Universidad Politécnica de Valencia

Máster en Computación Paralela y Distribuida

Trabajo Fin de Master

Optimización del cálculo de los modos lambda mediante la utilización de CUDA

Director: Vicent Vidal

Autor: Filippo Squillace

Año Académico 2011-2012

Índice general

Agradecimientos

En primer lugar quiero dar las gracias al director de la tesis Vicent Vidal Gimeno por su gran ayuda en la realización del mismo.

En general, quiero agradecer a todas las personas e instituciones que me han dado la posibilidad de finalizar mi trabajo a través del programa de investigación Santiago Grisolía financiado por parte de la Comunidad Valenciana y el Espacio Europeo de Investigación (EEI). Por eso quiero también agradecer el prof. Gumersindo Verdú por haber aceptado ser por mi parte el investigador responsable de este programa de investigación.

El trabajo se ha desarrollado en el departamento de Ingeniería Química y Nuclear (DIQN) de la Universidad Politécnica de Valencia y por eso quiero expresar agradecimiento a todas las personas que trabajan allí y que me han dado ayudas para una adecuada instalación en este Departamento.

Quiero agradecer a las personas e instituciones que me han dado acceso a los recursos informáticos, en particular el Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia y el Instituto de Telecomunicaciones y Aplicaciones Multimedia (iTEAM), que me han permitido acceder a los servidores para las pruebas de prestaciones sobre los algoritmos implementados.

Quiero dedicar este trabajo a mis padres por los esfuerzos hechos por todo mi largo tiempo de estudio por el hecho que han siempre creído en mis capacidades y posibilidades y me han ofrecido la oportunidad de desarrollar mi vida profesional; a mi hermana de la que siempre he sentido su cariño y apoyo.

Capítulo 1

Introducción

Los métodos para la manipulación de matrices dispersas son críticos en muchas aplicaciones. En particular, la operación de multiplicación matriz-vector (SpMV) ha mostrado ser de mucha importancia en la Ciencia de la Computación. En particular, esta operación representa el coste dominante de muchos métodos iterativos de resolución de sistemas de ecuaciones lineales y cálculo de valores propios de gran dimensión que aparecen en una gran variedad de aplicaciones científicas y de ingeniería. La parte remanente de los métodos iterativos típicamente se reduce en operaciones de álgebra lineal densas que se manejan con rutinas optimizadas de las librerías BLAS [9] y LAPACK [4].

Las modernas GPUs ofrecen procesadores con elevada productividad pero, para obtener ese potencial de computación se necesita diseñar un algoritmo paralelo con granularidad fina estructurando la computación de los hilos para mantener una regular ruta de ejecución y un rápido acceso a la memoria. Las operaciones con matrices densas son bastantes regulares; por contrario, las operaciones con matrices dispersas resultan mucho más irregulares en el acceso en memoria.

En este trabajo, se muestra el diseño de *kernels* masivamente paralelo para SpMV con matrices de diferente estructura y en particular con matrices que tienen muchos elementos cerca de la diagonal principal (que representan las matrices más usuales en el campo de ingeniera) en tarjeta gráfica GPU en CUDA [5]. Además, se utilizará la librería para el cálculo con matriz dispersa en CUDA denominada CUSP [3]. A través de esta librería analizaremos diferentes métodos de Krylov para la resolución de sistema de ecuaciones lineales y cálculo de valores propios. Al final de la memoria se presentaran las prestaciones obtenidas a través de las diferentes versiones de spMV y de métodos de Krylov con diferentes formatos de almacenamiento de la matriz. Se han utilizando diferentes matrices, y en particular las que se obtienen con la discretizacion de los reactores nucleares. Como veremos, es importante tener en cuenta la estructura de la matriz para obtener mayores prestaciones.

Este proyecto se ha realizado en colaboración con el Departamento de Ingeniería Química y Nuclear ya que necesita lograr el cálculo eficiente en problemas de Álgebra Lineal sobre matrices dispersas de gran tamaño asociado a la ecuación de la difusión neutrónica con varios grupos de energía. En particular, se expone un método eficiente para la resolución del cálculo de valores propios, es decir que se conoce el especifico tipo de problema y, entonces, para obtener altas prestaciones se adaptan los algoritmos para aprovechar al máximo los recursos disponibles. El cálculo sobre reactores se refieren, a menudo, a las ecuaciones de difusión neutronica multi-grupos [10, 14]. El sistema original de ecuaciones diferenciales se convierte en un problema algebraico generalizado de valores propios. Si las ecuaciones están modeladas para dos grupos de energía, entonces el problema puede ser abordado como la determinación de los valores propios y vectores propios de la ecuación 1.1.

$$\mathcal{L}\psi_i = \frac{1}{\lambda_i}\mathcal{M}\psi_i \tag{1.1}$$

donde

$$\mathcal{L} = \begin{bmatrix} L_{11} & 0 \\ -L_{21} & L_{22} \end{bmatrix}$$

y

$$\mathcal{M} = \begin{bmatrix} M_{11} & M_{22} \\ 0 & 0 \end{bmatrix} \quad , \quad \phi_i = \begin{bmatrix} \psi_{1i} \\ \psi_{2i} \end{bmatrix}$$

siendo L_{11} y L_{22} matrices dispersas no singulares, y M_{11}, M_{12} y L_{21} son matrices diagonales. Despejando ψ_{2i}, obtenemos el siguiente problema standard no simétrico, ecuación 1.2.

$$A\psi_{1i} = \frac{1}{\lambda_i}\psi_{1i} \tag{1.2}$$

donde la matriz A se obtiene a través

$$A = L_{11}^{-1}(M_{11} + M_{12}L_{22}^{-1}L_{21}) \tag{1.3}$$

Visto que no es posible lograr de forma explícita la matriz A, a causa de los errores de redondeo y al relleno en la matriz, es necesario aplicar un cálculo a través de la forma implícita de A en ecuación 1.3, así que el cálculo de la operación matriz-vector SpMV de A por cualquier vector y, se traduce a la resolución de sistemas de ecuaciones lineales de las matrices L_{ii} y por lo tanto al producto matriz-vector. En las ecuaciones de difusión neutrónica cuando tenemos que calcular los valores propios la resolución se basa sobre la operación básica del producto matriz-vector; pero como no se dispone de la forma explícita de A el cálculo se traduce a tres productos matriz diagonal por vector y a dos resoluciones de sistemas de ecuaciones lineales. Los sistemas se resuelven por métodos de Krylov, en los que la operación básica es nuevamente el producto matriz por vector. Por ello, las operaciones producto matriz-vector y resolución de sistemas son los procesos que hemos analizado en este trabajo.

Por el comportamiento físico del problema se sabe que todos los valores propios de la matriz A son reales. En esta memoria estamos en particular interesado en obtener los valores propios dominantes con sus correspondientes vectores propios. El primer paso para el diseño del algoritmo se refiere a la operación SpMV y este trabajo está en particular enfocado para este propósito. En particular, para el cálculo eficiente es necesario implementar funciones (denominadas *kernels*) que se ejecuten directamente en GPU. La eficiencia de SpMV depende de la estructura de la matriz dispersa y del patrón de almacenamiento de la matriz en memoria. El estado del arte en el almacenamiento de matrices dispersas [8, 12] es considerable, pero pueden resultar no adecuados para matrices especificas como las que tratamos en este trabajo. Por ello hemos diseñado unos formatos que se pueden utilizar para explotar los recursos en tarjeta GPU.

El siguiente objetivo se basa en la implementación de un método iterativo para la resolución del problema de valores propios. Para ello se utilizan los métodos basados en subespacios de Krylov que reducen el problema a operaciones de álgebra lineal densas. El método en particular se denomina Método de Arnoldi con Reinicio Implícito (IRAM) [7, 1, 6] y representa uno de los algoritmos más eficientes para la resolución de valores propios. Una vez obtenidas las estructuras densas, la operaciones básicas se manejan a través de las librerías optimizadas como BLAS y LAPACK implementadas para ejecutarse directamente en GPU.

La implementación realizada utiliza una de la más poderosa librerías para el manejo de matrices dispersas denominada CUSP[3]. Esta escrita en CUDA y contiene muchos algoritmos conocidos en el estado del arte en álgebra lineal. Además, la librería para cálculo con matrices densas denominada CULA[2] maneja implementaciones de las librerias LAPACK y BLAS en GPU. El proyecto pretende extender la librería CUSP implementando diferentes formatos de almacenamiento y algoritmos específicos para su tratamiento. El paso final se refiere en la análisis experimental. Los resultados obtenidos se han comparado con los obtenidos por la librería CUSP.

En el Capítulo 2 analizamos las principales características en la programación en CUDA y las técnicas para la optimización del código. En el Capítulo 3 se presentan los principales formatos de almacenamiento para matrices dispersas existentes. El Capítulo 4 analiza las propriedades que permiten aplicar optimizaciones en las funciones SpMV. En el Capítulo 5 se analizan los formatos aportados en este trabajo para las matrices. Luego en el Capítulo 6 se muestran las funciones SpMV para cada formato diseñado. En el Capítulo 7 se analiza el método del cálculo de valores propio. En particular, se hablará del estado del arte de este tipo de problema y se presentará una implementación a través de un método iterativo basado en subespacios de Krylov. Por último, en el Capítulo 8 se presentan los resultados obtenidos para todos los problemas de métodos iterativos tratados.

Capítulo 2

Programación Paralela en CUDA

En la programación paralela en CUDA una aplicación es compuesta de una programa secuencial en el host que puede ejecutar programas paralelos llamados *kernels* sobre un dispositivo paralelo. Un *kernel* es un SPMD (Single Program Multiple Data) que se ejecuta utilizando un gran número de hilos paralelos. Cada hilo ejecuta el mismo programa secuencial. El programador organiza los hilos de un *kernel* en una grid de bloques. Los hilos de un bloque pueden cooperar entre ellos utilizando un barrera de sincronizacion y a través de una memoria compartida privada por cada bloque. Aunque los *kernels* de CUDA se pueden compilar en código secuencial que puede ser ejecutados sobre cualquiera arquitectura que tiene un compilador en C, los *kernels* propuestos para spMV están diseñados para ser ejecutados en arquitecturas masivamente paralelas como, en particular, las tarjetas GPU NVIDIA. Esas arquitecturas permiten (1) la ejecución de hilos en la forma SIMD y (2) operaciones lectura/escritura coaleciente. Una moderna NVIDIA GPU se construye de un array de multiprocesadores, cada uno que soporta hasta 1024 hilos en ejecución. Un único multiprocesador tiene 8 cores escalares, 16384 registros a 32 bit. Operaciones con enteros y *float* en simple precisión pueden ser calculados a través de los 8 cores del multiprocesador, mientras una única unidad compartida se utiliza para operaciones en punto flotante en doble precisión. La creación de hilos, *scheduling*, y su gestión se hace enteramente en hardware. La NVIDIA GPU de última generación como GeForce GTX 280 o Tesla C1060 contienen 30 multiprocesadores, por un total de 30K hilos.

Para manejar esa gran población de hilos, la GPU utiliza un arquitectura SIMT (*Single Instruction Multiple Thread*) donde los hilos de un bloque se ejecutan en grupos de 32 llamado *warp*. Un *warp* ejecuta una simple instrucción a la vez a través de todos sus propios hilos. Los hilos de un *warp* pueden libremente seguir su propia ruta de ejecución (*execution path*) y todas las divergencias que aparecen vienen automáticamente gestionadas por hardware. De todas formas, es muy eficiente que los hilos siguen la misma ruta de ejecución. Además, cada hilo de un *warp* puede libremente acceder a locaciones de memoria diferentes para operaciones de lectu-

ra/escritura. El acceso en memoria en diferentes locaciones implica una divergencia que lleva al multiprocesador a ejecutar una transacción por cada hilo. Si las locaciones están suficientemente cerca, las operaciones por cada hilo pueden ser coalecientes llevando, así, una mayor eficiencia en el acceso en memoria. La memoria global está organizada en segmentos de 128 bytes. Las peticiones de memoria están servidos por 16 hilos (midad *warp*) a la vez. Si una petición de memoria por mitad *warp* toca solo un segmento, se dice que esa petición es completamente coaleciente, y en el caso que cada hilo toca un segmento separado de los otros se dice que la petición es no coaleciente.

La estructura básica de un programa en CUDA se basa sobre la función *kernel* indicada a traves de la clausola __global__. Una llamada a la función *kernel* es: kernel_function <<<P,B>>> que invoca P bloques de B hilos cada uno. Esos valores pueden ser diferentes por cada llamada del *kernel*, y una barrera implícita sirve para garantizar que todos los hilos de la primera llamada al *kernel* han terminado su ejecución antes que la segunda llamada sea invocada. Cuando un *kernel* está invocado sobre la GPU, el *scheduler* asigna los bloques de hilos a los cores de los multiprocesadores y continua el asignación hasta que hay los recursos disponibles.

Capítulo 3

Formatos de almacenamiento de matrices dispersas

Hay muchos formatos de almacenamiento para matrices dispersas y, cada uno con diferentes propiedades, características computaciónales, y métodos para el acceso y manipulación de los valores de la matriz. En el contexto de spMV no se tendrá en cuenta todas las operaciones que insertan o quitan elementos de la matriz. En las secciones siguientes realizaremos una comparativa de los diferentes *kernel* CUDA que utilizan algunos formatos de matriz dispersa disponibles para el problema spMV.

3.1. Diagonal (DIA)

Cuando los valores non-ceros están ubicados en un pequeño número de diagonales, el formato de almacenamiento diagonal (DIA) es apropiado. El formato DIA esta compuesto de dos arrays: *data*, que contiene los valores non-zeros, y *offset*, que almacena el desplazamiento de cada diagonal desde la diagonal principal. La diagonal principal tiene un desplazamiento igual a cero, mientras i>0 representa la i-ésima super-diagonal y i<0 la i-ésima sub-diagonal. La representación DIA almacena la diagonales en orden por columnas y los elementos con símbolo * se utilizan como *padding* y pueden almacenar valores arbitrario.

Los beneficios principales de eso formato son dos. Primero, los índices de fila y columna de cada elemento no nulo están implícitamente definidos a través de la posición del elemento en la diagonal y el desplazamiento correspondiente. La indicación implícita reduce la cantidad de memoria para la matriz y, también, la cantidad de datos que se necesita transferir desde el host hasta el device durante la operación spMV. Segundo, todos los accesos en memoria son contiguos y eso permite la coaleciencia de los datos durante la transferencia. El inconveniente más grande es por el hecho que DIA almacena elementos que están fuera de la matriz y elementos igual a cero

que están en las diagonales. El mejor caso es cuando tenemos matrices con muchos elementos en las diagonales cercanas a la diagonal principal.

3.2. Ellpack (ELL)

Otro esquema de almacenamiento es ELLPACK (ELL). Para una matriz MxN con un máximo de K elementos no nulos por fila, el formato ELLPACK almacena los valores non nulos en un array MxK, donde las filas con menos elementos respecto a K tienen añadido elementos de *padding*. De forma similar, el correspondiente índice de columna está almacenado en otro array de igual tamaño. ELL es más general que DIA. Cuando el número máximo de elementos no nulos de cada una de las filas no es muy diferente de la media, el formato ELL es un buen formato de almacenamiento.

3.3. Coordenado (COO)

El formato coordinate (COO) es el esquema de almacenamiento más sencillo. Utiliza dos arrays de enteros para almacenar los índices de fila y columna, y otro array de reales para almacenar los elementos no nulos de la matriz. COO es una representación general para matrices dispersa visto que la cantidad de memoria reservada es siempre proporcional al número de elementos no nulos.

3.4. Fila Dispersa Comprimida (CSR)

El formato de fila dispersa comprimida (CSR), es la más popular representación de matrices dispersa. Como el formato COO, CSR en manera explícita almacena los índices de columna y los elementos no nulos en dos array separados. Hay otro array de enteros que almacena los punteros sobre las filas. Para una matriz MxN, el puntero tiene magnitud M +1 y almacena el desplazamiento en la i-ésima fila del puntero. El último elemento del array de puntero, que debería corresponder a la (M + 1)-ésima fila, almacena el *nnz* (número de elementos no nulos). El CSR se puede ver como una extensión natural de COO, como un sencillo esquema de compresión aplicado por los índices de filas.

3.5. Diagonales Dentadas (JAD)

El formato JAD (JAgged Diagonal) resulta ser una generalización del formato Ellpack-Itpack donde se quita la suposición sobre la longitud fija de la fila. Para construir la estructura JAD, primero se ordenan las filas en orden decreciente respecto al número de elementos no nulos por fila. Un ejemplo se muestra en la siguiente figura.

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{pmatrix}, \quad PA = \begin{pmatrix} 3 & 4 & 0 & 5 & 0 \\ 0 & 6 & 7 & 0 & 8 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 9 & 10 & 0 \\ 0 & 0 & 0 & 11 & 12 \end{pmatrix}$$

Figura 3.1: Ejemplo de conversión de matriz en JAD

En este ejemplo A es la matriz original mientras que PA es la matriz ordenada. La estructura JAD también consta de tres arrays: un array de datos reales *DJ* que contiene los elementos no nulos, un array de índices *JDIAG* que contiene los índices de columna de cada elemento no nulo y un array *IDIAG* que contiene la posición inicial de cada *jagged* diagonal. Del ejemplo anterior obtenemos la estructura de la Figura 3.2.

DJ	3	6	1	9	11	4	7	2	10	12	5	8
JDIAG	1	2	1	3	4	2	3	3	4	5	4	5
IDIAG	1	6	11	13								

Figura 3.2: Estructura de datos en JAD

Capítulo 4

Propiedades y estructuras de las matrices

Antes de ver las implementaciones de SpMV con los diferentes formatos es importante observar la estructura de las matrices que utilizaremos en el análisis experimental. Como se ha indicado anteriormente, vamos a trabajar con las matrices que aparecen en la discretización de la Ecuación de Difusión Neutrónica con varios grupos de energía. Para cada grupo de energía tendremos asociada una matriz L_{ii} con la estructura mostrada en Figura 4.1.

Como se puede ver la matriz, debido a las condiciones iniciales impuestas, (que llamaremos L_{ii}) es simétrica no sólo en estructura sino también los valores son simétricos. Además, todos los elementos se disponen en diagonales relativamente cercanas a la diagonal principal. No todo los elementos de estas diagonales son no nulos sino que tenemos desplazamiento entre los elementos. Con la rutina *dinfo* de la librería SPARSKIT se puede obtener diversas informaciones, y la salida se presenta en la Figura 4.2.

La matriz es de tamaño 4680x4680 con 49776 elementos no nulos, el ancho de banda es 469 y no hay elementos no nulos que estén fuera de esa banda, así que por suerte la estructura es bastante definida. Dentro de la banda hay solo 27 diagonales con elementos no nulos. El problema principal como se puede ver es por el hecho que las diagonales contienen muchos elementos nulos. Para hacer un resumen de las características más importantes de L_{ii}:

- No hay elementos muy dispersos, siempre están en diagonales con otros elementos no nulos;
- Todas las diagonales están muy cerca de la diagonal principal;
- Las diagonales tienen elementos desplazados.

El formato DIA podría solucionar muchos de los problemas de almacenamiento de tal matriz, aunque nos llevaría almacenar elementos nulos de las diagonales. Esta situación la podríamos

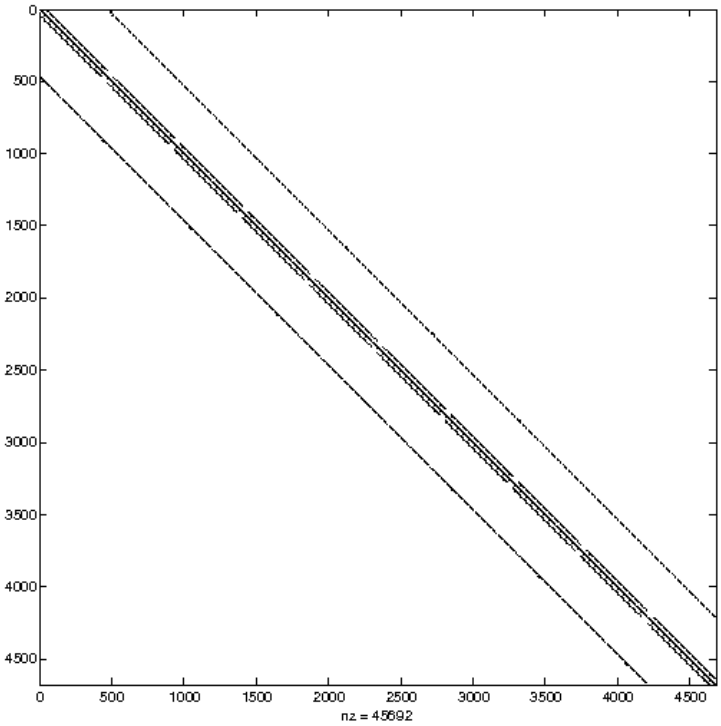

Figura 4.1: Matriz L_{ii}

resolver en parte a nivel del algoritmo para SpMV como veremos.

La otra matriz, que llamaremos T_{ii}, que se ha obtenido por discretizacion de un reactor nuclear, utilizando un mallado exagonal, se presenta en la Figura 4.3.

La matriz es 130075x130075 y tiene 5693803 elementos no nulos. Tal matriz presenta una forma bastante más compleja debido al hecho que el reactor resulta ser de una forma circular. En particular, si ampliamos la imagen (Figura 4.4) vemos como los elementos que están dispuestos en la diagonal llevan a una forma que no es rectilínea. Además, podemos ver que unos elementos están disperso por la matriz y no están dispuestos en las diagonales como los otros.

Una característica importante para el diseño del esquema de almacenamiento de los elementos, es que la forma de cada bloque es siempre idéntica en los otros bloques y lo que cambia son siempre los valores de los elementos de cada bloque.

```
1   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
2   * nforamcion de la matriz L11 a  traves de dinfo1                                       *
3   *                    Key =  ZL@@NN^@^@ , Type = RUA                                      *
4   * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
5       *  Dimension N                                      =       4680  *
6       *  Number of nonzero elements                       =      49775  *
7       *  Average number of nonzero elements/Column        =    10.6357  *
8       *  Standard deviation for above average             =     2.6881  *
9       *  Nonzero elements in strict lower part            =      22548  *
10      *  Nonzero elements in strict upper part            =      22548  *
11      *  Nonzero elements in main diagonal                =       4679  *
12      *  Weight of longest column                         =         16  *
13      *  Weight of shortest column                        =          6  *
14      *  Weight of longest row                            =         16  *
15      *  Weight of shortest row                           =          6  *
16      *  Matching elements in symmetry                    =      49775  *
17      *  Relative Symmetry Match (symmetry=1)             =     1.0000  *
18      *  Average distance of a(i,j)  from diag.           =  0.131E+03  *
19      *  Standard deviation for above average             =  0.197E+03  *
20      *-------------------------------------------------------------------*
21      *  Frobenius norm of A                              =  0.608E+05  *
22      *  Frobenius norm of symmetric part                 =  0.608E+05  *
23      *  Frobenius norm of nonsymmetric part              =  0.000E+00  *
24      *  Maximum element in A                             =  0.112E+04  *
25      *  Percentage of weakly diagonally dominant rows    =  0.753E+00  *
26      *  Percentage of weakly diagonally dominant columns =  0.753E+00  *
27      *-------------------------------------------------------------------*
28      *  Lower bandwidth  (max: i-j, a(i,j) .ne. 0)       =        469  *
29      *  Upper bandwidth  (max: j-i, a(i,j) .ne. 0)       =        469  *
30      *  Maximum Bandwidth                                =        938  *
31      *  Average Bandwidth                                =  0.852E+03  *
32      *  Number of nonzeros in skyline storage            =    3986742  *
33      *  90% of matrix is in the band of width            =        935  *
34      *  80% of matrix is in the band of width            =        935  *
35      *  The total number of nonvoid diagonals is         =         27  *
36      *  The 10 most important diagonals are (offsets)    :              *
37      *     0     4    -4   468  -468    44   -44     1    -1     3     *
38      *  The accumulated percentages they represent are   :              *
39      *   9.4  17.9  26.4  34.9  43.4  50.4  57.5  62.0  66.5  68.8     *
40      *-------------------------------------------------------------------*
```

Figura 4.2: Propiedades de la matriz L_{ii}

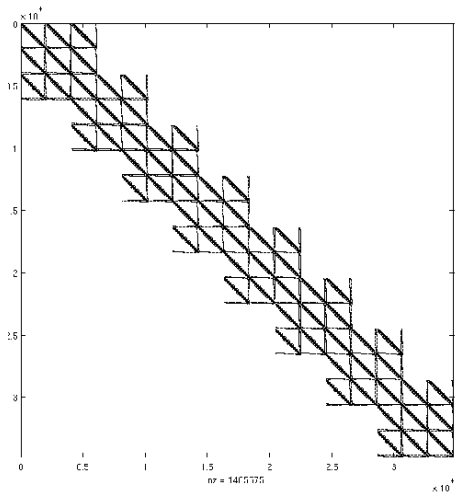

Figura 4.3: Matriz T_{ii}

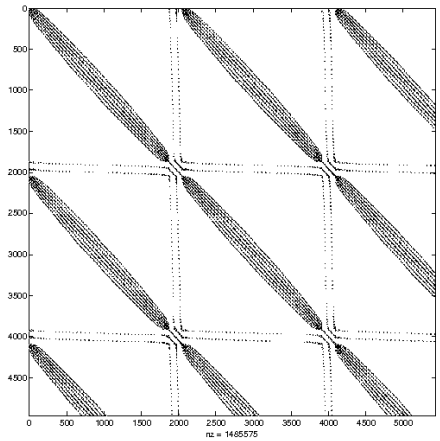

Figura 4.4: Zoom de la matriz T_{ii}

Capítulo 5

Análisis y justificación sobre los formatos de almacenamiento

Antes de ver las implementaciones podemos hacernos un idea de cual podría ser el formato de almacenamiento más adecuado en función de las características de las matrices anteriores. La multiplicación de matriz dispersa con el vector pide acceso a cada elemento de la matriz A una sola vez, mientras que cada elemento del vector x en el peor de los casos M veces (número de filas de A). Entonces, es muy fácil entender que poner elementos de A en memoria compartida no tiene mucho sentido mientras que se podría poner elementos de x en memoria compartida para que todos los hilos de un bloque puedan disfrutar de esos elementos cuando lo necesitan como si fuera una memoria cache. Si eso lo imaginamos en el formato COO eso no podría ser aplicable. En tal formato cada hilo de un bloque puede calcular con elementos de la matriz A que están también muy lejanos y de facto no hay una regularidad entre los elementos de la matriz. Para que todos esos hilos ejecutan elementos vecinos seria necesario hacer una lectura de todo el vector JA, y el coste seria prohibitivo debido al hecho que esta en la memoria global del device. Además, necesitaríamos hacer lectura de todo IA para individualizar la fila correspondiente del hilo. Por este motivo es claro que la técnica cache no es aplicable en el formato COO.

La idea sobre una multiplicación orientada a columna no es absolutamente aconsejado. Por ejemplo, una subdivisión en bloque de columna a cada hilo no conviene porque los resultados parciales que cada hilo obtiene de un elemento i-ésimo del vector necesitan ser sumados con los resultados obtenidos a través de otros hilos con la consiguiente comunicación entre estos hilos que están en bloques distintos y que podrían comunicar solo con la memoria global, añadiendo pasos de sincronización que reducen enormemente las prestaciones.

Otra idea seria la de utilizar formatos híbridos. En particular, dada la matriz A podríamos obtener el formato DIA sobre todos los elementos que están en diagonales con un elevado porcentaje de elementos no nulos, mientras que todos los otros elementos que están muy dispersos

en la matriz A, se podrían poner en otras estructuras más adecuadas como el COO. En realidad en la estructura de la matriz de la Figura 4.3, no hay elementos dispersos en toda la matriz y los elementos están situados en solo 27 diagonales. Así que, esta solución no nos llevaría a ningún resultado importante. Otra idea podría ser la de almacenar la mitad de la matriz para el memoria en caso de matriz simétrica, como sucede en algunos casos. Por ejemplo en el formato DIA se podría almacenar solo la mitad de diagonales. El principal problema es que el acceso en el array DIA del hilo asociado a la fila i-ésima no es contiguo ya que el hilo necesitaría también de elementos que están en otras filas. Como consecuencia no tendríamos un acceso coaleciente.

5.1. Formatos HYB y ELL a bloques

Viendo la matriz T_{ii} resulta claro que los formatos de almacenamiento se pueden organizar por bloques. En particular, se puede disponer de los formatos clásicos suministrados por la librería CUSP, como por ejemplo el formato híbrido, para construir un nuevo formato que maneja un conjunto de sub-matrices almacenadas con el formato híbrido y así poder llamar cada una de la rutina *kernel* ya implementada por CUSP. La principal ventaja de esta solución es el desarrollo de algoritmos que utilizan rutinas ya hechas y optimizadas de CUSP; el evidente problema es que la arquitectura de la tarjeta gráfica debe permitir la ejecución concurrente de rutinas *kernel* una por cada bloque de la matriz. En arquitecturas sencillas no esta permitida la ejecución concurrente de *kernels*, así que un tal algoritmo ejecutaría cada sub-bloque de la matriz de manera secuencial. Al final de todas las ejecución sería necesario sumar todos los vectores obtenidos por cada bloque de hilos.

Otra solución seria el modificar la rutina *kernel* existente para el el formato HYB de forma que los datos se organicen por bloques de filas y se asocia por cada una de ellos una sola vez el *kernel*. El problema reside en que el nuevo formato necesita muchas estructuras de datos: dos estructuras para las informaciones de los bloques y todas las estructuras de datos necesarias para almacenar los datos de la matriz en ELL y COO. Se puede reducir la complejidad de la implementación teniendo en cuenta solamente el formato ELL a bloques. El principal problema de esta solución es que dividiendo la matriz en muchos bloques pequeños se genera un número muy alto de elementos nulos almacenados en cada sub-matriz, con el consiguiente incremento en ejecución (hay muchas multiplicación entre elementos nulos), como se ve en la Figura 5.1.

A partir de esa consideración se puede aplicar formatos a bloque que disfrutan de la intrínseca replicación de información que tiene T_{ii}, como por ejemplo CSR a bloque.

5.2. Formatos CSR a bloques

Lo que parece bastante claro es que la matriz T_{ii} presenta un patrón en la estructura que se replica cada vez en toda la matriz. Eso significa que las informaciones de columnas y filas por

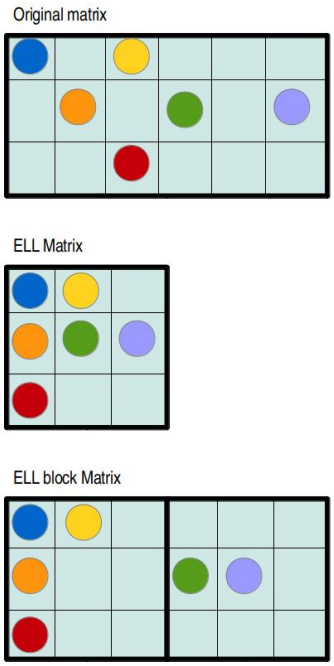

Figura 5.1: Conversión de la matriz en ELL block

cada elemento de la matriz puede ser almacenado una sola vez implicando un ahorro espacial muy grande. La información de la posición de cada bloque se almacena en dos estructuras de datos de forma parecida a lo que se hace con el CSR clásico sobre los elementos no nulos. En particular, tenemos que cada bloque no nulo (es decir, bloque que tiene por lo menos un elemento no nulo) está enumerado según las filas. La estructura *row_offset_block*, de tamaño igual al número de fila de bloque, tiene por cada fila de bloque el índice del primer bloque no nulo; mientras que *column_indices_block*, con tamaño igual al número de bloques no nulos, tiene la posición en columna de bloque por cada bloque no nulo. El almacenamiento de todos los elementos no nulos se hace de forma parecida al clásico CSR, utilizando las tres estructuras de datos, con la diferencia que se almacenan al principio los valores del primer bloque no nulo por fila y siguiendo así con los otros bloques. Esta forma de CSR a dos niveles permite un ahorro de diferente magnitud en el almacenamiento de los índices de cada elemento de la matriz respecto al clásico CSR (con respecto a la matriz T_{ii} el espacio ahorrado es alrededor de 60 veces). A continuación analizamos mediante un ejemplo la estructura comentada utilizando la en Figura 5.2.

Como se puede observar la matriz puede dividirse en bloques de manera parecida a la matriz

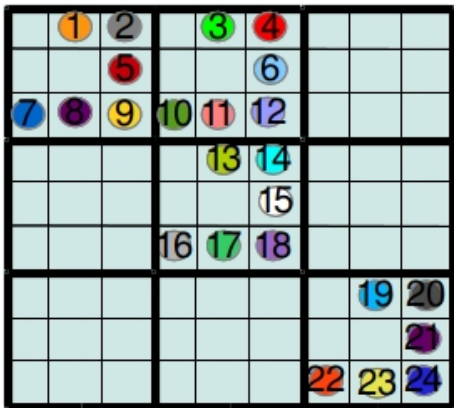

Figura 5.2: Ejemplo de matriz para convertirse en CSR block

T_{ii}. La información de los bloques se almacena en estructuras parecidas a la del CSR clásico como podemos ver en la Figura 5.3.

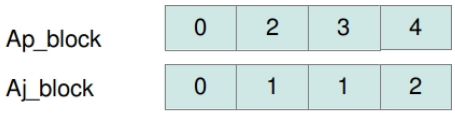

Figura 5.3: Estructuras de datos para los bloques en CSR block

En particular, *Ap_block* se refiere al desplazamiento hacia las filas de bloque y por eso tiene tamaño igual al número de fila de bloque más uno; mientras que *Aj_block* contiene los índices de columna de bloque de cada uno de los cuatro bloques no nulos. De manera parecida se almacena la estructura de los elementos dentro de cada bloque, y que representa el patrón que se repite por todos los bloques, como vemos en Figura 5.4.

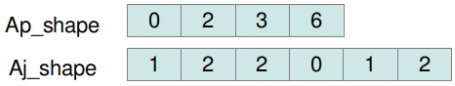

Figura 5.4: Estructuras de datos del patrón en CSR block

Por fin, todos los elementos se disponen dentro de un único array A_x donde los elementos están dispuestos según el orden de los índices de bloque que pertenecen.

De este formato se presentaran dos tipos de algoritmos de producto matriz-vector en la siguiente sección.

Figura 5.5: Estructura de datos A_x en CSR block

5.3. Formatos basados en JAD

Teniendo en cuenta las consideraciones obtenidas con los formatos analizados, los problemas encontrados han sido los siguientes:

- Coaleciencia parcial o nula;
- No homogénea distribución de la carga entre los bloques de hilos;
- Utilización no eficiente de la memoria compartida.

Con el JAD a bloque se solucionan totalmente o en parte los problemas anteriores. Además, se puede disponer de la memoria compartida almacenando completamente la estructura que almacena los desplazamientos de las diagonales *i_diag* en cuanto el tamaño depende del número de diagonales dentadas que muchas veces no resulta ser muy grande. Una consideración digna de nota es que cuando vamos a hacer la permutación de las filas, todos lo hilos del primer bloque tendrán más trabajo respecto a los otros bloques de hilos con consiguiente no homogénea distribución de la carga. Además, a diferencia del *kernel* CUSP CSR *scalar* que se adapta perfectamente con el formato JAD, el *kernel* CSR *vector* no conviene en el acceso de los elementos usando el formato JAD. En el CSR *vector* a cada fila se le asigna un entero *warp* (o potencia de dos hasta el número de hilos en un *warp*) de hilos resultando bastante eficiente en muchos tipos de matrices dispersas. La fila se va dividiendo y asignando cada elemento a un hilo, al final disfrutando de la memoria compartida e implícita sincronizacion de los hilos del mismo *warp* se hace la reducción de los resultados parciales de forma bastante rápida. Para hacer algo parecido en JAD es necesario una organización a bloque de la matriz y el esquema de almacenamiento tiene que ser de manera que hilos consecutivos, con asignada la misma fila, acceden en direcciones de memoria consecutivas. Para explicar el proceso de almacenamiento de JAD a bloques nos basaremos en el ejemplo de la Figura 5.6 cuya matriz tiene una forma parecida a la T_{ii}.

Cada bloque tiene la misma forma lo que cambia son los valores. La primera operación consiste en permutar las filas dentro de cada bloque obteniendo el resultado en Figura 5.7.

Como se puede observar ya en este paso, las filas con muchos elementos siguen manteniéndose distribuidas en la matriz así que todos los bloques de hilos tendrán la misma carga. Las estructuras de datos utilizadas para esta representación son muchas como veremos. La primera estructura sirve para almacenar la información sobre la permutación de las filas. A diferencia del formato

Figura 5.6: Ejemplo de matriz para convertirse en el formato JAD a bloque

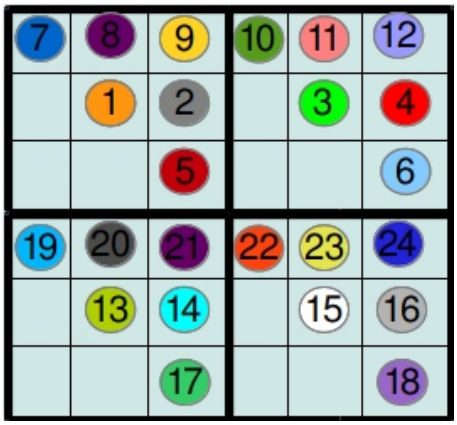

Figura 5.7: Permutación de las filas en bloques

JAD clásico no hace falta un vector de permutacion con tamaño igual al número de filas de la matriz, sino que sirve solo un vector de tamaño igual al tamaño de un bloque (en el nuestro ejemplo los bloques son matrices 3x3), entonces la estructura *perm* en Figura 5.8 almacenará solo tres elementos.

Figura 5.8: Estructura de datos para almacenar los índices de permutación

Exactamente como en el CSR a bloques tenemos que almacenar información sobre la posición de cada bloque enumerando los bloques no nulos según dos estructuras: una se refiere al deplazamiento de fila *Ap_block* y el otro indica el índice de columna por cada bloque no nulo *Aj_block*. Recordamos que en nuestro caso no tenemos bloques nulos, así que las estructuras son las que se muestran en Figura 5.9.

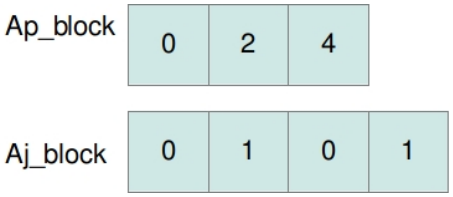

Figura 5.9: Estructuras de datos para JAD a bloque

La información dentro de un bloque se almacenan en dos estructuras similares a la del JAD clásico. Entonces, tenemos el desplazamiento de las diagonales dentadas *Ad_shape* y los índices de columna por cada elemento del bloque *Aj_shape* como se ve en Figura 5.10.

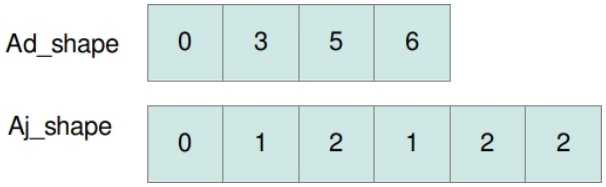

Figura 5.10: Estructuras de datos para el patrón en JAD a bloque

Por último en Figura 5.11 se muestra la estructura de los valores de todos los elementos contenidos en todos los bloques.

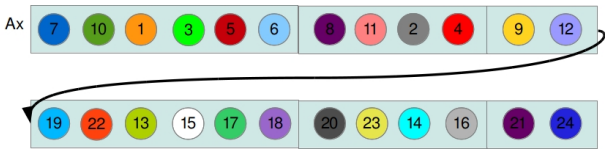

Figura 5.11: Estructura de datos para los elementos en JAD a bloque

Se cogen alternativamente elementos de las diagonales desplazadas en los bloques de una misma fila de bloque. Tal disposición de elementos sirve para garantizar un acceso coaleciente de parte de los hilos de un *warp* como veremos más en detalle cuando se analice la función *kernel* para este formato.

Capítulo 6

Multiplicación Matriz-Vector

La multiplicación matriz-vector (SpMV) es una de las operaciones más importante operación en la computación con matrices dispersas. Los métodos iterativos para la resolución de sistemas lineales ($Ax = b$) y problemas de valores propios ($Ax = \lambda x$) utilizan, generalmente, cientos o miles de productos matriz-vector para obtener la convergencia. En esta memoria consideramos, entonces, la operación $y = Ax + y$ donde A es una matriz dispersa y x e y son vectores columnas. El número de operaciones en coma flotante de $y = Ax + y$ es siempre dos veces el número de elementos no nulos de A (uno para el producto y el otro para la suma), independiente de la dimensión de la matriz. En ese capítulo se hablará de diferentes implementaciones de spMV para diferentes formatos de la matriz dispersa. Unas de las principales consideraciones son la convergencia de ejecución y el acceso a la memoria. El objetivo consiste en minimizar el número de rutas de ejecución que se van creado entre los hilos y permitir la coaleciencia cuando es posible.

6.1. Kernel DIA

Vista la natura de la matriz L_{ii}, el formato DIA resulta ser lo más adecuado. La paralelizacion de spMV es bastante sencilla: un hilo se asigna a cada fila de la matriz. Cada hilo al principio coge el índice de fila, y calcula el producto (disperso) entre la fila correspondiente y el vector x. Las diagonales de la matriz están almacenada por columna, y la linealización asegura que los hilos de un mismo *warp* acceden a la memoria de manera contigua. Contiguas filas, que corresponde a contiguos hilos, corresponden también a columnas consecutivas de la matriz, y hilos de un mismo *warp* acceden al vector x de manera contigua. La función *kernel* por el formato DIA se muestra en el Listado 6.1.

Listing 6.1: Kernel DIA

```
1 template <typename IndexType, typename ValueType, \
2 unsigned int BLOCK_SIZE, bool UseCache>
```

```
3  __launch_bounds__(BLOCK_SIZE,1)
4  __global__ void
5  spmv_dia_kernel(const IndexType num_rows,
6                  const IndexType num_cols,
7                  const IndexType num_diagonals,
8                  const IndexType pitch,
9                  const IndexType * diagonal_offsets,
10                 const ValueType * values,
11                 const ValueType * x,
12                       ValueType * y)
13 {
14     __shared__ IndexType offsets[BLOCK_SIZE];
15
16     const IndexType thread_id = BLOCK_SIZE * blockIdx.x + threadIdx.x;
17     const IndexType grid_size = BLOCK_SIZE * gridDim.x;
18
19     for(IndexType base = 0; base < num_diagonals; base += BLOCK_SIZE)
20     {
21         // read a chunk of the diagonal offsets into shared memory
22         const IndexType chunk_size = \
23         thrust::min(IndexType(BLOCK_SIZE), num_diagonals - base);
24
25         if(threadIdx.x < chunk_size)
26             offsets[threadIdx.x] = \
27             diagonal_offsets[base + threadIdx.x];
28
29         __syncthreads();
30
31         // process chunk
32         for(IndexType row = thread_id; row<num_rows; row+=grid_size)
33         {
34             ValueType sum = (base == 0) ? ValueType(0) : y[row];
35
36             // index into values array
37             IndexType idx = row + pitch * base;
38
39             for(IndexType n = 0; n < chunk_size; n++)
40             {
```

```
41                  const IndexType col = row + offsets[n];
42
43                  if(col >= 0 && col < num_cols)
44                  {
45                      const ValueType A_ij = values[idx];
46                      sum += A_ij * fetch_x<UseCache>(col, x);
47                  }
48
49                  idx += pitch;
50              }
51
52              y[row] = sum;
53          }
54
55          // wait until all threads are done reading offsets
56          __syncthreads();
57      }
58  }
```

En la implementacion original tenemos tres bucles *for* (19, 32, 39). En el primero coge BLOCK_SIZE diagonales por cada interacción de la estructura de datos *values* y cada hilo rellena un elemento de la variable compartida *offset*. En el segundo bucle (32) la variable *row* se refiere a la fila asignada al hilo y en el último bucle hace la multiplicación del vector x con la fila indicada por la variable *row*. La condición en 43 sirve para especificar si el elemento almacenado en el array resulta estar dentro de la matriz.

6.1.1. Técnica de la cache

Es posible mejorar las prestaciones de la precedente función añadiendo una memoria cache compartida entre los hilos de un bloque. Los elementos que están cerca de la diagonal principal son aquellos que con mucha probabilidad no serán nulos. Entonces, para cada bloque de hilos se puede asignar una memoria compartida para los elementos del vector x que están en correspondencia de la diagonal principal de la matriz.

Con una matriz que tiene un ancho de banda inferior a N, donde N es el número de hilos por bloque, tendremos que los tiempos de lecturas repetidas de elementos del vector x podrán ser reducida bastante a través de la memoria compartida *X_shar*. Es claro que en caso de diagonales que tienen un desplazamiento mayor de N necesitaran un acceso directo a la memoria global.

Listing 6.2: Kernel DIA_FAST

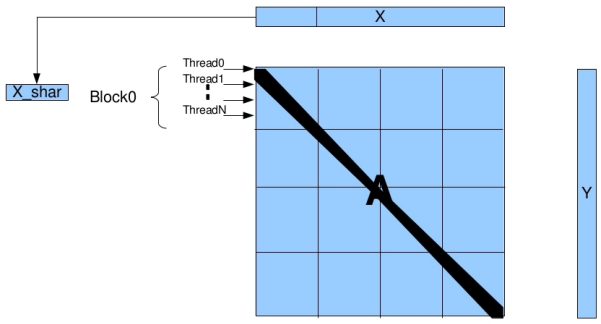

Figura 6.1: Técnica de la cache en el kernel DIA

```
1  template <typename IndexType, typename ValueType, \
2  unsigned int BLOCK_SIZE, bool UseCache>
3  __launch_bounds__(BLOCK_SIZE,1)
4  __global__ void
5  spmv_dia_fast_kernel(const IndexType base,
6                       const IndexType num_rows,
7                       const IndexType num_cols,
8                       const IndexType num_diagonals,
9                       const IndexType pitch,
10                      const IndexType * diagonal_offsets,
11                      const ValueType * values,
12                      const ValueType * x,
13                            ValueType * y)
14 {
15
16     __shared__ IndexType offsets[BLOCK_SIZE];
17      // Initialize the memory cache
18     __shared__ ValueType x_cache[BLOCK_SIZE];
19
20     if(threadIdx.x < num_diagonals)
21         offsets[threadIdx.x] = diagonal_offsets[threadIdx.x];
22
23     const IndexType row = large_grid_thread_id();
24     if(row >= num_rows){ return; }
25
```

```
26      const IndexType start_block = BLOCK_SIZE*(blockIdx.x);
27      const IndexType stop_block = BLOCK_SIZE*(blockIdx.x+1)-1;
28
29      x_cache[threadIdx.x] = (row<num_cols)? x[row]: ValueType(0);
30
31
32      __syncthreads();
33      ValueType sum = (base == 0) ? ValueType(0) : y[row];
34      // index into values array
35      IndexType idx = row;
36
37      for(IndexType n = 0; n < num_diagonals; n++)
38      {
39          const ValueType A_ij = values[idx];
40          if(A_ij!=ValueType(0)){
41
42              const IndexType col = row + offsets[n];
43              if(col >= start_block && col <=  stop_block)
44                  sum += A_ij * x_cache[col-start_block];
45
46              else
47                  sum += A_ij * fetch_x<UseCache>(col, x);
48          }
49          idx += pitch;
50      }
51      y[row] = sum;
52  }
```

En esta versión se declara la memoria *x_cache* (18). Además, se reduce el cálculo de cada hilo poniendo el bucle principal con la variable base fuera de la función *kernel*. Otra optimización sería el control del elemento de la matriz A en (40). Visto que tenemos elementos desplazados en la diagonal, es muy probable tener un elemento a cero, así que no tiene sentido hacer todo el cálculo que comportaría otras operaciones como una suma y multiplicación en coma flotante. Del otro lado esa operación nos lleva a una divergencia en la ruta de ejecución de los hilos, que podría comportar además a *branch misprediction*, pero desde el punto de vista experimental se observa que tal control nos lleva a un buen rendimiento.

6.2. Kernel ELL

El *kernel* ELL, como DIA, usa un hilo por cada fila de la matriz. La estructura del *kernel* ELL es bastante parecida al *kernel* DIA, con la principal excepción que los índices de columnas son explícitos en ELL e implícitos en DIA. De todas formas, el *kernel* ELL no hace un acceso contiguo sobre el vector x.

6.3. Kernel CSR

Consideramos la versión secuencial de spMV sobre el *kernel* ejecutado a través de la CPU. Visto que el producto entre una fila de la matriz y x puede ser computado de manera independiente de las otras filas, el *kernel* CSR se puede parallelizar fácilmente. La primera versión en CUDA del spMV con el formato CSR seria asociando un hilo a cada fila. Uno de los principales problemas de esta versión es el acceso del hilo dentro de un *warp*. El problema más significativo en esta solución está en la manera que los hilos dentro de un *warp* acceden a los arrays de datos e índice en CSR. Mientras los índices de columna y los valores no nulos están almacenados en manera contiguas para una fila, no se accede a esos valores simultáneamente. Una posible solución es asignar un *warp* por cada fila de la matriz. A diferencia de la versión escalar en la versión vectorial necesita coordinación entre los hilos de un *warp* que es posible obtener de manera automática gracias a la sincronizacion implícita de los hilos de un mismo *warp*. Por ejemplo, es necesaria una operación de reducción para la suma de los resultados de cada hilo en un mismo *warp*. En este caso el acceso a los índices de columna y a los valores pasa de manera contigua por todo el *warp*, superando el problema de la versión escalar. El formato CSR, en general, permite representar muchas clases de matrices dispersas de forma eficiente pero esa flexibilidad lleva a divergencia de los hilos. Por ejemplo, en la versión escalar donde cada hilo se le asigna una fila, en una matriz con alta variabilidad de elementos no nulos por cada fila, es muy probable que pasará un desequilibrio de la carga.

6.4. Kernel COO

El formato DIA y el ELL parecen más idóneos para el caso especifico de la matriz dispersa L_{ii}, mientras que el formato CSR resulta más general pero tiene el problema debido a la divergencia de los hilos cuando las filas presentan un número variable de elementos no nulos. El *kernel* COO permite que la computación del producto matriz-vector no dependa de la disparidad de la estructura de la matriz, y la idea es que a cada hilo se le asigna un elemento no nulo.

6.5. Kernel CSR a bloque 1

Del formato CSR a bloque se presentan dos tipos de algoritmos SpMV. El primero resulta ser más intuitivo y elegante (aunque eso no es sinónimo de eficiencia). Se asigna a cada bloque de hilo un bloque no nulo y cada bloque de hilos, siendo independiente de los otros, puede computar el cálculo con su propria matriz y obtener así el vector parcial. Cuando todos los bloques completan sus ejecuciones, otro *kernel* será ejecutado para hacer la suma de los vectores parciales.

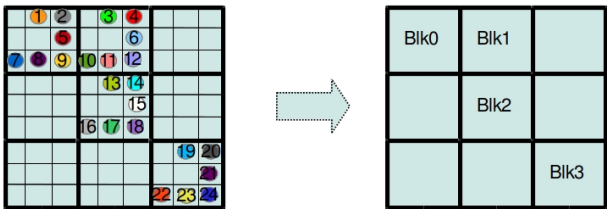

Figura 6.2: Asignación de sub-bloques de la matriz A a los bloques de hilos

Aunque hemos dicho antes que la idea sobre una multiplicación orientada a columna no es absolutamente aconsejada y resulta prohibitiva por el hecho de tener diferentes bloques de columnas, para nuestro caso particular la matriz presenta unos pocos bloques bien definidos por fila (tenemos solo 3 o 5 bloques por cada fila de bloque). La asignación de los bloques de hilos con los bloques de matriz es uno-a-uno como se puede observar en Figura 6.2. Esto permite una sencilla implementación en cuanto los datos en el CSR están almacenados con orden según los bloques de la matriz y por cada bloque de hilos se le asigna el desplazamiento en el array correspondiente (ver el Listado 6.3)

Listing 6.3: Kernel CSR block

```
1  template <bool UseCache,
2            typename IndexType,
3            typename ValueType,
4            unsigned int BLOCK_SIZE>
5  __launch_bounds__(BLOCK_SIZE,1)
6  __global__ void
7  spmv_csr_block_scalar_kernel(
8                          const IndexType num_rows,
9                          const IndexType num_cols,
10                         const IndexType nb_row,
11                         const IndexType nb_col,
12                         const IndexType num_blocks,
13                         const IndexType block_size,
```

```
14                          const IndexType * Ap_shape,
15                          const IndexType * Aj_shape,
16                          const IndexType * Ap_block,
17                          const IndexType * Aj_block,
18                          const ValueType * Ax,
19                          const ValueType * x,
20                                ValueType * y_tmp)
21 {
22
23
24     const IndexType row_offset_block=blockIdx.x*Ap_shape[block_size];
25     const IndexType column_index_block=block_size* \
26         Aj_block[blockIdx.x];
27     const IndexType row_index_block=block_size*blockIdx.x;
28
29     spmv_csr_block_scalar_device<UseCache, IndexType, ValueType,\
30         BLOCK_SIZE>(block_size, column_index_block,
31                         &Ap_shape[0],
32                         &Aj_shape[0],
33                         &Ax[row_offset_block],
34                         &x[0],
35                         &y_tmp[row_index_block]);
36
37 }
```

Donde las primeras tres constantes informan el desplazamiento y la función *spmv_csr_block_scalar_device* se muestra en el Listado 6.4 tiene la misma estructura del CSR secuencial.

Listing 6.4: Función SpMV por CSR block

```
1  template <bool UseCache,
2            typename IndexType,
3            typename ValueType,
4            unsigned int BLOCK_SIZE>
5  __device__ void
6  spmv_csr_block_scalar_device(const IndexType num_rows,
7                               const IndexType init_off_x,
8                          const IndexType * Ap,
9                          const IndexType * Aj,
10                         const ValueType * Ax,
```

```
11                          const ValueType * x,
12                                ValueType * y)
13 {
14     for(IndexType row=threadIdx.x; row<num_rows; row+=BLOCK_SIZE)
15     {
16         const IndexType row_start = \
17             fetch_vect_int_1<UseCache>(row, Ap);
18         const IndexType row_end = \
19             fetch_vect_int_1<UseCache>(row+1, Ap);
20
21         ValueType sum = 0;
22
23         for (IndexType jj = row_start; jj < row_end; jj++){
24             const IndexType Ajj = Aj[jj];
25             sum += Ax[jj] * \
26                 fetch_x<UseCache>(init_off_x + Ajj, x);
27         }
28         y[row] = sum;
29     }
30 }
```

Una vez que todos los bloques han calculado su vector parcial, la función *kernel* de reducción (Listado 6.5) sirve para sumar todos esos vectores.

Listing 6.5: Función de reducción por CSR block

```
1  template <typename IndexType,
2            typename ValueType,
3            unsigned int BLOCK_SIZE>
4  __global__ void
5  reduction_kernel(
6                          const IndexType num_rows,
7                          const IndexType num_cols,
8                          const IndexType nb_row,
9                          const IndexType nb_col,
10                         const IndexType num_blocks,
11                         const IndexType block_size,
12                         const IndexType * Ap_block,
13                         const ValueType * y_tmp,
14                               ValueType * y)
```

```
15 {
16
17     IndexType row_block_start = \
18         block_size*Ap_block[blockIdx.x];
19     IndexType row_block_end   = \
20         block_size*Ap_block[blockIdx.x+1];
21
22     for(IndexType row=threadIdx.x; row<block_size; row+=BLOCK_SIZE)
23     {
24         ValueType sum = 0;
25         for(IndexType b=row_block_start; b<row_block_end; \
26             b+=block_size){
27             sum += y_tmp[b + row];
28         }
29         y[blockIdx.x*block_size + row] = sum;
30     }
31 }
```

6.6. Kernel CSR a bloque 2

Esta segunda versión del *kernel* CSR resulta más eficiente. Para resolver el problema del retraso debido a la comunicación por la suma de vectores entre bloques de hilos hacia la misma fila de bloque, realizamos que cada bloque de hilos se le asigna una fila entera de bloques y se disfruta del mismo patrón repetido en la fila cargando los datos necesarios una sola vez en memoria compartida. Aplicando esas modificaciones aún tenemos unos problemas. Lo primero y más importante es que los datos del patrón son muchos para situarlos todos en memoria compartida. También en las ultimas arquitecturas de tarjeta NVIDIA® se puede llegar a un máximo de 48KB y las estructura de datos para el patrón son el *row_offset_shape* (5203 elementos enteros uno por cada fila más uno) y *column_índices_shape* (igual al número de elementos non nulos del patrón que son 58699) con un total necesario de 250KB. El segundo problema es debido al hecho que por como están organizados los elementos no nulos el acceso en cada fila resulta ser no secuencial y cada hilo del mismo *warp* puede acceder en segmentos de memoria diferentes.

En este caso se asigna todos los bloques de una fila por cada bloque de hilos. De esta manera almacenando las estructuras de datos (índices de columna y fila) sobre los bloques directamente en la memoria compartida el acceso sobre esos datos será más eficiente.

Listing 6.6: Kernel por CSR block2

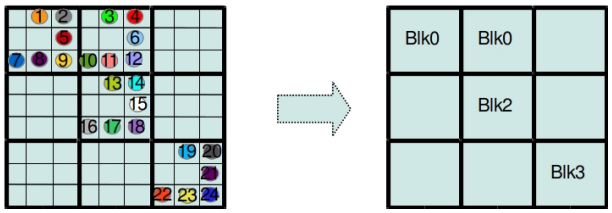

Figura 6.3: Asignación de sub-bloques de la matriz A a los bloques de hilos

```
1  extern __shared__ int array[];
2  template <bool UseCache,
3            typename IndexType,
4            typename ValueType,
5            unsigned int BLOCK_SIZE>
6  __launch_bounds__(BLOCK_SIZE,1)
7  __global__ void
8  spmv_csr_block_scalar2_kernel(
9                        const IndexType num_rows,
10                       const IndexType num_cols,
11                       const IndexType nb_row,
12                       const IndexType nb_col,
13                       const IndexType num_blocks,
14                       const IndexType block_size,
15                       const IndexType * __restrict__ Ap_shape,
16                       const IndexType * __restrict__ Aj_shape,
17                       const IndexType * __restrict__ Ap_block,
18                       const IndexType * __restrict__ Aj_block,
19                       const ValueType * __restrict__ Ax,
20                       const ValueType * __restrict__ x,
21                             ValueType * __restrict__ y)
22 {
23
24     // Aj_block in shared mem
25     IndexType* Aj_block_sh = (IndexType*)array;
26
27     const size_t start_row_block = Ap_block[blockIdx.x];
28     const size_t end_row_block = Ap_block[blockIdx.x + 1];
29     const size_t lenght_row_block = end_row_block-start_row_block;
```

```
30
31      for(size_t i=threadIdx.x; i<lenght_row_block; i+=BLOCK_SIZE)
32      {
33          Aj_block_sh[i] = Aj_block[start_row_block + i];
34      }
35
36      __syncthreads();
37
38      const IndexType num_entries_shape = Ap_shape[block_size];
39      const size_t row_index_block = block_size * blockIdx.x;
40
41      const size_t offset_block = start_row_block*num_entries_shape;
42
43      for(size_t row=threadIdx.x; row<block_size; row+=BLOCK_SIZE)
44      {
45
46          const IndexType row_start = Ap_shape[row];
47          const IndexType row_end   = Ap_shape[row+1];
48
49          ValueType sum = 0;
50          for (IndexType jj = row_start; jj < row_end; jj++){
51              const IndexType Aj = Aj_shape[jj];
52
53              #pragma unroll
54              for(size_t jj_block=0; jj_block<lenght_row_block; \
55                  jj_block++){
56                  const IndexType col_offset = Aj_block_sh[jj_block]*\
57                      block_size;
58                  sum += \
59                      Ax[offset_block + jj_block*num_entries_shape + jj]*\
60                      fetch_x<UseCache>(col_offset + Aj, x);
61                  }
62          }
63          y[row_index_block + row] = sum;
64      }
65  }
```

En la implementación tenemos unos bucles debido a la jerarquía del tal formato y los bucles

están organizados para que el más interno sea el que lleve menos iteraciones y se refiere al acceso a los datos que ya están en la memoria compartida (los índices de los bloques). Desafortunadamente, como hemos ya dicho, los datos de índices de columna y fila de CSR dentro del bloque (que recordamos son iguales para cualquier bloque de la matriz) de la matriz T_{ii} son de tamaño muy grande para poder estar almacenados en memoria compartida, así que hubiera sido posible hacer un único acceso en memoria global por todos los bloques de una fila. El problema principal de ambos kernels CSR a bloque está en la asignación estática entre bloque de hilos y bloque de matriz. Aunque eso simplifica la implementación, en la mayoría de los casos los bloques de la matriz son grandes y de pocos elementos, así que la ejecución de un *kernel* se hace con un número pequeño de bloques de hilos cada uno con una carga muy alta. Entonces, una solución más adecuada seria la de disponen de todo los hilos posibles en función de las prestaciones de la arquitectura.

6.7. Kernel JAD

Como en el caso del *kernel* CSR ya presente en CUSP, tenemos que solo un hilo trabaja sobre una fila disfrutando de un paralelismo con granularidad fina. Se puede observar que los datos y los índices de columna están almacenados en diagonales dentadas, así que el *kernel* JAD no tiene los mismos problemas encontrados que en el *kernel* CSR. Consecutivos hilos pueden acceder en direcciones de memoria contiguas mejorando el ancho de banda a través de transacciones de memoria coalecientes. El patrón de acceso a la memoria se muestra en la Figura 6.4.

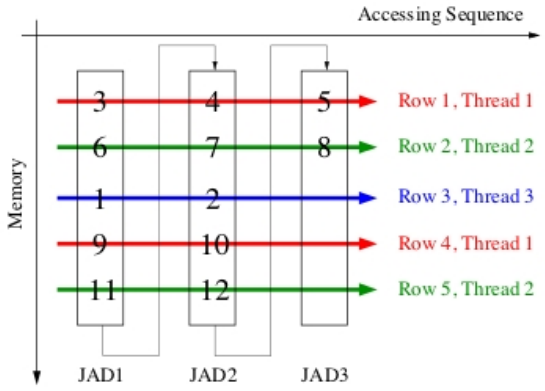

Figura 6.4: Patrón de acceso a la memoria global en JAD

El *kernel* con el formato JAD esta organizado de manera que cada hilo ejecuta una fila de la matriz. En cada bloque de hilos se almacena la estructura de datos *diagonal_offsets* que define el desplazamiento de cada diagonal dentada en cuanto en la mayoría de los casos esta estructura

tiene un tamaño pequeño. Cada hilo tendrá asignada la fila i-ésima y por cada diagonal dentada es necesario controlar si la longitud de la diagonal es mayor o igual a *i*. Si resulta que esta condición no se satisface, el hilo puede parar su ejecución en cuanto las diagonales dentadas están ordenada de manera decreciente, así que el hilo en la fila *i* nunca encontrará elementos de las otras diagonales dentadas remanentes. La función *kernel* para el formato JAD es presentado en el Listado 6.7.

Listing 6.7: Kernel por JAD

```
1
2  template <bool UseCache, unsigned int BLOCK_SIZE,
3            typename IndexType,
4            typename ValueType>
5  __launch_bounds__(BLOCK_SIZE, 1)
6  __global__ void
7  spmv_jad_scalar_kernel(const IndexType num_rows,
8                         const IndexType num_cols,
9                         const IndexType num_jagged_diagonals,
10                        const IndexType * __restrict__ Ad,
11                        const IndexType * __restrict__ Aj,
12                        const ValueType * __restrict__ Ax,
13                        const ValueType * __restrict__ x,
14                              ValueType * __restrict__ y)
15 {
16
17     const IndexType thread_id = blockDim.x*blockIdx.x+threadIdx.x;
18     const IndexType grid_size = gridDim.x*blockDim.x;
19
20     extern __shared__ IndexType array[];
21     volatile IndexType* Ad_s = (IndexType*)array;
22
23
24
25     for(int i=threadIdx.x; i<= num_jagged_diagonals; i+=BLOCK_SIZE)
26         Ad_s[i] = Ad[i];
27
28     __syncthreads();
29
30     for(register IndexType row=thread_id; row<num_rows; row+=grid_size)
```

```
31      {
32          ValueType sum = 0;
33
34          IndexType dia_start = Ad_s[0];
35          IndexType dia_stop = Ad_s[1];
36          IndexType dia_lenght = dia_stop - dia_start;
37          register IndexType dia = 0;
38          while((dia_lenght >row) && dia < num_jagged_diagonals){
39
40              sum += Ax[dia_start + row] * \
41                  fetch_x<UseCache>(Aj[dia_start + row], x);
42
43              dia++;
44              if(dia<num_jagged_diagonals){
45                  dia_start = dia_stop;
46                  dia_stop = Ad_s[dia+1];
47                  dia_lenght = dia_stop - dia_start;
48              }
49          }
50          y[row] = sum;
51      }
52  }
```

6.8. Kernel JAD a bloque

A través del formato JAD a bloque se quiere asignar a cada fila de la matriz un entero *warp* o una porción de este para llegar a una granularidad aún más fina que con el formato JAD clásico.

Como en el caso del CSR a bloque se asigna a cada bloque de hilos una fila entera de bloque, así que en el ejemplo que se muestra en Figura 6.5 tendremos dos bloques de hilos. Además, se puede observar desde la figura que es necesaria una operación de permutación a priori para obtener una nueva matriz $\mathbf{A}_p$ con filas permutadas, como se ha visto en el capítulo anterior (5). Una vez realizada la multiplicación con esta matriz lo que obtenemos será un vector permutado y_p, así que una nueva función *kernel* deberá actuar la permutación para obtener el vector y deseado.

A cada hilo le será asignado una fila dentro de un bloque, así que el hilo 0 tendrá los elementos 7, 8, 9; el hilo 1 los elementos 10, 11, 12; el hilo 2 los elementos 1, 2; y así sucesivamente. Está claro que si almacenamos los elementos por fila no será posible asegurarse un acceso coalesciente. El

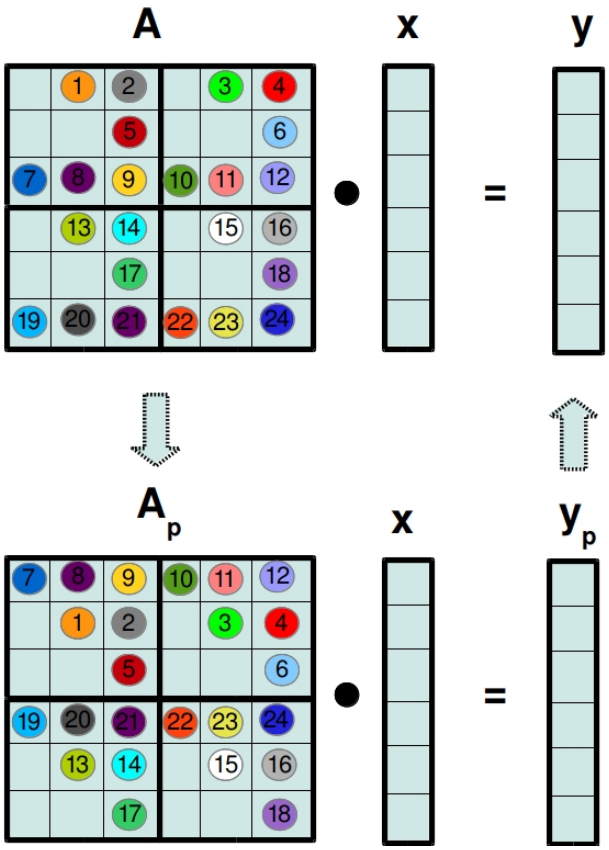

Figura 6.5: Ejemplo de matriz a bloque para convertirse en el formato JAD block

patrón de acceso a la memoria se muestra en Figura 6.6.

Entonces, para lograr la coalescencia, vamos cogiendo alternativamente elementos que pertenecen a las diagonales dentadas entre los bloques de la misma fila de bloque. Las flechas pequeña indican el siguiente elemento almacenado por orden, mientras las flechas grandes se refieren a los accesos de parte de cada hilo. Los hilos que pertenecen a un fila están siempre dentro de un *warp* así que, una vez que cada hilo ha calculando su resultado parcial, se aplica la reducción utilizando la memoria compartida.

6.9. Resumen

El *kernel* COO tiene la más fina granularidad (un hilo por elemento no nulo), mientras la versión vector de CSR y el JAD a bloque asigna una fila a un *warp*, así que una porción de fila se le asigna a un hilo. Todos los otros *kernel* tienen una granularidad de fila por hilo. Es importante recordar que la utilización de la GPU pide muchos hilos activos. Entonces, una granularidad

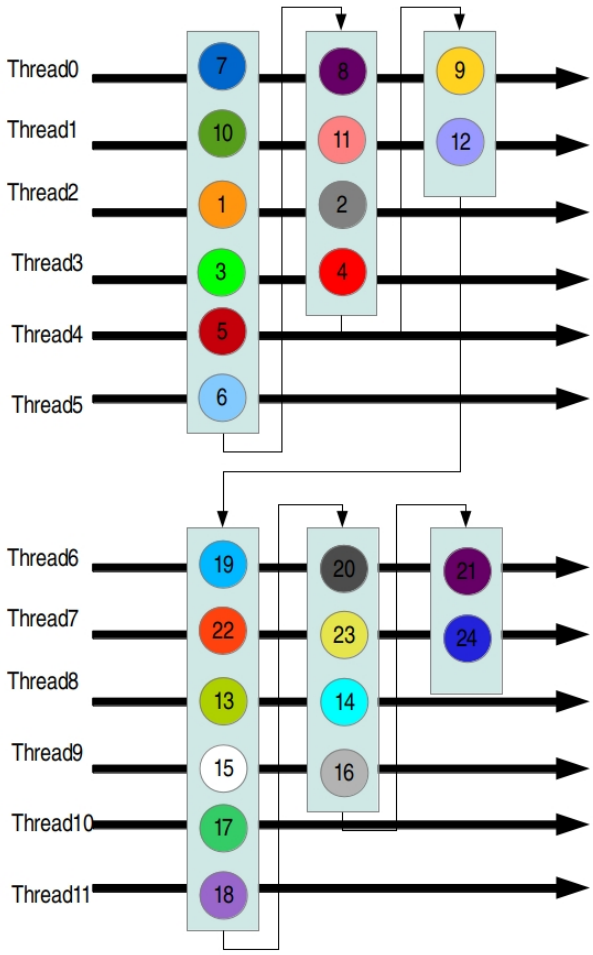

Figura 6.6: Patrón de acceso con el formato JAD block

más fina como el CSR vector, JAD a bloque o COO toma muchas ventajas con matrices con un número limitado de filas. A excepción de CSR todos los métodos tienen una coaleciencia perfecta para el acceso a la matriz dispersa. El *kernel* DIA tiene el menor ratio de bytes por FLOP, por lo que tiene la mayor intensidad computacional. El formato COO almacena de manera explicita el índice de fila y columna, y, así, tiene la más baja intensidad. El JAD a bloque permite una mejor distribución de las filas a los bloques de hilos, asegurando una carga constantes por todos los bloques, aunque desde el punto de vista de la implementación resulta lo más complejo en cuanto lleva muchas estructuras que añaden más overhead al sistema. Todos los *kernel* de SpMV utilizan la cache de textura que mejora significativamente las prestaciones.

Como hemos dicho la operación SpMV resulta fundamental en muchos métodos iterativos. La librería CUSP [3] nos da una serie de métodos iterativos basados en el subespacio de Krylov

para la resolución de sistemas lineales. En la Sección 8 hablaremos de las prestaciones obtenidas tanto en SpMV como en la resolución de sistemas de ecuaciones lineales y del cálculo de valores propios.

En la siguiente Sección trataremos el problema del cálculo de valores propios. En CUSP no existen aún implementaciones para la resolución de este problema que resulta ser uno de los más importantes.

Capítulo 7

Cálculo de Valores Propios

En Este capítulo se presenta la teoría sobre los métodos que utilizan proyecciones de sub-espacio de Krylov para el cálculo de valores propios. El método utilizado es el presentado en la implementación de la librería ARPACK [1] que se denomina Método de Arnoldi con Reinicio Implícito (IRAM). La primera parte del capítulo explica las nociones necesarias para entender los orígenes, motivaciones, y el comportamiento de tal algoritmo. EL argumento empieza tratando la estructura del problema de valores propios y unos métodos básicos numéricos que están relacionados con IRAM. La operación más importante es la factorización de Arnoldi que representa la manera de construir una base ortogonal para el sub-espacio de Krylov. El método con Reinicio Implícito se introduce como una eficiente manera de superar el almacenamiento y el cálculo que resulta intratable en el método original de Arnoldi. Gracias a esa característica, el método resulta apto para problemas de valores propios de gran tamaño. Además, el método permite aproximar un número de valores propios definido desde el usuario con un espacio proporcional a $n \times k$ donde k es el número de valores propios deseados por el usuario. El método básico de IRAM es el siguiente.

- *Comienzo*: Construye la factorización de Arnoldi de longitud m: $\mathbf{AV}_m = \mathbf{V}_m\mathbf{H}_m + \mathbf{f}_m\mathbf{e}_m^T$ con el vector inicial $\mathbf{v}_1$
- *Iteración*: Hasta la convergencia
  1. Calcula los valores propios $\{\lambda_j : j = 1, 2, \ldots, m\}$ de $\mathbf{H}_m$. Ordena los valores propios según el usuario en el conjunto de valores propios deseados $\{\lambda_j : j = 1, 2, \ldots, k\}$ y no deseados $\{\lambda_j : j = k+1, k+2, \ldots, m\}$.
  2. Realiza $m - k = p$ pasos de factorización QR sobre los valores propios no deseados haciendo operaciones de shift y obteniendo $\mathbf{H}_m\mathbf{Q}_m = \mathbf{Q}_m\mathbf{H}_m^+$.
  3. *Reiniciar*: Post multiplica la factorización de Arnoldi para la matriz $\mathbf{Q}_k$ que representa la principal submatriz de $\mathbf{Q}_m$ para obtener la factorización de Arnoldi de longitud k:

$\mathbf{AV}_m\mathbf{Q}_k = \mathbf{V}_m\mathbf{Q}_k\mathbf{H}_k^+ + \mathbf{f}_k^+\mathbf{e}_k^T$, donde $\mathbf{H}_k^+$ es la principal submatriz de orden k de $\mathbf{H}_m^+$ y $\mathbf{V}_k \leftarrow \mathbf{V}_m\mathbf{Q}_k$.

4. Adjunta la factorización de Arnoldi de longitud k con la factorización de Arnoldi de longitud m.

En el algoritmo mostrado, $\mathbf{H}_m$ es una matriz $m \times m$ de Hessenberg superior, $\mathbf{V}_m^T\mathbf{V}_m = \mathbf{I}_m$, y el vector residuo $\mathbf{f}_m$ es ortogonal a las columnas de $\mathbf{V}_m$.

7.1. Estructura del Problema de Valores Propios

En esta sección se introduce una pequeña discusión sobre la estructura matemática del problema de valores propios. La discusión se refiere al caso que los elementos de la matriz son reales y los valores propios resultan también reales.

El conjunto $\sigma(\mathbf{A}) = \{\lambda \in R : rank(\mathbf{A} - \lambda\mathbf{I}) < n\}$ se llama *espectro* de $\mathbf{A}$. Los elementos de este conjunto representan los valores propios de $\mathbf{A}$ y pueden ser caracterizados como las n raíz del polinomio característico $p_{\mathbf{A}}(\lambda) = det(\lambda\mathbf{I} - \mathbf{A})$. Por cada distinto valor propio $\lambda \in \sigma(\mathbf{A})$ corresponde un vector propio $\mathbf{x}$ tal que $\mathbf{Ax} = \lambda\mathbf{x}$. Este vector se denomina vector propio derecho de $\mathbf{A}$ correspondiente al valor propio λ.

7.2. Algoritmo QR con Desplazamiento Implícito

El algoritmo descrito en esa sección resulta fundamental para la trasformación de la matriz $\mathbf{A}$ en la forma triangular superior. El principal objetivo del algoritmo es obtener una descomposición de Schur que permite el desarrollo para una estabilidad numérica del algoritmo. En particular, el algoritmo, que se denomina QR con Desplazamiento Implícito, produce una secuencia de trasformaciones con $\mathbf{Q}_j$ que de manera iterativa reduce $\mathbf{A}$ en forma triangular superior.

La descomposición de Schur afirma que por cada matriz cuadrada existe una similar representación de matriz triangular superior.

Teorema 1. *(Descomposición de Schur). Dado $\mathbf{A} \in R^{nxn}$, hay una matriz unitaria $\mathbf{Q}$ y una matriz triangular superior $\mathbf{R}$ tal que*

$$\mathbf{AQ} = \mathbf{QR} \tag{7.1}$$

Los elementos de la diagonal de $\mathbf{R}$ son los valores propios de A.

El algoritmo empieza con una trasformación inicial de $\mathbf{A}$ por $\mathbf{V}$ a la forma $\mathbf{AV} = \mathbf{VH}$ donde $\mathbf{H}$ es una matriz de Hessenberg. En una matriz de Hessenberg se tiene que todos los elementos por debajo de la subdiagonal son nulos por lo que resulta ser casi una matriz triangular superior.

Algorithm 1 Algoritmo QR con Desplazamiento Implícito

Input: ($\mathbf{A}$, $\mathbf{V}$, $\mathbf{H}$) con $\mathbf{AV} = \mathbf{VH}$, $\mathbf{V^T V} = \mathbf{I}$, H es una matriz Hessenberg Superior;
for $i = 1, 2, \ldots,$ *hasta convergencia* **do**
  Selecciona uno shift $\mu \leftarrow \mu_j$;
  Calcula $\mathbf{Q}, \mathbf{R} = \mathbf{qr}(\mathbf{H} - \mu\mathbf{I})$;
  $\mathbf{H} \leftarrow \mathbf{Q^T HQ}; \mathbf{V} \leftarrow \mathbf{VQ}$;
end for

El algoritmo QR con Desplazamiento Implícito se presenta en el Algoritmo 1. La factorización de $\mathbf{H} - \mu\mathbf{I}$ proporciona la matriz ortogonal $\mathbf{Q}$ y la matriz triangular $\mathbf{R}$. La ejecución continua hasta que los elementos de la subdiagonal de $\mathbf{H}$ llegan a cero.

A pesar de la rapidez en la convergencia y la eficiencia en el almacenamiento, el algoritmo QR con Desplazamiento Implícito no es aconsejable para problemas de gran tamaño y resulta, también, difícil de paralelizar.

7.3. Métodos de Proyección basados en Subespacios de Krylov

Los métodos de Proyección tienen la ventaja sobre la secuencia de vectores obtenidos con el método de la potencia. Los sucesivos vectores obtenidos con el método de la potencia pueden contener importante información sobre la dirección de los valores propios. Entonces, combinaciones lineales de los vectores pueden construirse para mejorar la convergencia en el cálculo de los vectores propios. El espacio de Krylov se define:

$$\mathbf{K}_k(\mathbf{A}, \mathbf{v}_1) = Span\{\mathbf{v}_1, \mathbf{A}\mathbf{v}_1, \mathbf{A}^2\mathbf{v}_1, \ldots, \mathbf{A}^{k-1}\mathbf{v}_1\} \tag{7.2}$$

En particular, se puede demostrar que $\mathbf{K}_k$ es un subespacio invariante por $\mathbf{A}$ si y sólo si el vector inicial $\mathbf{v}_1$ es una combinación lineal de los vectores generadores del subespacio invariante de $\mathbf{A}$. Esta importante observación nos llevará, como veremos en la siguiente sección, a que los valores y vectores propios de la matriz $\mathbf{A}$ se pueden obtener a través de la matriz de Hessenberg $\mathbf{H}$ obtenida de la factorización de Arnoldi.

7.4. Factorización de Arnoldi

Definición 1. *Si* $\mathbf{A} \in R^{n \times n}$ *entonces una factorización de la forma*

$$\mathbf{AV}_k = \mathbf{V}_k\mathbf{H}_k + \mathbf{f}_k\mathbf{e}_k^T \tag{7.3}$$

donde $\mathbf{V}_k \in R^{n\times k}$ *tiene columnas ortogonales,* $\mathbf{V}_k^T\mathbf{f}_k = 0$ *y* $\mathbf{H}_k \in R^{k\times k}$ *es una matriz de Hessenberg superior con elementos de la subdiagonal no negativos, se llama Factorización de Arnoldi de paso* k *de la matriz* $\mathbf{A}$*.*

Una alternativa manera de escribir la factorización es

$$\mathbf{A}\mathbf{V}_k = (\mathbf{V}_k, \mathbf{v}_{k+1})\begin{pmatrix}\mathbf{H}_k \\ \beta_k \mathbf{e}_k^T\end{pmatrix} \quad \text{donde } \beta_k = \|\mathbf{f}_k\| \text{ y } \mathbf{v}_{k+1} = \frac{1}{\beta_k}\mathbf{f}_k. \tag{7.4}$$

El algoritmo que define los pasos necesarios para formar una factorización de Arnoldi de k-pasos se muestra en Algoritmo 2.

Algorithm 2 Algoritmo para la Factorización de Arnoldi de k pasos

Input: $(\mathbf{A}, \mathbf{v}_1)$
$\mathbf{v}_1 = \mathbf{v}/\|\mathbf{v}_1\|$; $\mathbf{w} = \mathbf{A}\mathbf{v}_1$; $\alpha_1 = \mathbf{v}_1^T\mathbf{w}$;
$\mathbf{f}_1 \leftarrow \mathbf{w} - \mathbf{v}_1\alpha_1$; $\mathbf{V}_1 \leftarrow (\mathbf{v}_1)$; $\mathbf{H}_1 \leftarrow (\alpha_1)$;
for $j = 1, 2, 3, \ldots, k-1$ **do**
  $\beta_j = \|\mathbf{f}_j\|$; $\mathbf{v}_{j+1} \leftarrow \mathbf{f}_j/\beta_j$;
  $\mathbf{V}_{j+1} \leftarrow (\mathbf{V}_j, \mathbf{v}_{j+1})$; $\hat{\mathbf{H}}_j \leftarrow \binom{\mathbf{H}_j}{\beta_j \mathbf{e}_j^T}$;
  $\mathbf{w} \leftarrow \mathbf{A}\mathbf{v}_{j+1}$;
  $\mathbf{h} \leftarrow \mathbf{V}_{j+1}^T\mathbf{w}$; $\mathbf{f}_{j+1} \leftarrow \mathbf{w} - \mathbf{V}_{j+1}\mathbf{h}$;
  $\mathbf{H}_{j+1} \leftarrow (\hat{\mathbf{H}}_j, \mathbf{h})$;
end for

Las columnas de $\mathbf{V}_j$ representan una base ortonormal del subespacio de Krylov y $\mathbf{H}_j$ es la proyección ortogonal de $\mathbf{A}$ sobre tal espacio.

Desde el punto de vista computacional las operaciones de matriz-vector densas y las actualizaciones de los vectores pueden ser calculados utilizando rutinas de según nivel del BLAS. Por ejemplo, como veremos en la implementación del algoritmo, GEMV es la rutina del producto matriz-vector que puede ser sencillamente paralizada teniendo también un mejor nivel de ratio entre operaciones en coma flotante y cantidad de datos de trasferencia [11] respecto el nivel uno del BLAS.

El resultado que se obtiene depende de como se elige el vector inicial $\mathbf{v}_1$. En particular, los valores propios pueden no aparecer hasta que k tome un valor muy grande. Además, se necesita almacenar grandes cantidades de elementos debido al tamaño de $\mathbf{V}_k$ y la búsqueda de valores propios de $\mathbf{H}_k$ resulta prohibitivo con un coste de $\mathcal{O}(k^3)$.

Se demuestra que la perdida de ortogonalidad de las columnas de $\mathbf{V}_k$ pasa exactamente cuando los valores propios de $\mathbf{H}_j$ están cerca de los valores propios de $\mathbf{A}$. Es deseable que $\|\mathbf{f}_k\|$ sea lo más pequeña posible, porque eso indica que los valores propios de $\mathbf{H}_k$ resultan cerca de los valores propios de $\mathbf{A}$.

7.5. Método de Arnoldi con Reinicio Implícito

Un importante problema del proceso de Arnoldi es que no se puede conocer apriori cuantos pasos son necesarios para que los valores propios de la matriz **H** sean una buena aproximación. En particular, eso pasa cuando la matriz de **A** tiene un ancho rango de valores propios, y tales valores propios están dispuestos en clusters.

Un método de reinicio fue propuesto por Saad como solución iterativa para sistemas de ecuaciones lineales. Saad [13] propuso reiniciar la factorización con un vector precondicionado que está más cerca en un subespacio invariante k-dimensional. La técnica se llama de reinicio implícito y combina el QR desplazado implícito con una factorización de k pasos de Arnoldi para obtener una forma truncada del QR Desplazamiento Implícito. La dificultad numérica y de almacenamiento presentadas en la Sección 7.4 están resueltos a través de este método. El algoritmo es capaz de calcular k valores propios, donde k representa el número de valores propios queridos por el usuario, utilizando una memoria proporcional a $2nk + \mathcal{O}(k^2)$.

El método de Arnoldi con reinicio implícito se muestra en Algoritmo 3. Una factorización de Arnoldi de $m = k + p$-pasos

$$\mathbf{AV}_m = \mathbf{V}_m\mathbf{H}_m + \mathbf{f}_m\mathbf{e}_m^T, \tag{7.5}$$

se comprime a una factorización de longitud k que retiene los valores propios de interés. Eso se obtiene haciendo p shift QR implícitos. Las k primeras columnas quedan en una relación de Arnoldi y representan el punto de inicio para la siguiente iteració.

Algorithm 3 Método de Arnoldi con Reinicio Implícito (IRAM)

Input: ($\mathbf{A}$, $\mathbf{V}$, $\mathbf{H}$, $\mathbf{f}$) con $\mathbf{AV}_m = \mathbf{V}_m\mathbf{H}_m + \mathbf{f}_m\mathbf{e}_m^T$, una Factorización de Arnoldi de m-pasos;
for $l = 1, 2, \ldots,$ *hasta convergencia* **do**
  Calcula $\sigma(\mathbf{H}_m)$ y selecciona según un criterio un conjunto con p valores propios $\mu_1, \mu_2, \ldots, \mu_p$ en $\sigma(\mathbf{H}_m)$;
  $\mathbf{q}^T \leftarrow \mathbf{e}_m^T$;
  for j=1,2,...,p **do**
    $\mathbf{Q}, \mathbf{R} = qr(\mathbf{H}_m - \mu_j\mathbf{I})$;
    $\mathbf{H}_m \leftarrow \mathbf{Q}^T\mathbf{H}_m\mathbf{Q}$; $\mathbf{V}_m \leftarrow \mathbf{V}_m\mathbf{Q}$;
    $\mathbf{q} \leftarrow \mathbf{q}^T\mathbf{Q}$;
  end for
  $\mathbf{f}_k \leftarrow \mathbf{v}_{k+1}\hat{\beta}_k + \mathbf{f}_m\sigma_k$;
  $\mathbf{V}_k \leftarrow \mathbf{V}_m(1:n, 1:k)$; $\mathbf{H}_k \leftarrow \mathbf{H}_m(1:k, 1:k)$;
  Empieza con factorización de Arnoldi con k-pasos $\mathbf{AV}_k = \mathbf{V}_k\mathbf{H}_k + \mathbf{f}_k\mathbf{e}_k^T$, y aplica p pasos de Arnoldi para obtener una factorización de Arnoldi con m-pasos $\mathbf{AV}_m = \mathbf{V}_m\mathbf{H}_m + \mathbf{f}_m\mathbf{e}_m^T$.
end for

Una vez aplicado los shift QR implícito, el primer paso que se obtiene es

$$\mathbf{A}\mathbf{V}_m^+ = \mathbf{V}_m^+\mathbf{H}_m^+ + \mathbf{f}_m\mathbf{e}_m^T\mathbf{Q}, \tag{7.6}$$

donde $\mathbf{V}_m^+ = \mathbf{V}_m\mathbf{Q}, \mathbf{H}_m^+ = \mathbf{Q}^T\mathbf{H}_m\mathbf{Q}$ y $\mathbf{Q} = \mathbf{Q}_1\mathbf{Q}_2\ldots\mathbf{Q}_p$.

Cada $\mathbf{Q}j$ es una matriz ortogonal asociada a μ_j y su estructura es de tipo Hessenberg, así que los primeros $k-1$ elementos de la última fila de $\mathbf{Q}$ son igual a cero. Eso implica, también, las k columnas de la Ecuación 7.6 quedan en una relación de Arnoldi

$$\mathbf{A}\mathbf{V}_k^+ = \mathbf{V}_k^+\mathbf{H}_k^+ + \mathbf{f}_k^+\mathbf{e}_k^T, \tag{7.7}$$

y actualizado el residuo $\mathbf{f}_k^+ = \mathbf{V}_m^+\mathbf{e}_{k+1}\hat{\beta}_k + \mathbf{f}_m\sigma$. El último paso del Algoritmo 3 consiste en aplicar p-pasos adicional para obtener la forma original con m-pasos.

Aplicar los shift permite obtener un vector inicial como combinación lineal de los vectores propios queridos $\mathbf{v}_1 = \sum_{j=1}^{k}\mathbf{x}_j\alpha_j$ donde $\mathbf{A}\mathbf{x}_j = \mathbf{x}_j\lambda_j$, así que $\mathbf{f}_k = 0$, $\mathbf{A}\mathbf{V}_k = \mathbf{V}_k\mathbf{H}_k$ y $\mathbf{V}_k$ es una base ortonormal y el espectro de $\mathbf{H}_k$ representará los valores propios deseados: $\sigma(\mathbf{H}_k) = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$.

Los valores propios que se obtienen son los k valores propios dominantes.

7.5.1. Criterio de Parada

Uno de los pasos más importante en el método IRAM y, en general, en métodos iterativos es el criterio de parada. En esa sección se considera, entonces, el problema de determinar cuando un factorización de Arnoldi de longitud m ha calculado una buena aproximación de los valores propios.

Dado $\mathbf{H}_m\mathbf{s} = \mathbf{s}\theta$ donde $\|s\| = 1$ y $\hat{x} = \mathbf{V}_m\mathbf{s}$. Entonces,

$$\|\mathbf{A}\hat{x} - \hat{x}\theta\| = \|\mathbf{A}\mathbf{V}_m\mathbf{s} - \mathbf{V}_m\mathbf{H}_m\mathbf{s}\| = \|\mathbf{f}_m\||\mathbf{e}_m^T\mathbf{s}|. \tag{7.8}$$

El par $(\hat{x}, \theta)$ se denomina par de Ritz y es una buena aproximación al par de valor y vector propios de A si la última componente de un vector propio de $\mathbf{H}_m$ es pequeño. Si la cantidad $\|\mathbf{f}_m\|$ es pequeña todos los m valores propios de $\mathbf{H}_m$ son una buena aproximación a los valores propios de $\mathbf{A}$. La ventaja principal en utilizar este tipo de estimación para el criterio de parada, denominado estimación de Ritz, es que tal estimación evita obtener informaciones explícitas del cálculo del residuo $\mathbf{A}\hat{\mathbf{x}} - \hat{\mathbf{x}}\theta$.

En general, los valores propios de una matriz no simétrica puede ser muy sensible a perturbaciones como las que introducen los errores de redondeo. Tal sensibilidad está sujeta a la normalidad de la matriz. Si la matriz $\mathbf{A}$ es cercana a una matriz mal-condicionada, entonces se puede decir algo sobre la precisión de los valores y vectores propios calculados. El criterio adoptado es el mismo utilizado en [1] que asegura un pequeño error. Esta estrategia afirma que

el par de Ritz $(\hat{\mathbf{x}}, \theta)$ se considera convergido cuando

$$\|\mathbf{f}_m\||\mathbf{e}_m^T\mathbf{s}| \leq max(\epsilon_M\|\mathbf{H}_m\|, tol|\theta|) \tag{7.9}$$

esta satisfecha donde ϵ_M es la precisión de la maquina. Esta condición se controla por cada valor propio deseado.

7.5.2. Implementación

En esa sección se mostrará la implementación del Método de Arnoldi con Reinicio Implícito descrito en la Sección 7.5. El método IRAM se basa sobre las principales rutinas BLAS y LAPACK que contribuyen mucho en la precisión, robustez, y prestación computacional en el cálculo con matrices densas. Además, otra importante operación resulta el producto matriz-vector SpMV. Para un valor fijo de valores propios deseados k y un valor fijo de la base de Arnoldi m, el coste computacional es lineal con n, el orden de la matriz. El índice de ejecución en FLOPS para el IRAM es asintótico sobre el índice de ejecución de SpMV.

Librerías utilizadas

Para desarrollar el algoritmo IRAM en CUDA se han utilizado diferentes librerías cada una maneja partes distintas del algoritmo. La librería más importante es CUSP que representa una librería para álgebra lineal dispersa en CUDA. CUSP proporciona una interfaz flexible y de alto nivel para manejar matrices dispersas y solucionar sistemas lineales. CUSP esta basado en otra librería, denominada THRUST que proporciona una interfaz de alto nivel para mejorar las prestaciones y la portabilidad entre GPU. THRUST actúa interoperatividad entre diferentes tecnologías (como CUDA y OpenMP). Eso nos permite escribir código para la implementación del algoritmo que soluciona el problema de álgebra lineal una sola vez y que será ejecutado tanto en GPU como en CPU.

La Figura 7.1 muestra las dependencias entre las librerías utilizadas. El algoritmo IRAM depende de CUSP para el calculo SpMV contenido en el método de Arnoldi (ver el Algoritmo 2) además, IRAM depende de THRUST para optimizar operaciones sencillas echas con matrices densas como por ejemplo actualizaciones de elementos en la diagonal principal, suma de elementos entre matrices, etc.

Para el calculo con matrices densa se han utilizado, como veremos en la siguiente sección, una librería que implementa las rutinas del BLAS y LAPACK, y también CUSP que implementa rutinas básicas del BLAS1 como el calculo de norma de un vector, producto entre vectores y axpy que sirven para el método IRAM (ver el Algoritmo 3).

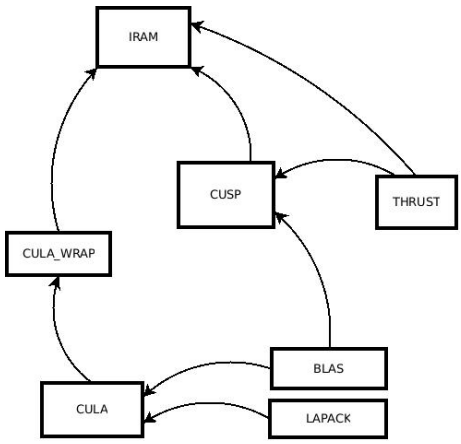

Figura 7.1: Dependencias entre las librerías utilizadas por IRAM

Cálculo con matrices densas

Como hemos dicho el algoritmo IRAM se basa sobre diferentes operaciones con matrices densas que necesitan de rutinas BLAS y LAPACK. Por ejemplo, en el Algoritmo 1 que se refiere al operación de Deplazamiento Implícito, hay diferentes operaciones con matriz densa como el producto matriz-matriz y factorización QR que pueden ser hecha utilizando las rutinas BLAS y LAPACK convencionales. La librería que implementa esa rutinas en CUDA es denominada CULA [2]. CULA es capaz de utilizar tanto recursos CPU como recursos en GPU.

CULA sigue la convención de nombramiento que es parecida al sistema adoptado en LAPACK. Las rutinas CULA están nombradas utilizando un sistema donde:

- La función empieza con *cula*. Para las rutinas en el Device se utiliza el prefijo *culaDevice*;
- El siguiente carácter en mayúscula representa el tipo de dato;
- Los siguientes dos caracteres representan el tipo de matriz;
- Los últimos dos caracteres representan el cálculo hecho da la rutina.

Una primera dificultad es la de integrar tal librería con CUSP. En particular, la ejecución de una rutina en CULA depende de dos factores 1) El tipo de dato; 2) el espacio de memoria. Por ejemplo, en la multiplicación matriz-vector hay cuatro posibles rutinas CULA y sólo una de esa puede ser llamada en función de los parámetros de ingreso. Si los datos en ejecución son una matriz y un vector en doble precisión en el espacio de memoria del Device es necesario llamar la rutina *culaDeviceDgemv*. Entonces, se ha desarrollado un wrapper nombrado *cula_wrap* donde cada método elige una de las rutinas CULA disponibles en función de los datos en ingreso integrando perfectamente el código de CULA con lo de la librería CUSP. Desde los algoritmos

vistos en las secciones anteriores (Algoritmo 1, 3), se puede ver que las rutinas necesarias en el wrapper son:

- *Xgemv(Array2d A, Array1d x, Array1d y, bool transA)*; Calcula el producto matriz-vector $\mathbf{y} = \mathbf{Ax}$ o también $\mathbf{A}^T\mathbf{x}$ si $transA = true$.
- *Xgemm(Array2d A, Array2d B, Array2d C, bool transA, bool transB)*; Calcula el producto matriz-matriz $\mathbf{C} = \mathbf{A}^{OP}\mathbf{B}^{OP}$ donde OP depende da los valores de $transA$ y $transB$.
- *Xgeqrf(Array2d A, Array2d Q, Array2d R)*; Calcula la factorización QR de la matriz $\mathbf{A}$.
- *Xgeev(Array2d A, Array1d eigvals, Array2d eigvects)*; Calcula los valores y vectores propios de una matriz densa.

Algoritmo IRAM

El algoritmo 4 presenta una implementación del método IRAM 4. Definimos $\mathbf{H}_j$ como una matriz de Hessenberg superior de orden $j \times j$, $\mathbf{V}_j^T\mathbf{B}\mathbf{V}_j = \mathbf{I}_j$, y el vector residuo $\mathbf{f}_j$ es ortogonal a las columnas de $\mathbf{V}_j$. El entero k es el número de valores propios deseados y m se refiere a la dimensión de la factorización de Arnoldi.

Algorithm 4 Implementación de IRAM

- Genera un vector aleatorio $\mathbf{V}_m\mathbf{e}_1 = \mathbf{v}_1$;
- Calcula la factorización de Arnoldi inicial de longitud k: $\mathbf{A}\mathbf{V}_k = \mathbf{V}_k\mathbf{H}_k + \mathbf{f}_k\mathbf{e}_k^T$;

for $iter = 1, \ldots, maxiter$ **do**

- Añade la factorización de Arnoldi de longitud m;
- Calcula los valores propios de $\mathbf{H}_m$;
- Los valores propios se dividen en dos conjuntos Ω_w y Ω_u. Los k valores propios de Ω_w son los deseados valores propios; mientras que los otros valores propios en Ω_u se utilizan para la operación de Shift;
- Determina el número de valores de Ritz que satisfacen la convergencia;
- Termina si todos los k valores propios convergen o si $iter > maxiter$;
- Incrementa k según una estrategia. Determina $p = m - k$. Si $p = 0$ termina.
- Aplica los p pasos de QR Desplazamiento Implícito sobre los valores propios de Ω_u, para obtener la submatriz principal $\mathbf{H}_k$ de orden k y la matriz ortogonal $\mathbf{V}_k$ de orden $n \times k$ para construir una nueva factorización de Arnoldi de longitud k:$\mathbf{A}\mathbf{V}_k = \mathbf{V}_k\mathbf{H}_k + \mathbf{f}_k\mathbf{e}_k^T$;

end for

- Devuelve los valores propios θ de $\mathbf{A}$ a partir de los k valores de $\mathbf{H}_k$, y los vectores propios $\hat{\mathbf{x}}$ de $\mathbf{A}$ a partir los vectores propios $\mathbf{s}$ de $\mathbf{H}_m$ donde $\hat{\mathbf{x}} = \mathbf{V}_m\mathbf{s}$;

La implementación genera un vector aleatorio como primero vector columna de la matriz $\mathbf{V}_m$. Antes de aplicar el algoritmo de QR Desplazamiento Implícito es posible elegir una estrategia

para incrementar k. La estrategia adoptada es la siguiente: El número p de shift a aplicar se reduce de uno por cada valor de Ritz deseado que satisfacen el criterio de convergencia. Este esquema ayuda a prevenir situaciones de ausencia de progreso en la convergencia de los valores restante. Resulta que, si k se queda fijo se hace siempre más difícil converger todos los valores de Ritz cada vez que nos acercamos al número de valores deseados. Un control añadido sirve para que k no exceda $(m-k)/2$.

La iteración se termina cuando el número de valores de Ritz deseados satisface el criterio de convergencia, visto en Sección 7.5.1, o cuando el número de iteracciones llega al limite máximo. En los dos casos los valores de Ritz obtenidos desde la última factorización de Arnoldi representan los valores propios de la matriz $\mathbf{A}$.

En la versión original de ARPACK [1] el algoritmo IRAM llama una rutina ($[s,d]lahqrb$) que representa una versión modificada de la rutina en LAPACK ($[s,d]lahqr$). Esa rutina sirve para lograr la forma de Schur a partir de la matriz de Hessenberg superior. La principal diferencia con la rutina en LAPACK es que la versión en ARPACK calcula sólo las últimas componentes de los vectores asociados de Schur porque son sólo esos lo que sirven para la estimación del error. Tal rutina sirve para el calculo de valores propios de $\mathbf{H}_m$ como se puede ver en el Algoritmo 4. Dicha rutina permite calcular la descomposición de Schur de la matriz proyectada $\mathbf{H}_m$. Después obtenida la matriz triangular superior de Schur, la rutina puede calcular los valores propios que representaran los valores de Ritz.

En nuestro caso la primera versión se basará sobre una rutina que calcula los valores propios de $\mathbf{H}_m$ directamente con la rutina LAPACK *geev*. Esta primera sencilla implementación nos lleva al problema de no poder calcular correctamente los valores propios en presencia de cluster, es decir cuando hay valores propios cercanos entre ellos. En particular, resulta que no construyendo adecuadamente la matriz en forma de Schur, los vectores asociados a los k valores propios dominantes no estarán ordenados y podrían no estar directamente en la submatriz principal de la forma de Schur. Eso implica que aunque unos valores propios han convergido en una iteración del algoritmo IRAM puede ser posible que en la siguiente iteración los vectores asociados no pertenecerán a las k primeras columna de la forma de Schur.

Las matrices analizadas L_{ii} y T_{ii} no resultan adecuadas para la implementación realizada pues tienen muchos clusters y los resultados que logramos resultan diferentes. Aunque no dispongamos de un método estable en el cálculo de los valores propios, el coste de las operaciones por iteración será el mismo, así que eso nos permite igualmente de evaluar las prestaciones comparando el algoritmo IRAM sobre CPU, GPU y con los diferentes formato de almacenamiento, como veremos en el capítulo de resultados.

Capítulo 8

Resultados

Las pruebas se han realizado utilizando máquinas con distintas características para analizar el comportamiento de las implementaciones sobre diferentes arquitecturas. Las características de las máquinas en las que se han desarrollado las pruebas son las siguientes. Se dispone de tres máquinas en total y en orden de prestaciones se denominan Eleanorrigby, Golub y MyArch. Las Tablas 8.1, 8.2 y 8.3 muestran sus características. Además, MyArch resulta ser la de menores prestaciones y no tiene habilitada la función para el cálculo en doble precisión.

En este capítulo presentamos los resultados obtenidos para tres de las operaciones mas importantes del Álgebra Lineal. En la sección 8.1 se presentan las prestaciones para la operación SpMV. En la siguiente sección se presentan los resultados para la resolución de sistema de ecuaciones lineales, y por último en la sección 8.3 se presentan los resultados para el cálculo del problema de valores propios mediante el método IRAM.

Modelo	NVIDIA "TESLA C2070"
Cantidad total de memoria global	5636554752 bytes (5GB)
Multiprocessadores x (Cores/MP) = Cores	14 (MP) x 32 (Cores/MP) = 448(Cores)
Cantidad total de memoria compartida por bloque	49152 bytes
Número total de registros disponibles por bloque	32768
Cantidad total de memoria constante	65536 bytes
Dimensión del Warp	32
Máximo número de hilos por bloque	1024
Frecuencia de Clock	1.15 GHz

Cuadro 8.1: Características de la máquina Eleanorrigby

Modelo	NVIDIA "Quadro FX 5800"
Cantidad total de memoria global	4294246400 bytes (4GB)
Multiprocessadores x (Cores/MP) = Cores	30 (MP) x 8 (Cores/MP) = 240(Cores)
Cantidad total de memoria compartida por bloque	16384 bytes
Número total de registros disponibles por bloque	16384
Cantidad total de memoria constante	65536 bytes
Dimensión del Warp	32
Máximo número de hilos por bloque	512
Frecuencia de Clock	1.30 GHz
Ancho de Banda en Memoria	102 GB/sec

Cuadro 8.2: Características de la máquina Golub

Modelo	NVIDIA "GeForce 8400M GS"
Cantidad total de memoria global	268107776 bytes (256MB)
Multiprocessadores x (Cores/MP) = Cores	2 (MP) x 8 (Cores/MP) = 16(Cores)
Cantidad total de memoria compartida por bloque	16384 bytes
Número total de registros disponibles por bloque	8192
Cantidad total de memoria constante	65536 bytes
Dimensión del Warp	32
Máximo número de hilos por bloque	512
Frecuencia de Clock	0.80 GHz

Cuadro 8.3: Características de la máquina MyArch

8.1. Multiplicación Matriz-vector

8.1.1. Análisis sobre la matriz L_{ii}

Vamos a analizar los resultados obtenidos con MyArch midiendo el tiempo de ejecución de los diferentes *kernels*. Las pruebas se han realizado sobre matrices de Poisson por el hecho de la similaridad con la matriz L_{ii} obtenida por el proceso de mallado del reactor nuclear. Con la matriz de Poisson es possible cambiar el tamaño de la matriz manteniendo siempre una característica parecida a la que tenemos en L_{ii}.

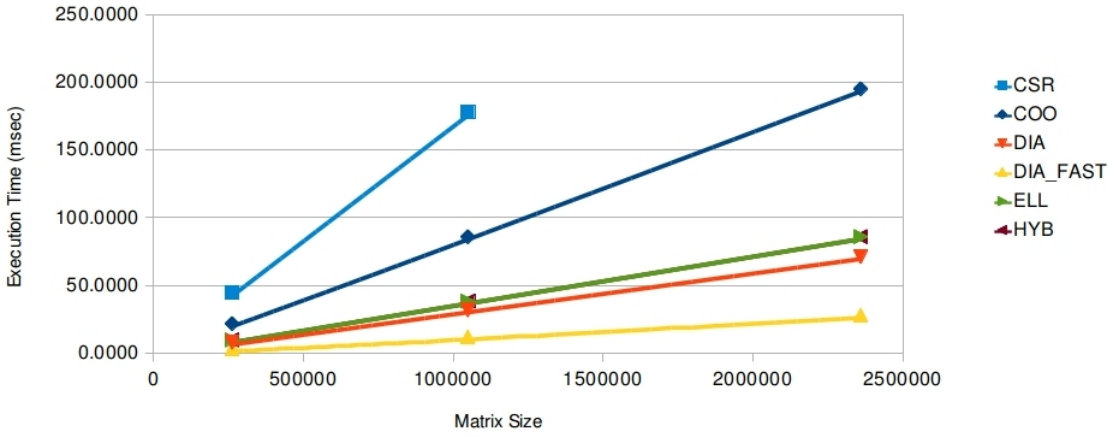

Figura 8.1: Tiempo de Ejecución para SpMV sobre una matriz de Poisson 5pt en simple precisión sobre myArch

Las optimizaciones aplicadas en el algoritmo *DIA* nos llevan a unos resultados importantes. En particular, podemos ver que también con matriz de tamaño muy grande el tiempo de ejecución de *DIA_FAST* se queda muy lejos de los otros en cuanto aprovecha mucho los recursos de la máquina. La peor solución se obtiene cuando se utilizan los formatos *CSR* y *COO*. Además, con este tipo de matriz el *CSR* no optimiza el almacenamiento de los elementos llegando a no poder contener matrices de tamaño muy grande. El comportamiento de los formatos *ELL* y *HYB* resulta ser prácticamente el mismo por el hecho que el híbrido almacena los datos exactamente como el *ELL* para el tipo de matriz diagonal. Calculando los GFLOP/sec podemos observar (Figura 8.2) como el rendimiento del algoritmo *DIA_FAST* resulta extraordinario respecto a las otras variantes.

Desafortunadamente, las prestaciones en la máquina Golub no resultan iguales. Eso es debido al hecho que todo lo que se ganaba a través de la técnica de la cache no sirve en un sistema que tiene velocidad de transferencia datos que puede llegar a más de 100 GB/sec. En la gráfica de la Figura 8.3 se nota como las prestaciones del DIA_fast se han reducido considerablemente con

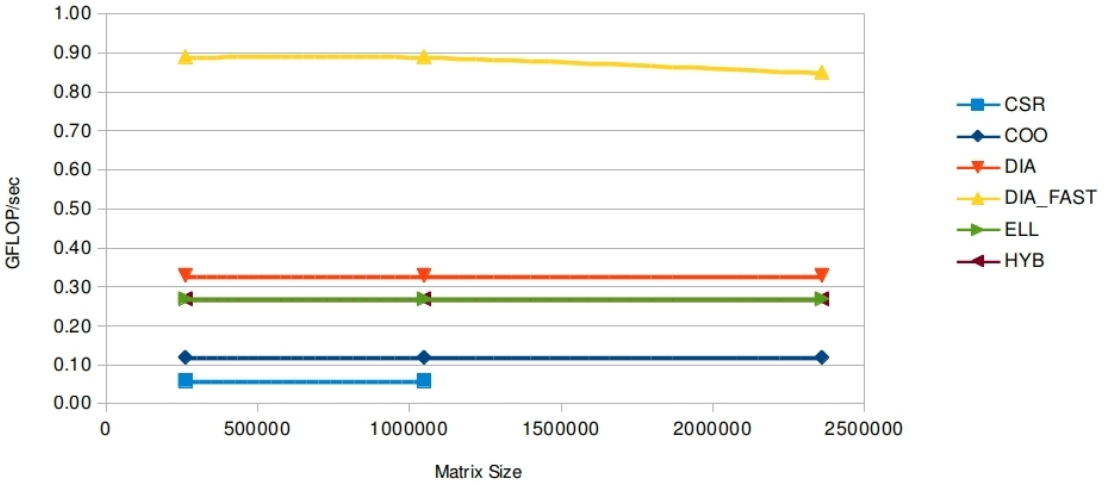

Figura 8.2: GFlops para SpMV sobre una matriz de Poisson 5pt en simple precisión sobre myArch

respecto a los demás formatos, incluso el DIA lo supera.

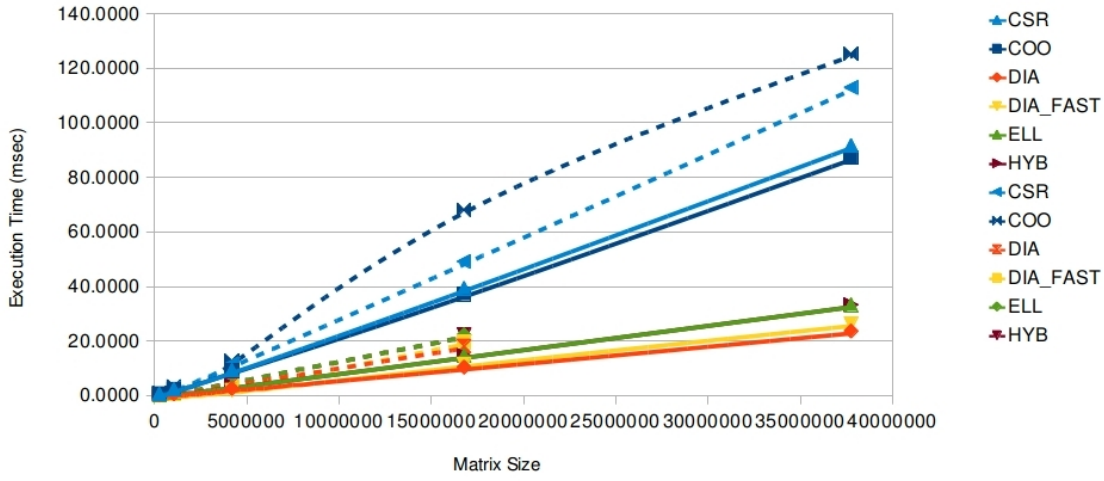

Figura 8.3: Tiempo de Ejecución para SpMV sobre una matriz de Poisson 5pt en simple y doble precisión sobre Golub

Además, si tenemos en cuenta los GFLOP/sec se nota como aumentando el tamaño de la matriz el rendimiento baja . La principal motivación de este resultado es por el *overhead* que obtenemos al añadir tal optimización (es decir ulteriores condiciones e instrucciones) con con-

siguiente "branch misprediction". En precisión doble se puede observar que los formatos DIA y ELL no son capaces de almacenar adecuadamente las matrices cuando se aumenta de mucho el tamaño. Los otros formatos resultan mas genéricos y permiten almacenar matrices de gran tamaño aunque los tiempos de ejecución quedan siempre muy altos.

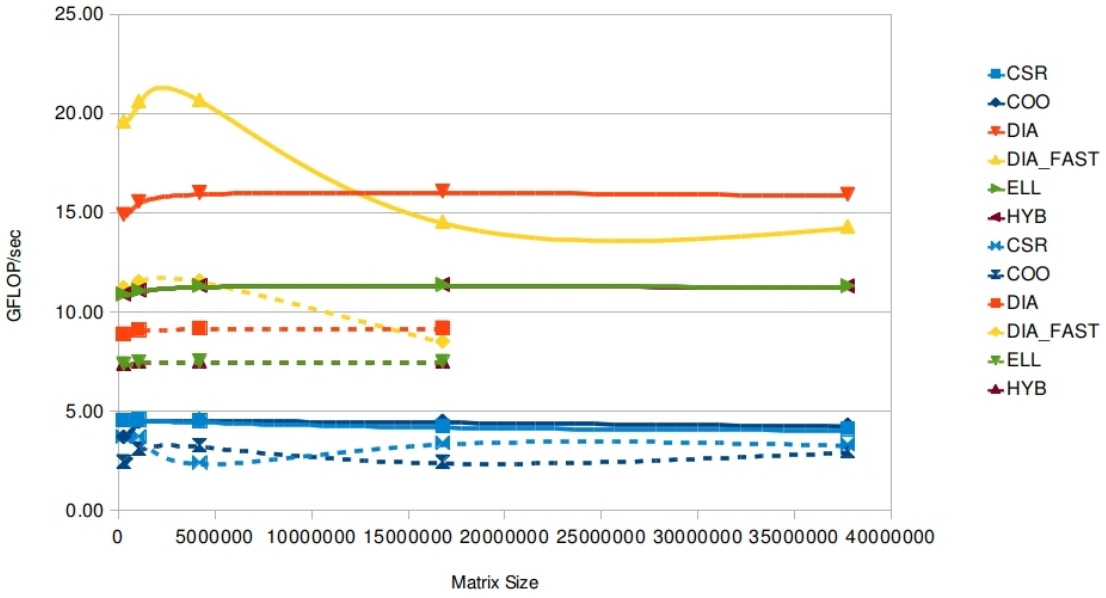

Figura 8.4: GFlops para SpMV sobre una matriz de Poisson 5pt en simple y doble precisión sobre Golub

8.1.2. Análisis sobre la matriz T_{ii}

A continuación, veremos los resultados obtenidos considerando la matriz T_{ii} obtenida a través del proceso mallado en 3D de un reactor nuclear con celdas hexagonales. Como hemos visto anteriormente, a diferencia de L_{ii} es posible observar que su estructura es más compleja en cuanto sus diagonales están muy desplazadas almacenando así una gran cantidad de ceros. También la malla es muy fina, así, que el tamaño de la matriz resulta ser mucho más grande que las otras matrices. Como se ha visto en el Capítulo 4, T_{ii} tiene 5693803 elementos con un tamaño de 130075x130075. Haciendo un zoom se nota como la matriz tiene lineas curvas y eso lleva a una imposibilidad de almacenar eficientemente una matriz de este estilo a través del formato *DIA*. Las figuras 8.5 y 8.6 muestran el tiempo de ejecución sobre la matriz T_{ii} y la matriz T_{ii_p} que presenta la misma estructura de T_{ii} pero con tamaño mas pequeño - 1485575 elementos no nulos y tamaño de 34799x34799.

Las dos versiones de *JAD* resultan bastante eficientes respecto a las otras soluciones. Aunque

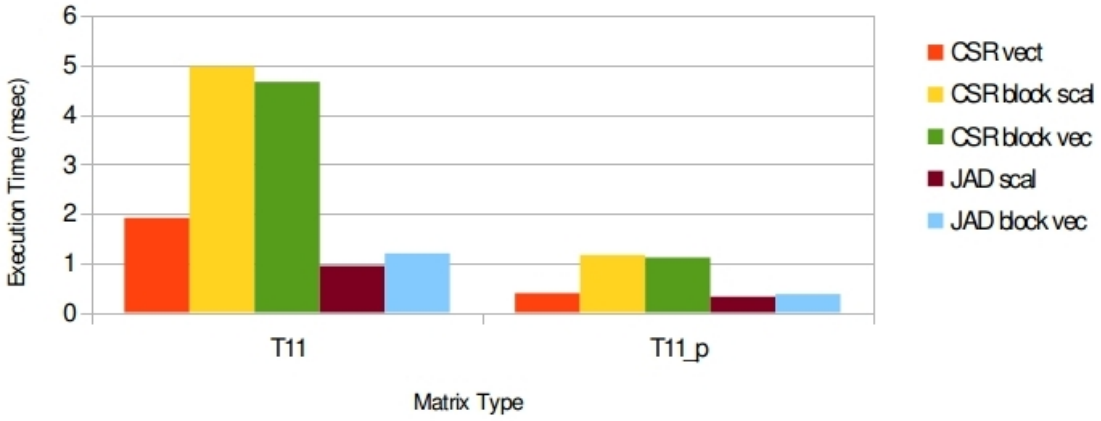

Figura 8.5: Tiempo de Ejecución para SpMV sobre la matriz T_{ii} en precisión simple sobre Eleanorrigby

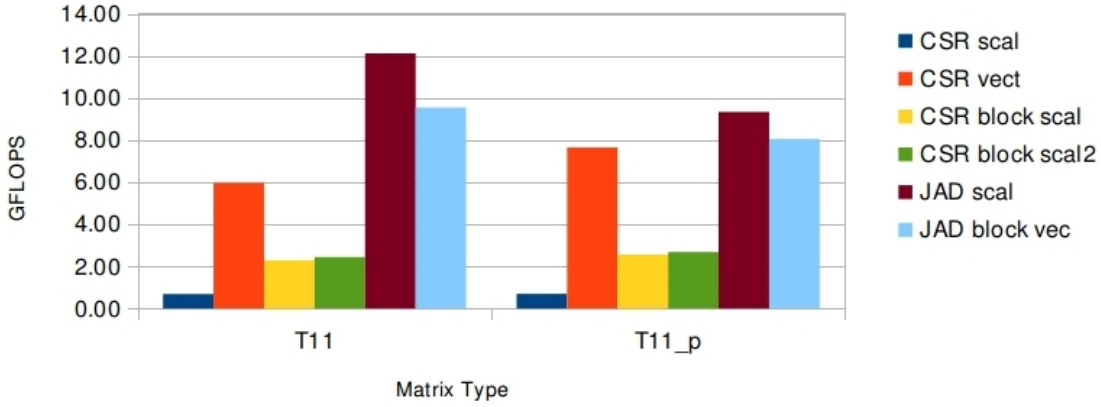

Figura 8.6: GFlops para SpMV sobre la matriz T_{ii} en precisión simple sobre Eleanorrigby

el *JAD Block* resulta más sofisticado porque maneja distribución de carga, coaleciencia etc, los resultados no superan los del *JAD* clásico que llega a obtener 12GFLOPS doblando el resultado del *CSR* de la librería CUSP. El *CSR Block* llega a 2GFLOP/sec superando bastante *CSR scalar* del CUSP. Esto sucede por la disposición de los elementos en *CSR Block* y el mayor uso de memoria compartida que permite ahorrar accesos en memoria global mientras que con *CSR* se hace sin coalesciencia.

Los gráficos de Figura 8.7 y 8.8 se refieren al cálculo SpMV en doble precisión. Comparando todos los algoritmos se puede observar como el comportamiento resulta parecido al de simple precisión aunque para todos los algoritmos bajamos bastante en GFLOP/sec.

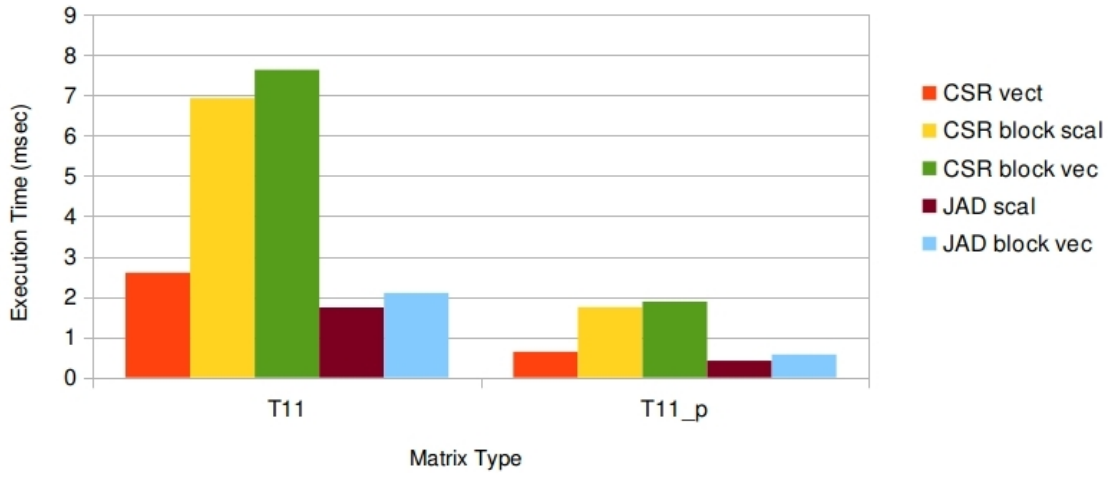

Figura 8.7: Tiempo de Ejecución para SpMV sobre la matriz T_{ii} en double precisión sobre Eleanorrigby

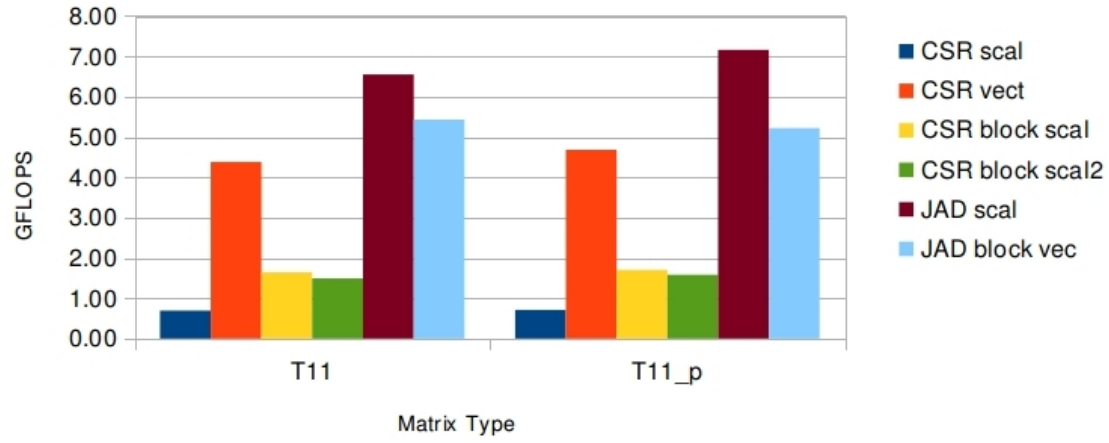

Figura 8.8: GFlops para SpMV sobre la matriz T_{ii} en double precisión sobre Eleanorrigby

8.1.3. Análisis sobre diferentes matrices

También se han realizado pruebas teniendo en cuenta diferentes tipos de matrices. Para la estructura de estas matrices algunos formatos de almacenamiento resultan, como veremos, inadecuados. Las matrices utilizadas para las pruebas tienen diferente estructura y sirven para analizar el comportamientos de los algoritmos en muchos casos. En las Figuras 8.9, 8.10, 8.11, 8.12 y 8.13 se muestran las diferentes matrices analizadas. Estas matrices se han obtenido desde el sitio web de NVIDIA donde es posible descargar diferentes matrices en formato Matrix Market

(.mtx).

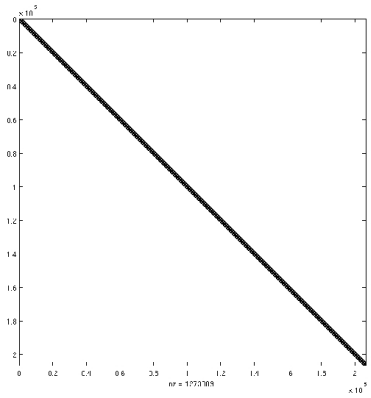

Figura 8.9: mac_econ_fwd500.mtx, tiene 1273389 elementos, tamaño de 206500x206500.

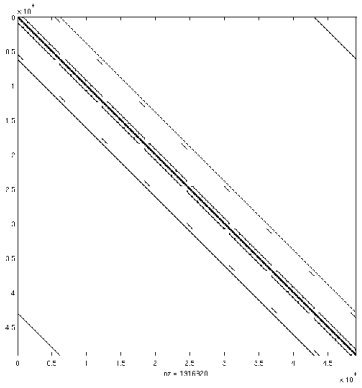

Figura 8.10: qcd5_4.mtx, tiene 1916928 elementos, tamaño de 49152x49152.

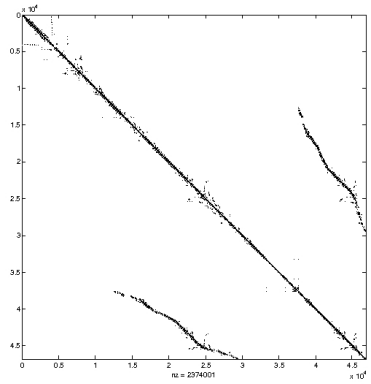

Figura 8.11: rma10.mtx, 2374001 elementos, tamaño 46835x46835.

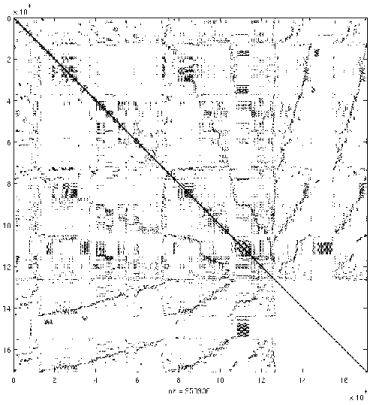

Figura 8.12: scircuit.mtx, 958936 elementos, tamaño 170998x170998.

Los resultados que se han obtenido se presentan en la Figura 8.14. En todos los casos excepto para la matriz L_{ii} el almacenamiento *DIA* no pudo utilizarse debido a sus estructuras. En general, el *CSR* se comporta mejor que los demás formatos aunque, si la matriz tiende a ser diagonal, como en el caso de la matriz QCD, *ELL* y *HYB* se comportan mejor.

A continuación, analizaremos los resultados obtenidos a través de la mejor máquina Eleanorrigby y añadiremos los nuevos formatos de almacenamiento *CSR Block*, *JAD* clásico y *JAD Block*. Se observa que, el formato *JAD Block* no puede ser aplicado a todas las matrices presentadas antes sino sólo a la matriz T_{ii} que esta organizada a bloques de tamaño 5203x5203 y, *T11_p* de tamaño 2047x2047.

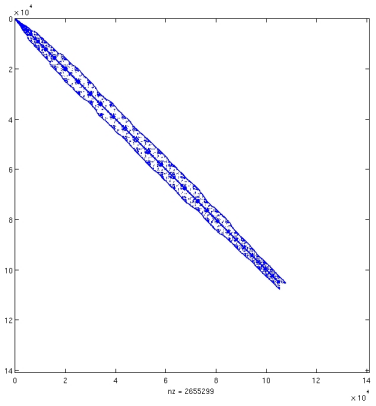

Figura 8.13: shipsec1.mtx es de 7813404 elementos con 140874x140874.

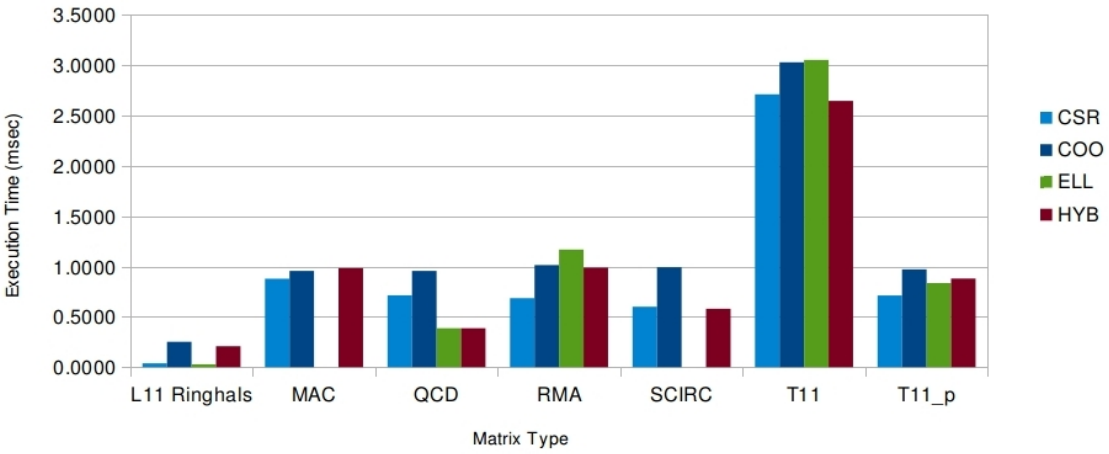

Figura 8.14: Tiempo de Ejecución para SpMV sobre diferentes matrices en simple precisión sobre Golub

Nótese que el *CSR scalar* toma un tiempo muy grande y por eso no figura en los tiempos de ejecución. Como vemos en los gráficos de Figura 8.15 y 8.16, en la mayoria de matrices el formato de almacenamiento más adecuado es el *CSR vector*.

8.2. Resolución de sistemas de ecuaciones lineales

Vamos ahora a presentar los resultados obtenidos en la resolución de sistemas de ecuaciones lineales con matrices dispersas.

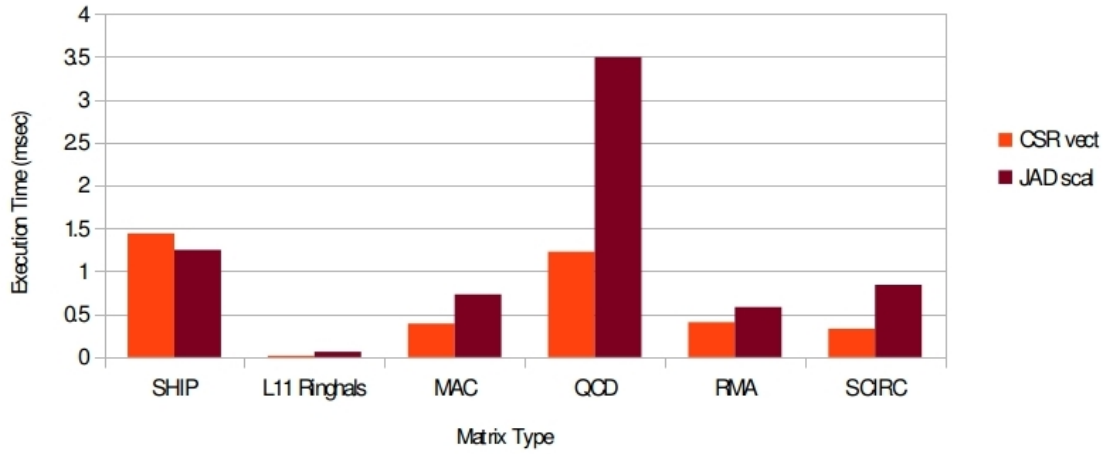

Figura 8.15: Tiempo de Ejecución para SpMV sobre diferentes matrices en simple precisión sobre Eleanorrigby

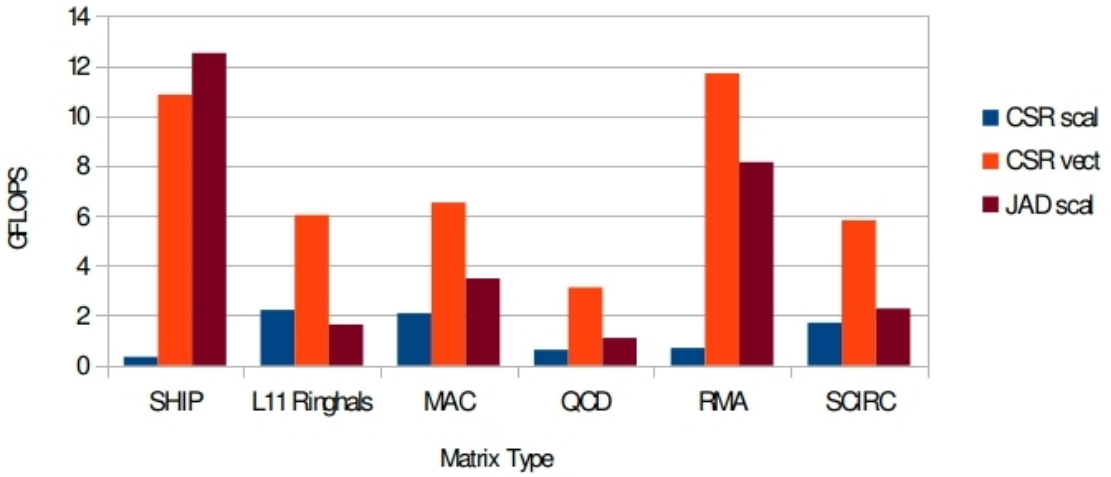

Figura 8.16: GFlops para SpMV sobre diferentes matrices en simple precisión sobre Eleanorrigby

8.2.1. Análisis sobre la matriz L_{ii}

Como en la sección 8.1.1 utilizaremos la matriz de Poisson 5pt para medir el tiempo de ejecución de los métodos con respecto al tamaño. También, como era de esperar, se observara una diferencia evidente entre los resultados obtenidos en Golub y MyArch.

Dada la estructura diagonal de la matriz lo que se puede ver en Figura 8.17 es que *DIA*, *DIA_FAST* y *ELL* se comportan de forma parecida mientras que, *COO* nos da el peor resultado

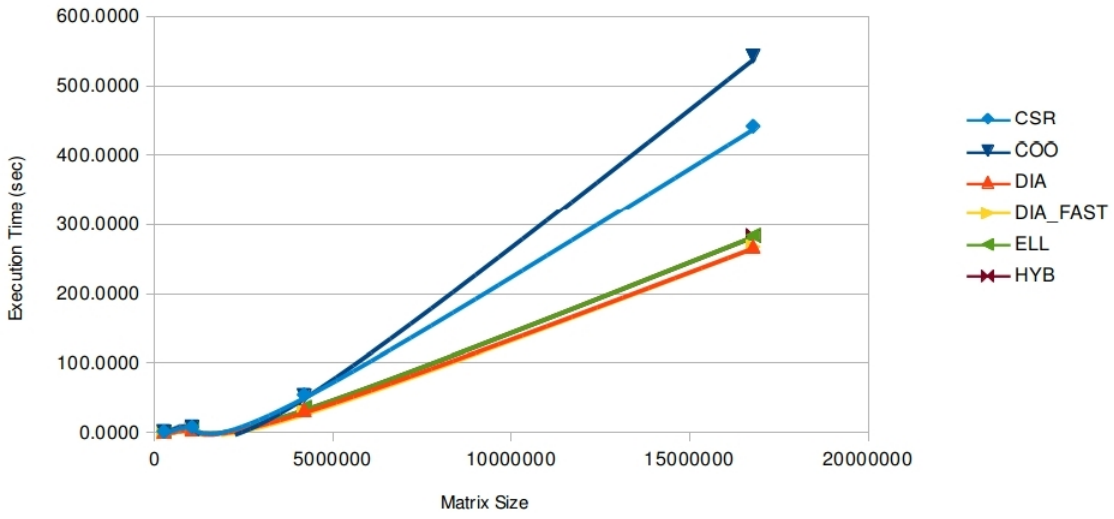

Figura 8.17: Tiempo de Ejecución para Resolución de Sistema de Ecuaciones Lineales utilizando el método CG sobre una matriz de Poisson 5pt en Golub

como en todos los otros casos.

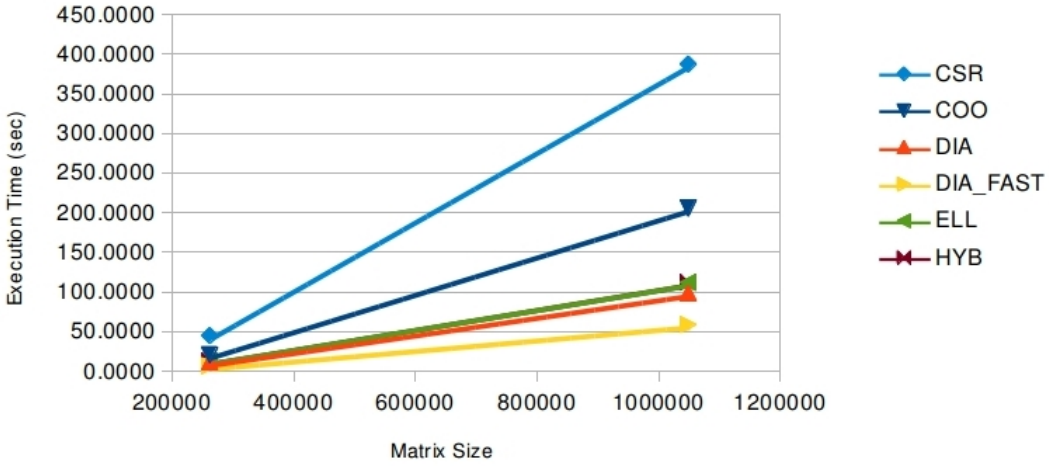

Figura 8.18: Tiempo de Ejecución para Resolución de Sistema de Ecuaciones Lineales utilizando el método CG sobre una matriz de Poisson 5pt en MyArch

En MyArch los resultados (Figura 8.18) cambian mucho en cuanto, como hemos visto anteriormente, la eficiencia de *DIA_FAST* resulta bastante evidente debido a la utilización de la memoria compartida. Vamos a ver ahora los tiempos considerando diferentes métodos iterativos

para la resolución del sistema con matrices dispersa.

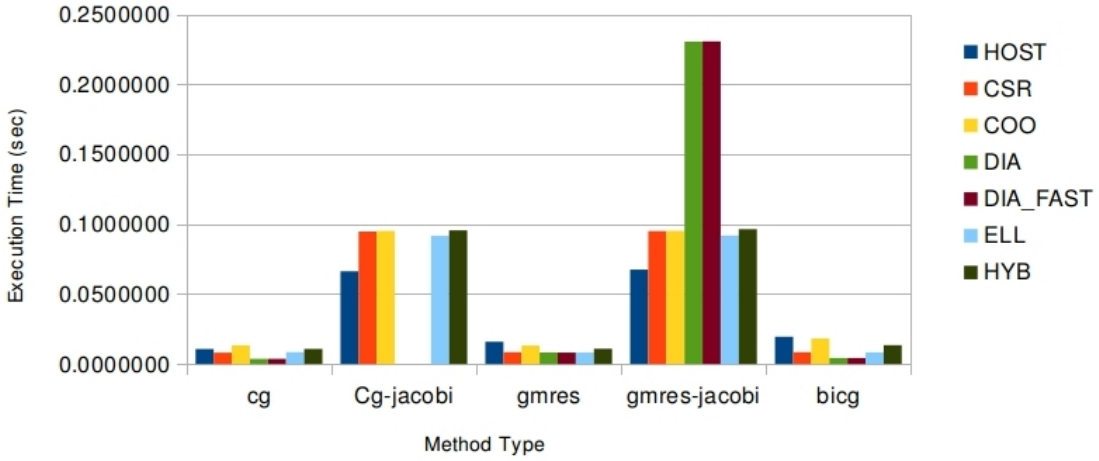

Figura 8.19: Tiempo de Ejecución para Resolución de Sistema de Ecuaciones Lineales sobre la matriz L_{ii} en Golub

Utilizando precondicionadores resulta que el número de iteracciones se reduce muchísimo, aunque el tiempo por cada iteración es mucho mayor, así que utilizando Jacobi el resultado empeora. El método más eficiente resulta ser CG en todos los formatos de almacenamiento también desde el punto de vista teórico resulta que siendo la matriz L_{ii} simétrica y definida positiva el CG es el método más adecuado.

8.2.2. Análisis sobre la matriz T_{ii}

A continuación, realizaremos un análisis de prestaciones para la matriz T_{ii}. En este caso también el método CG resulta ser el mejor método iterativo pero los tiempos son muy parecidos entre los diferentes formatos, aunque el *CSR* se mantiene siempre ligeramente a bajo de todos (Figura 8.20). Además, tenemos que observar que en los resultados no aparecen los tiempos obtenidos desde la CPU en cuanto esos tiempos resultan ser de diverso orden de magnitud mayor respecto lo que se obtiene con GPU. Vamos a analizar el comportamiento de los formatos de almacenamiento a bloque y de *JAD* en nuestra matriz principal T_{ii}. Como antes, queremos medir el tiempo de ejecución para la resolución de sistemas lineal con la matriz dispersa T_{ii}. Como esta claro, los resultados deberán ser muy parecidos a los que hemos obtenido en la multiplicación matriz-vector. Las Figuras en 8.21 y 8.22 muestran el tiempo de ejecución utilizando diferentes tipos de métodos con nuestros formatos de almacenamiento en Eleanorrigby en simple y doble precisión.

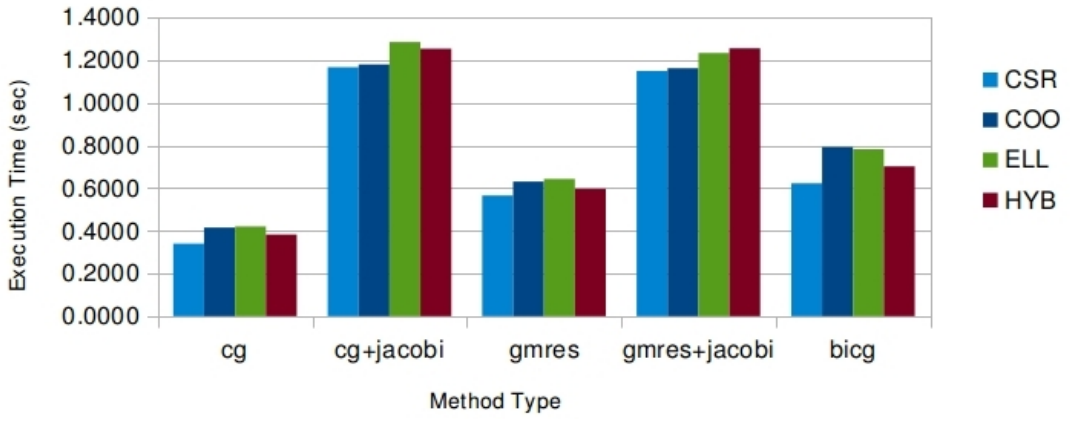

Figura 8.20: Tiempo de Ejecución para Resolución de Sistema de Ecuaciones Lineales sobre la matriz T_{ii} en Golub

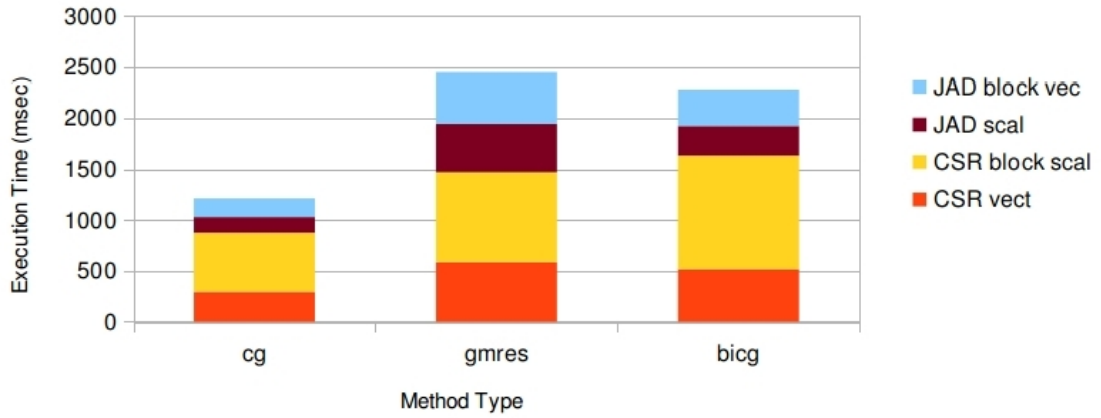

Figura 8.21: Tiempo de Ejecución para Resolución de Sistema de Ecuaciones Lineales sobre la matriz T_{ii} con simple precisión en Eleanorrigby

8.3. Cálculo de valores propios

En esta sección se muestran los resultados obtenidos a través de la Implementación del algoritmo IRAM presentado en el Capítulo 7. La versión actualmente realizada cumple el cálculo de valores y vectores propios sólo para matrices bien definidas con valores propios que no forman clusters, es decir, no hay valores propios muy cercanos entre ellos. La diferencia principal con el método clásico IRAM radica en que el cálculo de los valores de Ritz se obtienen calculando los valores propios de la matriz de Hessenberg Superior **H**.

Desafortunatamente, resulta que los valores propios de las matrices L_{ii} y T_{ii} forman clusters

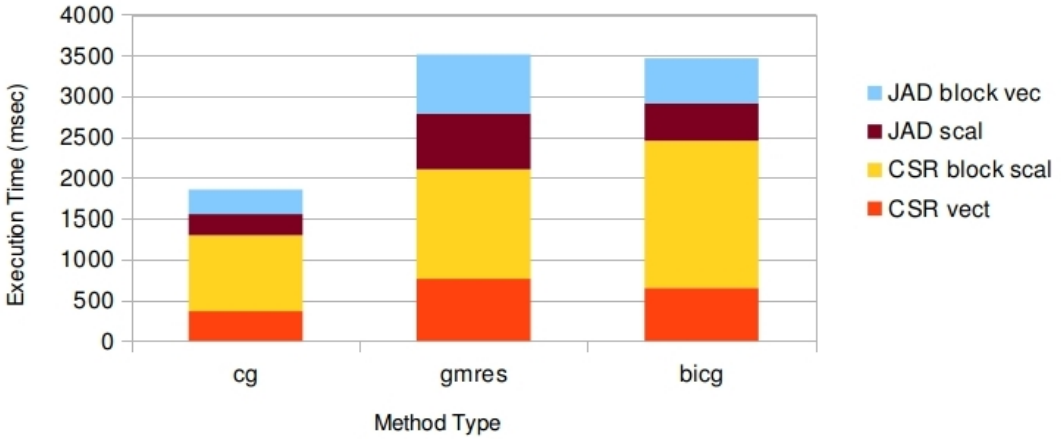

Figura 8.22: Tiempo de Ejecución para Resolución de Sistema de Ecuaciones Lineales sobre la matriz T_{ii} con double precisión en Eleanorrigby

así que la versión actual del método, en algunos casos no obtiene ordenados los valores propios dominantes, por lo que puede no devolver los k valores propios dominantes.

8.3.1. Análisis sobre la matriz L_{ii}

Aunque no dispongamos de un método estable (debido a los clusters y al reinicio) para el cálculo de los valores propios, el coste de las operaciones por iteración será parecido. Por tanto, para comprobar las prestaciones del algoritmo IRAM utilizando los adecuados formatos de almacenamiento sobre CPU y GPU, se han hecho pruebas para medir el tiempo de ejecución sobre las matrices analizadas. Las Figuras 8.23 y 8.24 muestran el tiempo medio de ejecución por iteración del algoritmo IRAM sobre la matriz L_{ii} con diferente tamaño del subespacio de Krylov en Eleanorrigby y MyArch. Como se puede observar el ahorro resulta bastante alto una vez incrementado el valor de la dimensión del subespacio a más de 128 (resulta alrededor de cuatro veces inferior al tiempo en GPU respecto al de CPU).

8.3.2. Análisis sobre la matriz T_{ii}

Se han hecho otras pruebas para comprobar las prestaciones en MyArch y Eleanorrigby sobre la matriz T_{ii} (Figuras 8.25, 8.26). Como en el caso anterior se notan diferencias en prestaciones en función de la arquitectura que se tiene en cuenta. En particular, los formatos de almacenamiento más específicos para las matrices ayudan a lograr mayor prestación en MyArch. Eso es debido al hecho que la maquina tiene muy pocos recursos y los resultados están muy sensibles a la manera de como estos recursos de la máquina se manejan por el algoritmo.

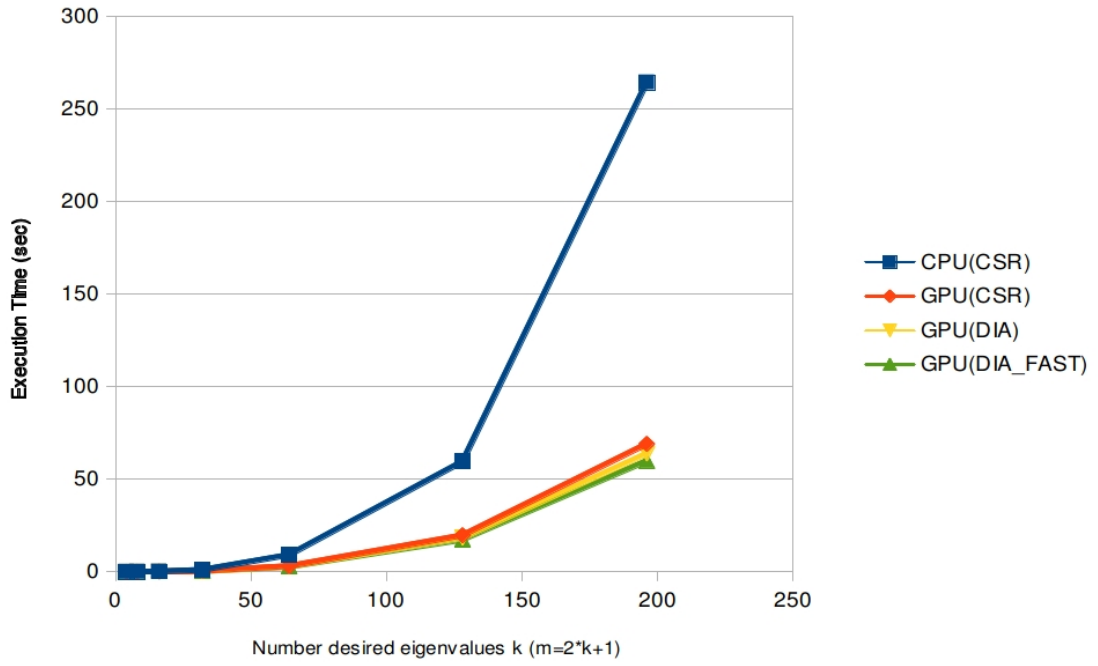

Figura 8.23: Tiempo medio de ejecución por iteración para IRAM sobre L_{ii} en Eleanorrigby sobre diferente tamaño del espacio de Krylov

8.3.3. Análisis de *profiling*

A través de los resultados obtenidos anteriormente puede ser hecha una importante consideración. Las prestaciones que obteníamos con los formatos de almacenamiento específico para cada matriz en SpMV o en el sistema de ecuaciones lineales, no aparecen en el algoritmo IRAM. En particular, haciendo prueba de *profiling* (Figura 8.27, 8.28) utilizando la herramienta NVIDIA Visual Profiler que pone a disposición CUDA Toolkit, se ha podido observar como el coste dominante no resulta ser la operación SpMV sino que otros cálculos entre matrices que se realizan en el algoritmo. En las figuras de *profiling* aparece el listado (lado izquierdo) de las rutinas con el relativo porcentaje de utilización y el número de invocaciones (entre paréntesis cuadra) ordenados por el porcentaje de utilización. En la parte de abajo se muestran las propiedades de las rutinas que toman mas tiempo. El porcentaje de utilización se refiere a la porción de tiempo que la GPU utiliza para la ejecución de un método respecto al tiempo total de actividad de la GPU. Como se puede observar la operación SpMV no resulta ser la operación dominante sino que la multiplicación entre las matrices densa resulta ser la que nos lleva a un mayor coste *gen_sgemm*. Esta operación está contenida en el algoritmo QR con Desplazamiento Implícito (Algoritmo 1) donde se aplican p (número de valores propios no deseados) iteraciones y por cada una de esta se aplican multiplicaciones entre matriz de gran tamaño (por ejemplo por la actualización de la

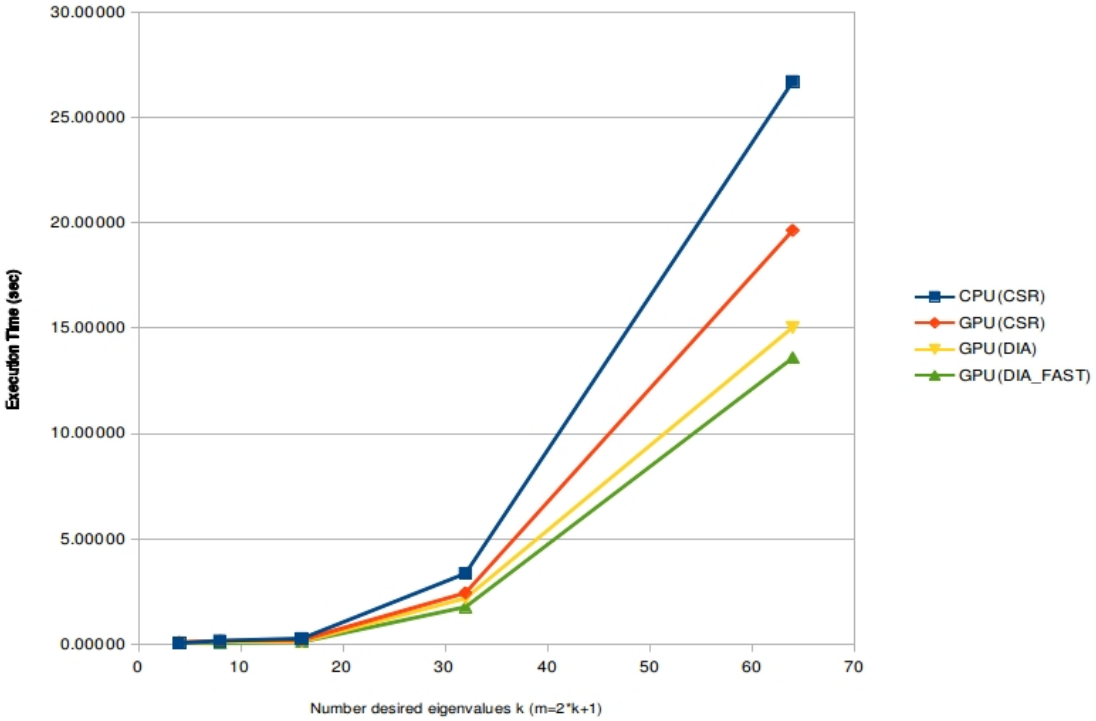

Figura 8.24: Tiempo medio de ejecución por iteración para IRAM sobre L_{ii} en MyArch sobre diferente tamaño del espacio de Krylov

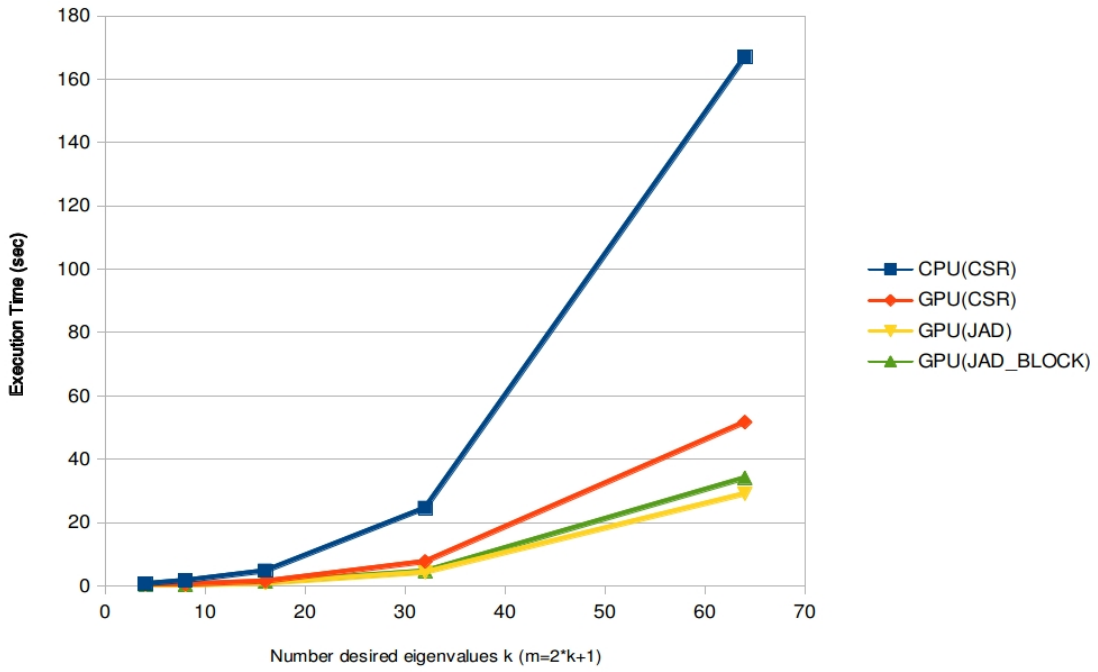

Figura 8.25: Tiempo medio de ejecución por iteración para IRAM sobre T_{ii} en MyArch sobre diferente tamaño del espacio de Krylov

matriz **V** que representa el subespacio de la base de Krylov). Hay también que observar como existen muchas operaciones secuenciales o operaciones sencillas que se ejecutan sobre la matriz de Hessenberg y que llevan a reducir enormemente el *speedup* del algoritmo.

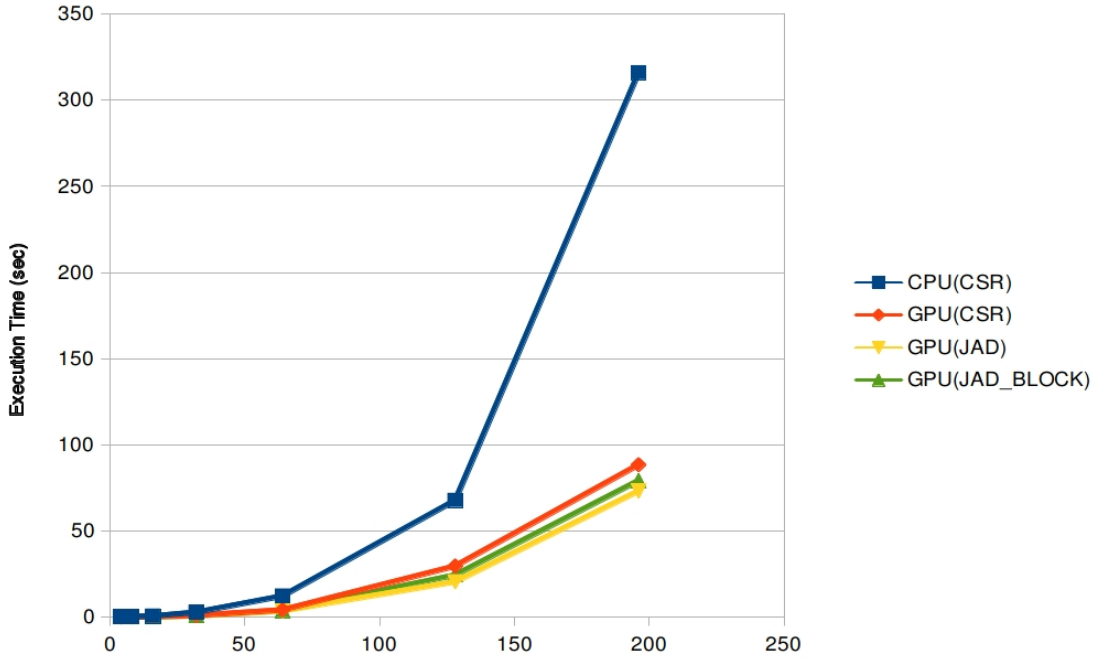

Figura 8.26: Tiempo medio de ejecución por iteración para IRAM sobre T_{ii} en Eleanorrigby sobre diferente tamaño del espacio de Krylov

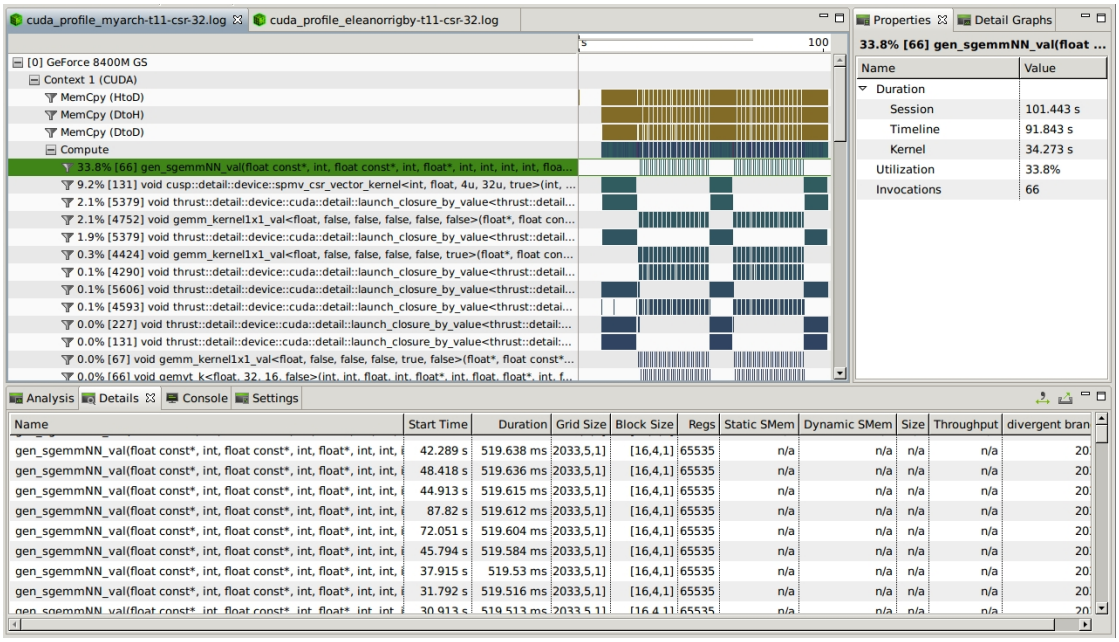

Figura 8.27: Profiling de IRAM en MyArch

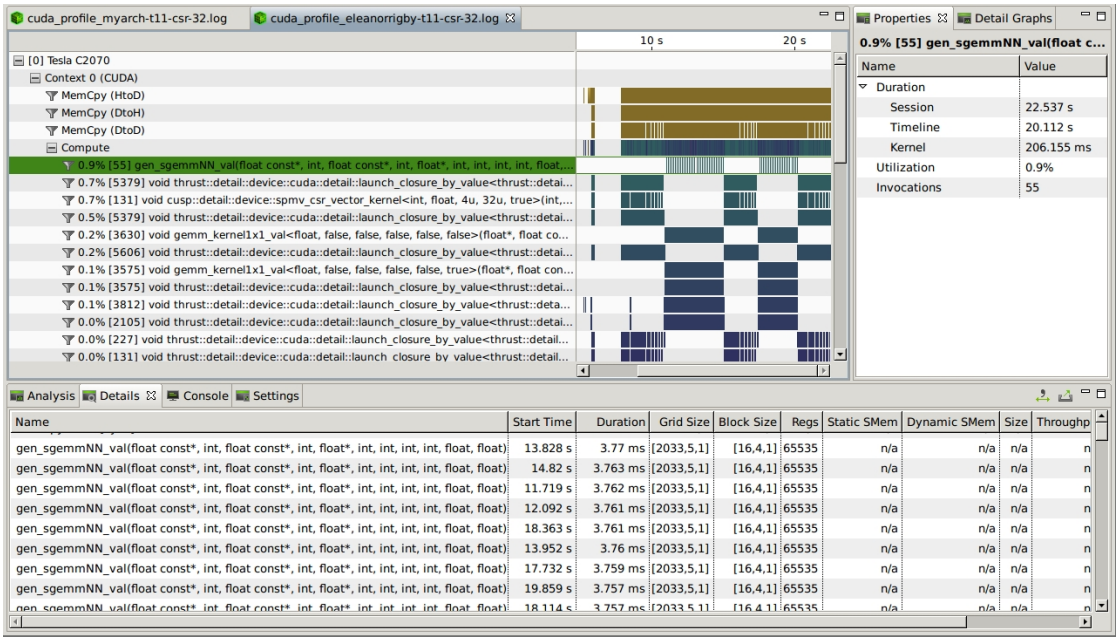

Figura 8.28: Profiling de IRAM en Eleanorrigby

Capítulo 9

Conclusiones

En esta memoria hemos analizado implementaciones para la resolución de sistemas de ecuaciones lineales y cálculo de valores propios con matriz dispersa utilizando métodos iterativos en CUDA. Desde el punto de vista computacional el producto Matriz-vector tiene el coste mayor, y es por eso que es necesario optimizar este cálculo. Desde el punto de vista experimental, en general, un procesador CPU moderno no puede llegar a un nivel de paralelismo como una unidad multi-cores GPU. Aunque la eficiencia, vista en término de capacidad de aprovechar todas las unidades de cálculo disponibles, no es tan alta en GPU por la dificultad de asociar los datos con los recursos disponibles, la GPU nos permite un elevado nivel de paralelismo que se adapta por al tipo de problema que hemos tenido en consideración en esta memoria.

Una vez completado el algoritmo optimizado para SpMV se puede diseñar todos los tipos de métodos iterativos para matrices dispersas. En particular, para el problema tratado en esta memoria (ver [10] para mayor detalles) nos hemos enfocados en la resolución de sistemas lineales y cálculo de valores propios. En los dos casos, es necesario diseñar métodos iterativos basados en el sub-espacio de Krylov. Sin embargo, en esta memoria hemos mostrado los resultados obtenidos que resultan estar relacionados a los de SpMV visto que tal multiplicación representa una de la operación con mayor coste.

Los principales problemas que se han presentado en el análisis son que la aplicación de las optimizaciones en CUDA están muy dependientes del tipo de la arquitectura utilizada. Haciendo pruebas con tarjetas gráficas NVIDIA muy diferentes en prestaciones, se ha podido observar como las mejoras computacionales obtenidas con *DIA_FAST* en la máquina MyArch no aparecen en una máquina con grandes capacidades como Golub o Eleanorrigby. En particular, las ventajas principales de *DIA_FAST* aparecen en arquitectura donde el ancho de banda no es muy alto (como en la maquina MyArch) en cuanto a través de *DIA_FAST* se reduce enormemente el coste debido al acceso en memoria global utilizando la memoria compartida existente en cada bloque de hilo como una memoria cache. En arquitecturas como Golub y Eleanorrigby donde el ancho de

banda es de diferentes orden de magnitud mayor a la de MyArch, las optimizaciones aplicadas con *DIA_FAST* no resultan tan evidentes como se ha visto desde el análisis experimental y, además, pueden representar un *overhead* en presencia de matrices de gran tamaño.

De todas formas, particular interés lo tienen las matrices, que hemos llamado L_{ii} y T_{ii}, obtenidas a través de de la Ecuación de la difusión Neutrónica en reactores nucleares. Lo que se puede ver es que sus estructuras son muy diferentes por el hecho que la primera se basa sobre un proceso de mallado con celdas cuadradas y la otra matriz sobre celdas de forma hexagonal. En cualquiera de las máquinas que hemos analizado, la matriz L_{ii} resulta más adecuada con un formato de almacenamiento DIA, mientras que para T_{ii}, siendo que los elementos no están alineados en diagonales resulta el más adecuado el *JAD* y su variante. En particular, el variante específico para la matriz T_{ii} llamado *JAD Block* esta diseñado para garantizar una perfecta coaleciencia entre hilos de un mismo *warp*, mayor utilización de la memoria compartida y distribución de la carga entre todos los bloques de hilos que se ejecutan en los cores de la GPU. Las optimizaciones manejadas con *JAD Block* pueden ser utilizadas para matrices de gran tamaño (actualmente se están estudiando matrices veinte veces mayores respecto con T_{ii}).

Desde el punto de vista experimental se ha demostrado que el *JAD Block*, aunque resulta ser el más eficiente respecto a cualquier formato presente en CUSP, respecto al clásico *JAD* no obtenemos los resultados esperados. Eso es principalmente debido al overhead sobre las optimizaciones aplicadas y que resulta difícil de evaluar en la etapa del diseño teórico sobre el formato de almacenamiento. Otra importante consideración es el hecho que en CUSP el coste de añadir precondicionadores resulta muy alto, aunque el precondicionador de Jacobi nos permite una convergencia en términos de número de iteracciones diferente orden de magnitud inferior.

La primera versión del algoritmo IRAM [7, 1] para el cálculo de valores propios permite lograr valores y vectores propios sobre matrices bien definidas y con valores propios muy distintos. En una primera análisis experimental se ha podido observar como la ejecución del algoritmo en arquitectura con GPUs permite lograr un alto nivel de paralelismo mejorando la eficiencia de diferentes magnitud respecto a la ejecución del mismo algoritmo en CPU. Se puede observar que los recursos utilizados durante la ejecución de IRAM en GPUs resulta muy alto y sólo se obtiene uno *speedup* máximo alrededor de cuatro. La principal razón puede ser debida al hecho que el algoritmo IRAM resulta ser un método iterativo donde hay diferentes partes secuenciales que no pueden ser ejecutadas en paralelo, como por ejemplo el control de la convergencia, copia de vectores y matrices de soporte, etc. También, la reducción en el subespacio de Krylov nos lleva a hacer cálculos sobre una matriz de Hessenberg de tamaño muy pequeño. Entonces, aunque el IRAM está caracterizado por una importante operación que puede ser hecha en paralelo como SpMV aprovechando muchísimo de los recursos de la GPU, las partes secuenciales del algoritmo reducen el *speedup* en un nivel teórico que no depende de las cantidades de recursos que se utilizan sino sólo de la fracción secuencial que en IRAM aparece. Teniendo en consideración la ley de Amdhal y el resultado obtenido en IRAM, es posible decir que con una porción de

sólo el 25 % de parte secuencial, podemos llegar a un *speedup* máximo teórico de cuatro. Desde el otro lado, hay también que observar que el tiempo de ejecución obtenido en la CPU no es secuencial sino que se basa sobre una implementación que utiliza MKL y OpenMP para lograr paralelismo directamente en CPU. Eso significa que el *speedup* obtenido resulta mayor de lo que se puede deducir en la Figura 8.23, 8.26. Además, a través de una análisis de *profiling* se ha podido observar que para el algoritmo IRAM el coste dominante no resulta ser el SpMV sino que hay otras operaciones como producto de matrices que implican tiempos de ejecución muy altos, así que no es posible explotar los recursos adecuadamente para lograr buenas prestaciones utilizando formatos de almacenamiento específico como en el caso de SpMV o del Sistema de ecuaciones lineales. Se ha observado que para máquinas con pocos recursos como MyArch los formatos de almacenamiento específicos ofrecen un mejor resultado por el hecho que aprovechan más de los recursos disponibles.

9.1. Trabajos futuros

A través de los resultados obtenidos con el algoritmo IRAM es posible añadir nuevas líneas de investigación que permitan de obtener mejores resultados para matrices genéricas. En particular, un trabajo futuro consistirá en generalizar el algoritmo IRAM para matrices más complejas que presenten *clusters* de valores propios (como L_{ii} y T_{ii}), substituyendo la rutina que calcula los valores propios para matrices densas por la descomposición de Schur.

Otra mejoría del algoritmo IRAM consiste en introducir una buena estrategia para la selección de los valores propios y adaptar este algoritmo con el problema de valores propios generalizado.

En el análisis de *profiling* se ha observado que unas procesos del algoritmo IRAM, como el algoritmo QR con Desplazamiento Implícito, representa el mayor coste del algoritmo. Estas porciones del algoritmo IRAM puede ser mejoradas añadiendo una adecuada estructura de paralelización.

Bibliografía

[1] *ARPACK Users' Guide:Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.*

[2] *CULA libraries.*

[3] *CUSP libraries.*

[4] *LAPACK Users' Guide. Society for Industrial and Applied Mathematics.*

[5] *Nvidia Cuda C Programming Guide.*

[6] *Introduction to Scientific Computing: A Matrix-Vector Approach Using MATLAB*, 1997.

[7] *Numerical Methods for Large Eigenvalue Problems*, 2011.

[8] N. Bell and M. Garland, *Efficient sparse matrix-vector multiplication on CUDA*, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[9] D. R. K. C. L. Lawson, R. J. Hanson and F. T. Krogh, *Basic linear algebra subprograms for fortran usage.*

[10] J. R. Gumersindo Verdù, Damian Ginestar and V. Vidal, *3d alpha modes of a nuclear power reactor*, Nuclear Science of Technology, (2010).

[11] S. H. J. J. Dongarra, J. DuCroz and R. J. Hanson, *An extended set of fortran basic linear algebra subprograms.*, ACM Trans. on Math. Software, 1 (1988), pp. 1–17.

[12] R. Li and Y. Saad, *Gpu-accelerated preconditioned iterative linear solvers.*

[13] Y. Saad, *Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems*, Mathematics of Computation, 42 (1984), pp. 567–588.

[14] J. R. Weston and M. Stacey, *Space-time nuclear reactor kinetics*, Academic Press, (1969).