_colL; j++) {
2673
2674          clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
2675          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2676              &globalWorkSize, &localWorkSize, 0, NULL,
2677              NULL);
2678      }
2679
2680      //7. devolviendo el control de memoria af::array a ArrayFire
2681      af_unlock_array(elmL);
2682      af_unlock_array(colL);
2683      af_unlock_array(rowL);
2684      af_unlock_array(b);
2685
2686      //copiando al argumento de salida
2687      af_copy_array(dC, b);
2688
2689      af_release_array(b);
2690  }
2691
2692  void AFire::fac_sparse_chol_sks(af_array elmA,
2693      af_array idxA) {
2694      //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
2695      //cl_context af_context;
```

```
2696      static cl_context af_context = afcl::getContext();
2697      static cl_device_id af_device_id = afcl::getDeviceId();
2698      static cl_command_queue af_queue = afcl::getQueue();
2699
2700      //2. Obteniendo parámetros necesarios
2701      //longitud de los vectores
2702      dim_t _order[AF_MAX_DIMS];
2703      af_get_dims(&_order[0], &_order[1], &_order[2],
2704          &_order[3], elmA);
2705      size_t size_elmL = _order[0];
2706
2707      af_get_dims(&_order[0], &_order[1], &_order[2],
2708          &_order[3], idxA);
2709      size_t size_idxL = _order[0];
2710
2711      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
2712      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
2713
2714      int status = CL_SUCCESS;
2715
2716      af_dtype typef;
2717      af_get_type(&typef, elmA);
2718
2719      int msize = 0;
2720      if (typef == f64)
2721          msize = sizeof(double);
2722      else if (typef == f32)
2723          msize = sizeof(float);
2724      else;
2725
2726      //3.obteniendo las referencias cl_mem de los objetos af::array
2727      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
2728          CL_MEM_READ_WRITE, msize*size_elmL,
2729          NULL, &status);
2730      af_get_device_ptr((void**)d_elmL, elmA);
2731
2732      cl_mem *d_idxL = (cl_mem*)clCreateBuffer(af_context,
2733          CL_MEM_READ_ONLY, sizeof(int)*size_idxL,
2734          NULL, &status);
2735      af_get_device_ptr((void**)d_idxL, idxA);
2736
2737      size_t program_length = strlen(Cholesky_source);
2738
2739      //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
          de entrada
2740      // para el Kernel
2741      cl_program program = clCreateProgramWithSource(af_context,
2742          1, (const char **)&Cholesky_source, &program_length,
2743          &status);
2744      status = clBuildProgram(program, 1, &af_device_id,
2745          NULL, NULL, NULL);
2746
2747      char* kernelName;
2748      if (typef == f64)
2749          kernelName = "chol_sparse_sks";
2750      else if (typef == f32)
```

```
2751            kernelName = "chol_sparse_sks_sp";
2752        else;
2753        cl_kernel kernel = clCreateKernel(program, kernelName,
2754            &status);
2755
2756        // 5.estableciendo los argumentos
2757        int i = 0;
2758        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
2759        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idxL);
2760        clSetKernelArg(kernel, i++, sizeof(cl_int), &size_elmL);
2761        clSetKernelArg(kernel, i++, sizeof(cl_int), &size_idxL);
2762        clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
2763        clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
2764
2765        cl_int sstep = 0;
2766        //6.Ejecutando el kernel
2767        for (cl_int j = 0; j < size_idxL - 1; j++)
2768        {
2769            //Se modifican los elementos de la columna j a partir
2770            //de la fila j + 1.
2771            //A(n, j) = A(n, j) / sqrt(A(j, j)) n > j
2772            sstep = 0;
2773            clSetKernelArg(kernel, 6, sizeof(cl_int), &j);
2774            clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
2775            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2776                &globalWorkSize, &localWorkSize, 0, NULL,
2777                NULL);
2778
2779            //para una columna s, s > j, se modifican las filas n >= s
2780            //A(n,s)=A(n,s)-A(s,j)*A(n,j) s>j, n>=s
2781            sstep++;
2782            clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
2783            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2784                &globalWorkSize, &localWorkSize, 0, NULL,
2785                NULL);
2786        }
2787        sstep++;
2788        clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
2789        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2790            &globalWorkSize, &localWorkSize, 0, NULL,
2791            NULL);
2792
2793        //7. devolviendo el control de memoria af::array a ArrayFire
2794        af_unlock_array(elmA);
2795        af_unlock_array(idxA);
2796    }
2797
2798    void AFire::SELchol_sparse_sks(af_array* dC, af_array elmL,
2799        af_array idxL, af_array dB) {
2800        //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
2801        //cl_context af_context;
2802        static cl_context af_context = afcl::getContext();
2803        static cl_device_id af_device_id = afcl::getDeviceId();
2804        static cl_command_queue af_queue = afcl::getQueue();
2805
2806        //creando copia de dB
```

```
2807      af_array b;
2808      af_copy_array(&b, dB);
2809
2810      //2. Obteniendo parámetros necesarios
2811
2812      //longitud de los vectores
2813      dim_t _order[AF_MAX_DIMS];
2814      af_get_dims(&_order[0], &_order[1], &_order[2],
2815          &_order[3], elmL);
2816      size_t size_elmL = _order[0];
2817
2818      af_get_dims(&_order[0], &_order[1], &_order[2],
2819          &_order[3], idxL);
2820      size_t size_idxL = _order[0];
2821
2822      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
2823      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
2824
2825      int status = CL_SUCCESS;
2826
2827      af_dtype typef;
2828      af_get_type(&typef, elmL);
2829
2830      int msize = 0;
2831      if (typef == f64)
2832          msize = sizeof(double);
2833      else if (typef == f32)
2834          msize = sizeof(float);
2835      else;
2836
2837      //3.obteniendo las referencias cl_mem de los objetos af::array
2838      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
2839          CL_MEM_READ_ONLY, msize*size_elmL,
2840          NULL, &status);
2841      af_get_device_ptr((void**)d_elmL, elmL);
2842
2843      cl_mem *d_idxL = (cl_mem*)clCreateBuffer(af_context,
2844          CL_MEM_READ_ONLY, sizeof(int)*size_idxL,
2845          NULL, &status);
2846      af_get_device_ptr((void**)d_idxL, idxL);
2847
2848      cl_mem d_b = (cl_mem)clCreateBuffer(af_context,
2849          CL_MEM_READ_WRITE, msize*size_idxL,
2850          NULL, &status);
2851      af_get_device_ptr((void**)d_b, b);
2852
2853      size_t program_length = strlen(Cholesky_source);
2854
2855      //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
          de entrada
2856      // para el Kernel
2857      cl_program program = clCreateProgramWithSource(af_context,
2858          1, (const char **)&Cholesky_source, &program_length,
2859          &status);
2860      status = clBuildProgram(program, 1, &af_device_id,
2861          NULL, NULL, NULL);
```

```
2862
2863        char* kernelName;
2864        if (typef == f64)
2865            kernelName = "chol_sparse_sks";
2866        else if (typef == f32)
2867            kernelName = "chol_sparse_sks_sp";
2868        else;
2869        cl_kernel kernel = clCreateKernel(program, kernelName,
2870            &status);
2871
2872        cl_int kk = 0;
2873        // 5.estableciendo los argumentos
2874        int i = 0;
2875        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
2876        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idxL);
2877        clSetKernelArg(kernel, i++, sizeof(cl_int), &kk);
2878        clSetKernelArg(kernel, i++, sizeof(cl_int), &size_idxL);
2879        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_b);
2880        clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
2881
2882        //6. ejecutando el kernel
2883        //resolviendo el sistema Ly=b (y=transpose(L)*x)
2884        cl_int sstep = 3;
2885        clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
2886        for (cl_int j = 0; j < size_idxL - 1; j++) {
2887
2888            clSetKernelArg(kernel, 6, sizeof(cl_int), &j);
2889            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2890                &globalWorkSize, &localWorkSize, 0, NULL,
2891                NULL);
2892        }
2893        sstep++;
2894        clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
2895        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2896            &globalWorkSize, &localWorkSize, 0, NULL,
2897            NULL);
2898
2899        //resolviendo el sistema transpose(L)*x=y
2900        sstep++;
2901        clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
2902        for (cl_int j = 0; j < size_idxL; j++) {
2903            clSetKernelArg(kernel, 6, sizeof(cl_int), &j);
2904            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
2905                &globalWorkSize, &localWorkSize, 0, NULL,
2906                NULL);
2907        }
2908
2909        //7. devolviendo el control de memoria af::array a ArrayFire
2910        af_unlock_array(elmL);
2911        af_unlock_array(idxL);
2912        af_unlock_array(b);
2913
2914        //copiando al argumento de salida
2915        af_copy_array(dC, b);
2916
2917        af_release_array(b);
```

```
2918 }
2919
2920 void AFire::fac_sparse_ldlt_c(af_array elmA, af_array colA,
2921     af_array rowA, af_array elmL, af_array colL,
2922     af_array rowL) {
2923     //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
2924     //cl_context af_context;
2925     static cl_context af_context = afcl::getContext();
2926     static cl_device_id af_device_id = afcl::getDeviceId();
2927     static cl_command_queue af_queue = afcl::getQueue();
2928
2929     //2. Obteniendo parámetros necesarios
2930
2931     //longitud de los vectores
2932     dim_t _order[AF_MAX_DIMS];
2933     af_get_dims(&_order[0], &_order[1], &_order[2],
2934         &_order[3], elmA);
2935     size_t size_elmA = _order[0];
2936
2937     af_get_dims(&_order[0], &_order[1], &_order[2],
2938         &_order[3], colA);
2939     size_t size_colA = _order[0];
2940
2941     af_get_dims(&_order[0], &_order[1], &_order[2],
2942         &_order[3], rowA);
2943     size_t size_rowA = _order[0];
2944
2945     af_get_dims(&_order[0], &_order[1], &_order[2],
2946         &_order[3], elmL);
2947     size_t size_elmL = _order[0];
2948
2949     af_get_dims(&_order[0], &_order[1], &_order[2],
2950         &_order[3], colL);
2951     size_t size_colL = _order[0];
2952
2953     af_get_dims(&_order[0], &_order[1], &_order[2],
2954         &_order[3], rowL);
2955     size_t size_rowL = _order[0];
2956
2957     size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
2958     size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
2959
2960     int status = CL_SUCCESS;
2961
2962     af_dtype typef;
2963     af_get_type(&typef, elmA);
2964
2965     int msize = 0;
2966     if (typef == f64)
2967         msize = sizeof(double);
2968     else if (typef == f32)
2969         msize = sizeof(float);
2970     else;
2971
2972     //3.obteniendo las referencias cl_mem de los objetos af::array
2973     cl_mem *d_elmA = (cl_mem*)clCreateBuffer(af_context,
```

```
2974          CL_MEM_READ_ONLY, msize*size_elmA,
2975          NULL, &status);
2976      af_get_device_ptr((void**)d_elmA, elmA);
2977
2978      cl_mem *d_colA = (cl_mem*)clCreateBuffer(af_context,
2979          CL_MEM_READ_ONLY, sizeof(int)*size_colA,
2980          NULL, &status);
2981      af_get_device_ptr((void**)d_colA, colA);
2982
2983      cl_mem *d_rowA = (cl_mem*)clCreateBuffer(af_context,
2984          CL_MEM_READ_ONLY, sizeof(int)*size_rowA,
2985          NULL, &status);
2986      af_get_device_ptr((void**)d_rowA, rowA);
2987
2988      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
2989          CL_MEM_READ_WRITE, msize*size_elmL,
2990          NULL, &status);
2991      af_get_device_ptr((void**)d_elmL, elmL);
2992
2993      cl_mem *d_colL = (cl_mem*)clCreateBuffer(af_context,
2994          CL_MEM_READ_WRITE, sizeof(int)*size_colL,
2995          NULL, &status);
2996      af_get_device_ptr((void**)d_colL, colL);
2997
2998      cl_mem *d_rowL = (cl_mem*)clCreateBuffer(af_context,
2999          CL_MEM_READ_WRITE, sizeof(int)*size_rowL,
3000          NULL, &status);
3001      af_get_device_ptr((void**)d_rowL, rowL);
3002
3003      size_t program_length = strlen(ldlt_source);
3004
3005      //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
         de entrada
3006      // para el Kernel
3007      cl_program program = clCreateProgramWithSource(af_context,
3008          1, (const char **)&ldlt_source, &program_length,
3009          &status);
3010      status = clBuildProgram(program, 1, &af_device_id,
3011          NULL, NULL, NULL);
3012
3013      char* kernelName;
3014      if (typef == f64)
3015          kernelName = "ldlt_sparse_c";
3016      else if (typef == f32)
3017          kernelName = "ldlt_sparse_c_sp";
3018      else;
3019      cl_kernel kernel = clCreateKernel(program, kernelName,
3020          &status);
3021
3022      int step = 0;
3023      int sstep = -1;
3024
3025      // 5.estableciendo los argumentos
3026      int i = 0;
3027      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmA);
3028      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colA);
```

```
3029      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowA);
3030      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
3031      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colL);
3032      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowL);
3033      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_elmA);
3034      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_colA);
3035      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
3036      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
3037      clSetKernelArg(kernel, i++, sizeof(cl_int), &step);
3038      clSetKernelArg(kernel, i++, sizeof(cl_int), &sstep);
3039
3040      //6. ejecutando el kernel
3041
3042      //primera ejecución con sstep=-1 copiará los elementos
3043      //de elmA en el lugar correspondiente de elmL, tomar
3044      //en cuenta que los elementos iniciales del elmL son
3045      //todos cero.
3046      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3047          &globalWorkSize, &localWorkSize, 0, NULL,
3048          NULL);
3049
3050      //6.Ejecutando el kernel
3051      for (int j = 0; j < size_colA - 1; j++)
3052      {
3053          //Se modifican los elementos de la columna j a partir
3054          //de la fila j + 1.
3055          //A(n,j)=A(n,j)/A(j,j) n>j
3056          int sstep = 0;
3057          clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
3058          clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
3059          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3060              &globalWorkSize, &localWorkSize, 0, NULL,
3061              NULL);
3062
3063          //para una columna s, s > j, se modifican las filas n >= s
3064          //A(n,s)=A(n,s)-A(s,j)*A(n,j)*A(j,j) s>j, n>=s
3065          sstep++;
3066          clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
3067          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3068              &globalWorkSize, &localWorkSize, 0, NULL,
3069              NULL);
3070      }
3071
3072      //7. devolviendo el control de memoria af::array a ArrayFire
3073      af_unlock_array(elmA);
3074      af_unlock_array(colA);
3075      af_unlock_array(rowA);
3076      af_unlock_array(elmL);
3077      af_unlock_array(colL);
3078      af_unlock_array(rowL);
3079  }
3080
3081  void AFire::SELldlt_sparse_c(af_array* dC, af_array elmA,
3082      af_array colA, af_array rowA, af_array elmL,
3083      af_array colL, af_array rowL, af_array dB) {
3084
```

```
3085      AFire::fac_sparse_ldlt_c(elmA, colA, rowA, elmL,
3086          colL, rowL);
3087      AFire::SELldlt_sparse_c(dC, elmL, colL, rowL, dB);
3088  }
3089
3090  void AFire::SELldlt_sparse_c(af_array* dC, af_array elmL,
3091      af_array colL, af_array rowL, af_array dB) {
3092      //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3093      //cl_context af_context;
3094      static cl_context af_context = afcl::getContext();
3095      static cl_device_id af_device_id = afcl::getDeviceId();
3096      static cl_command_queue af_queue = afcl::getQueue();
3097
3098      //creando copia de dB
3099      af_array b;
3100      af_copy_array(&b, dB);
3101
3102      //2. Obteniendo parámetros necesarios
3103
3104      //longitud de los vectores
3105      dim_t _order[AF_MAX_DIMS];
3106      af_get_dims(&_order[0], &_order[1], &_order[2],
3107          &_order[3], elmL);
3108      size_t size_elmL = _order[0];
3109
3110      af_get_dims(&_order[0], &_order[1], &_order[2],
3111          &_order[3], colL);
3112      size_t size_colL = _order[0];
3113
3114      af_get_dims(&_order[0], &_order[1], &_order[2],
3115          &_order[3], rowL);
3116      size_t size_rowL = _order[0];
3117
3118      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3119      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
3120
3121      int status = CL_SUCCESS;
3122
3123      af_dtype typef;
3124      af_get_type(&typef, elmL);
3125
3126      int msize = 0;
3127      if (typef == f64)
3128          msize = sizeof(double);
3129      else if (typef == f32)
3130          msize = sizeof(float);
3131      else;
3132
3133      //3.obteniendo las referencias cl_mem de los objetos af::array
3134      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
3135          CL_MEM_READ_WRITE, msize*size_elmL,
3136          NULL, &status);
3137      af_get_device_ptr((void**)d_elmL, elmL);
3138
3139      cl_mem *d_colL = (cl_mem*)clCreateBuffer(af_context,
3140          CL_MEM_READ_WRITE, sizeof(int)*size_colL,
```

```
3141            NULL, &status);
3142        af_get_device_ptr((void**)d_colL, colL);
3143
3144        cl_mem *d_rowL = (cl_mem*)clCreateBuffer(af_context,
3145            CL_MEM_READ_WRITE, sizeof(int)*size_rowL,
3146            NULL, &status);
3147        af_get_device_ptr((void**)d_rowL, rowL);
3148
3149        cl_mem *d_b = (cl_mem*)clCreateBuffer(af_context,
3150            CL_MEM_READ_WRITE, msize*size_colL,
3151            NULL, &status);
3152        af_get_device_ptr((void**)d_b, b);
3153
3154        size_t program_length = strlen(ldlt_source);
3155
3156        //4.creando el programa, construyendo el ejecutable y extrayendo el punto
               de entrada
3157        // para el Kernel
3158        cl_program program = clCreateProgramWithSource(af_context,
3159            1, (const char **)&ldlt_source, &program_length,
3160            &status);
3161        status = clBuildProgram(program, 1, &af_device_id,
3162            NULL, NULL, NULL);
3163
3164        char* kernelName;
3165        if (typef == f64)
3166            kernelName = "ldlt_sparse_c";
3167        else if (typef == f32)
3168            kernelName = "ldlt_sparse_c_sp";
3169        else;
3170        cl_kernel kernel = clCreateKernel(program, kernelName,
3171            &status);
3172
3173        int step = 0;
3174        int sstep = 0;
3175        int kk = 0;
3176        // 5.estableciendo los argumentos
3177        int i = 0;
3178        clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
3179        clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
3180        clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
3181        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
3182        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_colL);
3183        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowL);
3184        clSetKernelArg(kernel, i++, sizeof(cl_int), &kk);
3185        clSetKernelArg(kernel, i++, sizeof(cl_int), &size_colL);
3186        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_b);
3187        clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
3188        clSetKernelArg(kernel, i++, sizeof(cl_int), &step);
3189        clSetKernelArg(kernel, i++, sizeof(cl_int), &sstep);
3190
3191        //6. ejecutando el kernel
3192
3193        //resolviendo el sistema Ly=b (y=D*transpose(L)*x)
3194        sstep = 2;
3195        clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
```

```
3196      for (int j = 0; j < size_colL - 1; j++) {
3197
3198          clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
3199          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3200              &globalWorkSize, &localWorkSize, 0, NULL,
3201              NULL);
3202      }
3203
3204      //resolviendo el sistema Dz=y, z=transpose(L)*x
3205      sstep++;
3206      clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
3207      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3208          &globalWorkSize, &localWorkSize, 0, NULL,
3209          NULL);
3210
3211      //resolviendo el sistema transpose(L)*x=z
3212      sstep++;
3213      clSetKernelArg(kernel, 11, sizeof(cl_int), &sstep);
3214
3215      for (int j = 0; j < size_colL; j++) {
3216
3217          clSetKernelArg(kernel, 10, sizeof(cl_int), &j);
3218          clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3219              &globalWorkSize, &localWorkSize, 0, NULL,
3220              NULL);
3221      }
3222
3223      //7. devolviendo el control de memoria af::array a ArrayFire
3224      af_unlock_array(elmL);
3225      af_unlock_array(colL);
3226      af_unlock_array(rowL);
3227      af_unlock_array(b);
3228
3229      //copiando al argumento de salida
3230      af_copy_array(dC, b);
3231
3232      af_release_array(b);
3233  }
3234
3235  void AFire::fac_sparse_ldlt_sks(af_array elmA,
3236      af_array idxA) {
3237      //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3238      //cl_context af_context;
3239      static cl_context af_context = afcl::getContext();
3240      static cl_device_id af_device_id = afcl::getDeviceId();
3241      static cl_command_queue af_queue = afcl::getQueue();
3242
3243      //2. Obteniendo parámetros necesarios
3244      //longitud de los vectores
3245      dim_t _order[AF_MAX_DIMS];
3246      af_get_dims(&_order[0], &_order[1], &_order[2],
3247          &_order[3], elmA);
3248      size_t size_elmL = _order[0];
3249
3250      af_get_dims(&_order[0], &_order[1], &_order[2],
3251          &_order[3], idxA);
```

```
3252      size_t size_idxL = _order[0];
3253
3254      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3255      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
3256
3257      int status = CL_SUCCESS;
3258
3259      af_dtype typef;
3260      af_get_type(&typef, elmA);
3261
3262      int msize = 0;
3263      if (typef == f64)
3264          msize = sizeof(double);
3265      else if (typef == f32)
3266          msize = sizeof(float);
3267      else;
3268
3269      //3.obteniendo las referencias cl_mem de los objetos af::array
3270      cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
3271          CL_MEM_READ_WRITE, msize*size_elmL,
3272          NULL, &status);
3273      af_get_device_ptr((void**)d_elmL, elmA);
3274
3275      cl_mem *d_idxL = (cl_mem*)clCreateBuffer(af_context,
3276          CL_MEM_READ_ONLY, sizeof(int)*size_idxL,
3277          NULL, &status);
3278      af_get_device_ptr((void**)d_idxL, idxA);
3279
3280      size_t program_length = strlen(ldlt_source);
3281
3282      //4.creando el programa, construyendo el ejecutable y extrayendo el punto ↵
             de entrada
3283      // para el Kernel
3284      cl_program program = clCreateProgramWithSource(af_context,
3285          1, (const char **)&ldlt_source, &program_length,
3286          &status);
3287      status = clBuildProgram(program, 1, &af_device_id,
3288          NULL, NULL, NULL);
3289
3290      char* kernelName;
3291      if (typef == f64)
3292          kernelName = "ldlt_sparse_sks";
3293      else if (typef == f32)
3294          kernelName = "ldlt_sparse_sks_sp";
3295      else;
3296      cl_kernel kernel = clCreateKernel(program, kernelName,
3297          &status);
3298
3299      // 5.estableciendo los argumentos
3300      int i = 0;
3301      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
3302      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idxL);
3303      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_elmL);
3304      clSetKernelArg(kernel, i++, sizeof(cl_int), &size_idxL);
3305      clSetKernelArg(kernel, i++, sizeof(cl_mem), 0);
3306      clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
```

```
3307
3308        //6.Ejecutando el kernel
3309        for (int j = 0; j < size_idxL - 1; j++)
3310        {
3311            //Se modifican los elementos de la columna j a partir
3312            //de la fila j + 1.
3313            //A(n,j)=A(n,j)/A(j,j) n>j
3314            int sstep = 0;
3315            clSetKernelArg(kernel, 6, sizeof(cl_int), &j);
3316            clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
3317            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3318                &globalWorkSize, &localWorkSize, 0, NULL,
3319                NULL);
3320
3321            //para una columna s, s > j, se modifican las filas n >= s
3322            //A(n,s)=A(n,s)-A(s,j)*A(n,j)*A(j,j) s>j, n>=s
3323            sstep++;
3324            clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
3325            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3326                &globalWorkSize, &localWorkSize, 0, NULL,
3327                NULL);
3328        }
3329
3330        //7. devolviendo el control de memoria af::array a ArrayFire
3331        af_unlock_array(elmA);
3332        af_unlock_array(idxA);
3333    }
3334
3335    void AFire::SELldlt_sparse_sks(af_array* dC, af_array elmL,
3336        af_array idxL, af_array dB) {
3337        //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3338        //cl_context af_context;
3339        static cl_context af_context = afcl::getContext();
3340        static cl_device_id af_device_id = afcl::getDeviceId();
3341        static cl_command_queue af_queue = afcl::getQueue();
3342
3343        //creando copia de dB
3344        af_array b;
3345        af_copy_array(&b, dB);
3346
3347        //2. Obteniendo parámetros necesarios
3348
3349        //longitud de los vectores
3350        dim_t _order[AF_MAX_DIMS];
3351        af_get_dims(&_order[0], &_order[1], &_order[2],
3352            &_order[3], elmL);
3353        size_t size_elmL = _order[0];
3354
3355        af_get_dims(&_order[0], &_order[1], &_order[2],
3356            &_order[3], idxL);
3357        size_t size_idxL = _order[0];
3358
3359        size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3360        size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
3361
3362        int status = CL_SUCCESS;
```

```
3363
3364        af_dtype typef;
3365        af_get_type(&typef, elmL);
3366
3367        int msize = 0;
3368        if (typef == f64)
3369            msize = sizeof(double);
3370        else if (typef == f32)
3371            msize = sizeof(float);
3372        else;
3373
3374        //3.obteniendo las referencias cl_mem de los objetos af::array
3375        cl_mem *d_elmL = (cl_mem*)clCreateBuffer(af_context,
3376            CL_MEM_READ_ONLY, msize*size_elmL,
3377            NULL, &status);
3378        af_get_device_ptr((void**)d_elmL, elmL);
3379
3380        cl_mem *d_idxL = (cl_mem*)clCreateBuffer(af_context,
3381            CL_MEM_READ_ONLY, sizeof(int)*size_idxL,
3382            NULL, &status);
3383        af_get_device_ptr((void**)d_idxL, idxL);
3384
3385        cl_mem d_b = (cl_mem)clCreateBuffer(af_context,
3386            CL_MEM_READ_WRITE, msize*size_idxL,
3387            NULL, &status);
3388        af_get_device_ptr((void**)d_b, b);
3389
3390        size_t program_length = strlen(ldlt_source);
3391
3392        //4.creando el programa, construyendo el ejecutable y extrayendo el punto
            de entrada
3393        // para el Kernel
3394        cl_program program = clCreateProgramWithSource(af_context,
3395            1, (const char **)&ldlt_source, &program_length,
3396            &status);
3397        status = clBuildProgram(program, 1, &af_device_id,
3398            NULL, NULL, NULL);
3399
3400        char* kernelName;
3401        if (typef == f64)
3402            kernelName = "ldlt_sparse_sks";
3403        else if (typef == f32)
3404            kernelName = "ldlt_sparse_sks_sp";
3405        else;
3406        cl_kernel kernel = clCreateKernel(program, kernelName,
3407            &status);
3408
3409        cl_int kk = 0;
3410        // 5.estableciendo los argumentos
3411        int i = 0;
3412        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_elmL);
3413        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idxL);
3414        clSetKernelArg(kernel, i++, sizeof(cl_int), &kk);
3415        clSetKernelArg(kernel, i++, sizeof(cl_int), &size_idxL);
3416        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_b);
3417        clSetKernelArg(kernel, i++, msize*localWorkSize, 0);
```

```
3418
3419        //6. ejecutando el kernel
3420        //resolviendo el sistema Ly=b (y=D*transpose(L)*x)
3421        cl_int sstep = 2;
3422        clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
3423        for (cl_int j = 0; j < size_idxL - 1; j++) {
3424
3425            clSetKernelArg(kernel, 6, sizeof(cl_int), &j);
3426            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3427                &globalWorkSize, &localWorkSize, 0, NULL,
3428                NULL);
3429        }
3430
3431        //resolviendo el sistema Dz=y, z=transpose(L)*x
3432        sstep++;
3433        clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
3434        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3435            &globalWorkSize, &localWorkSize, 0, NULL,
3436            NULL);
3437
3438        //resolviendo el sistema transpose(L)*x=z
3439        sstep++;
3440        clSetKernelArg(kernel, 7, sizeof(cl_int), &sstep);
3441        for (cl_int j = 0; j < size_idxL; j++) {
3442            clSetKernelArg(kernel, 6, sizeof(cl_int), &j);
3443            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3444                &globalWorkSize, &localWorkSize, 0, NULL,
3445                NULL);
3446        }
3447
3448        //7. devolviendo el control de memoria af::array a ArrayFire
3449        af_unlock_array(elmL);
3450        af_unlock_array(idxL);
3451        af_unlock_array(b);
3452
3453        //copiando al argumento de salida
3454        af_copy_array(dC, b);
3455
3456        af_release_array(b);
3457    }
3458    //---------------
3459    //Pruebas y otros
3460    //---------------
3461    void AFire::fenceTest_sp(af::array &A, af::array &B) {
3462        //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3463        static cl_context af_context = afcl::getContext();
3464        static cl_device_id af_device_id = afcl::getDeviceId();
3465        static cl_command_queue af_queue = afcl::getQueue();
3466
3467        //obteniendo las referencias cl_mem de los objetos af::array
3468        cl_mem * d_A = A.device<cl_mem>();
3469        cl_mem * d_B = B.device<cl_mem>();
3470        //-------------------------------------
3471        //detalles importantes sobre el kernel
3472        //-------------------------------------
3473        /**/
```

```
3474
3475      size_t order = (int)A.dims(0);
3476
3477      size_t program_length = strlen(GJordan_source);
3478      int status = CL_SUCCESS;
3479
3480      //creando el programa, construyendo el ejecutable y extrayendo el punto
            de entrada
3481      // para el Kernel
3482      cl_program program = clCreateProgramWithSource(af_context, 1, (const char
            **)&GJordan_source, &program_length, &status);
3483      status = clBuildProgram(program, 1, &af_device_id, NULL, NULL, NULL);
3484      cl_kernel kernel = clCreateKernel(program, "fenceTest_sp", &status);
3485
3486      // estableciendo los argumentos
3487      int i = 0;
3488      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
3489      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_B);
3490      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
3491
3492      size_t localWorkSize[] = { BLOCK_SIZE, BLOCK_SIZE };
3493      size_t globalWorkSize[] =
3494      { shrRoundUp(localWorkSize[0], order),shrRoundUp(localWorkSize[1],
            order) };
3495      //ejecutando el Kernel
3496      clEnqueueNDRangeKernel(af_queue, kernel, 2, 0,
3497          globalWorkSize, localWorkSize, 0, NULL, NULL);
3498
3499      //devolviendo el control de memoria af::array a ArrayFire
3500      A.unlock();
3501      B.unlock();
3502  }
3503
3504  void AFire::test_1(af::array &A, af::array &B)
3505  {
3506      //B = A;
3507      size_t len = A.dims(0);
3508      seq idx(0, len - 1, 1);
3509      af::copy(B, A, idx);
3510  }
3511
3512  void AFire::test_2(af_array dA)
3513  {
3514      //1. Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3515      //cl_context af_context;
3516      static cl_context af_context = afcl::getContext();
3517      static cl_device_id af_device_id = afcl::getDeviceId();
3518      static cl_command_queue af_queue = afcl::getQueue();
3519
3520      //2. Obteniendo parámetros necesarios
3521
3522      //longitud de los vectores
3523      dim_t _order[AF_MAX_DIMS];
3524      af_get_dims(&_order[0], &_order[1], &_order[2],
3525          &_order[3], dA);
3526      size_t size_elmA = _order[0];
```

```
3527
3528        af_array zeros;
3529        dim_t size[] = { size_elmA };
3530        af_constant(&zeros, 0, 1, size, s32);
3531
3532        size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3533        size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
3534
3535        int status = CL_SUCCESS;
3536
3537        af_dtype typef;
3538        af_get_type(&typef, dA);
3539
3540        int msize = 0;
3541        if (typef == f64)
3542            msize = sizeof(double);
3543        else if (typef == f32)
3544            msize = sizeof(float);
3545        else;
3546
3547        //3.obteniendo las referencias cl_mem de los objetos af::array
3548        cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
3549            CL_MEM_READ_WRITE, msize*size_elmA,
3550            NULL, &status);
3551        af_get_device_ptr((void**)d_A, dA);
3552
3553        cl_mem *d_zeros = (cl_mem*)clCreateBuffer(af_context,
3554            CL_MEM_READ_WRITE, sizeof(int)*size_elmA,
3555            NULL, &status);
3556        af_get_device_ptr((void**)d_zeros, zeros);
3557
3558        size_t program_length = strlen(gc_source);
3559
3560        //4.creando el programa, construyendo el ejecutable y extrayendo el punto
               de entrada
3561        // para el Kernel
3562        cl_program program = clCreateProgramWithSource(af_context,
3563            1, (const char **)&gc_source, &program_length,
3564            &status);
3565        status = clBuildProgram(program, 1, &af_device_id,
3566            NULL, NULL, NULL);
3567
3568        char* kernelName;
3569        if (typef == f64)
3570            kernelName = "prueba_1";
3571        else if (typef == f32)
3572            kernelName = "prueba_1_sp";
3573        else;
3574        cl_kernel kernel = clCreateKernel(program, kernelName,
3575            &status);
3576
3577        // 5.estableciendo los argumentos
3578        int i = 0;
3579        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
3580        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_zeros);
3581        clSetKernelArg(kernel, i++, sizeof(cl_int), &size_elmA);
```

```
3582
3583        //6. ejecutando el kernel
3584        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3585            &globalWorkSize, &localWorkSize, 0, NULL,
3586            NULL);
3587
3588        af_print_array(zeros);
3589        //7. devolviendo el control de memoria af::array a ArrayFire
3590        af_unlock_array(dA);
3591        af_unlock_array(zeros);
3592
3593        af_release_array(zeros);
3594    }
3595
3596    void AFire::global_sync_test(af_array* dC, af_array dA,
3597        af_array dB) {
3598        //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3599        //cl_context af_context;
3600        static cl_context af_context = afcl::getContext();
3601        static cl_device_id af_device_id = afcl::getDeviceId();
3602        static cl_command_queue af_queue = afcl::getQueue();
3603
3604        //obteniendo el número de multiprocesadores, necesario
3605        //para determinar el número de bloques para los kernels,
3606        //y hacer posible el uso de las funciones de sincronización
3607        //ver artículo:
3608        //Inter-Block GPU Communication via Fast Barrier Synchronization
3609        //Shucai Xiao and Wu-chun Feng
3610        //Department of Computer Science Virginia Tech
3611        //2009/9/19
3612        //Pág.5
3613        cl_uint compute_units;
3614        clGetDeviceInfo(af_device_id,
3615            CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint),
3616            &compute_units, NULL);
3617        compute_units = BLOCK_SIZE;
3618        //acoplando A con B
3619        af_array Ac;
3620        af_join(&Ac, 1, dA, dB);
3621
3622        //vectores de sincronización
3623        const dim_t nBlks[] = { compute_units };
3624        af_array syncIn;
3625        af_array syncOut;
3626        af_constant(&syncIn, 0, 1, nBlks, s32);
3627        af_constant(&syncOut, 0, 1, nBlks, s32);
3628
3629        dim_t _order[AF_MAX_DIMS];
3630        af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], dA);
3631        size_t order = _order[0];
3632
3633        size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3634        size_t globalWorkSize = localWorkSize * compute_units;
3635
3636        int status = CL_SUCCESS;
3637
```

```
3638      af_dtype typef;
3639      af_get_type(&typef, dA);
3640
3641      int msize = 0;
3642      if (typef == f64)
3643          msize = sizeof(double);
3644      else if (typef == f32)
3645          msize = sizeof(float);
3646      else;
3647
3648      //obteniendo las referencias cl_mem de los objetos af::array
3649      cl_mem *d_A = (cl_mem*)clCreateBuffer(af_context,
3650          CL_MEM_READ_WRITE, msize * order*(order + 1),
3651          NULL, &status);
3652      af_get_device_ptr((void**)d_A, Ac);
3653
3654      cl_mem *d_syncIn = (cl_mem*)clCreateBuffer(af_context,
3655          CL_MEM_READ_WRITE, sizeof(int)*compute_units,
3656          NULL, &status);
3657      af_get_device_ptr((void**)d_syncIn, syncIn);
3658
3659      cl_mem *d_syncOut = (cl_mem*)clCreateBuffer(af_context,
3660          CL_MEM_READ_WRITE, sizeof(int)*compute_units,
3661          NULL, &status);
3662      af_get_device_ptr((void**)d_syncOut, syncOut);
3663
3664      size_t program_length = strlen(GJordan_source);
3665
3666      //creando el programa, construyendo el ejecutable y extrayendo el punto ↵
            de entrada
3667      // para el Kernel
3668      cl_program program =
3669          clCreateProgramWithSource(af_context, 1,
3670          (const char **)&GJordan_source, &program_length,
3671              &status);
3672      status = clBuildProgram(program, 1, &af_device_id,
3673          NULL, NULL, NULL);
3674
3675      char* kernelName;
3676      if (typef == f64)
3677          kernelName = "global_sync_test";
3678      else if (typef == f32)
3679          kernelName = "global_sync_test_sp";
3680      else;
3681      cl_kernel kernel = clCreateKernel(program, kernelName,
3682          &status);
3683
3684      // estableciendo los argumentos
3685      int i = 0;
3686      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_A);
3687      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
3688      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_syncIn);
3689      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_syncOut);
3690
3691      int device;
3692      af_get_device(&device);
```

```
3693
3694        for (int j = 0; j < order; j++)
3695        {
3696            clSetKernelArg(kernel, 4, sizeof(cl_int), &j);
3697            //ejecutando el Kernel
3698            clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3699                &globalWorkSize, &localWorkSize, 0, NULL,
3700                NULL);
3701            //af_sync(device);
3702        }
3703
3704        //devolviendo el control de memoria af::array a ArrayFire
3705        af_unlock_array(Ac);
3706
3707        //hasta aqui Ac contiene en su última columna
3708        //y en su diagonal principal, los valores
3709        //finales que deben dividirse para obtener
3710        //la solución final
3711
3712        //extrayendo la última columna
3713        af_array Au;
3714        af_index_t* indexers = 0;
3715        af_create_indexers(&indexers);
3716        af_set_seq_param_indexer(indexers, 0, order - 1, 1,
3717            0, false);
3718        af_set_seq_param_indexer(indexers, order, order, 1,
3719            1, false);
3720        af_index_gen(&Au, Ac, 2, indexers);
3721
3722        //extrayendo la diagonal
3723        af_array Ad;
3724        af_diag_extract(&Ad, Ac, 0);
3725
3726        //dividiendo
3727        af_release_array(Ac);
3728        af_div(&Ac, Au, Ad, false);
3729
3730        // copiando el resultado en dC
3731        af_copy_array(dC, Ac);
3732
3733        af_release_array(Au);
3734        af_release_array(Ad);
3735        af_release_array(Ac);
3736        af_release_indexers(indexers);
3737    }
3738
3739    void AFire::sks_util_1(af_array* index, af_array row,
3740        af_array col) {
3741        //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3742        //cl_context af_context;
3743        static cl_context af_context = afcl::getContext();
3744        static cl_device_id af_device_id = afcl::getDeviceId();
3745        static cl_command_queue af_queue = afcl::getQueue();
3746
3747        double orderd;
3748        double helper;
```

```
3749      af_max_all(&orderd, &helper, col);
3750      int order = (int)orderd;
3751      order++;
3752
3753      af_dtype typef;
3754      af_get_type(&typef, row);
3755
3756      af_array idx;
3757      dim_t d_order[] = { order };
3758      af_constant(&idx, 0, 1, d_order, typef);
3759
3760      dim_t _order[AF_MAX_DIMS];
3761      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], row);
3762      size_t lennz = _order[0];
3763
3764      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3765      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
3766
3767      int status = CL_SUCCESS;
3768
3769      //obteniendo las referencias cl_mem de los objetos af::array
3770      cl_mem *d_col = (cl_mem*)clCreateBuffer(af_context,
3771          CL_MEM_READ_ONLY, sizeof(int)*lennz,
3772          NULL, &status);
3773      af_get_device_ptr((void**)d_col, col);
3774
3775      cl_mem *d_row = (cl_mem*)clCreateBuffer(af_context,
3776          CL_MEM_READ_ONLY, sizeof(int)*lennz,
3777          NULL, &status);
3778      af_get_device_ptr((void**)d_row, row);
3779
3780      cl_mem *d_idx = (cl_mem*)clCreateBuffer(af_context,
3781          CL_MEM_WRITE_ONLY, sizeof(int)*order,
3782          NULL, &status);
3783      af_get_device_ptr((void**)d_idx, idx);
3784
3785      size_t program_length = strlen(util_src);
3786
3787
3788      //creando el programa, construyendo el ejecutable y
3789      //extrayendo el punto de entrada
3790      // para el Kernel
3791      cl_program program =
3792          clCreateProgramWithSource(af_context, 1,
3793          (const char **)&util_src, &program_length,
3794              &status);
3795      status = clBuildProgram(program, 1, &af_device_id,
3796          NULL, NULL, NULL);
3797
3798      char* kernelName;
3799      kernelName = "sks_util_1";
3800
3801      cl_kernel kernel = clCreateKernel(program, kernelName,
3802          &status);
3803
3804      // estableciendo los argumentos
```

```
3805      int i = 0;
3806      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idx);
3807      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_row);
3808      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_col);
3809      clSetKernelArg(kernel, i++, sizeof(cl_int), &lennz);
3810
3811
3812      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3813              &globalWorkSize, &localWorkSize, 0, NULL,
3814              NULL);
3815
3816      //devolviendo el control de memoria af::array a ArrayFire
3817      af_unlock_array(col);
3818      af_unlock_array(row);
3819
3820      // copiando el resultado en dC
3821      af_copy_array(index, idx);
3822
3823      af_release_array(idx);
3824  }
3825
3826  void AFire::sks_util_2(af_array sks, af_array ptr,
3827      af_array nz, af_array row, af_array col) {
3828      //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3829          //cl_context af_context;
3830      static cl_context af_context = afcl::getContext();
3831      static cl_device_id af_device_id = afcl::getDeviceId();
3832      static cl_command_queue af_queue = afcl::getQueue();
3833
3834      double orderd;
3835      double helper;
3836      af_max_all(&orderd, &helper, col);
3837      int order = (int)orderd;
3838      order++;
3839
3840      af_dtype typef;
3841      af_get_type(&typef, nz);
3842
3843      dim_t _order[AF_MAX_DIMS];
3844      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], row);
3845      size_t lennz = _order[0];
3846
3847      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], sks);
3848      size_t lensks = _order[0];
3849
3850      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3851      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
3852
3853      int status = CL_SUCCESS;
3854
3855      int msize = 0;
3856      if (typef == f32)
3857          msize = sizeof(float);
3858      else if (typef == f64)
3859          msize = sizeof(double);
3860      else;
```

```
3861
3862      //obteniendo las referencias cl_mem de los objetos af::array
3863      cl_mem *d_sks = (cl_mem*)clCreateBuffer(af_context,
3864          CL_MEM_WRITE_ONLY, msize*lensks,
3865          NULL, &status);
3866      af_get_device_ptr((void**)d_sks, sks);
3867
3868      cl_mem *d_ptr = (cl_mem*)clCreateBuffer(af_context,
3869          CL_MEM_READ_ONLY, sizeof(int)*order,
3870          NULL, &status);
3871      af_get_device_ptr((void**)d_ptr, ptr);
3872
3873      cl_mem *d_nz = (cl_mem*)clCreateBuffer(af_context,
3874          CL_MEM_READ_ONLY, msize*lennz,
3875          NULL, &status);
3876      af_get_device_ptr((void**)d_nz, nz);
3877
3878      cl_mem *d_row = (cl_mem*)clCreateBuffer(af_context,
3879          CL_MEM_READ_ONLY, sizeof(int)*lennz,
3880          NULL, &status);
3881      af_get_device_ptr((void**)d_row, row);
3882
3883      cl_mem *d_col = (cl_mem*)clCreateBuffer(af_context,
3884          CL_MEM_READ_ONLY, sizeof(int)*lennz,
3885          NULL, &status);
3886      af_get_device_ptr((void**)d_col, col);
3887
3888
3889      size_t program_length = strlen(util_src);
3890
3891
3892      //creando el programa, construyendo el ejecutable y
3893      //extrayendo el punto de entrada
3894      // para el Kernel
3895      cl_program program =
3896          clCreateProgramWithSource(af_context, 1,
3897          (const char **)&util_src, &program_length,
3898              &status);
3899      status = clBuildProgram(program, 1, &af_device_id,
3900          NULL, NULL, NULL);
3901
3902      char* kernelName;
3903      if (typef == f32)
3904          kernelName = "sks_util_2_sp";
3905      else if (typef == f64)
3906          kernelName = "sks_util_2";
3907      else;
3908      cl_kernel kernel = clCreateKernel(program, kernelName,
3909          &status);
3910
3911      // estableciendo los argumentos
3912      int i = 0;
3913      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_sks);
3914      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_ptr);
3915      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_nz);
3916      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_row);
```

```
3917      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_col);
3918      clSetKernelArg(kernel, i++, sizeof(cl_int), &lennz);
3919
3920
3921      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
3922          &globalWorkSize, &localWorkSize, 0, NULL,
3923          NULL);
3924
3925      //devolviendo el control de memoria af::array a ArrayFire
3926      af_unlock_array(sks);
3927      af_unlock_array(ptr);
3928      af_unlock_array(nz);
3929      af_unlock_array(row);
3930      af_unlock_array(col);
3931  }
3932
3933  void AFire::csc_util_1(af_array* index, af_array row,
3934      af_array col) {
3935      //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
3936      //cl_context af_context;
3937      static cl_context af_context = afcl::getContext();
3938      static cl_device_id af_device_id = afcl::getDeviceId();
3939      static cl_command_queue af_queue = afcl::getQueue();
3940
3941      double orderd;
3942      double helper;
3943      af_max_all(&orderd, &helper, col);
3944      int order = (int)orderd;
3945      order++;
3946
3947      af_dtype typef;
3948      af_get_type(&typef, row);
3949
3950      af_array idx;
3951      dim_t d_order[] = { order };
3952      af_constant(&idx, 0, 1, d_order, typef);
3953
3954      dim_t _order[AF_MAX_DIMS];
3955      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], row);
3956      size_t lennz = _order[0];
3957
3958      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
3959      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
3960
3961      int status = CL_SUCCESS;
3962
3963      //obteniendo las referencias cl_mem de los objetos af::array
3964      cl_mem *d_col = (cl_mem*)clCreateBuffer(af_context,
3965          CL_MEM_READ_ONLY, sizeof(int)*lennz,
3966          NULL, &status);
3967      af_get_device_ptr((void**)d_col, col);
3968
3969      cl_mem *d_row = (cl_mem*)clCreateBuffer(af_context,
3970          CL_MEM_READ_ONLY, sizeof(int)*lennz,
3971          NULL, &status);
3972      af_get_device_ptr((void**)d_row, row);
```

```
3973
3974        cl_mem *d_idx = (cl_mem*)clCreateBuffer(af_context,
3975            CL_MEM_WRITE_ONLY, sizeof(int)*order,
3976            NULL, &status);
3977        af_get_device_ptr((void**)d_idx, idx);
3978
3979        size_t program_length = strlen(util_src);
3980
3981
3982        //creando el programa, construyendo el ejecutable y
3983        //extrayendo el punto de entrada
3984        // para el Kernel
3985        cl_program program =
3986            clCreateProgramWithSource(af_context, 1,
3987            (const char **)&util_src, &program_length,
3988                &status);
3989        status = clBuildProgram(program, 1, &af_device_id,
3990            NULL, NULL, NULL);
3991
3992        char* kernelName;
3993        kernelName = "csc_util_1";
3994
3995        cl_kernel kernel = clCreateKernel(program, kernelName,
3996            &status);
3997
3998        // estableciendo los argumentos
3999        int i = 0;
4000        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idx);
4001        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_row);
4002        clSetKernelArg(kernel, i++, sizeof(cl_mem), d_col);
4003        clSetKernelArg(kernel, i++, sizeof(cl_int), &lennz);
4004
4005
4006        clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
4007            &globalWorkSize, &localWorkSize, 0, NULL,
4008            NULL);
4009
4010        //devolviendo el control de memoria af::array a ArrayFire
4011        af_unlock_array(col);
4012        af_unlock_array(row);
4013
4014        // copiando el resultado en dC
4015        af_copy_array(index, idx);
4016
4017        af_release_array(idx);
4018    }
4019
4020    void AFire::csc_util_2(af_array* del, af_array index,
4021        af_array row) {
4022        //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
4023        //cl_context af_context;
4024        static cl_context af_context = afcl::getContext();
4025        static cl_device_id af_device_id = afcl::getDeviceId();
4026        static cl_command_queue af_queue = afcl::getQueue();
4027
4028        dim_t _order[AF_MAX_DIMS];
```

```
4029      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], index);
4030      size_t order = _order[0];
4031
4032      af_get_dims(&_order[0], &_order[1], &_order[2], &_order[3], row);
4033      size_t lennz = _order[0];
4034
4035      af_dtype typef;
4036      af_get_type(&typef, index);
4037
4038      af_array idx;
4039      dim_t d_order[] = { order };
4040      af_constant(&idx, 0, 1, d_order, typef);
4041
4042      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
4043      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
4044
4045      int status = CL_SUCCESS;
4046
4047      //obteniendo las referencias cl_mem de los objetos af::array
4048
4049      cl_mem *d_index = (cl_mem*)clCreateBuffer(af_context,
4050          CL_MEM_READ_ONLY, sizeof(int)*order,
4051          NULL, &status);
4052      af_get_device_ptr((void**)d_index, index);
4053
4054      cl_mem *d_row = (cl_mem*)clCreateBuffer(af_context,
4055          CL_MEM_READ_ONLY, sizeof(int)*lennz,
4056          NULL, &status);
4057      af_get_device_ptr((void**)d_row, row);
4058
4059      cl_mem *d_idx = (cl_mem*)clCreateBuffer(af_context,
4060          CL_MEM_WRITE_ONLY, sizeof(int)*order,
4061          NULL, &status);
4062      af_get_device_ptr((void**)d_idx, idx);
4063
4064      size_t program_length = strlen(util_src);
4065
4066      //creando el programa, construyendo el ejecutable y
4067      //extrayendo el punto de entrada
4068      // para el Kernel
4069      cl_program program =
4070          clCreateProgramWithSource(af_context, 1,
4071          (const char **)&util_src, &program_length,
4072              &status);
4073      status = clBuildProgram(program, 1, &af_device_id,
4074          NULL, NULL, NULL);
4075
4076      char* kernelName;
4077      kernelName = "csc_util_2";
4078
4079      cl_kernel kernel = clCreateKernel(program, kernelName,
4080          &status);
4081
4082      // estableciendo los argumentos
4083      int i = 0;
4084      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_idx);
```

```
4085      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_index);
4086      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_row);
4087      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
4088
4089
4090      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
4091          &globalWorkSize, &localWorkSize, 0, NULL,
4092          NULL);
4093
4094      //devolviendo el control de memoria af::array a ArrayFire
4095      af_unlock_array(index);
4096      af_unlock_array(row);
4097
4098      // copiando el resultado en dC
4099      af_copy_array(del, idx);
4100
4101      af_release_array(idx);
4102  }
4103
4104  void AFire::csc_util_3(af_array del, af_array ptr) {
4105      //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
4106      //cl_context af_context;
4107      static cl_context af_context = afcl::getContext();
4108      static cl_device_id af_device_id = afcl::getDeviceId();
4109      static cl_command_queue af_queue = afcl::getQueue();
4110
4111      dim_t _order[AF_MAX_DIMS];
4112      af_get_dims(&_order[0], &_order[1], &_order[2],
4113          &_order[3], ptr);
4114      size_t order = _order[0];
4115
4116      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
4117      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
4118
4119      int status = CL_SUCCESS;
4120
4121      //obteniendo las referencias cl_mem de los objetos af::array
4122
4123      cl_mem *d_del = (cl_mem*)clCreateBuffer(af_context,
4124          CL_MEM_READ_ONLY, sizeof(int)*order,
4125          NULL, &status);
4126      af_get_device_ptr((void**)d_del, del);
4127
4128      cl_mem *d_ptr = (cl_mem*)clCreateBuffer(af_context,
4129          CL_MEM_READ_WRITE, sizeof(int)*order,
4130          NULL, &status);
4131      af_get_device_ptr((void**)d_ptr, ptr);
4132
4133      size_t program_length = strlen(util_src);
4134
4135      //creando el programa, construyendo el ejecutable y
4136      //extrayendo el punto de entrada
4137      // para el Kernel
4138      cl_program program =
4139          clCreateProgramWithSource(af_context, 1,
4140          (const char **)&util_src, &program_length,
```

```
4141              &status);
4142      status = clBuildProgram(program, 1, &af_device_id,
4143          NULL, NULL, NULL);
4144
4145      char* kernelName;
4146      kernelName = "csc_util_3";
4147
4148      cl_kernel kernel = clCreateKernel(program, kernelName,
4149          &status);
4150
4151      // estableciendo los argumentos
4152      int i = 0;
4153      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_del);
4154      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_ptr);
4155      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
4156      clSetKernelArg(kernel, i++,
4157          sizeof(int)*localWorkSize, 0);
4158
4159
4160      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
4161          &globalWorkSize, &localWorkSize, 0, NULL,
4162          NULL);
4163
4164      //devolviendo el control de memoria af::array a ArrayFire
4165      af_unlock_array(del);
4166      af_unlock_array(ptr);
4167  }
4168
4169  void AFire::csc_util_4(af_array csc, af_array rowc,
4170      af_array ptr, af_array nz, af_array row, af_array del) {
4171      //Obteniendo el dispositivo, contexto y la cola usada por ArrayFire
4172          //cl_context af_context;
4173      static cl_context af_context = afcl::getContext();
4174      static cl_device_id af_device_id = afcl::getDeviceId();
4175      static cl_command_queue af_queue = afcl::getQueue();
4176
4177      dim_t _order[AF_MAX_DIMS];
4178      af_get_dims(&_order[0], &_order[1], &_order[2],
4179          &_order[3], ptr);
4180      size_t order = _order[0];
4181
4182      af_get_dims(&_order[0], &_order[1], &_order[2],
4183          &_order[3], row);
4184      size_t lennz = _order[0];
4185
4186      af_get_dims(&_order[0], &_order[1], &_order[2],
4187          &_order[3], csc);
4188      size_t lencsc = _order[0];
4189
4190      af_dtype typef;
4191      af_get_type(&typef, nz);
4192
4193      size_t localWorkSize = BLOCK_SIZE * BLOCK_SIZE;
4194      size_t globalWorkSize = localWorkSize * BLOCK_SIZE;
4195
4196      int status = CL_SUCCESS;
```

```
4197
4198        int msize = 0;
4199        if (typef == f32)
4200            msize = sizeof(float);
4201        else if (typef == f64)
4202            msize = sizeof(double);
4203        else;
4204
4205        //obteniendo las referencias cl_mem de los objetos af::array
4206        cl_mem *d_csc = (cl_mem*)clCreateBuffer(af_context,
4207            CL_MEM_WRITE_ONLY, msize*lencsc,
4208            NULL, &status);
4209        af_get_device_ptr((void**)d_csc, csc);
4210
4211        cl_mem *d_rowc = (cl_mem*)clCreateBuffer(af_context,
4212            CL_MEM_WRITE_ONLY, sizeof(int)*(lencsc - order),
4213            NULL, &status);
4214        af_get_device_ptr((void**)d_rowc, rowc);
4215
4216        cl_mem *d_ptr = (cl_mem*)clCreateBuffer(af_context,
4217            CL_MEM_READ_ONLY, sizeof(int)*order,
4218            NULL, &status);
4219        af_get_device_ptr((void**)d_ptr, ptr);
4220
4221        cl_mem *d_nz = (cl_mem*)clCreateBuffer(af_context,
4222            CL_MEM_READ_ONLY, msize*lennz,
4223            NULL, &status);
4224        af_get_device_ptr((void**)d_nz, nz);
4225
4226        cl_mem *d_row = (cl_mem*)clCreateBuffer(af_context,
4227            CL_MEM_READ_ONLY, sizeof(int)*lennz,
4228            NULL, &status);
4229        af_get_device_ptr((void**)d_row, row);
4230
4231        cl_mem *d_del = (cl_mem*)clCreateBuffer(af_context,
4232            CL_MEM_READ_ONLY, sizeof(int)*order,
4233            NULL, &status);
4234        af_get_device_ptr((void**)d_del, del);
4235
4236        size_t program_length = strlen(util_src);
4237
4238        //creando el programa, construyendo el ejecutable y
4239        //extrayendo el punto de entrada
4240        // para el Kernel
4241        cl_program program =
4242            clCreateProgramWithSource(af_context, 1,
4243            (const char **)&util_src, &program_length,
4244                &status);
4245        status = clBuildProgram(program, 1, &af_device_id,
4246            NULL, NULL, NULL);
4247
4248        char* kernelName;
4249        if (typef == f32)
4250            kernelName = "csc_util_4_sp";
4251        else if (typef == f64)
4252            kernelName = "csc_util_4";
```

```
4253      else;
4254      cl_kernel kernel = clCreateKernel(program, kernelName,
4255          &status);
4256
4257      // estableciendo los argumentos
4258      int i = 0;
4259      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_csc);
4260      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_rowc);
4261      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_ptr);
4262      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_nz);
4263      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_row);
4264      clSetKernelArg(kernel, i++, sizeof(cl_mem), d_del);
4265      clSetKernelArg(kernel, i++, sizeof(cl_int), &order);
4266
4267
4268      clEnqueueNDRangeKernel(af_queue, kernel, 1, 0,
4269          &globalWorkSize, &localWorkSize, 0, NULL,
4270          NULL);
4271
4272      //devolviendo el control de memoria af::array a ArrayFire
4273      af_unlock_array(csc);
4274      af_unlock_array(ptr);
4275      af_unlock_array(nz);
4276      af_unlock_array(row);
4277      af_unlock_array(del);
4278  }
4279
```