UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Particionamiento óptimo de matrices dispersas

Raúl Ignacio Marichal

Proyecto de Grado en Ingeniería en Computación
Facultad de Ingeniería
Universidad de la República

UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Particionamiento óptimo de matrices dispersas

Raúl Ignacio Marichal

Co-directores:
Dr. Ing. Ernesto Dufrechou
Dr. Ing. Pablo Ezzatti

Marichal, Raúl Ignacio

Particionamiento óptimo de matrices dispersas / Raúl Ignacio Marichal. - Montevideo: Universidad de la República, Facultad de Ingeniería, 2021.

xv, 91 p. 29, 7cm.

Co-directores:

Ernesto Dufrechou

Pablo Ezzatti

Tesis de Grado – Universidad de la República, Programa en Ingeniería en Computación, 2021.

Referencias bibliográficas: p. 79 – 86.

1. matrices dispersas, 2. almacenamiento óptimo, 3. reordenamiento, 4. múltiples precisiones. *et al.* II. Universidad de la República, Proyecto de Grado en Ingeniería en Computación. III. Título.

INTEGRANTES DEL TRIBUNAL DE DEFENSA DE TESIS

_____

D.Sc. Prof. Nombre del 1er Examinador Apellido

_____

Ph.D. Prof. Nombre del 2do Examinador Apellido

_____

D.Sc. Prof. Nombre del 3er Examinador Apellido

Agradecimientos

Quisiera agradecer, en primer lugar, a mis hermanos y al resto de mi familia que me apoyó y motivó durante todo mi proceso educativo. Agradezco, también a todas las personas de que una manera u otra, me brindaron su apoyo, compartiendo su valioso tiempo, a veces recursos, para ayudarme a que continúe. Además quiero dar las gracias a mis amigos, que supieron escucharme y compartir, a veces desde la distancia entendiendo cuando no pude acompañarlos. He de agradecer también, a las instituciones, Fondo de Solidaridad y Bienestar Universitario, que me brindaron apoyo financiero a lo largo de la carrera de no haber contado con ellas, hubiese sido, sin duda, mucho más difícil llegar hasta aquí.

Sin quitarles mérito, quisiera agradecer a mis tutores, Ernesto y Pablo, por su apoyo y guía durante todo el proceso, siendo un pilar importantísimo para la finalización del proyecto, motivándome a siempre seguir adelante, incluso cuando creía haber perdido la fuerza de voluntad.

RESUMEN

Las matrices dispersas tienen múltiples aplicaciones en el ámbito de la ciencia y la ingeniería, ya que son una herramienta fundamental para la resolución de problemas de gran escala que no pueden ser modelados por matrices densas como, por ejemplo, las simulaciones de circuitos electrónicos, la resolución de ecuaciones diferenciales parciales utilizando FEM, o incluso operaciones con grafos de redes sociales. La creciente importancia de las matrices dispersas para la comunidad científica motiva el estudio de técnicas que permitan el manejo eficiente, tanto del almacenamiento como del cómputo de las operaciones asociadas con este tipo de matrices. En general, estas técnicas buscan reducir el tráfico de datos con la memoria principal mediante formatos de almacenamiento que permitan ubicar los elementos no nulos dentro de la matriz transfiriendo la menor cantidad de datos posibles.

El objetivo principal de este proyecto es avanzar en el estudio y comprensión de estas estrategias. En particular, se evalúan estrategias de particionamiento y procesamiento de matrices para el uso eficiente de técnicas de almacenamiento centradas, principalmente, en la aplicación de reordenamientos, el uso de múltiples precisiones para almacenar los índices de los elementos no nulos y formatos híbridos que permitan almacenar la matriz mediante una parte regular, en general densa, y una parte irregular dispersa. El trabajo incluye, en primer lugar, la actualización del estado del arte respecto a formatos de almacenamiento disperso. Luego se desarrollaron un conjunto de heurísticas que tienen por objetivo optimizar el espacio de almacenamiento de las matrices dispersas mediante el particionamiento de las mismas, alcanzando resultados alentadores. Por último, se extendió la evaluación experimental midiendo el impacto de la compresión de índices luego de aplicar los reordenamientos. Este estudio permitió identificar los importantes ahorros en cuanto a espacio de almacenamiento que se pueden obtener al comprimir índices y, además, resaltó la importancia de combinar estrategias de reordenamiento para dicha tarea.

Palabras claves:
matrices dispersas, almacenamiento óptimo, reordenamiento, múltiples precisiones.

Tabla de contenidos

Capítulo 1

Introducción

Las matrices dispersas son aquellas que poseen una fracción relativamente pequeña (frecuentemente muy menor al 1 %) de sus coeficientes distintos a cero, lo que motiva el uso de estructuras de almacenamiento que aprovechen esta particularidad. Específicamente, se suelen almacenar únicamente los coeficientes distintos de cero, acompañados por índices que permitan deducir sus coordenadas en la matriz. Su importancia en el ámbito de la ciencia y la ingeniería radica en que son una herramienta fundamental para la resolución de problemas de gran escala que no pueden ser modelados por matrices densas, por ejemplo, algunos problemas de optimización [29] o la resolución de ecuaciones diferenciales parciales utilizando el Método de Elementos Finitos (o FEM por Finite Element Method) [55]. Los FEMs [6] permiten resolver ecuaciones diferenciales asociadas a un problema físico sobre geometrías complicadas (por ejemplo [4]). Son usados en el diseño y mejora de productos y aplicaciones industriales, así como en la simulación de sistemas físicos y biológicos complejos. La variedad de problemas a los que puede aplicarse ha crecido enormemente, siendo el requisito básico que las ecuaciones constitutivas y ecuaciones de evolución temporal del problema sean conocidas de antemano, representadas comúnmente a la hora de resolverlas como matrices dispersas. Una completa revisión del uso de matrices dispersas en la computación científica, en las etapas tempranas de desarrollo, hasta el año 1977, se puede encontrar en el trabajo de I. Duff [22].

En los últimos años, las matrices dispersas han ido cobrando cada vez más relevancia en el campo de la computación científica, debido, por ejemplo, al impulso del crecimiento de las aplicaciones relacionadas con las redes sociales.

En este contexto, las matrices representan, en general, un grafo de las relaciones entre los diferentes usuarios, tal como se presenta en [39], alcanzando matrices de dimensiones extraordinariamente grandes y, al mismo tiempo, con muy pocos coeficientes no nulos.

Esta evolución, descrita en los párrafos anteriores, motiva el estudio de técnicas eficientes, tanto desde el punto de vista del almacenamiento como del cómputo de las operaciones asociadas a las matrices dispersas. Dado que las características de estas matrices, como pueden ser su tamaño, la proporción de elementos distintos de cero o la posición de los mismos, varían fuertemente según la aplicación, esta área de trabajo se encuentra aún en constante desarrollo.

Uno de los principales desafíos que presentan los problemas con matrices dispersas (a través de operaciones como la multiplicación matriz dispersa-vector -SpMV-) es que, en su gran mayoría, son inherentemente limitados por los accesos a memoria, por lo que mejorar la forma de almacenar las matrices para optimizar las comunicaciones de datos entre la memoria y el procesador es crucial. En este caso, suelen ser útiles las técnicas de ordenamiento que permiten reorganizar los coeficientes no nulos de la matriz de forma que éstos puedan almacenarse de manera más eficiente. Otra de las ideas interesantes que apuntan a optimizar el trabajo con matrices dispersas, son aquellas que intentan reducir la precisión de los elementos almacenados, reduciendo las comunicaciones con la memoria. Hartwig Anzt et al. [34], por ejemplo, presenta estrategias enfocadas en almacenar los coeficientes en punto flotante de precisión adaptable. Sin embargo, parte importante de las comunicaciones en problemas dispersos están dedicadas al manejo de los índices que permiten direccionar dichos coeficientes. Por este motivo es interesante explorar técnicas que permitan, al menos para ciertos tipos de matriz dispersa, almacenar dichos índices de una forma más eficiente. Algunos de los esfuerzos se centran en lograr comprimirlos, aplicando codificaciones distintas, por ejemplo, el delta encoding [57]. Otros intentan agrupar elementos contiguos y direccionarlos con un índice. Como último punto, presenta un gran interés el hecho de combinar las estrategias antes mencionadas, estudiando la aplicación de ordenamientos con una posterior compresión de índices.

En el contexto descrito anteriormente, el objetivo de este proyecto es avanzar en el estudio y comprensión de estrategias de optimización de uso de matrices dispersas. En particular, se busca evaluar estrategias de particionamiento y

procesamiento de matrices para el uso eficiente de técnicas de almacenamiento. Para alcanzar el objetivo general del proyecto se plantean los siguientes objetivos específicos:

- Actualizar el estado del arte del uso de matrices dispersas.
- Actualizar el estado del arte del uso de computación de alta performance (HPC) y en especial, su uso para acelerar la resolución de problemas de álgebra lineal numérica (ALN) dispersa.
- Estudiar las técnicas híbridas, reordenamientos y el uso de múltiples precisiones para matrices dispersas.
- Desarrollar estrategias (formatos, procedimientos, etc.) para matrices dispersas que permitan alcanzar un uso más eficiente de los datos/cómputo.

El resto del documento se estructura de la forma que se describe a continuación. El Capítulo 2 presenta, de forma acotada, los conceptos y la teoría en la que está basada este proyecto. Entre otros temas, se aportan detalles sobre matrices dispersas, la multiplicación Matriz dispersa-Vector (SpMV), estrategias de reordenamiento de matrices y hardware para HPC. En el Capítulo 3 se resumen los principales antecedentes relacionados con el uso de formatos de almacenamiento de matrices dispersas, el uso de múltiples precisiones y las técnicas de reordenamiento de matrices asociadas. Posteriormente, en el Capítulo 4, se presentan los esfuerzos realizados en el contexto del proyecto de grado. Esto incluye el estudio de técnicas de reordenamiento para formatos híbridos y la evaluación de técnicas de múltiples precisiones. El documento se cierra con un resumen de las conclusiones arribadas durante el desarrollo del proyecto y la identificación de líneas de trabajo para continuar el esfuerzo en el Capítulo 5.

Capítulo 2

Fundamentos teóricos

Este capítulo incluye un resumen de los enfoques, teorías y conceptos en los cuales se fundamenta el trabajo del proyecto de grado. Se basa, principalmente, en la exposición de otros trabajos sobre los temas estudiados, buscando cierto nivel de auto-contención en el documento. En otras palabras, el objetivo de este capítulo es guiar al lector en la interpretación de trabajos que se han ocupado previamente de la cuestión central de esta tesis, incluyen conocimiento de base para este trabajo u ofrecen herramientas analíticas o interpretativas para los estudios realizados. Específicamente, en este capítulo se incluyen conceptos introductorios sobre matrices dispersas, en especial los formatos más comunes para almacenarlas, así como el uso de hardware para computación de alta performance (HPC por su sigla en inglés –High Performance Computing–) con foco en el uso de los procesadores gráficos o simplemente GPUs.

2.1. Matrices dispersas

Para iniciar los conceptos, primero se presenta una definición formal de matriz. Una matriz es un arreglo bidimensional de números dispuestos en filas y columnas, donde dichos números frecuentemente representan los coeficientes de un sistema lineal. En cuanto a la notación, típicamente las matrices se llaman con letras mayúsculas (A) y quedan definidas por sus dimensiones, es decir, la cantidad de filas y columnas que dispone. Las entradas del arreglo se denominan coeficientes, comúnmente identificados con letra minúscula y dos subíndices que indican fila y columna respectivamente (a_{ij}). Ver Figura 2.1 por un resumen gráfico.

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

Figura 2.1: Matriz de tamaño $m \times n$.

El concepto de matriz dispersa no cuenta con una definición específica. En forma conceptual, se puede considerar como aquellas matrices "grandes" que incluyen "una porción importante" de coeficientes con valor nulo. Es claro que "grandes" y "una porción importante" son términos ambiguos para referirse a las características de las matrices ya que dependen del contexto de aplicación. Por ejemplo, una matriz en los años 60s, podía considerarse grande y en la actualidad ser una matriz pequeña. Según Tim Davis [62] las matrices dispersas que surgen de los problemas del mundo real, tanto en ciencia, ingeniería, matemática y otras áreas, tienden a ser dispersas. Los algoritmos que trabajan con dichas matrices, se encuentran en la intersección de la teoría de grafos y el álgebra lineal numérica. Un grafo puede representar las conexiones entre variables en el modelo matemático, como el voltaje a través de un componente de un circuito, un enlace de una página web a otra, las fuerzas físicas entre dos puntos en una estructura mecánica, etc. El álgebra lineal numérica surge porque estas matrices representan sistemas de ecuaciones cuya solución nos dice algo sobre cómo se comporta el problema del mundo real. Por ejemplo, el algoritmo *page rank* [49] de Google, requiere el cálculo de un vector propio para una matriz con tantas filas y columnas como páginas en la Web.

Otro aspecto relacionado con las matrices dispersas es el almacenamiento. Para hablar de matrices dispersas es necesario que se utilice un formato de almacenamiento que saque partido (al menos que lo intente) de los coeficientes con valor 0[1]. Es decir, si se tiene una matriz grande y con muchos coeficientes nulos, pero está almacenada en el formato tradicional, no es posible sacar partido de esta propiedad y, en la práctica, su tratamiento será idéntico al de una matriz densa.

En el contexto de este proyecto, para la definición de matriz dispersa, se

[1]En campos de aplicación específicos podría ser dispersa con respecto a otro valor. En este caso, se usa 0 como valor nulo.

pondrá foco en el uso de alguna estrategia de almacenamiento que saque partido de la proporción de coeficientes no nulos. Para esto, se presenta el concepto de densidad de una matriz dispersa [48], magnitud que permite de cierto modo cuantificar cuán dispersa es una matriz. Esta se define como $\rho = \frac{nnz}{m \times n}$, donde nnz es la cantidad de elementos no nulos de la matriz, siendo m y n la cantidad de filas y columnas respectivamente. Se considera dispersas, en general, a aquellas matrices que presentan valores de densidad por debajo del 1 %. Sobre las dimensiones de la matriz no se pondrán restricciones, pero recordando que, en los dispositivos actuales, las matrices que al menos tengan centenas o miles de filas/columnas son las que presentan real interés.

En este contexto, es claro que dichas matrices se pueden almacenar en memoria de diferentes formas, omitiendo al menos la gran mayoría de los elementos nulos. Como ejemplo, notar que, utilizando la forma convencional de almacenamiento (matriz densa), una matriz de $10^4 \times 10^5$ elementos utilizando aritmética de punto (o coma) flotante de doble precisión (`64 bits`, i.e. `8 Bytes`) para los coeficientes, implica el uso de un total de $10^4 \times 10^5 \times 8$ `Bytes` $= 8 \times 10^9$ `Bytes` $= 8$ `GBytes`. En caso de tener una densidad del 1 %, es decir 10^7 coeficientes no nulos y poder almacenar únicamente estos coeficientes, implicaría un almacenamiento de 0.08 `GBytes`. Este razonamiento permite observar los posibles ahorros en memoria, aunque es claro que se necesitará información extra para indicar al menos las posiciones de los coeficientes.

Si bien se desprende de los párrafos anteriores que, se pueden lograr grandes ahorros en memoria con el simple hecho de guardar solamente los elementos no nulos y sus correspondientes coordenadas o índices que permitan identificar cada una de estas entradas dentro de la matriz, esto no garantiza una representación óptima (o eficiente) a la hora de operar con dicha matriz. Interesa, entonces, encontrar cierto balance en las características del formato de almacenamiento, de modo de poder optimizar tanto requerimientos de memoria como de cómputo.

Como se mencionó antes, las características de las matrices pueden ser muy variadas dependiendo de los problemas que éstas representan o intentan resolver, dando lugar a estructuras particulares que algunos formatos intentan explotar. Muy a grandes rasgos, según las propiedades o características de las matrices, determinadas por la disposición de los elementos no nulos (concepto denominado patrón de la matriz [7]), se las puede clasificar en las siguientes categorías:

- **de banda:** se dice que una matriz es de banda cuando tanto el ancho de banda (la mayor distancia de un elemento no nulo a la diagonal) inferior como el superior son razonablemente "pequeños". Expresado matemáticamente, una matriz A de tamaño $n \times n$, es una matriz de banda si todos sus elementos no nulos están comprendidos dentro de un rango o zona entorno a la diagonal principal. Entonces $a_{ij} \neq 0$ si $i-j < k_1$ o $j-i < k_2$, donde k_1 y k_2 corresponden a los semi-anchos izquierdo (inferior) y derecho (superior) respectivamente. En la literatura, algunos autores definen el ancho de banda de una matriz en base a los semi-anchos, obteniendo:

  $$\beta(A) = max\{k_1, k_2\}.$$

  Otros, de forma equivalente, directamente definen el ancho de banda en base a los índices de los coeficientes no nulos. Por ejemplo, George et al. [26] definen el ancho de banda de una matriz A de la siguiente manera:

  $$\beta(A) = max\{|i-j| \mid a_{ij} \neq 0\}.$$

  También es importante considerar el concepto de *profile*. El profile de una matriz A, simétrica, está definido como: $profile(A) = \sum_{i=1}^{n} \beta_i(A)$, donde $\beta_i(A)$ es el ancho de banda en la fila i-ésima de la matriz A. Es decir, la distancia máxima de un valor no nulo de la fila i a la diagonal.
- **diagonal:** un caso extremo de matriz de banda, son las matrices diagonales, donde la banda está compuesta por una única diagonal.
- **simétricas** (o **hermíticas** en el caso complejo): una matriz A es simétrica sí y solo sí $A = A^t$ o $a_{ij} = a_{ji}$, para todo valor válido de i y j, es decir, que sean menores o iguales a la cantidad de filas y columnas respectivamente. En el caso complejo, $a_{ij} = \overline{a_{ji}}$. Un ejemplo común de donde aparecen estas matrices son los grafos bidireccionales o no dirigidos.
- **de bloques:** una matriz de bloques, es una matriz que se puede interpretar como la composición de varias sub-matrices o bloques cuadrados de dimensiones n_b. En particular, para matrices dispersas, la división de estos bloques se realiza de forma que estos contengan todos los elementos no nulos, dejando por fuera la mayor cantidad de elementos nulos, ver apartados 2.2.7 y 2.2.6.
- **cantidad constante de elementos por fila/columna:** son las ma-

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 & 4 & 0 \\ 0 & 5 & 0 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 9 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 4 & 5 & 0 \\ 0 & 0 & 6 & 0 & 7 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9 \end{pmatrix}$$

Figura 2.2: Matriz A ejemplo.

trices que presentan la misma cantidad de elementos no nulos en todas las filas (o columnas). Una clase importante de problemas ofrecen esta característica y los formatos asociados también son particulares.

2.2. Formatos de almacenamiento

A la hora de almacenar este tipo de matrices, como se menciona en la Sección 2.1 existen varias estrategias, clasificadas principalmente en dos grandes grupos, estáticas y dinámicas. Parece importante mencionar, y de forma breve explicar, como funcionan los formatos clásicos para almacenar matrices dispersas.

- Estrategias estáticas: Utilizadas normalmente cuando la estructura de la matriz (el patrón de coeficientes no nulos) no sufrirá grandes modificaciones. Por lo general, utilizan menos memoria y poseen mayor velocidad de acceso a los coeficientes que las estrategias dinámicas. Entre las estrategias estáticas que se utilizan frecuentemente se destacan: el formato simple o coordenada-valor (COOrdinate format), el formato comprimido por fila (Compress Row Storage - CRS), el formato comprimido por columna (Compressed Column Storage - CCS), almacenamiento por diagonales (Compressed Diagonal Storage) y el formato Ellpack-itpack (ELL).
- Estrategias dinámicas: Se eligen cuando la frecuencia de modificación de la estructura de la matriz es alta, ya que permiten agregar nuevos coeficientes de forma simple. Entre las distintas estrategias dinámicas se pueden destacar: lista bidimensional doblemente enlazada (*Linked List Row-Column Storage* - LLRCS), listas enlazadas por filas (*Linked List*

Row Storage - LLRS) y listas enlazadas por columna (*Linked List Column Storage* - LLCS).

Además de la anterior, existen otras formas de clasificar los formatos. Por ejemplo, Bell y Garland [10, 11] proponen la siguiente categorización, que va desde formatos generales, que no asumen nada de la matriz o no exigen características determinadas (ej. COO), y específicos, que buscan explotar propiedades particulares de la matriz (ej. BCRS o DIA). Se puede encontrar, además, otra gran cantidad de estrategias de almacenamiento tanto estáticas como dinámicas, así como propuestas de almacenamiento disperso especialmente diseñadas para determinados tipos de matrices y/o para utilizar con determinados algoritmos. A continuación se describen algunas de las estrategias de almacenamiento disperso más difundidas. Para facilitar la comprensión de las estrategias, en algunos casos se muestra la aplicación de las estrategias descritas sobre la matriz A, presentada en la Figura 2.2.

2.2.1. COO (COOrdinate format)

$$
\begin{aligned}
d &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} \\
f &= \begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 3 & 3 & 3 & 4 & 4 & 5 & 5 & 5 & 6 & 6 & 6 & 7 \end{pmatrix} \\
c &= \begin{pmatrix} 1 & 2 & 5 & 6 & 2 & 4 & 2 & 3 & 4 & 1 & 4 & 2 & 5 & 6 & 3 & 5 & 6 & 7 \end{pmatrix}
\end{aligned}
$$

Figura 2.3: Matriz A almacenada en formato COO.

El formato simple o formato coordenada (COOrdinate format), quizás la forma más natural de representar una matriz dispersa, utiliza tres vectores para el almacenamiento de la matriz. Un vector d de tipo punto flotante para los datos (entradas de la matriz) y dos vectores (f y c) de enteros para los índices (fila, columna) como se muestra en la Figura 2.3. Cada valor no nulo de la matriz está asociado a 3 valores, uno por cada vector componente del formato. En cuanto a las necesidades de memoria del método, si la matriz tiene nnz coeficientes no nulos entonces se utilizan nnz entradas de tipo punto flotante y $2nnz$ entradas de tipo entero para los índices.

Si se desea acceder al valor $A(i, j)$ es necesario saber el p tal que $f(p) = i$ y $c(p) = j$ y el valor buscado es $d(p)$. Se pueden almacenar los coeficientes conservando algún orden (recorriendo por fila o columna) o en forma aleatoria.

Como ventaja de la estrategia de almacenamiento se puede destacar que es sumamente sencilla e intuitiva. En contra, vale la pena mencionar la necesidad de memoria, ya que si la densidad por fila/columna es mayor a uno (situación normal) utiliza más memoria que otros métodos. Además el formato no ofrece ventajas de acceso ni por filas ni por columnas.

2.2.2. CRS (Compressed Row Storage)

El formato comprimido por fila (también conocida como CSR-*Compressed Sparse Row*) utiliza, al igual que el formato simple, tres vectores. Un vector d de tamaño nnz y tipo punto flotante[1], en el que se almacenan los valores de los coeficientes. Otro vector c de tamaño nnz en el que se almacenan los números de columna de los elementos distintos de cero, por último, un vector f de tamaño $n+1$ siendo n la cantidad de filas de la matriz, en el cual se almacenan las posiciones de los primeros elementos de cada fila accesibles a través de c y d.

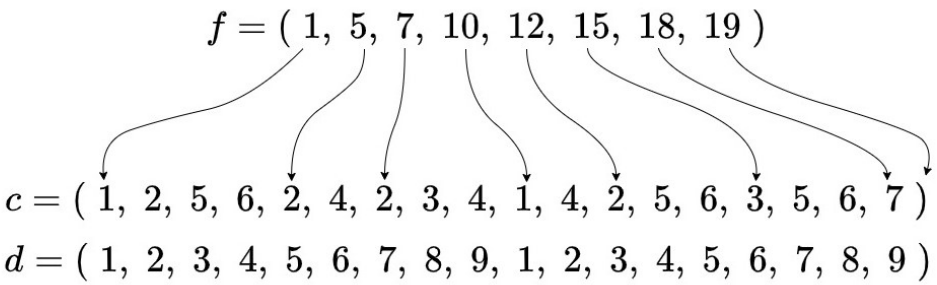


Figura 2.4: Matriz A almacenada en formato CRS.

En la Figura 2.4 se muestra cómo se representa la matriz A utilizando la estrategia CRS. Para acceder al valor $A(i,j)$ de la matriz A, se obtienen primero los índices $p_1 = f(i)$ y $p_2 = f(i+1)$, posteriormente se busca el índice p tal que $p_1 \leq p < p_2$ y $c(p) = j$, y luego se puede obtener el valor buscado accediendo a $d(p)$.

Si la matriz tiene más coeficientes no nulos que cantidad de filas, la estrategia CRS utiliza menos memoria que el formato simple. Es necesario conocer todas las posiciones de los coeficientes para generar la estructura en forma eficiente o, dicho de otra forma, es muy difícil agregar un nuevo valor a la

[1]Podría ser otro tipo de dato.

estructura, a menos que sean posteriores en la matriz al último elemento ya presente. Este formato permite acceder fácilmente a todos los elementos de una fila, pero no a los de una columna.

2.2.3. CCS (Compressed Column Storage)

El formato comprimido por columna es similar al CRS pero utilizando el vector comprimido para las columnas. En la Figura 2.5 se muestra como se aplica esta estrategia a la matriz A.

$$
\begin{aligned}
d &= \begin{pmatrix} 1 & 1 & 2 & 5 & 7 & 3 & 8 & 6 & 6 & 9 & 2 & 3 & 4 & 7 & 4 & 5 & 8 & 9 \end{pmatrix} \\
f &= \begin{pmatrix} 1 & 4 & 1 & 2 & 3 & 5 & 3 & 6 & 2 & 3 & 4 & 1 & 5 & 6 & 1 & 5 & 6 & 7 \end{pmatrix} \\
c &= \begin{pmatrix} 1 & 3 & 7 & 9 & 12 & 15 & 18 & 19 \end{pmatrix}
\end{aligned}
$$

Figura 2.5: Matriz A almacenada en formato CCS.

Se mantienen las necesidades de memoria del formato CRS y en cuanto a las características de acceso se invierten los conceptos con respecto a filas y columnas.

2.2.4. DIA (DIAgonal format)

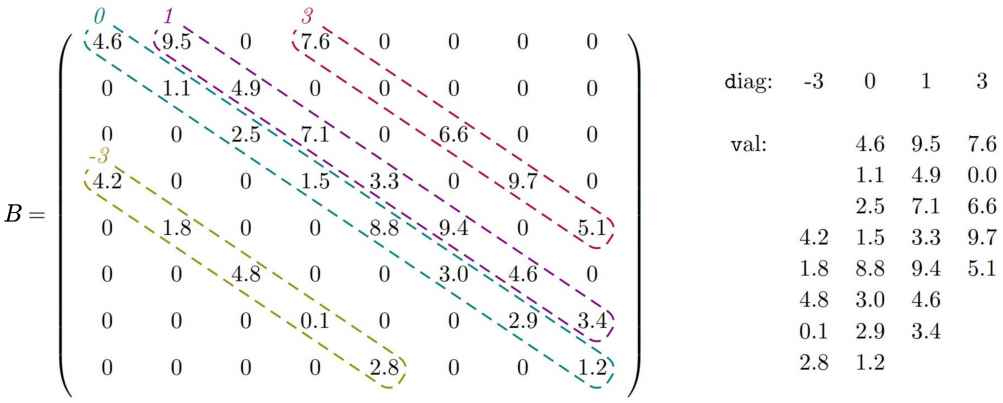


Figura 2.6: Matriz B almacenada en formato DIA.

Cuando la matriz a almacenar es de banda, se pueden utilizar estrategias de almacenamiento por diagonales. Entonces se almacena la matriz original en

una matriz rectangular de tamaño $n \times d$ siendo n la dimensión de la matriz y d la cantidad de diagonales.

Si se desea almacenar una matriz que posee d diagonales no consecutivas, una variante de formato es el *DIAgonal storage Format* que utiliza una matriz de tamaño $n \times d$ en la que se almacenan las d diagonales y un vector de tamaño d en el que se especifica el desplazamiento de cada diagonal con la diagonal principal, estando la diagonal principal asociada con 0, las diagonales "por encima" con valores positivos y "por debajo" con valores negativos. En la Figura 2.6 se puede ver un ejemplo.

2.2.5. ELL (Ellpack-itpack)

En el formato *Ellpack-itpack*, se utiliza una estructura densa de tamaño $n \times k$, donde n es la cantidad de filas y $k = max_i(nnz\,(A_i))$, con A_i la fila i-ésima de la matriz, es decir, la cantidad máxima de elementos no nulos por fila. La matriz dispersa es almacenada en dos matrices "densas" de tamaño $n \times k$, una con las entradas no nulas de la matriz y otra con los índices de las columnas. Es necesario agregar explícitamente valores nulos para completar la primer matriz (*zero padding*). Este problema es menor cuando todas las filas de la matriz son de largos similares (el caso ideal son las matrices con cantidad constante de elementos por fila/columna). En la Figura 2.7, se puede observar como se almacena una matriz en formato ELL.

Dado que la estructura elegida es de tamaño $n \times k$, es decir, tiene la misma cantidad de filas que la matriz original, no es necesario almacenar explícitamente los índices de fila ya que están implícitos en la estructura, a diferencia de otros formatos, como por ejemplo COO.

2.2.6. BCRS (Block Compressed Row Storage)

Este formato se utiliza cuando la matriz se puede dividir/almacenar eficientemente como sub-bloques regulares.

Se tiene entonces, una matriz de $n \times m$ con nnz_b bloques de dimensión n_b[1], con entradas de números en punto flotante, enfocada a almacenar los elementos no nulos, además de dos vectores de números enteros, uno para el índice de columna de cada sub-bloque y el otro con el índice (puntero) al comienzo de

[1]En general los bloques son cuadrados y tanto m como n son múltiplos de n_b.

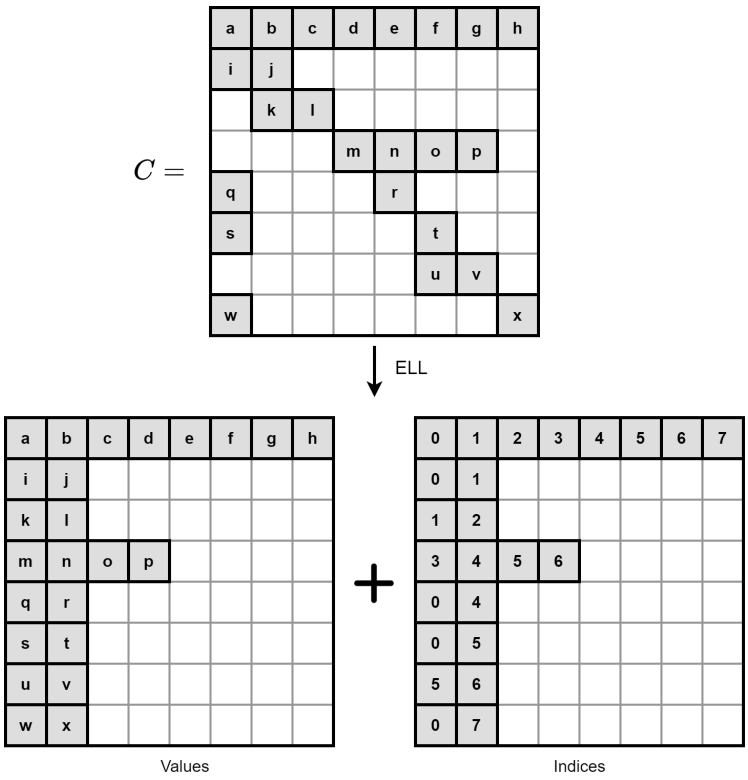


Figura 2.7: Matriz C almacenada en formato ELL.

cada fila.

Notar que este formato funciona razonablemente bien si la matriz posee bloques regulares.

2.2.7. BCCS (Block Compressed Column Storage)

Funciona de la misma manera que BCRS pero, a la hora de acceder a cada sub-bloque de la matriz, se utiliza CCS.

2.2.8. LLRCS (Linked List Row-Column Storage)

En esta estrategia se utiliza una multi-estructura bidimensional. Cada entrada correspondiente a un no-nulo contiene dos punteros a sus adyacentes por fila, y dos a sus adyacentes por columna. Además, dispone de dos vectores de tamaño igual a la cantidad de filas y columnas, con punteros al comienzo de cada fila o columna, respectivamente.

La memoria que se necesita para implementar la estrategia LLRCS, si nnz es la cantidad de coeficientes no nulos, se necesitan nnz posiciones de pun-

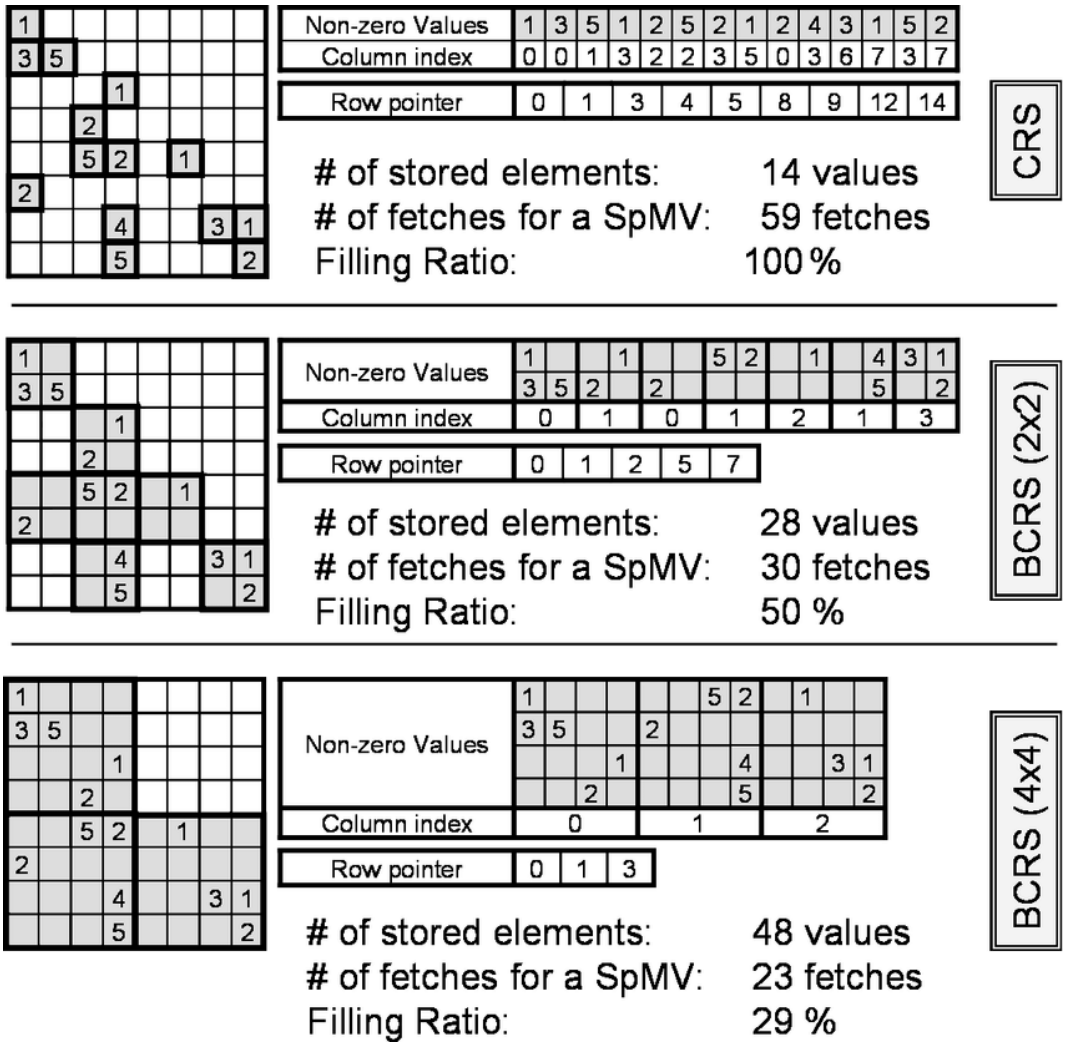


Figura 2.8: Ejemplo de matriz almacenada en formato BCRS aplicado con distintos tamaños de bloque y comparación con CRS. Extraído de [13].

to flotante, $4 \times nnz$ punteros, nnz enteros, más la memoria necesaria para almacenar los dos vectores ($2 \times n$ punteros).

Para acceder al valor $A(i, j)$ se puede recorrer la lista de la entrada i por el vector de filas buscando el valor cuya columna sea j, o recorrer la lista de la entrada j del vector de columnas hasta encontrar la fila i.

2.2.9. LLRS (Linked List Row Storage)

En esta estrategia se utiliza una estructura unidimensional, en la cual se emplea un vector de tamaño igual a la cantidad de filas, y de cada posición del vector se puede obtener la lista de entradas de esa fila.

En cuanto a memoria, se necesitan nnz posiciones de punto flotante, nnz enteros, nnz punteros, más el vector de entrada (n punteros).

2.2.10. LLCS (Linked List Column Storage)

Al igual que en la propuesta anterior, en la estrategia LLCS se utiliza una estructura unidimensional, pero el vector base de la estructura es de tamaño igual a la cantidad de columnas y de cada posición del vector se puede obtener la lista de coeficientes no nulos de esa columna. La estrategia LLCS posee las mismas necesidades de memoria que la LLRS.

2.3. Multiplicación Matriz-Vector (SpMV)

Si bien el objetivo del proyecto no se centra en estudiar las matrices dispersas para una operación particular, la importancia que implica la operatoria al trabajar con éstas obliga a mostrar algunas operaciones específicas. Por esta razón, en este apartado se presenta la multiplicación Matriz Dispersa-Vector (SpMV por su nombre en Inglés Sparse Matrix-Vector). La SpMV [31] es una operación de la forma $y = Ax$, donde A es una matriz dispersa de tamaño $m \times n$, los vectores x e y son densos de tamaño n y m respectivamente. Este kernel es altamente utilizado y con enormes aplicaciones en la computación científica. Dada su importancia, la implementación de esta operación ha sido fuertemente estudiada en variedad de contextos, desde diferentes formatos dispersos, hasta diversas arquitecturas de hardware, entre ellas: GPUs, FPGAs e incluso en dispositivos como TPUs [36, 43, 53].

Si bien las operaciones necesarias para su cómputo son simples (sumas y multiplicaciones), la SpMV, presenta ciertas limitantes a la hora de su ejecución. El cuello de botella se da en los accesos a memoria (principalmente por la aleatoriedad de los accesos). Este hecho ha motivado que muchos esfuerzos por optimizar la SpMV se centren en definir formatos de almacenamiento adecuados. Esto ha permitido que en muchas de las implementaciones de la operación SpMV, los accesos a la matriz A sean ordenados y bien aprovechados, obteniendo accesos predecibles y eficientes. Mientras que, en general, el vector x es accedido de forma irregular debido a la estructura de la matriz A, sacando poco beneficio de la localidad de datos y desaprovechando del uso de memorias cache.

A modo de ejemplo, en el Algoritmo 1, se presenta en alto nivel el procedimiento secuencial que computa SpMV, utilizando el formato CSR para almacenar la matriz A. El primer ciclo `for` (i) recorre las filas de la matriz A, y el segundo `for` interno (j) calcula, accediendo a los elementos no nulos almacenados en d a través de los punteros en f, para cada fila, la multiplicación por los coeficientes correspondientes del vector x (accedido en las posiciones donde existen elementos no nulos indicado por c) y se almacena el resultado en la posición del vector y pertinente.

Algoritmo 1: Cálculo secuencial de la multiplicación matriz dispersa-vector (SpMV) con la matriz dispersa A almacenada en el formato CSR. El vector d almacena los valores distintos de cero de A por filas, f contiene los índices que especifican el primer elemento de cada fila en el vector d; y c contiene el índice de columna de cada elemento en la matriz original. Los elementos distintos de cero dentro de cada fila están ordenados por índice de columna.

1 **Input:** f, c, d, x
2 **Output:** y
$y = 0$
for $i = 1$ **to** m **do**
  for $j = f[i]$ **to** $f[i+1] - 1$ **do**
    $y[i] = y[i] + d[j] \cdot x[c[j]]$
  end for
end for

2.4. Estrategias de reordenamiento

Como se menciona en las secciones anteriores, muchas de las matrices dispersas con las que se trabaja en la actualidad corresponden a discretizaciones de problemas de redes eléctricas y distribución de energía, ingeniería estructural, etc. y, más en general, formulaciones de sistemas de ecuaciones diferenciales parciales. Estos sistemas son aproximados mediante ecuaciones con una cantidad finita de incógnitas, y resultan en sistemas de la forma $Ax = b$, con A una matriz dispersa. Buscando explotar estas características, por muchos años, la estrategia más difundida fue intentar llevar las matrices a una matriz equivalente pero de banda. Notar que, una matriz con un ancho de banda pequeño es útil principalmente por dos razones. La primera es que permite utilizar una estructura de datos simple en métodos directos como la factorización LU para resolver sistemas lineales dispersos. Segundo, también es útil en métodos iterativos como el método del Gradiente Conjugado, porque los elementos distintos de cero serán agrupados cerca de la diagonal, mejorando así la localidad de los datos.

En este contexto se desarrollaron las técnicas de reordemaniento para las matrices dispersas. Reordenar es encontrar una permutación p que se aplica tanto para filas como para las columnas de la matriz. Este reordenamiento puede buscar diferentes objetivos. Dada una matriz simétrica A, un reordenamiento de reducción de ancho de banda apunta a encontrar una permutación P de modo que el ancho de banda de PAP^t sea pequeño (en el ideal, mínimo).

Notar que en los problemas que provienen de estructuras de grilla, reordenar la matriz es equivalente a renumerar los nodos de la grilla. Este aspecto también ha sido especialmente abordado en dicho campo de estudio (la generación de grillas de discretización).

Dado que el problema de encontrar el reordenamiento que minimice el ancho de banda para una matriz es un problema $\mathcal{NP}$-completo [50], se ha trabajado en estudiar e implementar múltiples heurísticas con el objetivo de encontrar buenas soluciones con esfuerzos computacionales razonables. Una familia importante de estos algoritmos, trata la reducción del ancho de banda de una matriz, como se dijo antes, bajo la perspectiva de un problema de etiquetado de grafos. De esta manera, el problema de reordenar una matriz dispersa A equivale al de etiquetar los vértices o nodos del grafo correspondiente a interpretar A como la matriz de adyacencia asociada. Esto es interpretar

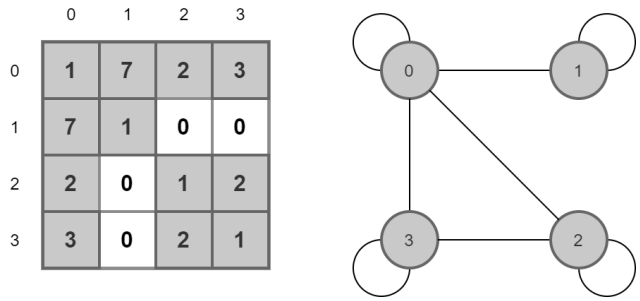


Figura 2.9: Ejemplo de grafo y su matriz de adyacencia correspondiente.

cada fila o columna i como un nodo de un grafo, y cada entrada no nula j de dicha fila como una arista que conecta el nodo i con el j, tal como muestra la Figura 2.9.

A continuación, son presentadas las heurísticas clásicas para la reducción del ancho de banda, Cuthill-McKee (CM) [19], Reverse Cuthill-McKee (RCM) [25] y Gibbs-Poole-Stockmeyer (GPS) [28]. Es preciso mencionar que, a pesar que estas herramientas funcionan relativamente bien, no hay que dejar de lado que son heurísticas, por lo tanto no son métodos que garanticen alcanzar la solución óptima, sino que una solución aceptable buscando un buen balance costo/beneficio.

2.4.1. Cuthill-McKee (CM)

El algoritmo de Cuthill-McKee [19], está basado en la aplicación de una estrategia de recorrida Breadth-First-Search (BFS), en la que el grafo asociado a la matriz es atravesado por niveles, determinados a partir del nodo raíz (nivel 0). A medida que se atraviesan los nodos de un nivel, a estos se los marca y numera en base al orden en que son recorridos. A continuación, se inspeccionan los vecinos de cada uno de estos nodos marcados, cada vez que se encuentra un vecino de un nodo visitado que no está numerado, se lo agrega a una lista y se etiqueta como el siguiente elemento del siguiente nivel. El orden en el que se atraviesa cada nivel da lugar a diferentes ordenamientos o permutaciones de filas y columnas. En el proceso de reordenar, en el algoritmo Cuthill-McKee, los nodos adyacentes a un nodo visitado siempre se atraviesan del grado más bajo al más alto.

Algoritmo 2: Cuthill-McKee. Esquema general en alto nivel del algoritmo Cuthill-McKee.

```
1 Input: Grafo G
2 Output: p
3 v_1 ← r; // Nodo inicial o raíz, en algunos casos el de menor grado -
    nodo periférico
4 for i = 1 to n do
5     Encontrar todos los nodos adyacentes de v_i sin numerar;
6     Etiquetar los nodos encontrados en orden creciente de grado;
7 end
```

2.4.2. Reverse Cuthill-McKee (RCM)

Variante del algoritmo CM, anteriormente descrito, en el que se "invierte" el orden obtenido para las filas/columnas. La heurística RCM produce como resultado matrices con el mismo ancho de banda que CM, pero con *profile* menor.

En la versión original de la heurística RCM propuesta por Alan George, también son seleccionados como vértices iniciales aquellos con grado mínimo en el grafo y se generan las estructuras de nivel desde estos vértices.

Las reducciones de ancho de banda y profile de la matriz resultante, obtenidas por las heurísticas CM y RCM, dependen fuertemente de la elección del vértice inicial. Por esto se han realizado varios estudios de como elegir el vértice inicial de manera de lograr optimizaciones en el proceso de encontrar ordenamientos aceptables.

2.4.3. Gibbs-Poole-Stockmeyer (GPS)

Antes de profundizar en esta heurística, es importante presentar algunas definiciones básicas útiles para la técnica, en especial considerando que los autores utilizan el concepto del grafo asociado a la matriz dispersa.

Definición (*distancia*). La distancia, función definida para dos vértices del grafo $d : V \times V \longrightarrow \mathbb{N}$. Sean u y v vértices, la distancia $d(u, v)$ está dada por el largo del camino mínimo entre éstos.

Definición (*excentricidad*). La excentricidad, definida sobre un vértice del grafo, $l : V \longrightarrow \mathbb{N}$, está dada por $l(v) = max_{v,u \in V}(d(v, u))$.

Definición (*diámetro del grafo*). El diámetro $\Phi(G) = max_{v \in V}(l(v)) = max_{v,u \in V}(d(u, v))$ del grafo $G(V, E)$ es la mayor excentricidad presente en G.

Gibs, Poole y Stockmeyer posteriormente del surgimiento de las heurísticas CM y RCM, verificaron que éstas heurísticas no eran adecuadas en los casos que la elección del vértice inicial implica un costo computacional grande. En dichas técnicas los vértices iniciales son los de menor grado en el grafo, y luego se generan las estructuras de nivel a partir de éstos. Notar, por ejemplo, que CM y RCM harían un cómputo innecesario en situaciones donde los vértices tienen todos el mismo grado.

Para comprender el algoritmo GPS, es necesario dar una definición de nodo o vértice pseudo-periférico:

Definición (*vértice periférico*). Un vértice v es considerado periférico si su excentricidad es igual al diámetro del grafo, i.e. $\mathbf{l}(v) = \Phi(G)$. Entonces, para Gibbs, Poole y Stockmeyer, un vértice es pseudo-periférico si su excentricidad es cercana al diámetro del grafo.

En GPS, en lugar de construir varias estructuras de nivel, como en las planteadas en CM y RCM, sólo se construyen dos (con nodos raíces dos vértices pseudoperiféricos). Gibbs, Poole y Stockmeyer, fueron los primeros autores en utilizar un vértice pseudo-periférico como inicial para la renumeración. Las bondades que ofrece elegir un vértice pseudo-periférico como vértice inicial para renumerar se dan debido a que en la estructura de nivel formada desde este vértice habrá mayor cantidad de niveles y, en consecuencia, menos ancho por nivel, comparado con un vértice que no está en la periferia o cerca de la periferia del grafo de adyacencia.

La heurística GPS se puede dividir principalmente en tres etapas. En el primer paso de la heurística GPS, se seleccionan dos nodos pseudo-periféricos, $v, u \in V$, y sus estructuras de nivel asociadas $L(v)$ y $L(u)$, respectivamente, tomando a estos como raíz. En la segunda etapa de la heurística GPS, en base a las estructuras anteriormente mencionadas en la etapa uno, $L(v)$ y $L(u)$, se crea una nueva estructura de nivel, $K(v, ...)$. Se combina las estructuras de nivel con el objetivo de reducir el ancho por nivel (que sería equivalente al ancho de banda). Este paso asegura que el ancho de nivel de la estructura $K(v, ...)$ sea, como máximo, el ancho de nivel más pequeño de las estructuras de nivel $L(v)$ y $L(u)$. Finalmente, en el último paso, se renumeran los vértices atravesando la estructura del nivel $K(v,)$ generada en la segunda etapa de la heurística GPS. Los niveles se recorren en orden inverso, es decir, desde el nivel $K_{l(v)}(v, ...)$ hasta el nivel $K_0(v, ...)$. Esta renumeración se realiza nivel a nivel de la estructura generada, en la que la renumeración de los vértices se realiza

en orden ascendente de grado.

2.4.4. Algoritmo Sloan (Sloan)

El algoritmo Sloan [56] es una heurística planteada por Scott W. Sloan en el año 1986 para reordenar matrices dispersas. Normalmente, se utiliza para acelerar el cálculo de sistemas de ecuaciones lineales dispersos como muchas de las otras heurísticas de reordenamientos.

La idea principal del algoritmo es numerar los vértices desde un punto cercano al diámetro del grafo, un pseudo-diámetro. En el primer paso, se busca un pseudo-diámetro del grafo y se eligen s y e como extremos de dicho camino. En el paso dos, a cada vértice del grafo se le asocia una cierta prioridad, en base al pseudo-diámetro anterior y al grado de éste, inmediatamente, el vértice inicial s se ordena primero. Luego, en cada etapa, se elige el siguiente vértice a numerar entre el conjunto factible con la mayor prioridad. Manteniendo así, un equilibrio entre los objetivos de mantener un *profile* pequeño y la incorporación de vértices elegibles que se han quedado atrás (lejos de e). La lista de vértices posibles para numerar en cada iteración, está formada por los vecinos de uno o más vértices ya renumerados y los vecinos de estos.

El algoritmo, puede entonces, dividirse principalmente en dos fases bien distintivas: (1) selección del vértice inicial s y final e, y (2) reordenamiento de vértices.

Para la prioridad de cada nodo, se utilizan dos pesos positivos W_1 y W_2 de forma de ponderar las dos propiedades que determinan la prioridad del nodo, en este caso son la distancia al nodo inicial y el grado, respectivamente. $P(n) = W_1 \times distance(n, s) - W_2 \times degree(n)$.

El Algoritmo 3 presenta un pseudocódigo de cómo funciona el algoritmo Sloan. En cada iteración, los nodos del grafo puede estar en uno de 4 estados: (i) *numerado*, (ii) *activo*, (iii) *preactivo*, para los nodos vecinos de algún nodo activo, y un cuarto estado (iv) *inactivo* para el resto de los nodos. Inicialmente, sólo el nodo inicial s está en estado *preactivo* mientras que el resto se encuentra en estado *inactivo*. Seguido, el algoritmo itera a través de todos los nodos del grafo, y en cada paso elige entre los nodos *activos* o *preactivos* en orden descendente de prioridad, es decir, maximizando en cada paso la prioridad. Posteriormente, se asignan nuevas prioridades a los vecinos y los vecinos de éstos son seleccionados para procesar.

Algoritmo 3: Algoritmo Sloan. Pseudocódigo para una posible implementación secuencial del algoritmo Sloan. Extraído de [61].

```
1 Input: Grafo G, Pesos W_1, W_2, Nodos s, e
2 Output: p
    nextId = 0;
    for each n Node in G do
      status(n) = inactive;
      priority(n) = W_1× distance(n,s) - W_2× degree(n);
    end for
    status(s) = preactive;
    working_queue = {s};
    for each n node in working_queue order by priority do
      for each v in Neighbors(n) do
        if status(n) == preactive and (status(v) == inactive or status(v)
        == preactive)) then
          update(priority(v));
          status(v) = preactive;
          updateFarNeighbors(v,working_queue);
        else if status(n) == preactive and status(v) == active) then
          update(priority(v));
        else if  status(n) == active status(v) == preactive) then
          update(priority(v));
          status(v) = active;
          updateFarNeighbors(v,working_queue);
        end if
        p[nextId++] = n;
        status(n) = numbered;
      end for
    end for
    for i = 1 to m do
      for j = f[i] to f[i + 1] − 1  do
        y[i] = y[i] + d[j] · x[c[j]]
      end for
    end for
```

Existen también, implementaciones en paralelo para este algoritmo, tal y como se plantea en [61].

2.5. Plataformas de hardware heterogéneas en HPC

La computación de alta performance (HPC), al día de hoy, es una de las herramientas más importantes para el avance de la computación científica. El desarrollo tecnológico ha motivado resolver problemas más grandes y/o con mayor precisión. Esto, a su vez, se retro-alimenta y genera aún mayores necesidades computacionales. En este contexto, la HPC aporta en soluciones tecnológicas que sirven de base para abordar este tipo de realidades. Entre otros campos, el uso de HPC ha permitido el desarrollo de áreas como el pronóstico del tiempo, la exploración de energía, la bioinformática [51], el aprendizaje automático, el análisis de big data [5] y el edge-computing [63].

Entre las arquitecturas de hardware utilizadas para la HPC se encuentran las unidades de procesamiento gráfico, o GPU (del inglés Graphics Processing Units). Son coprocesadores diseñados originalmente para alivianar la carga de la CPU en aplicaciones con una importante carga de trabajo dedicada a la representación de gráficos en la pantalla, como pueden ser videojuegos o simulaciones con un alto contenido de imágenes 3D, entre otras cosas [41]. En los últimos años, los avances tanto en el hardware como en distintos lenguajes de programación orientados a estas arquitecturas, que permiten, no sólo trabajar con procesamiento gráfico, sino que también con problemas de propósito general como simulaciones y modelos numéricos. Entre los lenguajes orientados a este tipo de arquitecturas se destaca `CUDA` (estándar de bibliotecas de la empresa `Nvidia`). Gracias a la programabilidad y flexibilidad de la arquitectura `CUDA`, las GPU han pasado a ser verdaderos procesadores masivamente paralelos, presentes en la mayoría de las plataformas de HPC modernas. Debido a la gran importancia que han tenido las GPUs en la computación científica, y en particular en el ALN, se dedica la Sección 2.5.1 al estudio de su arquitectura.

2.5.1. Arquitectura de las GPUs

En el año 1965 Gordon Moore planteaba la tan conocida Ley de Moore, en la que afirmaba, a grandes rasgos, que la cantidad de transistores dentro de los microprocesadores se duplicaría en el período de 2 años, fenómeno que se ha ido cumpliendo hasta el día de hoy. Si se observa la evolución de los procesadores CPU, durante el período comprendido entre los años 70-80 hasta el 2004, el

número de cores dentro del procesador no presentó cambios, en particular se mantuvo en uno. Esto se debe, principalmente, a que la arquitectura de la CPU estaba pensada para ser puramente secuencial, dando lugar a la pregunta de cómo se fue mejorando el rendimiento de los procesadores. Básicamente, la estrategia para mejorar el desempeño de los procesadores, fue aumentar la frecuencia a la que trabajaban, así como la cantidad de transistores en el chip. Observando la Figura 2.10, es posible notar que además de aumentar los transistores y la frecuencia a la que trabajan los procesadores aumenta, de forma conjunta, la potencia utilizada o requerida por los mismos. Esto se debe a que, físicamente, la potencia está directamente relacionada a la frecuencia así como el voltaje al que trabajan los procesadores, relaciones dadas por la siguientes ecuaciones de proporcionalidad:

$$P \propto CV^2 f,$$

donde C es la carga capacitiva, V el voltaje y f la frecuencia. Básicamente, a mayor frecuencia, dado que son voltajes bajos, la potencia queda en su mayoría determinada por la corriente que circula por el procesador, y a mayor cantidad de carga pasando por un conductor mayor es el calor.

Si bien no se muestra en la Figura 2.10, a medida que el consumo de potencia iba en aumento junto con la frecuencia, la primer estrategia para reducir el consumo energético de los procesadores fue reducir el voltaje. Se fue disminuyendo, pasando de manejar valores de voltaje en el rango de 5V a 1V. Esto permitió alcanzar frecuencias de alrededor de 3,6 GHz (sin disparar demasiado la potencia o con valores controlables) aumentando razonablemente el rendimiento de los procesadores. Llegado el punto, la reducción de voltaje ya no era viable (debido a que los 1s y 0s están representados por diferentes voltajes, seguir reduciendo provocaría que estos no puedan ser correctamente distinguidos) y seguir aumentando la frecuencia traía consigo el problema antes mencionado, consumo de potencia y calor disipado, encontrándose con el fenómeno denominado “The Power Wall”.

Debido a la barrera física con la que se enfrentan en caso de seguir con la misma estrategia, era necesaria la creación de sistemas más sofisticados de enfriamiento, como los que propuso `IBM` en su momento basados en enfriamiento liquido [23], lo que motivó la elección de una nueva dirección de desarrollo. Para mitigar y resolver este problema, en lugar de aumentar la frecuencia, se

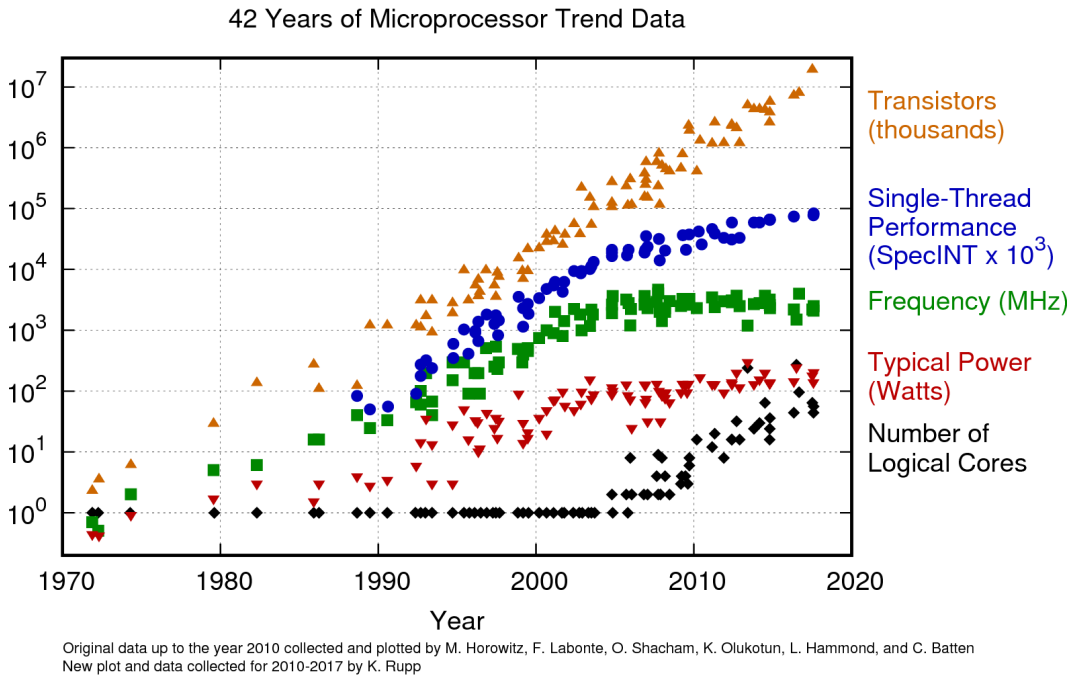


Figura 2.10: Evolución de los procesadores y sus principales factores. Extraído de [1].

adoptó la idea de aumentar la cantidad de cores dentro del procesador, pero funcionando a frecuencias más bajas. Estrategia mejor conocida como *multi-core*, que al día de hoy se mantiene con fuerza siendo así inspiración para las arquitecturas paralelas como la GPU, que explota esta idea en gran escala. En particular, las GPUs han ido agregando más cores (funcionando a frecuencias bajas), entre otros factores que han permitido el aumento de performance de estos dispositivos. Dicha evolución vuelve a estos dispositivos una arquitectura muy atractiva en ámbitos de la computación científica.

La GPU y la CPU presentan grandes diferencias a nivel de arquitectura, ya que en la primera, la gran mayoría de los transistores están dedicados al cómputo mientras que en la última están dedicados a intentar mejorar el tiempo de ejecución secuencial. En una CPU tradicional, gran parte de los transistores están destinados a realizar otro tipo de tareas que el cómputo, como por ejemplo:

- predicción de branches: Para ejecutar secuencialmente y explotar las técnicas de *pipelining*, es importante saber cuál instrucción es la siguiente a ejecutar, por lo que la CPU tiene mecanismos para intentar prever, por ejemplo, ante instrucciones condicionales, por cuál de estas ramifi-

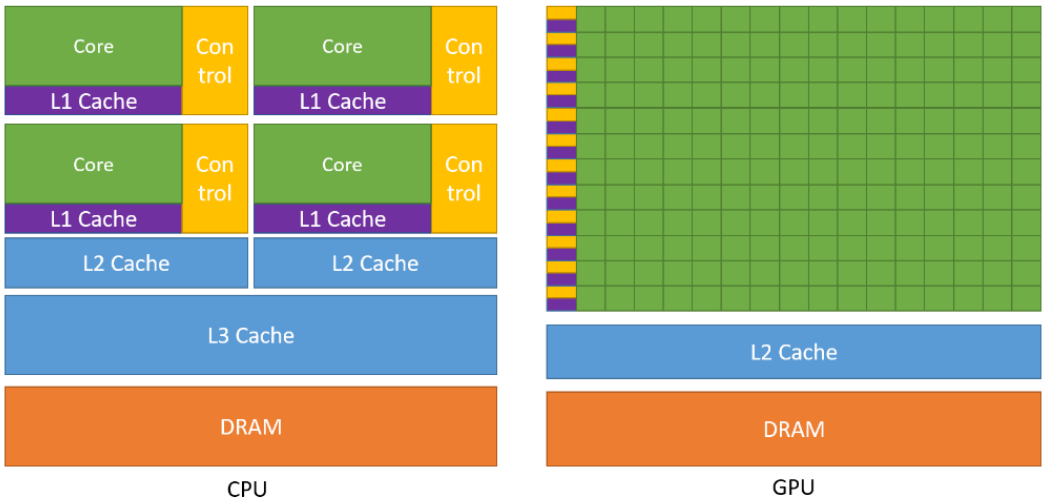


Figura 2.11: Diferencia de arquitectura entre una CPU y una GPU. Extraído de [17].

caciones podría seguir.

- prefetch de memoria: Cargar memoria, con datos que de antemano se sabe que la CPU va a necesitar o cree que puede llegar a utilizar, para no tener que esperar durante la ejecución la carga de esos datos.
- ejecución fuera de orden
- caché de datos

Entonces, la CPU se encarga de todos estos detalles, dedicando gran parte de sus recursos, dando la impresión al usuario de que el sistema funciona más rápido.

En cuanto a las GPUs, estos conceptos no están presentes, o aparecen en menor medida, dedicando la mayoría de los transistores al cálculo, agregando más unidades de cómputo. La Figura 2.11 muestra una comparativa aproximada de la proporción de transistores que está dedicado a cada componente dentro de una CPU y una GPU. Mientras que gran parte de la capacidad de las CPUs está dedicada a control, intentando mejorar el tiempo de ejecución secuencial, las GPUs enfocan estos transistores en agregar más unidades de cómputo, reduciendo las unidades de control.

Si se estudia la arquitectura de las GPUs, es necesario abordar también CUDA, término utilizado no solamente para describir la arquitectura de hardware presentada en 2006 por la compañía `NVIDIA`, sino también para referirse al modelo y lenguaje de programación que permite crear aplicaciones que eje-

cutan en estos dispositivos.

Desde el punto de vista del modelo de programación, la ejecución del programa se distribuye en hilos (*threads*), organizados en una grilla indexada llamada bloque (*threadblock*), y a su vez, estos bloques también forman una grilla indexada (*grid*). A priori, la ejecución de los hilos y bloques es totalmente independiente y puede darse en cualquier orden. El lenguaje de programación consiste en una extensión del lenguaje C, que permite, entre otras cosas, la creación de estas grillas de hilos, la programación de las funciones a ejecutar en el dispositivo (kernels) y las transferencias de datos entre la CPU y la GPU.

Por su parte, la arquitectura de hardware se forma entorno a una serie de multi-procesadores paralelos (Streaming Multiprocessors o SMs), cada uno formado por varios núcleos. La cantidad de procesadores y núcleos con los que cuenta la GPU varía según las diferentes generaciones de tarjetas. La memoria de las GPUs se organiza de forma jerárquica. En un primer nivel, existe una memoria global relativamente lenta pero accesible por todos los hilos. Se encuentra fuera de los multiprocesadores, por lo que el acceso a la misma implica una alta latencia, pero también posee un gran ancho de banda. Adicionalmente, a partir de la segunda generación de CUDA, se incluye una memoria caché de segundo nivel. Dentro de cada multiprocesador, existe una memoria de baja latencia pero de mucho menor tamaño con respecto a la global. Esta memoria es compartida físicamente por los bloques de hilos que residen en el multiprocesador. Por último, cada multiprocesador contiene un archivo de registros el cual es repartido entre todos los hilos residentes en el multiprocesador. Los registros son la memoria más rápida que brinda la GPU, pero también es la más reducida en tamaño.

Existe cierta correspondencia entre este modelo abstracto y la arquitectura de hardware. Siguiendo el espíritu de la clasificación de sistemas paralelos propuesta por Flynn en [24], NVIDIA clasifica su arquitectura como Single Instruction Multiple Thread o SIMT. A diferencia de la categoría SIMD, en la cual una misma instrucción se ejecuta simultáneamente sobre distintos elementos de un conjunto de datos, en SIMT el usuario puede especificar distintos flujos de ejecución para los distintos hilos, aunque dadas las características del hardware, el desempeño es mucho mayor cuando el comportamiento de la aplicación se asemeja al tipo SIMD.

Comenzada la ejecución del programa, cada multiprocesador se encarga de la ejecución concurrente de un grupo fijo de bloques, como se muestra en la

Figura 2.12. Los hilos pertenecientes a estos bloques son divididos, planificados y ejecutados en grupos de 32 hilos llamados warps. La división se realiza siempre de forma que los hilos con índice 0 a 31 forman el primer warp, los de índice 32 a 63 al segundo, y así sucesivamente. La ejecución se organiza de forma que los threads de un mismo warp deben ejecutar la misma instrucción en cada momento. Dado el caso, si en el flujo de ejecución de distintos hilos del mismo warp, dos hilos divergen, debiendo ejecutar distintas instrucciones, las mismas se ejecutan de forma secuencial y los hilos que ejecutan una de las instrucciones quedan inactivos hasta que el resto de los hilos del warp ejecuten la otra instrucción. Por esta razón, la máxima eficiencia es alcanzada cuando todos los hilos de un warp ejecutan la misma instrucción en todo momento, aunque la misma se ejecute sobre distintos elementos del conjunto de datos. La asignación de recursos a cada warp dentro de un multiprocesador se realiza de forma estática, asignando un segmento del archivo de registros (*register file*) a cada warp, y asignando una sección de la memoria compartida del multiprocesador a cada bloque que ejecuta en él. De esta forma, el cambio de contexto entre un warp y otro se realiza sin costo. Otro aspecto muy importante son los accesos a memoria. No sólo es una mejora importante que cada hilo de un mismo warp ejecute la misma instrucción, sino que es importante también que ejecuten utilizando datos que se encuentren en espacios de memoria contiguos, produciendo accesos *coalesced*. El acceso a memoria *coalesced*, refiere a combinar múltiples accesos a la memoria en una sola transacción.

Junto con CUDA, NVIDIA provee un conjunto de herramientas que continúan en desarrollo gracias a los aportes de investigadores, las cuales tienen como objetivo complementar la arquitectura mejorando su usabilidad en distintas áreas de aplicación. Entre estas herramientas se encuentran lenguajes como `CUDA FORTRAN`, `PyCUDA`, APIs como `OpenACC` o `PGI Accelerator Compiler`, herramientas de análisis, debugging, y un gran conjunto de bibliotecas optimizadas en GPU, como por ejemplo `cuBLAS` [16] y `cuSPARSE` [18].

2.5.2. Algoritmos en HPC

En general, el cuello de botella de la mayoría de las operaciones y algoritmos que se ejecutan en arquitecturas de HPC al día de hoy, se da en los accesos a memoria, debido principalmente a la alta latencia de los mismos en relación a la de las operaciones aritméticas (operaciones en punto flotante FLOPS).

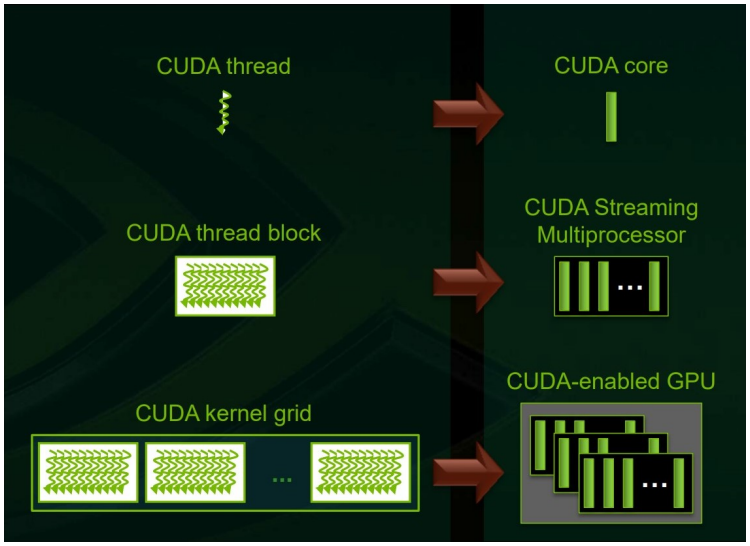


Figura 2.12: Mapeo entre estructuras CUDA y GPU. Extraído de [17].

Integer	
Add	
8 bit	0.03pJ
32 bit	0.1pJ
Mult	
8 bit	0.2pJ
32 bit	3 pJ

FP	
FAdd	
16 bit	0.4pJ
32 bit	0.9pJ
FMult	
16 bit	1pJ
32 bit	4pJ

Memory	
Cache	(64bit)
8KB	10pJ
32KB	20pJ
1MB	100pJ
DRAM	1.3-2.6nJ

Figura 2.13: Valores aproximados de energía consumida por operación (45nm). Extraído de [38].

Muchos cálculos científicos sólo aprovechan una fracción de la potencia computacional en las arquitecturas de alto rendimiento actuales. La dificultad radica, en muchos casos, en mapear dichas operaciones a las arquitecturas, intentando explotar al máximo los atributos de cómputo que presentan haciendo uso de, por ejemplo, jerarquías de memoria con el objetivo de mitigar la baja latencia de las memorias principales.

Al mismo tiempo, las operaciones de memoria son el principal consumidor de energía de las arquitecturas modernas, lo que afecta en gran medida el costo de los recursos, como se puede observar en la Figura 2.13. Un estudio detallado de este y otros problemas es el que plantea Horowitz en [38].

El ALN no escapa de este paradigma, en particular, las operaciones con matrices dispersas encajan perfectamente en esta categoría de problemas. En-

tre los varios problemas del ALN dispersa, la SpMV es un claro ejemplo de operación con un costo computacional razonable a nivel de operaciones, y que puede ser atacada de forma paralela, pero que se ve fuertemente limitada por el ancho de banda entre procesador y memoria, además su baja intensidad computacional, es decir, la baja cantidad de operaciones aritméticas realizadas por cada acceso a memoria, sumada a la aleatoriedad de los accesos que implica un pobre aprovechamiento de los caches, produciendo un ratio bajo de lecturas efectivas.

En este contexto, es de gran interés estudiar formatos "óptimos" para matrices dispersas, que logren reducir la cantidad de transacciones de memoria o, al menos, ofrezcan mejoras mediante el acceso de forma ordenada, aprovechando así en lo posible el ancho de banda limitado.

Capítulo 3

Revisión del estado del arte

Como se mencionó anteriormente el uso de matrices dispersas tiene múltiples aplicaciones en el ámbito de la ciencia y la ingeniería, desde simulaciones de circuitos electrónicos [20], pasando por problemas de optimización y hasta operaciones con grafos de redes sociales [14]. En este contexto, también tratado brevemente en la Sección 2.3, una de las principales operaciones involucradas es la Multiplicación Matriz dispersa-Vector (SpMV). Esto ha motivado que en los últimos 40 años muchos trabajos busquen optimizar la SpMV para poder atacar de forma más eficiente los problemas.

En la literatura se encuentran varios trabajos centrados en el diseño de formatos de almacenamiento que buscan optimizar diferentes aspectos de la SpMV[1]. Algunos de estos esfuerzos avanzaron en la implementación de formatos híbridos en [46], mientras que otros se concentraron en reducir la precisión de los números utilizados [65], buscando aprovechar mejor el ancho de banda de acceso a memoria. Las motivaciones para utilizar formatos novedosos son diversas. Mientras que, en ocasiones, se busca acotar el espacio de almacenamiento, implicando una posible reducción en los accesos a memoria, en otros casos el objetivo es mejorar alguna característica del cómputo de una operación específica, incluso para sacar partido de las características de determinado hardware. Por ejemplo, si se consideran los formatos de almacenamiento clásicos como COO y ELLPACK, que se utilizan para representar matrices dispersas, estructuras de datos como estas suelen emplear matrices con índices para el acceso indirecto a los vectores densos de entrada. Para cada operación de suma o multiplicación, ELLPACK requiere tres accesos a la memoria, mien-

[1] Aunque el proyecto no se centra en el estudio de formatos para una operación específica, la concentración de esfuerzos en la operación SpMV implica un lógico destaque.

tras que para COO, son necesarios cuatro accesos. Debido a la baja relación entre las operaciones de punto flotante y el número de accesos a la memoria, el rendimiento de la SpMV generalmente está limitado por el ancho de banda de memoria disponible en la arquitectura subyacente.

A continuación se presentan, de forma breve, las ideas abordadas en diferentes investigaciones que intentaron avanzar en las capacidades de los formatos dispersos. Si bien, en muchos casos, todos los conceptos abordados en este capítulo están fuertemente relacionados, se decidió clasificar los esfuerzos bajo las siguientes categorías: formato de almacenamiento por bloques, formatos híbridos, uso de múltiples precisiones y técnicas de reordenamiento, únicamente con la intención de organizar el estudio.

3.1. Estrategias de almacenamiento por bloques

Belgin et al. [8] presentan un enfoque llamado *Pattern-based Representation* (PBR), basado en identificar patrones de bloques, es decir, elementos no nulos agrupados en una zona pequeña de la matriz y reemplazar los índices de los elementos no nulos por máscaras en forma de bitmaps para reducir la sobrecarga del ancho de banda producida por los accesos a los índices para la mayoría de las matrices dispersas. Esta reducción se da debido a que, en lugar de un índice por elemento no nulo de la matriz, las representaciones por bloques emplean un índice por dicha subestructura. Sin embargo, la utilización de estrategias por bloques puede requerir un llenado explícito de elementos nulos (también llamado *zero-filling*), lo que puede aumentar los requerimientos de memoria y las operaciones en punto flotante. Por este motivo, este tipo de técnicas suelen ser útiles únicamente cuando la matriz presenta cierta estructura de bloques densos. PBR explota un análisis simple que identifica estructuras de bloques recurrentes que comparten el mismo patrón de coeficientes no nulos dentro de una matriz. Para cualquier patrón que cubra más que un número umbral de no nulos, PBR representa la submatriz formada por este patrón en el formato (BCOO), junto con una máscara de bits que describe el patrón repetido. Para la identificación de los patrones repetidos, se utiliza una estrategia de análisis simple. Dado un bloque de tamaño $R \times C$, se divide la matriz de dimensiones $m \times n$ en una grilla de $\lceil \frac{m}{R} \rceil \times \lceil \frac{n}{C} \rceil$ bloques rectangulares, contando cuán

frecuente es cada una de las combinaciones, entre las $2^{R \times C}$ posibles. Luego se representa la submatriz correspondiente a cada bloque con un patrón, registrando las coordenadas del bloque en formato COO, junto con un "bloque código", que es un vector de bits de tamaño $R \times C$ que codifica el patrón de los elementos no nulos. Los autores dan dos variantes para la implementación de la SpMV utilizando PBR: una secuencial y la otra paralela. En la estrategia paralela, se divide la matriz en particiones, asignando una a cada hilo. Además, cada hilo mantiene un vector y_i que representa el producto de $A_i x$ correspondiente a la submatriz que tiene asignada, realizando la adición de los distintos y_i mediante una reducción paralela al final. En cuanto a la evaluación, los autores reportan una reducción en el tiempo de ejecución al realizar operaciones como SpMV basadas en PBR, tanto en secuencial como en paralelo. Quizás como desventaja se podría destacar que PBR puede interferir en el principio de localidad de los accesos al vector x. Como ventaja, PBR no realiza asunciones sobre el patrón y la estructura de la matriz a la hora de identificar los bloques por patrón.

En otra investigación, Choi et al. [15] proponen técnicas de auto-ajuste para los parámetros como, por ejemplo, el tamaño de bloques en el formato BCSR basados en modelos del desempeño de la SpMV en GPUs. Posteriormente, presentan una implementación de un nuevo formato de compresión para matrices dispersas que denominaron *blocked* ELLPACK (BELLPACK) que combina las ventajas de sub-bloques densos de BCSR y la comodidad de los accesos por vector de ELLPACK. En sus primeros resultados observaron que si bien BCSR presentaba mejoras con respecto a CSR, igualmente no competía con la mejor implementación de la biblioteca cuSparse de NVIDIA, que en la mayoría de casos es HYB.

De forma similar a [15], Yan et al. [66] proponen basados en bloques, con el objetivo de optimizar la SpMV, una extensión de un formato clásico, en este caso COO. Los coeficientes, agrupados en bloques, son almacenados con un índice de columna y de fila, de modo de reducir la sobrecarga producida por los accesos a índices distintos, como a direcciones no contiguas. En esta investigación presentan dos variantes para COO, la primera, denominada blocked compressed common coordinate (BCCOO). Utiliza bits flags para reducir la sobrecarga de los accesos por índice de fila. Indicando con un bit en 1 el inicio de una fila de bloques.

Luego, para mejorar la tasa de aciertos de la caché para acceder al vector

multiplicado, proponen la segunda estrategia de almacenamiento, en la que se divide la matriz dispersa en slices verticales y estos son alineados antes de aplicar el formato BCCOO. Dicha variante con en particiones verticales, se conoce como formato BCCOO+.

3.2. Formatos híbridos

En esta sección se analizan propuestas de formatos dispersos basados principalmente en combinar formatos existentes. Notar que si la matriz A puede ser expresada, por ejemplo, como $A = A_\alpha + A_\beta$, entonces $Ax = A_\alpha \cdot x + A_\beta \cdot x$. Esta descomposición permite almacenar ambos sumandos en un formato distinto con el objetivo de optimizar el desempeño de las operaciones.

Quizás una de las primeras ideas de utilizar estrategias híbridas se puede encontrar en las técnicas que guardan la diagonal de la matriz en un vector separado del resto de la matriz. Esta estrategia es especialmente útil cuando se aplican precondicionadores sobre la diagonal en los métodos iterativos de resolución de sistemas lineales. Similar a esta idea, los autores Sun et al. presentan, en su investigación [59], el formato Compressed Row Segment with Diagonal-pattern (CRSD). En su planteo, centrado principalmente en matrices con patrones diagonales, agrupando o segmentando por filas, proponen almacenar las componentes diagonales en vectores cuyo índice corresponde al offset con respecto a la diagonal. Si la diagonal a almacenar se encuentra por encima de la principal tendrá un offset positivo, negativo cuando está por debajo. Excepto por los elementos más dispersos de la matriz, fuera de diagonales

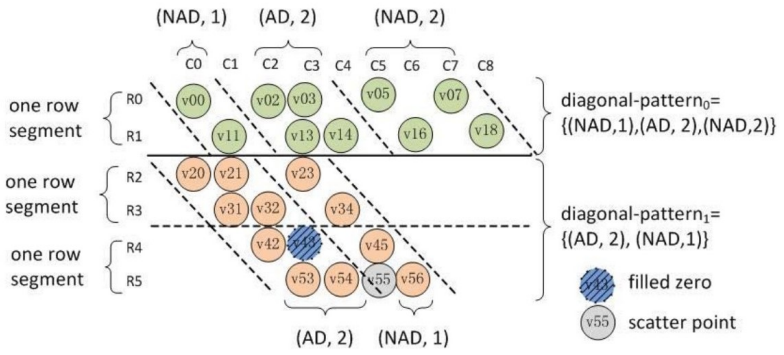


Figura 3.1: Sección de matriz diagonal, aplicando conceptos del CRSD. Extraído de [59].

densas, la matriz es almacenada a través de patrones diagonales. Aquellas filas que poseen estos elemetos dispersos, o como los denominan los autores scatter point, aquellos que están presentes en una sola diagonal, son agrupadas y almacenadas en forma completa con el formato ELL, debido a que de no utilizar otro formato, sería necesario realizar *zero padding* agregando overhead al procesamiento y trabajo con la matriz. Esta idea puede ser observada mejor en la Figura 3.1.

Otro ejemplo puede ser el formato HYB que combina los formatos ELL y COO, propuesto por Bell y Garland [10]. Este formato busca mitigar ciertas debilidades del esquema ELL, que si bien ofrece ventajas en cuanto a la localidad de datos, es poco eficiente en los casos donde la cantidad de elementos no nulos en cada fila varía considerablemente. En tales circunstancias, hay un aumento significativo del espacio total requerido (debido al padding necesario para completar los vectores de cada fila con menos de k elementos no nulos, donde k es la cantidad máxima de elementos en una fila de la matriz). Notar además, que los algoritmos operan con estos valores agregados aún siendo nulos. Para trabajar con estas matrices de forma más eficiente, una opción es utilizar un formato híbrido, que divida la matriz en dos, una componente en formato ELL y la otra por ejemplo en COO. La idea es tener una matriz A_{ELL} de tamaño $n \times k$ donde la cantidad de elementos por fila se aproxime bastante a k y otra matriz A_{COO} para el resto de los elementos. Elegir la columna k que determine la partición, se puede resolver, como en la implementación de la biblioteca *CUSP* [9], con una heurística donde dicho valor se elige de forma de que el número de filas con al menos k elementos no nulos, sea menos de un tercio de la cantidad total de filas de la matriz.

Otra investigación donde se combinan dos formatos de almacenamiento de matrices dispersas, enfocadas en mejorar el desempeño de la SpMV en GPUs, se puede encontrar en [35], en la cual Guo et al. presentan ELL and Vectored CSR Hybrid (EVC-HYB). El formato presentado se construye primero aplicando reordenamientos simples entre filas basado en los largos de éstas (nnz por fila), de menor a mayor, y seguido, se realiza una partición en dos grupos, filas largas y cortas. Centrado en esta partición, las entradas de la matriz son almacenadas en los formatos ELL o VCSR[1] según corresponda. Ambos formatos funcionan bien para ciertos tipos de clases de matrices, por ejemplo, ELL

[1]Vectorized CSR, implementación vectorizada de la SpMV optimizada para GPUs utilizando el formato CSR.

funciona bien con las matrices cuya cantidad de elementos no nulos por fila es baja y similares entre sí, debido a la localidad de los accesos, en cambio, es muy ineficiente cuando esta cantidad varía considerablemente entre las filas. Mientras que VCSR funciona mejor con matrices cuyas filas son de cierto largo suficiente y en lo posible múltiplos de 32 (warp en GPU), pero la propiedad de los accesos coalesced no puede ser explotada de forma óptima si las filas son de largos menores, particularmente menores a 16. Entonces, buscando aprovechar los beneficios que estos formatos presentan y evitar las debilidades, los autores plantean la estrategia híbrida EVC-HYB que combina ambos. Para la evaluación, en el estudio se comparan contra las mejores elecciones de CUSP para la SpMV, que en general son, CSR y HYB. Evaluaron, para 22 matrices distintas tomadas de la SSMC, la cantidad de operaciones punto flotante por segundo (FLOPS) en 500 iteraciones utilizando los diferentes formatos, obteniendo sin aplicar reordenamientos, mejoras en velocidad de hasta $1,64$, también con resultados que degradaron la cantidad de operaciones, que en el peor de los casos se registraron reducciones en un factor de $0,90$.

3.3. Múltiples precisiones

Operaciones como la SpMV, limitadas por el acceso a memoria [32], pueden obtener un beneficio inmediato si se logra reducir el uso de memoria en el almacenamiento. El simple hecho de sólo indexar los datos asociados a los elementos no nulos supone una oportunidad para la compresión de estos datos. La mayoría de los formatos dispersos, tienen cierto grado de compresión en los índices. Por ejemplo: los formatos basados en ELL (ver Sección 2.2.5) no almacenan explícitamente un índice para las filas, sino que este se infiere por la posición en la memoria. Otros formatos en bloque guardan un índice asociado a cada sub-bloque denso (en lugar de almacenar un índice por cada valor no nulo, reduciendo los datos de indexación) y luego con un desplazamiento acceden a los valores dentro de éste.

El uso de múltiples precisiones es una idea muy empleada en la historia del ALN. En general, centrado en la precisión de los coeficientes de las matrices, tanto por motivos de almacenamiento (por ejemplo trabajar en simple precisión reduce a la mitad el almacenamiento) como de cómputo (en una CPU la relación de desempeño entre simple precisión y doble precisión es 2 a 1 pero en otras plataformas de hardware, como por ejemplo algunas GPUs, las

diferencias suelen ser mucho mayores). En el caso del ALN disperso, es aún mayor la motivación para disminuir la precisión, dado que la principal restricción para el desempeño son los accesos a memoria, por lo que trabajar con precisiones menos demandantes disminuye la cantidad de datos que se mueven por la jerarquía de memorias.

En los últimos años, en el ALN dispersa, han aparecido varios esfuerzos por trabajar con precisiones reducidas o modificaciones de los formatos utilizados. Algunos ejemplos de estos esfuerzos son [3, 30], donde se evalúa cómo el uso de precisiones reducidas (como half y single) para almacenar algunos coeficientes de los precondicionardores obtenidos con el método de Jacobi, mejora el desempeño al utilizarlos en métodos iterativos para resolver sistemas de ecuaciones lineales. Específicamente, estos formatos buscan reducir el overhead producido por la transferencia de datos, almacenando adaptativamente los bloques diagonales del precondicionador de Jacobi en distintas precisiones. En [33, 34] se aplican ideas similares, pero desacoplando el formato de punto flotante utilizado para operaciones aritméticas del formato utilizado para almacenar los datos en la memoria.

Un enfoque complementario, es reducir la precisión de los índices asociados a los coeficientes. En este sentido, en el trabajo de Shiming Xu et al. [65] se propone una optimización de la SpMV, tomando como base el formato ELL, cuyo objetivo es disminuir la cantidad de bits necesarios para representar los índices. En este sentido, se estudia la posibilidad de utilizar como índice de la columna la distancia a la diagonal en lugar del valor real de la coordenada atacando, en este caso, un conjunto acotado de matrices cuadradas de tamaño $n \times n$, donde se busca reducir la distancia de los elementos no-nulos a la diagonal a través de reordenamientos o permutaciones como el método RCM (Reverse Cuthill-McKee). Este trabajo será discutido y analizado con mayor profundidad en la Sección 3.4.

Otra idea, es la planteada en el formato CoAdELL [46], donde se continúa la investigación realizada por los mismos autores [45], centrada en la división por warps de los cómputos con matrices almacenadas en formatos basados en ELL. Esta mejora consiste en una técnica de compresión para reducir el almacenamiento asociado a los índices de columna. La idea es utilizar una codificación basada en la diferencia entre los índices de dos elementos no nulos consecutivos en una misma fila, técnica que la mayoría de autores denomina *delta encoding* o *delta compression*, también utilizada con frecuencia en otros

campos de estudio [44]. Como estas distancias o deltas tendrán valores menores que los índices, se podrán representar con menor cantidad de bits. Por ejemplo, la secuencia 1, 2, 3, 4, 10, de índices de columna, pasa a valer 1, 1, 1, 1, 6. Se intenta entonces, estudiar la posibilidad de una compresión basada en la reducción de precisión de los nuevos coeficientes obtenidos aplicando esta técnica de codificación.

En otro trabajo, Kourtis et al. [42] emplean de manera similar la compresión de datos de índice, esta vez buscando reducir la sobrecarga producida por los mecanismos de descompresión debido a las predicciones erróneas de los flujos de cómputo de la operación. En particular, proponen dos métodos distintos apuntando a comprimir tanto índices como coeficientes no nulos utilizando el formato CSR como base para la investigación, los autores plantean mecanismos para la compresión y descompresión con el objetivo de optimizar la SpMV, atacando primero los índices de columna, estrategia que denominaron CSR Delta Unit (CSR-DU) y luego los coeficientes no nulos con otro formato que llamaron CSR Value Indexed (CSR-VI). Intentando explotar la distribución por columnas de los elementos no nulos, como se ha estudiado anteriormente utilizan la codificación delta, calculada como la diferencia de índice con respecto al elemento anterior, agregando a esta estrategia la idea de división o agrupamiento por unidades con largos variables, dónde cada una se representa con la cantidad de bits necesaria para el máximo valor de la unidad. Esta compresión trae consigo un cierto overhead en la etapa interna de la SpMV producido por la decodificación dependiendo de la cantidad de bits utilizada. Si los coeficientes fueran comprimidos todos por separado produciría, casi con total seguridad, ramificaciones en la multiplicación, degradando el rendimiento de toda la operación. Entonces, cabe destacar que la decisión de cómo es conformada cada unidad, es decir su tamaño, es de gran impacto en la estrategia. Por ejemplo, si las unidades son muy pequeñas la sobrecarga generada por las divergencias en la descompresión superará a las ganancias que se puedan obtener al comprimir. En cambio, si el tamaño indicado es muy grande, es probable que se encuentren menos oportunidades de compresión debido a que un delta grande impactaría en la cantidad de bits del resto de los índices de la unidad que podrían admitir precisiones menores.

Generalmente, la carga de trabajo de los distintos métodos se concentra en el procesamiento de los valores en punto flotante, ya que normalmente estos se almacenan en doble precisión utilizando 64 bits. Sin embargo, pese a que

la ganancia al aplicar compresión es potencialmente mayor, la compresión de valores en punto flotante no es tan sencilla como la de los números enteros (que permiten en algunos casos reducciones en la cantidad de bits si se conoce los rangos en los que estos trabajan), porque las operaciones aritméticas de punto flotante producen resultados redondeados, y reducir entonces la cantidad de bits implica posiblemente una pérdida de precisión. Pese a esto, los autores indican que existe un número significativo de matrices del conjunto experimental elegido en las que sólo una pequeña parte de sus valores son únicos. Esta redundancia puede ser explotada entonces, almacenando de forma única los valores comunes o repetidos y sus correspondientes referencias o punteros a su ubicación en la matriz, lo que conducirá a la reducción del conjunto de trabajo si la cantidad de valores redundantes es alta en relación al total. En consecuencia, una reducción considerable en este aspecto, derivará en una mejora del rendimiento en caso de poder compensar el costo derivado de la indirección en el acceso a los valores redundantes.

Las ganancias en desempeño a la hora de utilizar CSR-DU, según los autores, dependen del porcentaje del tamaño de información de índices sobre el tamaño total de la matriz. En el caso de matrices indexadas con 32-bits que contienen los valores numéricos en 64-bits, el porcentaje es cercano a 1/3, por lo que la ganancia está limitada por este factor.

Como conclusión general, una técnica de compresión puede ser beneficiosa para la SpMV siempre y cuando el método de descompresión no atosigue al procesador con ramificaciones adicionales irregulares y difíciles de predecir. Como se muestra en [32], el kernel SpMV es muy sensible a operaciones extra, así como predicciones erróneas en branches pueden fácilmente dañar el desempeño.

Tang et al. [60] proponen utilizar una familia de esquemas de compresión eficientes, que denominan *bit-representation optimized* (BRO), para reducir la cantidad de bits requerida para representar los índices. En particular, para el diseño de los esquemas de almacenamiento BRO, los autores tomaron en consideración aspectos importantes relacionados con las arquitecturas para las que fueron diseñadas las estrategias. Sin entrar en detalles de implementación, los autores se plantean estudiar el impacto de las etapas necesarias para aplicar estas técnicas. Por ejemplo, para poder realizar la descompresión en GPU, ésta debe ser relativamente liviana en comparación con las operaciones de suma y multiplicación de la SpMV, de forma que la mayoría de los ciclos de la GPU

se asignen a un trabajo útil y no se utilicen únicamente para descomprimir los datos del índice. Además, debido a que las GPU no contienen hardware complejo para la predicción de branches, la descompresión debe evitar los costosas penalizaciones por divergencia dentro de cada warp. En este sentido se presentan dos técnicas de compresión de índices, BRO-ELL y BRO-COO, basadas en los formatos ya estudiados ELL y COO, comprimiendo los índices de columna utilizando una codificación delta. La diferencia con otros autores que han trabajado con técnicas similares es que proponen una implementación escalable en GPUs. Por ejemplo, para el caso de BRO-ELL, una vez transformados los vectores de índices de columna en vectores con los delta, sugieren una división de cada uno de éstos en segmentos (llamados slices) de altura h. Posteriormente, cada slice es comprimido por un hilo independiente de acuerdo al número de bits necesarios para cada índice delta. Para hacer coincidir el modelo de ejecución SIMT de la GPU y garantizar que el acceso a los datos sea coalesced, se asigna un número fijo de bits para cada columna en un slice con el que se almacenan todos los valores delta de esa columna. Esto requiere encontrar el número máximo de bits necesarios para representar los valores en cada columna, así como estructuras adicionales para almacenar estos valores y la cantidad de columnas en cada slice.

Además de BRO-COO y BRO-ELL, presentan también BRO-HYB, útil en los casos cuando la cantidad de elementos no nulos por filas varía sustancialmente. Este formato es análogo a HYB, almacenando la componente regular en BRO-ELL y la irregular BRO-COO.

Willcock y Lumsdaine [64] desarrollaron dos métodos de compresión sin pérdidas para el formato CSR con el objetivo de reducir el ancho de banda de memoria requerido en la operatoria con matrices dispersas de grandes dimensiones. Ambos esquemas de compresión constan de dos etapas: comprimir los índices de una matriz utilizando una cantidad menor de bits antes de almacenar la matriz en la memoria, y descomprimir estos índices sobre la marcha como parte de la SpMV.

En el primer método, *Delta-Coded Sparse Row* (DCSR) se plantea un esquema de compresión basado en la codificación delta de los índices, codificando los índices como las diferencias entre las posiciones de columna de elementos distintos de cero en una fila, utilizando la cantidad mínima de bytes posible. Para esto se emplea un conjunto de seis códigos de comando para codificar los datos del índice.

El segundo método, *Row Pattern Compressed Sparse Row* (RPCSR), es un enfoque adaptativo que requiere más tiempo de compresión pero cuyo kernel de SpMV presenta un mejor rendimiento. Se basa en fusionar grupos de listas de intervalos de valores delta de los datos de índice. Se lograron aceleraciones de hasta un 30 % respecto a CSR, utilizando el método adaptativo. Ninguno de los dos esquemas de compresión presentados cambia los valores numéricos almacenados en la matriz, almacenándolos exactamente en el mismo orden y con la misma precisión que en formato CSR. La investigación está enmarcada en el caso de que la misma matriz se multiplicará repetidamente por muchos vectores, como es el caso de los solvers iterativos de sistemas de ecuaciones o valores propios, buscando que el tiempo ahorrado por multiplicaciones más rápidas puede compensar el tiempo de compresión.

3.4. Reordenamiento

En la Sección 3.3 se describieron varias estrategias que proponen reducir la precisión con las que trabajan las matrices dispersas con el fin de mejorar la velocidad de los accesos a memoria. Muchas de estas se basan en técnicas como *delta encoding*, que se ve beneficiada cuando los coeficientes no-nulos de la matriz se encuentran en posiciones cercanas entre sí, como sucede en matrices con ancho de banda pequeño. A continuación, se discutirán en más detalle aquellos que proponen aplicar reordenamientos para reducir ancho de banda, mejorar ciertos formatos de almacenamiento, u optimizar de cierta forma la compresión de matrices.

En [65], enfocados en la SpMV, los autores utilizan RCM como método de optimización para la reducción del ancho de banda, mostrando como la permutación obtenida por el método RCM puede mejorar la localidad de los accesos al vector x, además de permitir cierta compresión de la información de columna. Dado que x está, en general, expresado como un vector denso y de sólo lectura (read-only), se utiliza el *Texture Cache* (TC) de la GPU para almacenarlo en una memoria de rápido acceso, técnica también utilizada en [10, 12, 15]. En el caso del formato ELLPACK, dado que el desplazamiento de los accesos al vector x a la hora de computar la SpMV están indicados por los índices de columna de cada elemento no nulo, cuando se usa el TC para almacenar x, es importante que el patrón de acceso en x tenga las siguientes características de localidad: (1) las direcciones accedidas por hilos (o warps)

que se ejecutan cercanos en el tiempo sean cercanos en memoria, y (2) direcciones accedidas por un hilo en iteraciones consecutivas sean cercanas. El primer punto, requiere que los hilos dentro del mismo ThreadBlock (TB) posean direcciones similares, es decir, las filas adyacentes deben tener patrones de dispersión similares. El segundo requiere que los elementos no nulos en una fila deben estar lo más juntos posibles, es decir, agrupados en ciertas posiciones. Esto implica que permutaciones matriciales que mejoran las estructuras locales y generan bloques más densos pueden resultar en un mejor aprovechamiento del principio de localidad a la hora de acceder a x. Los autores utilizan el método RCM para mejorar la localidad. Para matrices de ancho de banda reducido luego de aplicar RCM, existe un buen límite superior para las regiones a las que accede cada TB: T + BW, donde T es el número de hilos por TB y BW es el ancho de banda de la matriz. En caso de que el ancho de banda no se reduzca de manera efectiva, RCM también tiende a generar filas con patrones de dispersión similares agregando elementos no nulos al perfil exterior de la matriz, debido a que la heurística inmediatamente luego de procesar un vértice procesa los vecinos, provocando que estas aristas aumenten el ancho de banda, pero sobre la misma columna. Esto corresponde a una mejora de la localidad espacial para threads adyacentes.

La reducción del ancho de banda de la matriz permite también la compresión de los índices para acceder a la información. Es decir, al pasar a una matriz de banda, hay una gran correlación entre los índices de fila y columna, r y c para cada elemento no nulo de la matriz. Además, se puede verificar que hay una diferencia de como máximo $BW/2$ entre ambos valores. Visto de otra forma los índices de fila y columna cumplen que: $r - BW/2 \leq c \leq r + BW/2$. Los autores proponen guardar la distancia del elemento hacia la diagonal en lugar de los índices de columna para matrices con anchos de banda reducidos. Dada la reducción del ancho de banda, BW tiende a ser pequeño, permitiendo así expresar $(c - r)$ con menor cantidad de bits. Notar que, como contrapartida, es necesario utilizar un formato de numeración con signo, dado que las entradas que están por debajo de la diagonal, tendrán un offset negativo, a diferencia de cuando se almacenaba c que permitía el uso de formatos sin signo. Los autores proponen utilizar una representación de 16-bits para almacenar la diferencia $(c - r)$, estudiando cuáles matrices del conjunto de prueba elegido aplican para la compresión de índice.

Para la evaluación, los autores evalúan y comparan la velocidad de los

accesos a las matrices, almacenadas aplicando la estrategia propuesta para SP y DP, y a los vectores x en la operación SpMV. En promedio, para matrices con ancho de banda reducido, se logra una reducción del 26 % y 33 % en el tiempo requerido para acceder al vector x. El ancho de banda reducido también permite la compresión del índice de columna utilizando precisiones reducidas, lo que resulta en una reducción del 25 % (para SP) y del 16 % (para DP) en el tiempo requerido para acceder a los datos de la matriz. En promedio, se alcanzan para SP y DP, 16 % y 12,6 % respectivamente, en todo el conjunto de prueba conformado por 11 matrices bien conocidas y utilizada en otras investigaciones [10].

En la literatura se han utilizado métodos de reordenamiento de matrices como el algoritmo RCM para reducir el ancho de banda de la matriz y disminuir el número de elementos no-nulos que se generan al aplicar la factorización LU o Cholesky. En [52] Pinar y Heath, proponen estructuras alternativas de almacenamiento para matrices dispersas, junto con algoritmos de reordenamiento para incrementar la efectividad de dichas estructuras, con el principal objetivo de reducir la cantidad de accesos indireccionados. Dicho de otra forma, un reordenamiento es aplicado para permutar los elementos no nulos de la matriz, llevándolos a ubicaciones contiguas, tanto como sea posible, para agrandar los bloques densos. En este contexto, presentan dos estrategias de almacenamiento basadas en bloques, cuya efectividad depende directamente de la disponibilidad de bloques densos. La primera presenta la idea de trabajar con la matriz dispersa expresada como la suma de varias matrices. En particular, descomponen la matriz original en dos matrices, donde una contiene bloques densos de tamaño fijo $r \times c$ (en el caso del trabajo en cuestión 1×2), y la otra con los restantes elementos. El uso de estos bloques densos puede reducir el número total de operaciones de carga, así como el total de memoria requerida. Dado que el tamaño de los bloques es fijo, sólo un índice es necesario para direccionar un bloque. La segunda estrategia de almacenamiento abordada también utiliza bloques, pero a diferencia de la anterior, éstos son de largo variable, permitiendo así empaquetar más elementos no nulos en un solo bloque. De nuevo, la ventaja que ofrecen estas estructuras es que sólo es necesario el índice del primer elemento no nulo del bloque para acceder a los elementos contiguos. Esta estrategia reduce la cantidad de operaciones de carga de índices, pero requiere un ciclo más en la SpMV generando cierto overhead. Debido a esto, la elección del tamaño de los bloques afecta directamente la efectividad

del método, si el tamaño de los bloques es muy pequeño la sobrecarga generada por el ciclo extra dominará sobre la ganancia de reducir la cantidad de operaciones de carga.

Dado que el desempeño de estos métodos se encuentra ligado a la existencia de bloques densos, se propone aplicar reordenamientos con el objetivo de mejorar dicho aspecto. Los autores demuestran que el problema de reordenar las columnas es $\mathcal{NP}$-completo y, por lo tanto, se deben utilizar heurísticas para obtener una solución práctica. Proponen entonces, un modelo basado en el Travelling Salesperson Problem (TSP) [54], usando heurísticas diseñadas para ese problema. Notar que los casos de estudio, TSP y reordenar la matriz, no corresponden directamente al mismo problema, ya que mientras que uno busca encontrar un ciclo o un camino cerrado, el otro busca solamente un camino que pase por todos los nodos. Utilizando esta estrategia se intenta encontrar el recorrido que maximice el peso del camino. Para esto definieron el peso de una arista como la cantidad de filas que tienen elementos no nulos en las columnas correspondientes a los vértices unidos por esa arista. Los autores muestran que las estructuras de datos y técnicas de reordenamiento presentadas producen mejoras de hasta un 33 % y mejoras de un 21 % en promedio.

En otra investigación, Monakov et al. [47] proponen, con el objetivo de mejorar el rendimiento de la SpMV en GPUs, un nuevo formato que denominaron sliced ELLPACK. Este formato tiene por parámetro principal S, que es el tamaño o la cantidad de filas de cada *slice*. Cada una de estas porciones o slices es almacenada en formato ELL. Si bien la sobrecarga de almacenamiento en el formato sliced ELLPACK se limita a los slices con un desequilibrio en el número de elementos no-nulos por fila, esto aún puede causar una degradación notable del rendimiento. Los autores proponen entonces, una heurística simple de reordenamiento que puede mejorar sustancialmente la performance de la implementación de la SpMV para el formato presentado. Esta se basa en reordenar las filas agrupando aquellas con el mismo número de elementos distintos de cero. Es importante destacar, sin entrar en detalles, que los autores tuvieron en cuenta la complejidad del algoritmo de reordenamiento, proponiendo uno lineal en el número de filas de la matriz. De forma breve, la heurística consiste en un mecanismo simple de B cubetas (*buckets*), numeradas de 0 a $B-1$, donde la cubeta z contiene las filas con z elementos, y aquellas filas con más de B elementos son asignadas a la última cubeta. Si al agregar una fila a una cubeta ésta alcanza las S filas, se vacía dicha cubeta agregando las filas

agrupadas a la matriz reordenada formando un slice con z elementos en cada fila. Esto se repite hasta recorrer todas las filas, y las restantes que aún no hayan sido agregadas a la matriz reordenada se agregan de forma arbitraria. A partir de la etapa experimental muestran que esta heurística simple puede mejorar en gran medida el rendimiento.

Con el objetivo de aprovechar los métodos de compresión, los mismos autores de [60] extienden sus primeras propuestas con el uso de una estrategia de ordenamiento que denominaron *BRO-aware reordering* (BAR). Esta estrategia consiste en acercar aquellas columnas que poseen patrones similares en cuanto a la cantidad de bits necesarios para su codificación, de modo de reducir el espacio total y, en consecuencia, reducir posiblemente el número de transacciones a la hora de operar con la matriz. La obtención de la permutación P, es formulada por los autores como un problema de clusterización. Consiste en encontrar v particiones iguales disjuntas entre sí del conjunto de filas $\mathcal{R}$ expresadas con los delta, minimizando el número de transacciones requeridas por la SpMV. Con este objetivo, plantean una serie de ecuaciones y funciones a minimizar e indican que una posible limitante de este problema es que la forma de clusterización elegida toma en cuenta sólo la localidad espacial, y no la temporal. En general, es sabido que encontrar la solución óptima global para problemas de este estilo es $\mathcal{NP}$-completo [40]. Proponen, entonces, una heurística greedy para particionar las filas de la matriz. La evaluación experimental muestra que *BRO-aware* obtuvo mejor desempeño y ahorro de memoria, luego de comprimir la matriz en formato BRO-ELL, que AMD[1] y RCM en la mayoría de los casos. En promedio con BAR se obtuvieron ahorros en el espacio de almacenamiento de 4 %, comparados contra un 1 % para RCM y AMD. Cabe destacar que el algoritmo greedy presentado puede que no llegue a soluciones de buena calidad siempre. Por ejemplo, para un caso (la matriz *cant*) RCM y AMD obtuvieron mejores resultados que BAR, debido principalmente a que no se toma en cuenta la localidad temporal, lo que genera muchas veces, múltiples fallas en el uso del caché.

Con el creciente uso del Aprendizaje Automático en múltiples ámbitos, y la capacidad de cómputo que estos necesitan, han surgido varios estudios que buscan la forma de optimizar operaciones como SpMM (Sparse-dense Matrix

[1]El Approximate Minimum Degree (AMD) [27] es un algoritmo de reordenamiento para matrices simétricas, utilizado generalmente, para reducir la cantidad de *fill in* producido en la factorización, por ejemplo, de Cholesky.

Multiplication) y SDDMM (Sampled Dense Dense Matrix Multiplication), frecuentes en este tipo de estrategias. Un estudio de este estilo es el presentado por Hong et al. [37], en dicho artículo se diseña una estrategia de ordenamiento basada en *tiling* adaptativo, aplicándola para mejorar el rendimiento de las dos primitivas, SpMM y SDDMM. El *tiling* es una técnica muy importante para la optimización de la localidad de datos. Consiste en agrupar elementos de la matriz en bloques o tiles, generalmente 2D, con los cuales se realiza cierta operación, por ejemplo multiplicaciones y convoluciones. Usada ampliamente en implementaciones de alto rendimiento de multiplicaciones matriz-matriz densas, tanto para CPU y GPU, que aplicando un *tiling* uniforme, todos los *tiles*, sin ser aquellos ubicados en los bordes, necesitan la misma cantidad de transferencia de datos, como operaciones. Sin embargo, el irregular patrón de acceso a datos dependiente de las matrices dispersas en la multiplicaciones dificulta el uso del *tiling* para mejorar la reutilización de datos. A continuación se presenta un breve resumen de los resultados obtenidos en [37] con la estrategia propuesta, Adaptive Spaese Tiling (ASpT). Los autores proponen, a diferencia de otras investigaciones similares que utilizan formatos particulares y personalizados para optimizar las operaciones, una implementación utilizando un formato estándar, en este caso CSR. Posiblemente una de las razones más importantes por las cuales es bueno utilizar un formato estándar como CSR es la compatibilidad con el código y las bibliotecas existentes. La idea es que el número medio de no ceros por segmento de fila/columna "activo" (es decir, al menos uno elemento no nulo) dentro de un bloque 2D juega un papel importante en la determinación de si es preferible para dicho bloque la ejecución aplicando *tiling* o no. Entonces, la matriz dispersa se divide en paneles de filas, y las columnas activas dentro de cada panel de filas son agrupadas en tiles 2D para la ejecución o se relegan a la ejecución sin *tiling* porque su densidad de columna "activa" es inadecuada. La propiedad adaptativa del *tiling* en ASpT viene dada por la combinación el formato CSR con un reordenamiento dentro de las filas, buscando mejorar la localidad de los accesos. Se diferencian de otros autores por utilizar un reordenamiento que no produce tanta sobrecarga como las estrategias clásicas basadas en algoritmos greedy para grafos. Además, la técnica propuesta reordena sólo los elementos no nulos manteniendo una estructura auxiliar y no renumera todo el grafo, es decir, los índices de los elementos para el formato CSR se mantienen. La idea principal es numerar los vértices de modo que a los vértices con muchos vecinos comunes se les

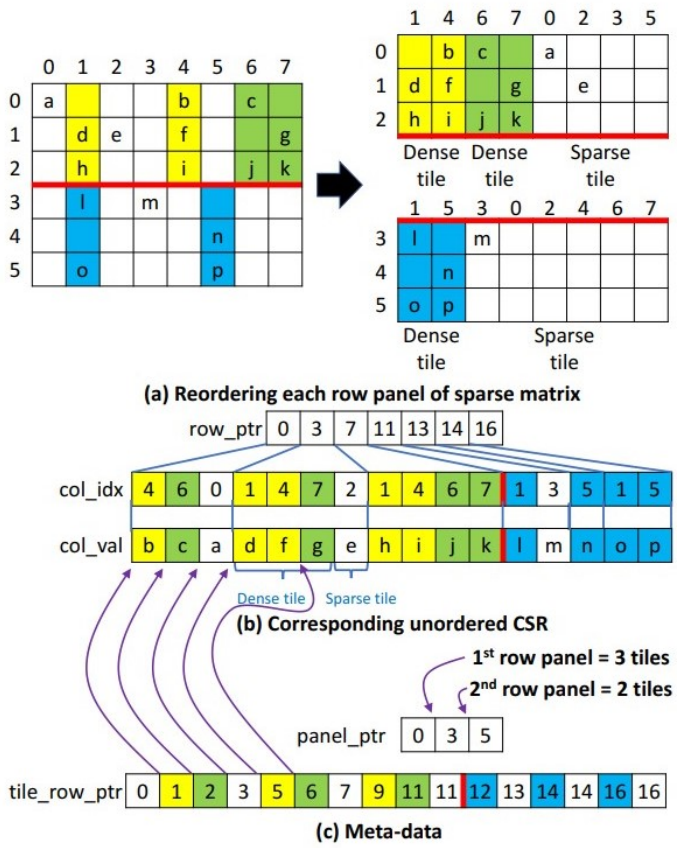


Figura 3.2: Modificación de CSR con reordenamiento. Extraída de [37].

asignen índices cercanos entre sí para mejorar la localidad de los accesos. Se puede observar mejor en la Figura 3.2.

Capítulo 4

Propuestas

En este capítulo se describen algunas propuestas que buscan incorporar ideas de distintos autores para lograr formatos de almacenamiento más eficientes para clases específicas de matrices dispersas. En particular, la primer parte de este capítulo, Sección 4.1, está orientada al estudio de heurísticas de reordenamiento utilizando estrategias evolutivas, variando las funciones de evaluación con el objetivo de encontrar nuevos ordenamientos que mejoren la eficiencia de técnicas de compresión de índices como *delta encoding*. Por otra parte, la segunda mitad de éste capítulo, presentada en la Sección 4.2, se centra en evaluar y comparar los resultados de combinar estrategias de reordenamiento con formatos de compresión.

Casos de estudio

En el proyecto, se trabaja con matrices de la colección SuiteSparse Matrix Collection [21] (anteriormente conocida como University of Florida Sparse Matrix Collection), un conjunto de matrices dispersas que surgen en aplicaciones reales, que continúa en crecimiento. La colección es ampliamente utilizada por la comunidad de álgebra lineal numérica para el desarrollo y evaluación del desempeño de algoritmos para matrices dispersos. Permitiendo a los investigadores realizar experimentos robustos (los resultados de rendimiento con matrices sintéticas generadas artificialmente, pueden ser engañosos) y repetibles (las matrices están disponibles públicamente en los formatos de archivo más comunes) [58].

Este conjunto de matrices cubre un amplio espectro de dominios, dividi-

dos en dos grandes grupos. Por un lado, los *Dominios geométricos 2D o 3D*, usualmente provienen de la discretización de EDPs en áreas como ingeniería estructural, dinámica de fluidos computacional, reducción de modelos, etc. Por otro lado, los *Dominios No-Geométricos* corresponden a problemas donde no existe una clara geometría subyacente, como por ejemplo la simulación de procesos químicos, simulación de circuitos, modelado económico y financiero, etc. Estas matrices y la metadata asociadas a cada una, serán accedidas mediante una de las `API` provistas por el grupo encargado de mantener la colección, en este caso `MATLAB`. A la fecha, la SSMC está conformada por 2893 matrices de

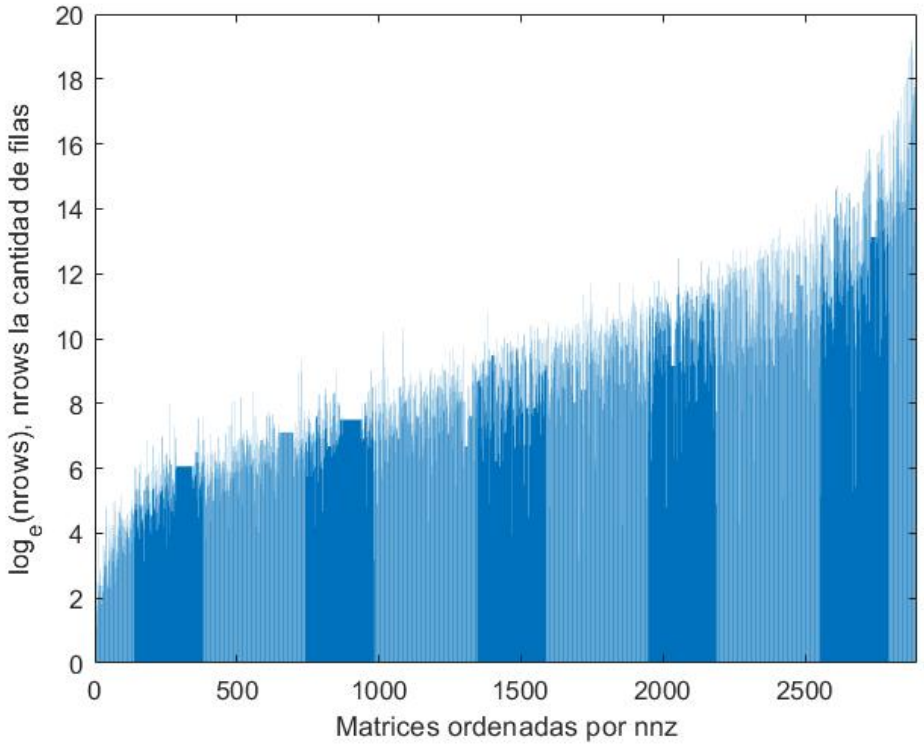


Figura 4.1: Cada barra del gráfico simboliza una de las 2893 matrices ordenadas según *nnz*, y su altura la dimensión de la matriz expresada logarítmicamente.

problemas diversos, con múltiples propiedades y patrones. Por ejemplo, 1407 de estas matrices presentan un patrón simétrico, 1185 de éstas tienen además simetría numérica, 601 corresponden a matrices con coeficientes binarios, entre otras posibles categorizaciones[1]. A su vez, las matrices pueden ser ordenadas y clasificadas por cantidad de elementos no nulos (*nnz*), así como cantidad de

[1]Notar que algunas de estas clases no son excluyentes entre sí, sino que una matriz puede presentar más de una de estas propiedades

filas y columnas.

Con respecto a la dimensión de los problemas, se pueden encontrar matrices de 2×2 (no tan comunes), hasta aquellas que poseen $226.196.185 \approx e^{19.24}$ filas y columnas, tal y como se puede apreciar en el gráfico de la Figura 4.1. Y en cuanto a la cantidad de elementos no nulos, el valor máximo de nnz es 11.588.725.964, para una matriz de $183.964.077 \times 183.964.077$.

4.1. Reducción de diagonales en matrices

Como se presentó en la Sección 3.3 algunos autores han utilizado técnicas de reducción del ancho de banda de las matrices para mejorar el uso de formatos de banda o híbridos.

En un principio, se puso especial foco en la reducción de la cantidad de diagonales, de forma de poder agrupar y representar las diagonales más densas sin la necesidad de utilizar índices para cada elemento no nulo, de manera similar a la planteada en [52] para los bloques. Si una matriz concentra una proporción importante de sus elementos no nulos en posiciones pertenecientes a unas pocas diagonales densas, se puede dividir la misma en dos componentes donde uno corresponde a las diagonales densas y el otro al resto de los elementos no nulos. A continuación se resumen algunos de los esfuerzos realizados en el proyecto en este sentido.

4.1.1. Heurísticas para reordenamiento

Con la idea de verificar y estudiar la posible compresión de matrices, se desarrolla un algoritmo evolutivo capaz de evaluar y encontrar, para la matriz M que se le indique, de tamaño $n \times n$, un vector de permutación p (de tamaño n) que, al aplicarlo, minimice lo más que se pueda alguna métrica establecida de antemano, como por ejemplo el ancho de banda de la matriz o la cantidad de diagonales densas. Aplicar la permutación expresada en p a una matriz M, implica remplazar cada fila/columna i por la fila/columna $p(i)$.

Los algoritmos evolutivos, buscan soluciones a cierto tipo de problemas sometiendo a una población de individuos a acciones y circunstancias aleatorias semejantes a las que actúan en la evolución biológica (recombinaciones y mutaciones por ejemplo), así como también a una selección de acuerdo con algún criterio, en función del cual se decide cuáles son los individuos más adaptados,

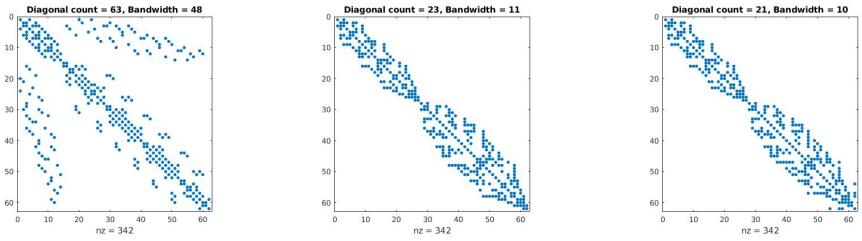
que sobreviven, y cuáles los menos aptos, que son descartados. En el Anexo 1 se describen con mayor profundidad los conceptos básicos de estos algoritmos.

En las siguientes sub-secciones se presentan variantes del algoritmo evolutivo en un esfuerzo por encontrar y evidenciar posibles características que presentan las matrices dispersas para explotar formatos basados en diagonales. En primer lugar, se estudia la posibilidad de reducción de ancho de banda para codificar los enteros de los índices en base a su distancia a la diagonal, buscando ahorrar espacio en el almacenamiento de los mismos. Luego, la idea fue reducir la cantidad de diagonales de las matrices, de forma de poder representar las diagonales más densas sin los índices. Dado que RCM es una heurística con objetivos similares a estos, porque la reducción del ancho de banda está relacionada muchas veces a la cantidad de diagonales, el estudio se centra en saber si, planteando estos nuevos objetivos, existen soluciones mucho mejores que las dadas por RCM. Por esta razón, se plantean las variantes del algoritmo evolutivo.

4.1.1.1. Reducción del ancho de banda

En este primer caso, con el algoritmo evolutivo se busca evaluar cuán lejos está la permutación generada por RCM de un posible reordenamiento óptimo. En otras palabras, el algoritmo se encargará de obtener, basado en la evaluación del ancho de banda de la matriz en cada iteración, una permutación de filas y columnas capaz de competir con RCM.

El algoritmo diseñado trabaja sobre matrices de tamaño $n \times n$. La familia de individuos o cromosomas serán vectores de permutación de tamaño n. La función a minimizar (*fitness*) será el ancho de banda β de la matriz. Es bien sabido que una componente importante del algoritmo o estrategia evolutiva son los operadores a los cuales se somete a la población de individuos, dado que son la herramienta que permite al algoritmo alcanzar cierta meta u óptimo. Para este problema, dado que los genotipos son vectores de permutación, en caso de incluir operadores de cruzamiento (o recombinación) es necesario utilizar estrategias especializadas en este tipo de codificación, quedando excluidos los operadores sencillos como el de un punto de la codificación binaria. Por esta razón, y contemplando el alto costo computacional que implican estos operadores (pruebas preliminares con el operador *Order Crossover* [2]) para cruzar estos individuos que presentan restricciones como por ejemplo que no



(a) bfwb62 original. **(b)** bfwb62 con RCM. **(c)** bfwb62 con el algoritmo evolutivo.

Figura 4.2: Tres estados de la matriz `bfwb62` con diferentes reordenamientos.

se repitan valores, se decidió no explorar a fondo este camino. De forma de intentar reforzar la carencia del operador de cruzamiento, se implementan dos operadores de mutación: `fliplr` y `swap`. El primero consiste en invertir el orden de los elementos que componen un individuo, y se aplica a la mitad de los individuos de cada generación, seleccionados aleatoriamente. El segundo operador de mutación, que se aplica con una cierta probabilidad (*Mutation rate*), consiste en realizar un intercambio (`swap`) entre dos elementos del individuo.

Para la evaluación del algoritmo evolutivo, en las primeras etapas se trabajó con matrices de dimensiones acotadas (y quizás poco representativas). Un ejemplo de estas matrices es `bfwb62`, correspondiente a un problema de electromagnetismo, de tamaño 62×62 y 342 elementos no nulos. En la Figura 4.2 se muestran tres instancias de la matriz con reordenamientos distintos. Pese a que las imágenes no muestran una diferencia significativa, principalmente entre RCM y la estrategia evolutiva, aplicando la permutación obtenida con el algoritmo evolutivo, se logra reducir en un elemento el ancho de banda con respecto al reordenamiento generado por RCM.

Evaluación experimental

El resto de las pruebas se realizaron sobre un subconjunto de matrices de problemas variados y con patrones distintos, buscando cierta heterogeneidad en las pruebas. En general, la dimensión de estas matrices es solo ligeramente mayor que la de la prueba anterior, debido a la creciente cantidad de cómputo necesaria que va a la par con la dimensión de los problemas.

Para las pruebas, el arreglo de los parámetros con los que se instanció el algoritmo evolutivo son los siguientes :

- Población inicial: 240 individuos divididos en tres grupos, una porción

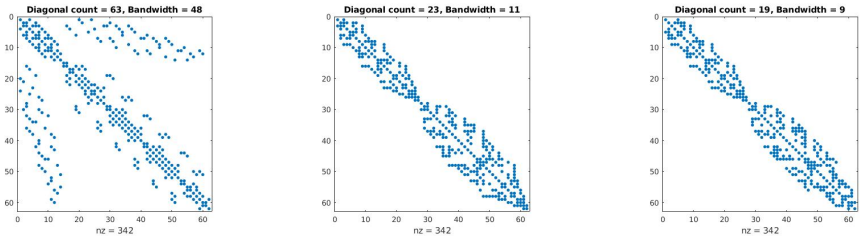
de individuos o permutaciones correspondiente a la identidad, es decir, un vector que contiene de forma ordenada los valores de 1 a n, siendo n la dimensión de la matriz, otra porción de individuos con la permutación obtenida de aplicar RCM, y el resto, la porción más grande, de permutaciones aleatorias.

- *Generations* o cantidad máxima de generaciones: 2500,
- *Mutation rate* o probabilidad de mutación: $0,2$,
- *Operadores:* los dos operadores de mutación antes mencionados.

Matriz	**Dim**(n)	***nnz***	**Original**		**RCM**		**AE**		***DC*reduction (%)**		
			DC	*BW*	*DC*	*BW*	*DC*	*BW*	RCM - Orig.	AE - Orig.	AE - RCM
'662_bus'	662	2474	463	335	237	118	237	118	64.78 %	64.78 %	0.00 %
'S10PI_n1'	528	1317	97	509	29	14	27	13	97.25 %	97.45 %	7.14 %
'Si2'	769	17801	1013	552	649	324	649	324	41.30 %	41.30 %	0.00 %
'Spectro_10NN'	531	7422	994	518	192	96	190	95	81.47 %	81.66 %	1.04 %
'Trefethen_700'	700	12654	21	512	655	327	655	327	36.13 %	36.13 %	0.00 %
'bfwb782'	782	5982	593	593	71	35	71	35	94.10 %	94.10 %	0.00 %
'dendrimer'	730	63024	1355	708	853	426	835	417	39.83 %	41.10 %	2.11 %
'dwt_503'	503	6027	477	452	129	64	129	64	85.84 %	85.84 %	0.00 %
'goddardRocket'	831	8498	723	777	575	287	573	286	63.06 %	63.19 %	0.35 %
'lowThrust_1'	584	6133	509	487	607	303	595	297	37.78 %	39.01 %	1.98 %
'lshp_577'	577	3889	325	563	53	26	53	26	95.38 %	95.38 %	0.00 %
'lshp_778'	778	5272	411	762	61	30	61	30	96.06 %	96.06 %	0.00 %
'nos6'	675	3255	7	30	33	16	33	16	46.67 %	46.67 %	0.00 %
'orsirr_2'	886	5970	437	554	245	122	245	122	77.98 %	77.98 %	0.00 %
'steam2'	600	5660	123	330	152	76	152	76	76.97 %	76.97 %	0.00 %
'young4c'	841	4089	5	29	59	29	59	29	0.00 %	0.00 %	0.00 %

Tabla 4.1: Resultados del algoritmo evolutivo intentando minimizar el *bandwidth* sobre el subconjunto de matrices.

Los resultados de la evaluación se presentan en la Tabla 4.1, dónde cada fila corresponde a los resultados para cada matriz, medidos o cuantificados con los parámetros: *bandwidth* (BW) y *diagonal count* (DC). Para el análisis se pondrá foco, principalmente, en las columnas porcentuales. La columna RCM-Orig. corresponde al porcentaje de reducción de cantidad de diagonales luego de aplicar el reordenamiento generado por RCM, comparado con la matriz original. De manera similar se presenta en la columna AE-Orig., dónde se muestran los porcentajes de reducción obtenidos por el algoritmo evolutivo con respecto al reordenamiento original. Y una columna interesante es la última, AE-RCM, que muestra el porcentaje de reducción del algoritmo evolutivo comparado con el ordenamiento de RCM. De esta tabla se puede observar que, en general, los valores de ancho de banda obtenidos con el algoritmo evolutivo no se alejaron mucho de los obtenidos mediante reordenamientos generados por la heurística RCM. Son pocos los casos en que el algoritmo logró una permutación



(a) bfwb62 original. **(b)** bfwb62 con RCM. **(c)** bfwb62 con el algoritmo evolutivo.

Figura 4.3: Tres estados de la matriz `bfwb62` con diferentes reordenamientos.

con mejor ancho de banda que RCM y, en particular, la mejora ronda en el 1 %. Siendo, posiblemente, un indicador de que los reordenamientos obtenidos con RCM son más que aceptables, dado su razonable costo computacional y midiendo la eficacia de éste según el ancho de banda resultante.

4.1.1.2. Reducción de la cantidad de diagonales

En esta segunda prueba para el algoritmo evolutivo, lo que se busca, es obtener permutaciones que logren reducir significativamente la cantidad de diagonales independientemente de si el ancho de banda (bandwidth) se ve optimizado o no.

Naturalmente, para este nuevo enfoque es necesaria la modificación de la función *fitness*, que para este caso se corresponderá con la cantidad de diagonales luego de permutar la matriz. El resto de los aspectos del algoritmo se mantienen incambiados.

Evaluación experimental

Siguiendo la metodología de evaluación utilizada en la Sección 4.1.1.1 se hizo una prueba de concepto con la matriz `bfwb62`. En la Figura 4.3, se puede observar una mejora significativa con respecto a su ordenamiento original y el de RCM. Comparado con RCM, se logra reducir en aproximadamente 21 % la cantidad de diagonales. Incluso sin que ese sea el objetivo, se produce también una reducción en el ancho de banda, logrando una reducción de 18 % con respecto a RCM, mejor resultado que el obtenido por el algoritmo en la Sección 4.1.1.1.

En la etapa de evaluación de esta segunda versión de la estrategia evolutiva, se utilizó el mismo conjunto de matrices, con la idea de obtener resultados

comparativos entre los dos enfoques. Los parámetros del algoritmo, a excepción de la función de *fitness* (en este caso cuenta la cantidad de diagonales), se mantienen.

Los resultados de las pruebas se pueden observar en la Tabla 4.2, análoga a la presentada en el apartado de evaluación de la Sección 4.1.1.1.

Matriz	**Dim**(n)	***nnz***	**Original**		**RCM**		**AE**		***DC*reduction (%)**		
			DC	*BW*	*DC*	*BW*	*DC*	*BW*	RCM - Orig.	AE - Orig.	AE - RCM
'662_bus'	662	2474	463	335	237	118	205	118	48.81 %	55.72 %	13.50 %
'S10PI_n1'	528	1317	97	509	29	14	27	13	70.10 %	72.16 %	6.90 %
'Si2'	769	17801	1013	552	649	324	649	324	35.93 %	35.93 %	0.00 %
'Spectro_10NN'	531	7422	994	518	192	96	190	95	80.68 %	80.89 %	1.04 %
'Trefethen_700'	700	12654	21	512	655	327	21	512	-3019.05 %	0.00 %	96.79 %
'bfwb782'	782	5982	593	593	71	35	71	35	88.03 %	88.03 %	0.00 %
'dendrimer'	730	63024	1355	708	853	426	773	386	37.05 %	42.95 %	9.38 %
'dwt_503'	503	6027	477	452	129	64	129	64	72.96 %	72.96 %	0.00 %
'goddardRocket'	831	8498	723	777	575	287	465	324	20.47 %	35.68 %	19.13 %
'lowThrust_1'	584	6133	509	487	607	303	427	468	-19.25 %	16.11 %	29.65 %
'lshp_577'	577	3889	325	563	53	26	53	26	83.69 %	83.69 %	0.00 %
'lshp_778'	778	5272	411	762	61	30	61	30	85.16 %	85.16 %	0.00 %
'nos6'	675	3255	7	30	33	16	7	30	-371.43 %	0.00 %	78.79 %
'orsirr_2'	886	5970	437	554	245	122	245	122	43.94 %	43.94 %	0.00 %
'steam2'	600	5660	123	330	152	76	123	330	-23.58 %	0.00 %	19.08 %
'young4c'	841	4089	5	29	59	29	5	29	-1080.00 %	0.00 %	91.53 %

Tabla 4.2: Resultados del algoritmo evolutivo intentando minimizar la cantidad de diagonales sobre sobre el subconjunto de matrices.

Dado que el conjunto de matrices fue elegido de modo que las mismas presenten características heterogéneas, los resultados se pueden dividir en tres grupos. El primero de ellos, son aquellas matrices que originalmente presentan un mejor ordenamiento que el que puede generar RCM (en algunos casos tanto para cantidad de diagonales como ancho de banda). En estos casos posiblemente el ordenamiento original sea el óptimo, por lo que el algoritmo evolutivo tiende a quedarse con dicho ordenamiento original. Por ejemplo, observar las filas de la Tabla 4.2 correspondientes a las matrices `Trefethen_700` y `nos6`. Como segundo caso, están aquellas matrices para las cuales RCM obtiene una muy buena permutación, por ejemplo, `lshp_577`, `lshp_778` y `dwt503`. El efecto del algoritmo evolutivo para estos casos, es similar producido en el experimento anterior no logrando mejoras sustanciales con respecto a la solución obtenida con RCM. Observar los resultados de 0 % en la columna AE-RCM. Y el tercer caso, el que presenta mayor interés, son aquellas que permiten evidenciar cuán lejos está RCM de un posible óptimo. Entonces, para analizar más en detalle los resultados, se centran los esfuerzos de la discusión sobre el tercer conjunto.

De las pruebas realizadas, se puede observar que para 4 de 6 matrices para las cuales el algoritmo presenta un mejor reordenamiento que el de RCM y el

original, el algoritmo evolutivo obtiene soluciones que lo sobrepasan a RCM por aproximadamente 10 %. Hay incluso, un caso donde se logra una mejora de un 29 % con respecto a RCM, y es para la matriz `lowThrust1`, donde RCM obtuvo un mal rendimiento medido en cantidad de diagonales.

En general, cuando RCM obtiene una permutación que no favorece en la cantidad de diagonales, es decir, aumenta significativamente este valor, el resultado obtenido por el algoritmo tiende a ser similar al ordenamiento original.

4.1.1.3. Cantidad fija de diagonales para formatos híbridos

Habiendo estudiado la posibilidad de reordenar las matrices intentando reducir ancho de banda y cantidad de diagonales, con un enfoque en la posible aplicación de formatos de compresión de índices, a continuación se plantea un estudio del mismo conjunto de matrices, ahora con el objetivo de emplear formatos híbridos (por ejemplo HYB, entre otros). En este caso se almacena la matriz, generalmente, en dos partes, una con una estructura razonablemente regular y la otra completamente dispersa. Modificando el algoritmo evolutivo, se intenta encontrar la permutación que, aplicada a las matrices, maximice la cantidad de elementos no nulos en cierta cantidad fija de diagonales, sin intentar optimizar el resto de parámetros. Es decir, se busca lograr una permutación que genere algunas diagonales densas (posiblemente cerca de la diagonal principal), de forma de poder almacenar las mismas en una estructura regular, en lugar de minimizar el ancho de banda y cantidad de diagonales. Notar que esta estrategia tiene cierto límite. En una matriz de tamaño $n \times n$, en el mejor de los casos, si se logra disponer los elementos en las d diagonales que son más próximas a la principal, igualmente puede suceder que la cantidad de elementos no nulos de la matriz sea mayor a las entradas disponibles para d diagonales. Cabe destacar que, para la evaluación, no se tomó en consideración que los elementos no nulos estuviesen en dichas diagonales próximas a la principal, sino que se calcula entre todas, la cantidad de elementos que cada una posee y posteriormente se selecciona aquellas d con mayor cantidad de *nnz*.

Evaluación experimental

Para la evaluación de esta estrategia, con respecto a las anteriores sólo cambia la función *fitness*. Ahora en lugar de minimizar, como se realizó para los parámetros elegidos en las Secciones 4.1.1.1 y 4.1.1.2, la función calcula la

Matriz	Dim (n)	nnz	2% DIAG	Original				RCM				AE			
				DC	BW	NNZ IN 2% DIAG	%NNZ IN 2% DIAG	DC	BW	NNZ IN 2% DIAG	%NNZ IN 2% DIAG	DC	BW	NNZ IN 2% DIAG	%NNZ IN 2% DIAG
'662_bus'	662	2474	14	463	335	951	38.44 %	237	118	966	39.05 %	529	651	1701	68.76 %
'S10PI_n1'	528	1317	10	97	509	1223	92.86 %	29	14	1058	80.33 %	65	513	1256	95.37 %
'Si2'	769	17801	16	1013	552	7029	39.49 %	649	324	4256	23.91 %	1099	768	7200	40.45 %
'Spectro_10NN'	531	7422	10	994	518	208	2.80 %	192	96	2182	29.40 %	340	530	3130	42.17 %
'Trefethen_700'	700	12654	14	21	512	9610	75.94 %	655	327	2818	22.27 %	21	512	9610	75.94 %
'bfwb62'	62	342	2	63	48	94	27.49 %	23	11	86	25.15 %	83	60	108	31.58 %
'bfwb782'	782	5982	16	593	593	3733	62.40 %	71	35	2926	48.91 %	609	597	3785	63.27 %
'dendrimer'	730	63024	14	1355	708	8000	12.69 %	853	426	4551	7.22 %	1425	729	9168	14.55 %
'dwt_503'	503	6027	10	477	452	2190	36.34 %	129	64	1919	31.84 %	537	493	2813	46.67 %
'goddardRocket'	831	8498	16	723	777	2937	34.56 %	575	287	1472	17.32 %	727	789	4291	50.49 %
'lowThrust_1'	584	6133	12	509	487	2377	38.76 %	607	303	836	13.63 %	721	547	2554	41.64 %
'lshp_577'	577	3889	12	325	563	2487	63.95 %	53	26	2680	68.91 %	277	571	3072	78.99 %
'lshp_778'	778	5272	16	411	762	3583	67.96 %	61	30	3920	74.36 %	355	766	4310	81.75 %
'nos6'	675	3255	14	7	30	3255	100.00 %	33	16	2893	88.88 %	7	30	3255	100.00 %
'orsirr_2'	886	5970	18	437	554	5170	86.60 %	245	122	2491	41.73 %	437	554	5170	86.60 %
'steam2'	600	5660	12	123	330	2413	42.63 %	152	76	2116	37.39 %	230	595	2518	44.49 %
'young4c'	841	4089	16	5	29	4089	100.00 %	59	29	2295	56.13 %	5	29	4089	100.00 %

Tabla 4.3: Resultados del algoritmo evolutivo intentando maximizar la cantidad de elementos no nulos en d diagonales para cada matriz del subconjunto de matrices.

máxima cantidad de elementos no nulos en una cantidad fija de diagonales d, expresada en la implementación como un porcentaje de n. Para estas pruebas, el porcentaje de diagonales fue un 2 % de n.

La Tabla 4.3 muestra que, luego de aplicar los ordenamientos, RCM tiende a disminuir la cantidad de elementos no nulos en las d diagonales más próximas a la principal. Por este motivo, se hace especial foco en comparar los resultados del algoritmo evolutivo con el ordenamiento original que presentan las matrices. En general, se puede apreciar que hay una aparente mejora de unos pocos puntos porcentuales por parte del algoritmo evolutivo. También se pueden observar casos en los que no se la logra maximizar la cantidad de elementos no nulos en las d diagonales en absoluto, indicando posiblemente que son matrices con estructuras diagonales, como es el caso de `nos6`, que tiene un patrón bien definido. Para 5 matrices en las que el reordenamiento original supera al de RCM, el algoritmo evolutivo logra aumentar un 3 % la cantidad de elementos no nulos alrededor de la diagonal principal. Para otras 4 matrices del conjunto de prueba, la mejora fue de un 14 %. Como máximos resultados de optimización, a 2 matrices se las logra optimizar en un 30.3 % y 39.4 %.

4.1.2. Resumen de los resultados obtenidos

Observando las Tablas 4.1 y 4.2, es bastante claro cómo, al utilizar la cantidad de diagonales como función *fitness*, los resultados obtenidos fueron mejores que con la estrategia del *bandwidth*. Cabe destacar que, en algunos casos, la reducción de diagonales también implicó una reducción del ancho de banda, obteniendo mejores resultados que en el algoritmo que emplea el ancho de

banda como función *fitness*. Si bien ambas estrategias no son eficientes comparadas con la mayoría de las heurísticas de reordenamiento para la reducción del ancho de banda, los resultados podrían evidenciar que quizás, en lugar de enfocar los esfuerzos en reducir directamente el ancho de banda, podría ser una mejor estrategia enfocarse en reducir la cantidad de diagonales.

Con respecto a la última idea evaluada, si bien RCM en los anteriores casos resultó ser un buen comienzo para el espacio de búsqueda de un reordenamiento opitmizado, como se puede observar en la Tabla 4.3 este no es el caso. Esto debido, probablemente, a la naturaleza del RCM no enfocado en directamente en lograr la menor cantidad de diagonales densas posibles.

4.2. Categorización de matrices (Estudio del espacio de matrices)

Para evaluar que tan efectivas podrían ser las técnicas de compresión de matrices dispersas basadas en la reducción de precisión para almacenar los índices, a continuación se presenta una clasificación o categorización de un gran grupo de matrices de `SuiteSparse Matrix Collection`. En particular, las matrices de la colección que son simétricas, a las que se les puede aplicar técnicas de reordenamiento para reducir su ancho de banda. Se evalúa para las matrices mencionadas, utilizando técnicas distintas, cuál es la cantidad de bits mínima con la que se podría almacenar las coordenadas si se utiliza otro valor como índice de columna. Entre las estrategias que se evalúan están, reducir la precisión de los índices actuales basados en las dimensiones de la matriz, sustituir el índice por la distancia a la diagonal, y por último utilizar la diferencia con el elemento no nulo anterior o *delta encoding*.

Para esta tarea se programaron rutinas (scripts y funciones) en `MATLAB`, que accediendo a través de la `API` de SSMC obtienen las matrices simétricas, y aplican las técnicas que se describen en las siguientes secciones.

4.2.1. Comprimir los índices de cada fila sin modificarlos

El primer estudio planteado consistió en evaluar la capacidad de compresión al utilizar los índices originales asociados al formato comprimido, ya sea

por filas o columnas (incluso COO). Lo importante es que se intenta atacar la componente no comprimida de los índices. Para este caso particular, las matrices están en formato CCS, por lo que se intentarán comprimir los índices de fila, los cuales son no-negativos. El estudio evalúa la cantidad mínima de bits (8, 16 o 32) para almacenar todos los coeficientes de la matriz. Es decir, se busca el índice que requiere más bits para ser representado y, en base a esto, se cuantifican los bits necesarios para la matriz. Notar que en este caso la cantidad de bits está dada por la dimensión de la matriz, que es el máximo valor que los coeficientes pueden alcanzar.

Como los índices a estudiar son no-negativos, y se quiere evaluar la posibilidad de utilizar representaciones con 8, 16 y 32 bits, se obtienen los siguientes rangos para poder clasificar cada matriz según sus índices máximos: $[0, 2^8 - 1]$, $[0, 2^{16} - 1]$ y $[0, 2^{32} - 1]$.

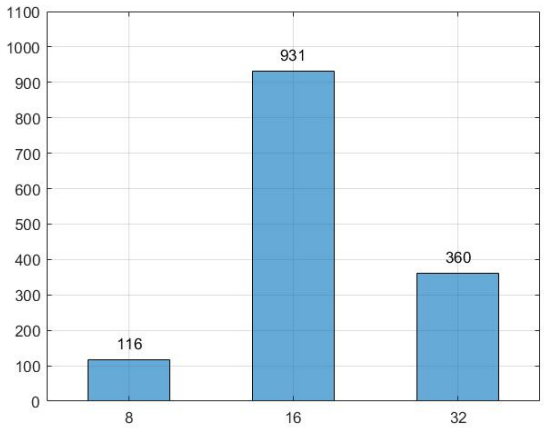


Figura 4.4: Histograma que muestra la cantidad de matrices cuyos índices de columna pueden ser representados en 8, 16 y 32 bits.

En la Figura 4.4 se presenta un gráfico con la clasificación de las matrices, dependiendo de si sus índices pueden ser almacenados en cada una de las precisiones. Se puede observar que, del espacio de 1407 matrices estudiadas, quedan catalogadas con 8, 16 y 32 bits, 116, 931 y 360 matrices, respectivamente. Es decir que aproximadamente el 25 % de las matrices necesitan 32 bits para ser almacenadas.

En esta y en las secciones posteriores, se estudiarán con mayor foco las matrices que dependiendo de la técnica de compresión analizada, pertenecen a la categoría de 32 bits. En este sentido, se presenta en la Figura 4.5 los resultados para dichas matrices, para observar efectivamente cuántas son las filas que

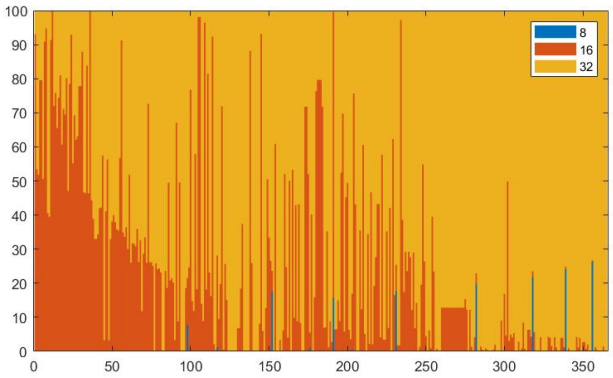


Figura 4.5: Cada barra del gráfico simboliza una matriz original, que necesita 32 bits para ser representada, y cada componente es el porcentaje de índices máximos por fila que pueden ser representadas con 8, 16 y 32 bits.

necesitan los 32 bits para ser representadas, o dicho de otra forma, comparar la cantidad de filas que puedan ser representadas con 8 y 16 bits, dentro de las matrices de esta categoría. Las barras del gráfico están subdivididas en 3 partes, indicando la proporción de filas de la matriz que pueden ser representadas con las cantidades de bits antes mencionadas. Notar como la división de los porcentajes de filas, está principalmente distribuida, en la mayoría de las matrices, entre 16 y 32 o, en otras palabras, pocas son las matrices que presentan una porción considerable de filas representables con 8 bits (si fueran una cantidad reducida de filas se podrían utilizar formatos híbridos, almacenando esas pocas filas representables con 32 en otra estructura y el resto con precisiones reducidas). Se puede apreciar, que cuanto menor es la cantidad de elementos no nulos, (izquierda del gráfico), menor es la cantidad de filas que necesitan 32 bits para ser almacenadas. A medida que crece dicha cantidad, hacia la derecha, parece aumentar progresivamente la cantidad de filas en 32 bits.

4.2.2. Diferencia a la diagonal

Comenzando a evaluar posibles técnicas de compresión enfocadas sobre los índices. En esta sección se realiza un estudio de la utilización de la diferencia a la diagonal como índice, en lugar del valor de columna original. Para dicho objetivo, es necesario calcular para cada fila de cada una de las matrices, la máxima distancia de un elemento no nulo a la diagonal (en valor absoluto, notar que es equivalente al ancho de banda por fila $\beta(A_i)$). Obtenidos estos

valores, se determina cuantos bits son los necesarios para su almacenamiento y se los clasifica. En este sentido, la Figura 4.6 resume la cantidad de matrices que necesitan 8, 16 y 32 bits para ser almacenadas utilizando esta técnica para representar el índice.

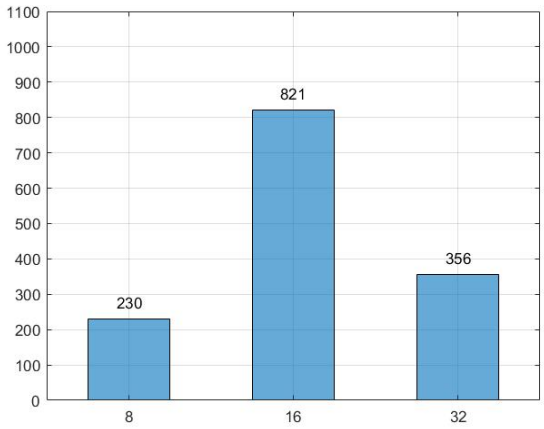


Figura 4.6: Histograma que muestra la cantidad de matrices cuyos índices de columna expresados como la distancia a la diagonal pueden ser representados en 8, 16 y 32 bits.

Notar cómo, a diferencia de la estrategia analizada anteriormente, donde los índices sólo trabajaban con valores positivos, para esta técnica se necesitarán enteros con signo[1]. En particular, se busca un rango simétrico con respecto a 0, valor que simboliza la diagonal. Se obtienen para 8, 16 y 32 bits los rangos: $[-2^7, 2^7]$, $[-2^{15}, 2^{15}]$ y $[-2^{31}, 2^{31}]$.

En la Figura 4.7, de manera similar a la anterior sección, se puede observar un gráfico de barras, del que se desprende que para la gran mayoría de las matrices que pertenecen a la categoría de 32 bits, a medida que crece la cantidad de elementos nulos (hacia la derecha), aumenta el porcentaje de filas en la categoría de 32 bits. También se puede observar cómo decrece el porcentaje de filas en 16 bits, notorio en la porción con menor cantidad de elementos no nulos, agrupada a la izquierda.

[1]Por ejemplo, el índice de una entrada no nula a la derecha de la diagonal tomará valores positivos, mientras que si está a la izquierda negativos.

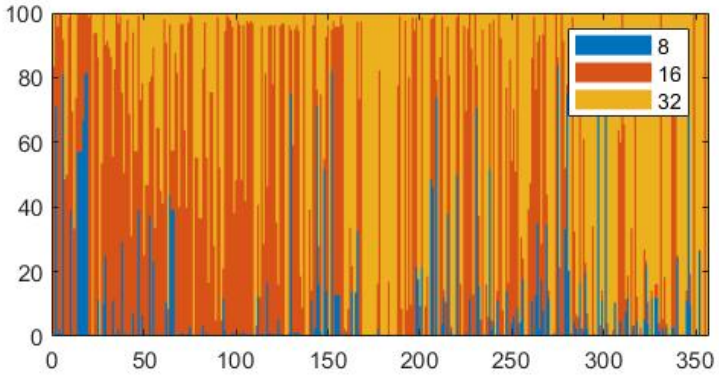


Figura 4.7: Cada barra del gráfico simboliza una matriz sin reordenar, que necesita 32 bits para ser representada, y cada componente es el porcentaje de distancias máximas por fila a la diagonal que pueden ser representadas con 8, 16 y 32 bits.

4.2.3. Diferencia a la diagonal con reordenamiento

En esta sección, se realiza un análisis del uso de técnicas de reordenamiento. En particular, y siguiendo las ideas propuesta por Xu et al. [65], se aplica la heurística RCM a la técnica de utilizar la diferencia a la diagonal en lugar del índice de columna, presentada en la Sección 4.2.2.

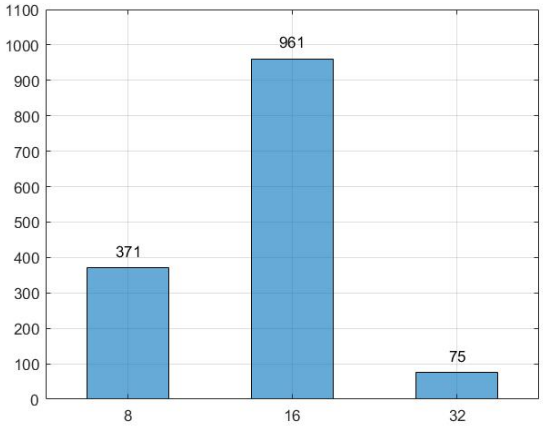


Figura 4.8: Histograma que muestra la cantidad de matrices. luego de aplicar RCM, cuyos índices de columna expresados como la distancia a la diagonal pueden ser representados en 8, 16 y 32 bits.

Una comparativa de cuántas matrices son representables, utilizando la técnica de distancia a la diagonal, con 8, 16 y 32 bits, con y sin aplicación de reordenamiento, se puede observar en las Figuras 4.6 y 4.8. De las gráficas se deduce que el uso de RCM ofrece importantes beneficios, en especial dismi-

nuyendo las matrices representables con 32 bits. Este artilugio, permite pasar de 353 a 63 matrices, del espacio representables con 32 bits, en otras palabras, 290 o el 82 % de dichas matrices se puede almacenar con precisiones menores, siempre y cuando se les aplique un reordenamiento. En el caso de 16 bits, si bien crece el número, muchas de estas son matrices que antes necesitaban 32 bits. De hecho, 154 matrices que necesitaban 16 pasan a 8 bits al aplicar RCM, como se puede observar en la Figura 4.9. Este gráfico, que a los efectos de este proyecto se llamará matriz de composición, es útil para comparar los beneficios de aplicar algún tipo de reordenamiento a una técnica. Cada una de las filas representa la cantidad de matrices que originalmente pueden ser representadas utilizando el número de bits indicada por la fila, a su vez, cada una está subdividida en las categorías a las que van a parar luego de aplicar RCM. Si se la ve por columnas, cada una representa el número de matrices originales que fueron a parar a cada categoría luego de aplicado el reordenamiento obtenido con RCM. En dicha figura se puede apreciar que, si bien los números del triángulo superior son muy bajos, hay valores por encima de 0 (y uno sólo en 0), indicando que algunas de éstas matrices, en su forma original, pueden ser representadas con una menor cantidad de bits que cuando se les aplica RCM. Para este caso hay 15 matrices que pasan de 8 a 16 bits, y 2 que pasan de 16 a 32. Notar que son cantidades mucho menores a aquellas que si permiten una compresión. De todos modos, evidencia que existen ciertos problemas para los que no sería conveniente aplicar esta técnica de reordenamiento.

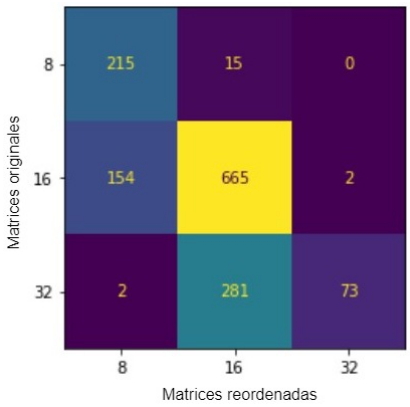


Figura 4.9: Matriz de composición, muestra en cada fila la distribución de las matrices en cada categoría luego del reordenamiento con RCM, en base a la distancia a la diagonal.

En la Figura 4.10 se presenta el ejemplo de la matriz `FIDAP/ex25` que pasa,

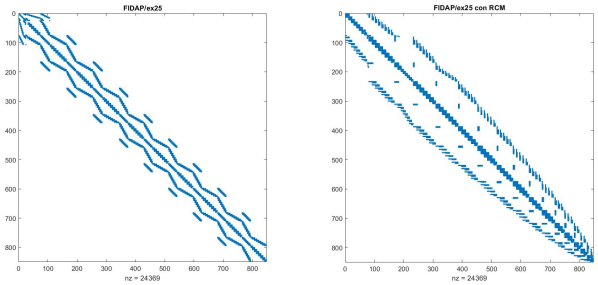


Figura 4.10: Ejemplo de matriz que al aplicarle RCM pasa de categoría, de 8 a 16 basado en la diferencia a la diagonal.

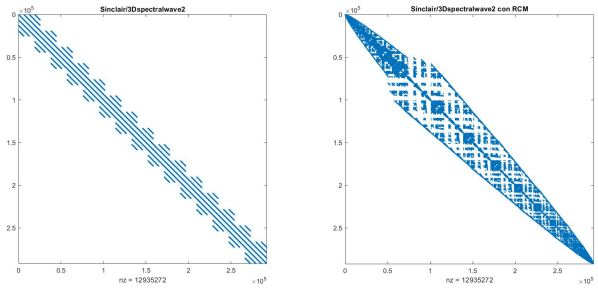


Figura 4.11: Ejemplo de matriz que al aplicarle RCM pasa de categoría, de 16 a 32 basado en la diferencia a la diagonal.

luego de aplicar el reodenamiento generado por RCM, de la categoría de 8 a 16 bits. En particular en el ordenamiento original, las 848 filas eran representables con 8 bits, pasando a tener 475 en esta categoría y 373 en la de 16 bits. Notar la periodicidad de la matriz, estructura que se pierde por la forma en la que reordena RCM, que pasando de vértices de grados menores a mayores, acaba por producir un leve ensanchamiento en el ancho de banda, rompiendo con la regularidad del ordenamiento original.

De igual forma, en la Figura 4.11 se plantea un ejemplo de matriz que pasa, en este caso, de 16 a 32 bits. `Sinclair/3Dspectralwave2` tiene un comportamiento muy similar al ejemplo anterior ante RCM.

De esto se puede concluir que en general el uso de las heurísticas de reordenamiento, en particular RCM, permiten ahorrar en el almacenamiento. Sin embargo, es necesario estudiar caso a caso porque en algunos ejemplos de matrices estructuradas, el reordenamiento rompe con dicha estructura llegando a aumentar los requerimientos de almacenamiento, obteniendo resultados con-

trarios al objetivo.

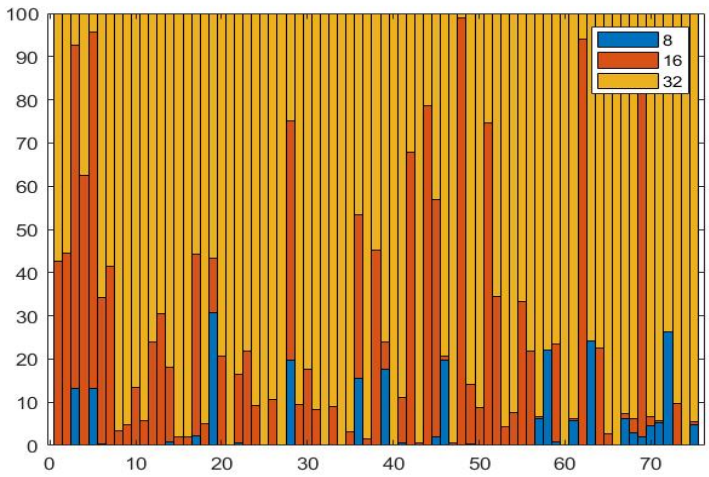


Figura 4.12: Cada barra del gráfico simboliza una matriz que necesita 32 bits para ser representada luego de aplicar RCM, y cada componente es el porcentaje de distancias máximas por fila a la diagonal que pueden ser representadas con 8, 16 y 32 bits.

Buscando profundizar el estudio, tal y como se realizó en los casos anteriores, se toman las matrices representables con al menos 32 bits y se evalúa qué proporción de sus filas, intentando comprimirlas, permiten un cambio de precisión, es decir, una compresión. Resultados que se pueden observar en la Figura 4.12, que en general indican que luego de aplicar RCM, las matrices que pertenecen a esta categoría, en su gran mayoría tienen una baja proporción de sus filas representables en precisiones menores que 32 bits. Tampoco se aprecia una correlación importante entre la cantidad de elementos no nulos y la cantidad de filas en precisiones menores.

4.2.4. Delta entre índices

Se estudió también, para cada matriz del conjunto definido, el delta o la distancia máxima entre elementos no nulos consecutivos para cada una de las filas y para cada matriz. Esta idea fue aplicada por Maggioni et al. [46], para su formato CoAdELL, presentado de forma breve en el Capítulo 3, así como Kourtis et al. [42] en su formato CSR-DU, entre otros autores [60, 64].

Al igual que en el estudio de los índices máximos por filas, presentado en la Sección 4.2.1, los valores que toman los índices utilizando este enfoque, son

también positivos, notar que el elemento anterior necesariamente tendrá un índice menor, por lo que su resta será positiva.

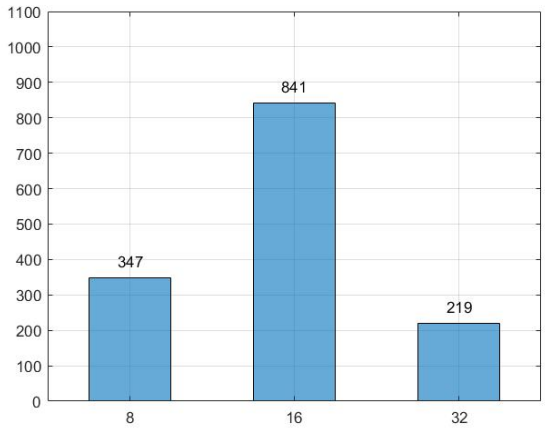


Figura 4.13: Histograma que muestra la cantidad de matrices cuyos índices de columna expresados como la distancia al elemento anterior no nulo pueden ser representados en 8, 16 y 32 bits.

A continuación se presentan algunos de los resultados obtenidos. En la Figura 4.13 se muestra la categorización del espacio de matrices si se representa el índice como la diferencia con el anterior no nulo. Notar que, con respecto a la técnica de diferencia a la diagonal sin reordenar (Figura 4.6), aplicando esta compresión se obtienen significativamente más matrices en la categoría de 8 bits, mientras que la de 32 bits se ve reducida en un porcentaje similar.

Cabe destacar que en este caso, los rangos de representación comienzan en cero (a diferencia de los rangos de la parte anterior, que utilizaban representaciones con signo), permitiendo utilizar 1 bit más dedicado, no al signo, sino a la numeración directamente.

Similar a lo que se realizó anteriormente, se evalúan las matrices que necesitan 32 bits para ver cuántas filas de esas matrices se pueden representar con 8 y 16 bits. Dicho estudio se resume en la Figura 4.14. Nuevamente, se puede observar cómo, para las matrices que poseen menores cantidades de *nnz*, ubicadas sobre la izquierda del gráfico, una gran porción presenta muy pocas filas que necesitan 32 bits. A medida que aumenta la cantidad de elementos no nulos de las matrices se puede apreciar, quizás no tan regularmente, un progresivo aumento de las filas en 32 bits.

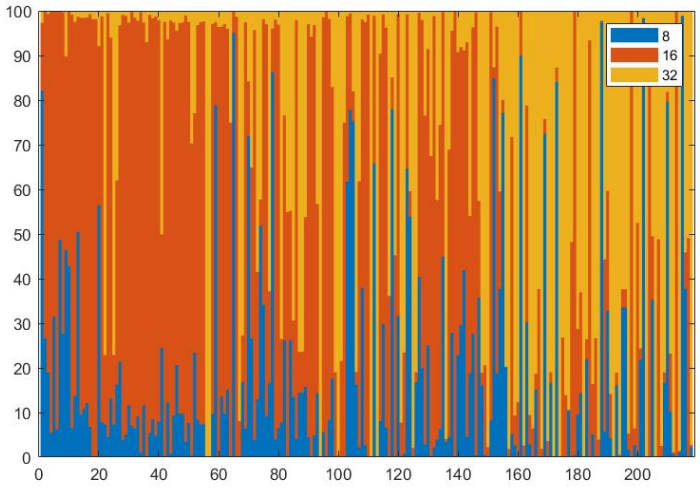


Figura 4.14: Cada barra del gráfico simboliza una matriz sin reordenar, que necesita 32 bits para ser representada, y cada componente es el porcentaje de deltas máximos por fila que pueden ser representadas con 8, 16 y 32 bits.

4.2.5. Delta entre índices con reordenamiento

Considerando los importantes beneficios de usar RCM para la técnica de sustituir los índices por la diferencia a la diagonal, presentado en 4.2.3, en este trabajo se propone extender las ideas de Maggioni et al. aplicando RCM pero, en este caso, en conjunto con la codificación delta de forma similar a la propuesta por [60], los resultados se resumen en la Figura 4.15.

Para esta conjunción de técnicas, se produce un efecto similar al ocurrido en la Sección 4.2.3. Aplicar el reordenamiento de RCM produce resultados mejores, incluso con un impacto más positivo si se comparan los efectos sobre la técnica de la diferencia a la diagonal, obteniendo menores cantidades de matrices en la categoría de 32 bits y mayores en la de 8 bits. Con respecto a las originales con la misma técnica de la distancia al anterior, se logra reducir en aproximadamente un 75 % la cantidad de matrices que se representaban originalmente con 32 bits.

De la misma manera, algunas matrices aumentaron la cantidad de bits necesaria para poder ser representadas luego de aplicar el reordenamiento de RCM. Estos resultados se pueden observar en la matriz de composición, presentada en la Figura 4.16. En este caso, las relaciones por encima de la diagonal corresponden a 22 matrices que pasan de 8 a 16 bits y 4 que pasan de 16 a 32. Notar que, estos son valores mayores a los obtenidos con la técnica de la

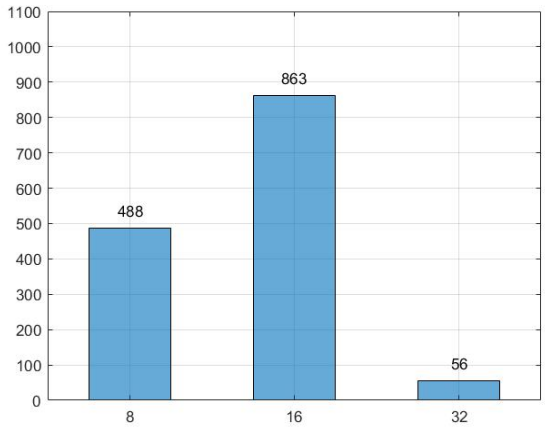


Figura 4.15: Histograma que muestra la cantidad de matrices, luego de aplicar RCM, cuyos índices de columna expresados como la distancia al elemento anterior no nulo pueden ser representados en 8, 16 y 32 bits.

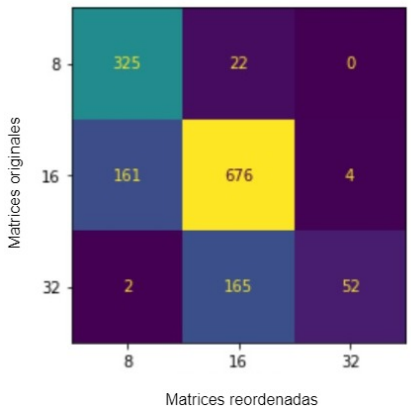


Figura 4.16: Matriz de composición, muestra en cada fila la distribución de las matrices en cada categoría luego del reordenamiento con RCM, en base a la diferencia con el elemento no nulo anterior.

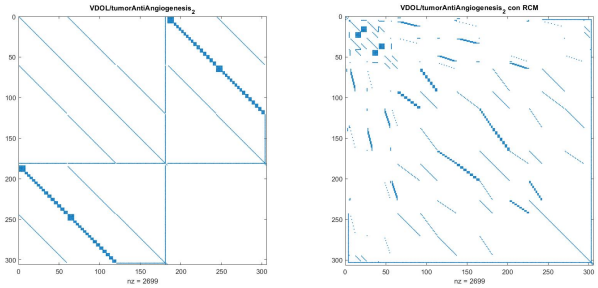


Figura 4.17: Ejemplo de matriz que al aplicarle RCM pasa de categoría, de 8 a 16 basado en la diferencia al elemento no nulo anterior.

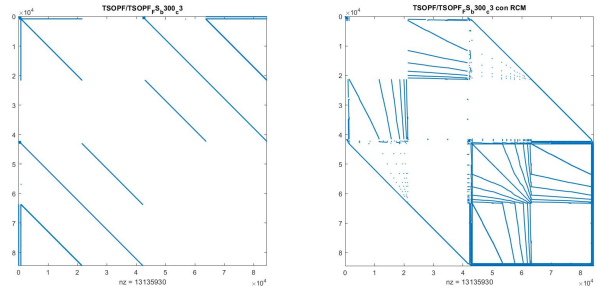


Figura 4.18: Ejemplo de matriz que al aplicarle RCM pasa de categoría, de 16 a 32 basado en la diferencia al elemento no nulo anterior.

diferencia a la diagonal. En las Figuras 4.17 y 4.18 se presentan dos ejemplos de matrices que padecieron esta situación. La Figura 4.17 corresponde a la matriz `VDOL/tumorAntiAngiogenesis_2` (la imagen a la izquierda corresponde a la matriz original y la derecha a la misma luego de aplicar RCM), pasando de una distancia máxima de 180 (representable con 8 bits 0-255) entre elementos contiguos en una misma fila, a 283 (con 8 bits no es suficiente). Si bien puede dar la impresión de que, en el desorden, las distancias entre elementos no nulos se acortaron, con un estudio minucioso se puede observar que no es así, constatando la existencia de filas que tienen distancias mayores a la original.

De igual forma, la Figura 4.18 que representa la matriz `TSOPF/TSOPF_FS_b300_c3`, pasando en este caso, de un delta máximo de 42438 representable con 16 bits, a 82215 excediendo dicho rango.

Otro fenómeno que se produce, quizás no se percibe en los gráficos, es el hecho de que muchas matrices luego de aplicarles RCM, si bien no pasan a categorías más grandes, sí sucede que el delta máximo en varios casos es mayor

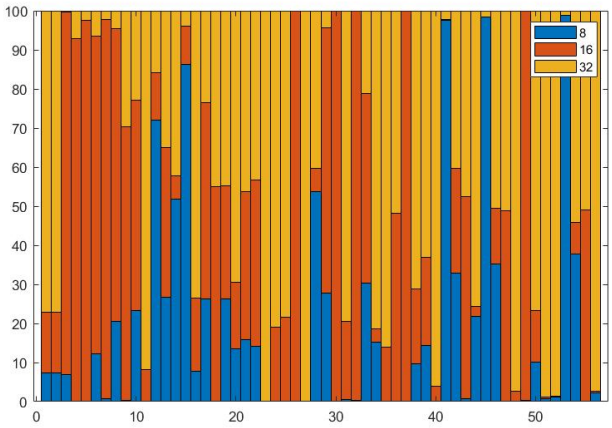


Figura 4.19: Cada barra del gráfico simboliza una matriz que necesita 32 bits para ser representada luego de aplicar RCM, y cada componente es el porcentaje de deltas máximos por fila que pueden ser representadas con 8, 16 y 32 bits.

cuando se aplica RCM.

Tal y como se ha realizado en las secciones previas, a continuación se intenta ahondar en las matrices representables con 32 bits, las que poseen posiblemente, (a nuestro criterio mayor) interés. Se presenta en la Figura 4.19, un gráfico donde se puede observar, para las 56 matrices ordenadas por cantidad de elementos no nulos, la proporción de filas que pueden ser representadas en las diferentes precisiones. Con un fenómeno muy similar a su contraparte sin ordenar, pero a mucho menor escala, se obtiene un resultado muy variado en la proporción de filas, mientras que las primeras, las de menor *nnz*, presentan pocas filas que necesitan 32 bits. Luego, a medida que aumenta la cantidad de elementos no nulos, paulatinamente crece la proporción de filas que pueden ser representadas sólo con 32 bits.

4.2.6. Resumen de la evaluación

Para resumir y analizar los datos obtenidos, se presenta en la Tabla 4.4 la clasificación del espacio de matrices utilizado, aplicando las diferentes técnicas anteriormente discutidas. De ésta se desprenden varios resultados. Probablemente la primera observación que surge analizando la columna de la cantidad de matrices que pertenecen a la categoría de 32 bits, es la reducción de dicha cantidad al aplicar cualquiera de las técnicas propuestas respecto a utilizar los índices originales, incluso sin aplicar reordenamientos como RCM. En el

Variante	8	16	32
4.3.1. Índice de columna	116	931	360
4.3.2. Diferencia a la diagonal	230	821	356
4.3.3. Diferencia a la diagonal con RCM	371	961	75
4.3.4. Delta entre índices	347	841	219
4.3.5. Delta entre índices con RCM	488	863	56

Tabla 4.4: Resumen de los resultados de clasificación de las diferentes estrategias evaluadas.

caso de diferencia a la diagonal la reducción no es tan notoria, sólo 4 matrices menos, mientras que utilizando el delta entre índices se obtiene resultados más favorables.

Como segunda observación, las técnicas de reordenamiento abordadas mejoran significativamente la clasificación de las matrices, fenómeno que puede ser observado en la reducción de la cantidad de matrices que se clasifican con 32 bits. Esto motiva a continuar profundizando acerca de las técnicas para compresión de matrices aplicando reordenamientos en el futuro. Otra línea interesante es la exploración de herramientas para optimizar operaciones como la SpMV utilizando estas técnicas, explotando también las características de distintas arquitecturas de hardware. A pesar de que RCM no es una heurística específicamente diseñada para reordenar matrices con el objetivo de reducir el espacio de almacenamiento, cabe destacar que los resultados obtenidos, aplicando esta técnica, resultan más que favorables. A esta misma conclusión llegaron múltiples autores en sus investigaciones, tal y como se presenta en el Capítulo 3.

Capítulo 5

Conclusión y trabajo futuro

En este capítulo se detallan las conclusiones más importantes inferidas a través del trabajo realizado en este proyecto de grado. Adicionalmente, se resumen algunas de las posibles líneas de trabajo futuro identificadas que permitirían extender el trabajo.

5.1. Conclusiones

El proyecto tenía por objetivo principal, como se menciona en el Capítulo 1, avanzar en el estudio y comprensión de estrategias de optimización para el uso de matrices dispersas. Poniendo especial foco en el estudio de estrategias de almacenamiento híbridas y explorando el uso de técnicas de reordenamiento en conjunto con la aplicación de estrategias de precisiones reducidas. Teniendo en cuenta el objetivo planteado, se desprenden los siguientes objetivos específicos: (i) Realizar una actualización del estado del arte del uso de matrices dispersas; (ii) Actualizar el estado del arte del uso de computación de alta performance (HPC) y en especial, su uso para acelerar la resolución de problemas de álgebra lineal numérica (ALN) dispersa; (iii) Estudiar estrategias de almacenamiento para matrices dispersas, que apliquen técnicas híbridas, reordenamientos y el uso de múltiples precisiones; (iv) En base a lo relevado, desarrollar estrategias (formatos, procedimientos, etc.) para matrices dispersas que permitan alcanzar un uso más eficiente de los datos/cómputo.

En primera instancia, se destaca que la ejecución del proyecto permitió cumplir con los objetivos originalmente planteados. En particular, se realizó un relevamiento del estado del arte del uso de matrices dispersas, describiendo

varios de los formatos de almacenamiento más conocidos y usados. También se conceptualizó una breve explicación de algunas de las estrategias de reordenamiento más extendidas a la hora de trabajar con matrices dispersas. En el mismo marco teórico, se analizó el uso de arquitecturas de HPC y su aplicación para acelerar la resolución de problemas de ALN dispersa, comentando las principales dificultades de operar con este tipo de matrices, y motivando así el estudio de formatos que permitan operar de forma eficiente. También se avanzó en la actualización del estado del arte centrado en las técnicas para almacenar y operar con matrices dispersas. Especialmente, aplicando diferentes estrategias que, buscando optimizar ciertos aspectos al operar, trabajan tanto sobre los índices de la matriz, el orden de las filas y columnas, o con los coeficientes de éstas. Como conclusión de esta etapa, se puede destacar que, la gran mayoría de las investigaciones relevadas, tienen su foco en una operación en particular, la SpMV. Y también muchas de ellas están enfocadas en optimizaciones para alguna arquitectura de hardware específica. Se puede notar también que, una cantidad importante de estas investigaciones aplican reordenamientos, ya sea para optimizar cierta propiedad antes de almacenar la matriz, o directamente sobre el formato disperso planteado (por ejemplo para mejorar los accesos a memoria), dejando en claro la importancia de este tipo de enfoques. Se resalta también, que otro conjunto importante de investigaciones buscan disminuir el tráfico de datos entre los distintos niveles de la jerarquía de memoria mediante la reducción de la precisión utilizada, tanto en lo referido a los coeficientes como los índices.

En cuanto al foco principal del trabajo, se proponen y evalúan, en primera instancia, diferentes estrategias para reordenar matrices utilizando heurísticas. En particular, se desarrollaron algoritmos evolutivos, teniendo por objetivo encontrar reordenamientos que permitan explotar técnicas de compresión para lograr un uso eficiente de memoria. Entre los resultados más interesantes que se obtuvieron se puede destacar que, al intentar minimizar la cantidad de diagonales con el algoritmo evolutivo planteado, se logra optimizar también el ancho de banda de la matriz, incluso más que el algoritmo que tenía esta medida por función objetivo.

Por otro lado, se realizó un estudio y discusión sobre los posibles ahorros de memoria al intentar comprimir los índices. Específicamente, se comparan distintas técnicas que explotan el almacenamiento de índices con diferentes precisiones. Para la evaluación experimental se emplean las matrices disper-

sas, con patrón simétrico, de la colección SSMC, siguiendo un procedimiento sistemático sobre todo el conjunto de prueba y los métodos a evaluar. Este estudio evidencia que, para una gran cantidad de problemas, el abordaje de estos enfoques ofrece importantes mejoras. Entre otros resultados se destacan los beneficios al aplicar técnicas alternativas para almacenar los índices, como son el *delta encoding* y la diferencia a la diagonal. Estas mejoras se ven incrementadas si se combinan con la aplicación de las estrategias de reordenamiento, por ejemplo mediante la heurística RCM.

En cuanto a los resultados obtenidos en la evaluación experimental, es necesario enfatizar las importantes reducciones en los requerimientos de almacenamiento alcanzadas. Obteniendo que, en promedio, entre todas las técnicas que no aplican reordenamientos, un 61 % de las matrices analizadas pueden ser almacenadas con 16 bits (entre 9 y 16), y un 16 % pueden ser almacenadas con 8 bits. Como aspecto negativo, entre las matrices que requieren 32 bits, no se identifican tendencias que permitan avanzar con el uso parcial de precisiones menores, es decir no hay un número grande de filas que puedan ser almacenadas con una cantidad menor de bits. Por último, y especialmente destacado, la aplicación de técnicas de reordenamientos, incluso con heurísticas no diseñados para optimizar los parámetros estudiados, ofrecen una importante mejora de las técnicas evaluadas. En este caso el promedio de las matrices que pueden ser almacenadas con 16 bits sube a 64 % y a 30 % las que se pueden almacenar con 8 bits.

5.2. Trabajo futuro

El desarrollo de este trabajo permitió avanzar en la comprensión de distintas líneas de investigación relacionadas con el almacenamiento de matrices dispersas. No obstante, existen ciertos puntos importantes vinculados al objetivo del proyecto que, debido al alcance y tiempo, no pudieron se abordados. A continuación, se detallan algunas de estas ideas que podrían extender el proyecto como trabajo futuro:

- Una línea interesante es desarrollar algoritmos y heurísticas que permitan acercarse más, en cuanto a costos computacionales razonables, a soluciones como las obtenidas por el algoritmo evolutivo de la Sección 4.1.
- Otro aspecto prometedor es implementar y evaluar operaciones matri-

ciales utilizando los formatos propuestos, en particular, para alguna arquitectura de hardware de interés científico.

- Por último, sería importante implementar una biblioteca de ALN dispersa capaz de manipular matrices almacenadas con los formatos abordados, operar sobre ellas y así poder estudiar en mayor profundidad los beneficios de estos paradigmas.

Bibliografía

[1] *42 Years of Microprocessor Trend Data*. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/ (Accedido el 04-10-2020).

[2] Umbarkar A.J. y Sheth P.D. "CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW". En: *ICTACT Journal on Soft Computing* 06.01 (oct. de 2015), págs. 1083-1092. DOI: 10.21917/ijsc.2015.0150.

[3] Hartwig Anzt, Jack Dongarra, Goran Flegar, Nicholas J. Higham y Enrique S. Quintana-Ortí. "Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers". En: *Concurrency and Computation: Practice and Experience* 31.6 (mar. de 2018), e4460. DOI: 10.1002/cpe.4460.

[4] *Audi piston rod - Stiffness matrix*. URL: https://www.cise.ufl.edu/research/sparse/matrices/Koutsovasilis/F2.html (Accedido el 29-12-2020).

[5] Tulasi B, Rupali Sunil Wagh y Balaji S. "High Performance Computing and Big Data Analytics Paradigms and Challenges". En: *International Journal of Computer Applications* 116.2 (abr. de 2015), págs. 28-33. DOI: 10.5120/20311-2356.

[6] Klaus-Jürgen Bathe. *Finite Element Procedures*. Klaus-Jürgen Bathe, ago. de 2014. ISBN: 0979004950.

[7] Mehmet Belgin, Godmar Back y Calvin J. Ribbens. "Applicability of Pattern-based sparse matrix representation for real applications". En: *Procedia Computer Science* 1.1 (mayo de 2010), págs. 203-211. DOI: 10.1016/j.procs.2010.04.023.

[8] Mehmet Belgin, Godmar Back y Calvin J. Ribbens. "Pattern-based sparse matrix representation for memory-efficient SMVM kernels". En: *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*. ACM Press, 2009. DOI: 10.1145/1542275.1542294.

[9] N. Bell y M. Garland. *Cusp library*. 2012. URL: https://github.com/cusplibrary/cusplibrary.

[10] N. Bell y M. Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". En: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, págs. 1-11. DOI: 10.1145/1654059.1654078.

[11] Nathan Bell. *Iterative methods for sparse linear systems on GPU - NVIDIA Research*. Mechanical Engineering, Pan-American Advanced Studies Institute, Boston University. URL: https://www.bu.edu/pasi/courses/iterative-methods-for-sparse-linear-systems/.

[12] Nathan Bell y Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004. NVIDIA Corporation, dic. de 2008.

[13] Luc Buatois, Guillaume Caumon y Bruno Lévy. "Concurrent number cruncher: a GPU implementation of a general sparse linear solver". En: *International Journal of Parallel, Emergent and Distributed Systems* 24.3 (jun. de 2009), págs. 205-223. DOI: 10.1080/17445760802337010.

[14] Anwesha Chakraborty, Trina Dutta, Sushmita Mondal y Asoke Nath. "Application of Graph Theory in Social Media". En: *International Journal of Computer Sciences and Engineering* 6.10 (oct. de 2018), págs. 722-729. DOI: 10.26438/ijcse/v6i10.722729.

[15] Jee W. Choi, Amik Singh y Richard W. Vuduc. "Model-driven autotuning of sparse matrix-vector multiply on GPUs". En: *ACM SIGPLAN Notices* 45.5 (mayo de 2010), págs. 115-126. DOI: 10.1145/1837853.1693471.

[16] *cuBLAS :: CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/cublas/index.html (Accedido el 17-08-2020).

[17] *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (Accedido el 13-08-2020).

[18] *cuSPARSE :: CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/cusparse/index.html (Accedido el 17-08-2020).

[19] E. Cuthill y J. McKee. "Reducing the bandwidth of sparse symmetric matrices". En: *Proceedings of the 1969 24th national conference*. ACM Press, 1969, págs. 157-172. DOI: 10.1145/800195.805928.

[20] T. Davis y E. P. Natarajan. "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems". En: *ACM Trans. Math. Softw.* 37 (2010), 36:1-36:17.

[21] Timothy A. Davis y Yifan Hu. "The university of Florida sparse matrix collection". En: *ACM Transactions on Mathematical Software* 38.1 (nov. de 2011), págs. 1-25. DOI: 10.1145/2049662.2049663.

[22] I. S. Duff. "A survey of sparse matrix research". En: *Proceedings of the IEEE* 65.4 (1977), págs. 500-535. DOI: 10.1109/PROC.1977.10514.

[23] M.J. Ellsworth, L.A. Campbell, R.E. Simons, M.K. Iyengar, R.R. Schmidt y R.C. Chu. "The evolution of water cooling for IBM large server systems: Back to the future". En: *2008 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*. IEEE, mayo de 2008. DOI: 10.1109/itherm.2008.4544279.

[24] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". En: *IEEE Transactions on Computers* C-21.9 (sep. de 1972), págs. 948-960. DOI: 10.1109/tc.1972.5009071.

[25] Alan George. "Computer implementation of the finite element method". Tesis doct. CA, USA: Computer Science Department, School of Humanities y Sciences, STANFORD UNIVERSITY, 1971.

[26] Alan George, Joseph Liu y Esmond Ng. *Computer Solution of Sparse Linear Systems*. 1994.

[27] Alan George y Joseph W.H. Liu. "The Evolution of the Minimum Degree Ordering Algorithm". En: *SIAM Review* 31.1 (mar. de 1989), págs. 1-19. DOI: 10.1137/1031001.

[28] Norman E. Gibbs, Jr. William G. Poole y Paul K. Stockmeyer. "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix". En: *SIAM Journal on Numerical Analysis* 13.2 (abr. de 1976), págs. 236-250. DOI: 10.1137/0713023.

[29] Philip E. Gill, Walter Murray, Michael A. Saunders y Margaret H. Wright. "Sparse Matrix Methods in Optimization". En: *SIAM Journal on Scientific and Statistical Computing* 5.3 (sep. de 1984), págs. 562-589. DOI: 10.1137/0905041.

[30] Fritz Goebel, Hartwig Anzt, Terry Cojean, Goran Flegar y Enrique S. Quintana-Ortí. "Multiprecision Block-Jacobi for Iterative Triangular Solves". En: *Euro-Par 2020: Parallel Processing*. Springer International Publishing, 2020, págs. 546-560. DOI: 10.1007/978-3-030-57675-2_34.

[31] Gene Golub. *Matrix computations*. Baltimore: Johns Hopkins University Press, 1996. ISBN: 978-0801854149.

[32] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis y Nectarios Koziris. "Understanding the Performance of Sparse Matrix-Vector Multiplication". En: *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. IEEE, feb. de 2008. DOI: 10.1109/pdp.2008.41.

[33] Thomas Grützmacher, Terry Cojean, Goran Flegar, Hartwig Anzt y Enrique S. Quintana-Ortí. "Acceleration of PageRank with Customized Precision Based on Mantissa Segmentation". En: *ACM Transactions on Parallel Computing* 7.1 (abr. de 2020), págs. 1-19. DOI: 10.1145/3380934.

[34] Thomas Grützmacher, Terry Cojean, Goran Flegar, Fritz Göbel y Hartwig Anzt. "A customized precision format based on mantissa segmentation for accelerating sparse linear algebra". En: *Concurrency and Computation: Practice and Experience* 32.15 (jul. de 2019). DOI: 10.1002/cpe.5418.

[35] Dahai Guo, William Gropp y Luke N Olson. "A hybrid format for better performance of sparse matrix-vector multiplication on a GPU". En: *The International Journal of High Performance Computing Applications* 30.1 (jul. de 2015), págs. 103-120. DOI: 10.1177/1094342015593156.

[36] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski y Trevor Mudge. "Sparse-TPU: adapting systolic arrays for sparse matrices". En: *Proceedings of the 34th ACM International Conference on Supercomputing*. ACM, jun. de 2020. DOI: 10.1145/3392717.3392751.

[37] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh y P. Sadayappan. "Adaptive sparse tiling for sparse matrix multiplication". En: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. ACM, feb. de 2019. DOI: 10.1145/3293883.3295712.

[38] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". En: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, feb. de 2014. DOI: 10.1109/isscc.2014.6757323.

[39] Norman P. Hummon y Patrick Doreian. "Computational methods for social network analysis". En: *Social Networks* 12.4 (dic. de 1990), págs. 273-288. DOI: 10.1016/0378-8733(90)90011-w.

[40] Anil K. Jain. "Data clustering: 50 years beyond K-means". En: *Pattern Recognition Letters* 31.8 (jun. de 2010), págs. 651-666. DOI: 10.1016/j.patrec.2009.09.011.

[41] David B. Kirk y Wen-mei W. Hwu. *Programming Massively Parallel Processors*. Elsevier, 2013. DOI: 10.1016/c2011-0-04129-7.

[42] Kornilios Kourtis, Georgios Goumas y Nectarios Koziris. "Optimizing sparse matrix-vector multiplication using index and value compression". En: *Proceedings of the 2008 conference on Computing frontiers - CF '08*. ACM Press, 2008. DOI: 10.1145/1366230.1366244.

[43] Anderson Kuei-An Ku, Jenny Yi-Chun Kuo y Jingling Xue. "Hardware Support for Efficient Sparse Matrix Vector Multiplication". En: *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. IEEE, dic. de 2008. DOI: 10.1109/euc.2008.154.

[44] Y Ma, H Noda, I Ito y A Nishihara. "Modified delta encoding and its applications to speech signal". En: *TENCON 2010 - 2010 IEEE Region 10 Conference*. IEEE, nov. de 2010. DOI: 10.1109/tencon.2010.5686714.

[45] Marco Maggioni y Tanya Berger-Wolf. "AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs". En: *2013 42nd International Conference on Parallel Processing*. IEEE, oct. de 2013. DOI: 10.1109/icpp.2013.10.

[46] Marco Maggioni y Tanya Berger-Wolf. "CoAdELL: Adaptivity and Compression for Improving Sparse Matrix-Vector Multiplication on GPUs". En: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, mayo de 2014. DOI: 10.1109/ipdpsw.2014.106.

[47] Alexander Monakov, Anton Lokhmotov y Arutyun Avetisyan. "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures". En: *High Performance Embedded Architectures and Compilers*. Springer Berlin Heidelberg, 2010, págs. 111-125. DOI: 10.1007/978-3-642-11515-8_10.

[48] *Number of nonzero matrix elements - MATLAB nnz*. (Accedido el 15-07-2020).

[49] Lawrence Page, Sergey Brin, Rajeev Motwani y Terry Winograd. "The PageRank Citation Ranking: Bringing Order to the Web". En: (nov. de 1998).

[50] Ch. H. Papadimitriou. "The $\mathcal{NP}$-Completeness of the bandwidth minimization problem". En: *Computing* 16.3 (sep. de 1976), págs. 263-270. DOI: 10.1007/bf02280884.

[51] Horacio Peréz-Sánchez, Afshin Fassihi, José M. Cecilia, Hesham H. Ali y Mario Cannataro. "Applications of High Performance Computing in Bioinformatics, Computational Biology and Computational Chemistry". En: *Bioinformatics and Biomedical Engineering*. Ed. por Francisco Ortuño e Ignacio Rojas. Cham: Springer International Publishing, 2015, págs. 527-541. ISBN: 978-3-319-16480-9.

[52] Ali Pinar y Michael T. Heath. "Improving performance of sparse matrix-vector multiplication". En: *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '99*. ACM Press, 1999. DOI: 10.1145/331532.331562.

[53] Istvan Reguly y Mike Giles. "Efficient sparse matrix-vector multiplication on cache-based GPUs". En: *2012 Innovative Parallel Computing (InPar)*. IEEE, mayo de 2012. DOI: 10.1109/inpar.2012.6339602.

[54] G Reinelt. *The Traveling Salesman : Computational Solutions for TSP Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994. ISBN: 978-3-540-48661-9.

[55] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial y Applied Mathematics, ene. de 2003. DOI: 10.1137/1.9780898718003.

[56] S. W. Sloan. "An algorithm for profile and wavefront reduction of sparse matrices". En: *International Journal for Numerical Methods in Engineering* 23.2 (feb. de 1986), págs. 239-251. DOI: 10.1002/nme.1620230208.

[57] Steven Smith. *The scientist and engineer's guide to digital signal processing*. San Diego, Calif: California Technical Pub, 1997. ISBN: 0966017641.

[58] *SuiteSparse Matrix Collection*. URL: https://sparse.tamu.edu/about (Accedido el 12-02-2020).

[59] Xiangzheng Sun, Yunquan Zhang, Ting Wang, Xianyi Zhang, Liang Yuan y Li Rao. "Optimizing SpMV for Diagonal Sparse Matrices on GPU". En: *2011 International Conference on Parallel Processing*. IEEE, sep. de 2011. DOI: 10.1109/icpp.2011.53.

[60] Wai Teng Tang, Wen Jun Tan, Rajarshi Ray, Yi Wen Wong, Weiguang Chen, Shyh-hao Kuo, Rick Siow Mong Goh, Stephen John Turner y Weng-Fai Wong. "Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes". En: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, nov. de 2013. DOI: 10.1145/2503210.2503234.

[61] Lucia Catabriga Thiago Nascimento Rodrigues Maria Claudia Silva Boeres. "A Parallel Sloan Algorithm for the Profile Minimization Problem of Sparse Matrices". En: *XLIX Simpósio Brasileiro de Pesquisa Operacional* (ago. de 2017).

[62] *Tim Davis*. URL: https://people.engr.tamu.edu/davis/welcome.html (Accedido el 15-02-2020).

[63] Wanqing Tu, Florin Pop, Weijia Jia, Jie Wu y Mauro Iacono. "High-Performance Computing in Edge Computing Networks". En: *Journal of Parallel and Distributed Computing* 123 (ene. de 2019), pág. 230. DOI: 10.1016/j.jpdc.2018.10.014.

[64] Jeremiah Willcock y Andrew Lumsdaine. “Accelerating sparse matrix computations via data compression”. En: *Proceedings of the 20th annual international conference on Supercomputing - ICS ’06*. ACM Press, 2006. DOI: 10.1145/1183401.1183444.

[65] Shiming Xu, Hai Xiang Lin y Wei Xue. “Sparse Matrix-Vector Multiplication Optimizations based on Matrix Bandwidth Reduction using NVIDIA CUDA”. En: *2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science*. IEEE, ago. de 2010. DOI: 10.1109/dcabes.2010.162.

[66] Shengen Yan, Chao Li, Yunquan Zhang y Huiyang Zhou. “yaSpMV”. En: *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP ’14*. ACM Press, 2014. DOI: 10.1145/2555243.2555255.

ANEXOS

Anexo 1

Algoritmo evolutivos

En la computación científica se pueden encontrar numerosos problemas con diferentes complejidades. De forma genérica, estos pueden ser clasificados como problemas "sencillos" y problemas "difíciles". Esta clasificación depende de la evaluación de la complejidad de los algoritmos capaces de resolver los problemas en diferentes casos. Formalmente un problema es "sencillo", si puede ser resuelto en tiempo polinomial en una computadora determinística. A este tipo de problemas se los llama de clase $\mathcal{P}$. Un problema "difícil" o que pertenece a la clase $\mathcal{NP}$, es aquel que puede ser resuelto en tiempo polinomial pero en una computadora no determinística. (Estos problemas una vez obtenida la solución, son fáciles de verificar, pero difíciles de encontrar.) En 1971 se planteó la pregunta ¿si $\mathcal{P}$=$\mathcal{NP}$? Al día de hoy no se ha podido demostrar si se cumple, se cree que $\mathcal{P} \neq \mathcal{NP}$, siendo uno de los principales desafíos en el campo de la computación.

Problemas $\mathcal{NP}$, son la clase de problemas "difíciles" de resolver. Un ejemplo muy común y conocido es el "Travelling Salesperson Problem" (TSP) . Este consiste, básicamente, en encontrar un una permutación que represente el recorrido de una serie de ciudades (vértices) conectadas entre si (aristas), de tal forma que todas sean visitadas (sólo una vez), minimizando la distancia total viajada. A continuación, con el objetivo de observar cuan difícil puede ser resolver este problema de manera óptima. Considerando n ciudades, la dimensión del espacio de búsqueda (permutaciones) es: $(n-1)!/2$, entonces:

- Para $n = 10$, hay 181.440 permutaciones posibles.
- Para $n = 12$ hay 19.958.400 permutaciones posibles.
- Para $n = 20$ hay 60.822.550.204.416.000 permutaciones posibles.

Estos ejemplos evidencian de forma clara, la inviabilidad de buscar una solución por fuerza bruta o una búsqueda exhaustiva. Entonces, para este tipo de problemas, es necesario utilizar algoritmos de resolución más eficientes que la búsqueda exhaustiva.

Una clase particular de problemas $\mathcal{NP}$ son los problemas de optimización, los que tienen por objetivo hallar la(s) solucione(s) óptima(s) de un problema (de acuerdo a una función objetivo determinada).

Existen varias técnicas para "resolver" estos problemas, entre ellas: analíticas, métodos exactos (backtracking, programación dinámica, método simplex, etc.), métodos de aproximación, métodos aleatorios y las que se describen con más profundidas en esta sección, heurísticas y metaheurísticas. Las heurísticas son métodos de resolución basados en procedimientos conceptualmente simples para encontrar soluciones de buena calidad (no necesariamente hallan la solución óptima) a problemas difíciles, de un modo sencillo y eficiente.

Entonces, por ejemplo, el problema de reordenar una matriz de modo de reducir el ancho de banda (u otra formulación donde se busca la cantidad mínima de diagonales) interpretando la matriz como una matriz de adyacencia, i.e. encontrar un reordenamiento que cumpla lo esperado, es equivalente al problema de dar una permutación de los vértices del grafo asociado a la matriz, de forma de minimizar la función ancho de banda. Como prueba Papadimitriou [50] y discutido brevemente en la Sección 2.4, este es un problema $\mathcal{NP}$-completo.

La computación evolutiva engloba un amplio conjunto de técnicas que siguen un mecanismo análogo a los procesos de evolución natural, permite resolver problemas "difíciles", similares al planteado, en tiempos y costos computacionales razonables.

Un Algoritmo Evolutivo (AE) trabaja sobre una población (P) de individuos que representan potenciales soluciones al problema a resolver. En el Algoritmo 4 se presenta un esquema general en alto nivel de como trabaja un AE. Formalmente, la representación del individuo es el genotipo, la solución el fenotipo. Otro concepto importante presente en estos algoritmos, es una función de *fitness*, que evalúa los individuos de acuerdo a su adaptación para la resolución del problema.

La estructura general de un AE consiste en un ciclo que conformado por cuatro etapas.

1. **Evaluación**: se asigna un valor de *fitness* a cada individuo.

Algoritmo 4: Esquema genérico de un AE que trabaja sobre una población P

```
Inicializar(P(0));
generacion = 0;
mientras (no CriterioParada) hacer
    Evaluar(P(generacion));
    Padres = Seleccionar(P(generacion));
    Hijos = Aplicar Operadores Evolutivos(Padres);
    NuevaPoblacion = Remplazar(Hijos,P(generacion));
    generacion++;
    P(generacion) = NuevaPoblacion;
fin
retornar Mejor Solucion Encontrada
```

2. **Selección**: se determinan candidatos adecuados para la generación de la nueva generación.
3. **Aplicación de los operadores evolutivos**: se genera un conjunto de descendientes a partir de los individuos seleccionados, mediante operadores que emulan la evolución natural.
4. **Reemplazo**: mecanismo que realiza el recambio generacional.

La población P es inicializada, en general, mediante mecanismos aleatorios o guiado por heurísticas específicas. La aplicación de distintas políticas de selección y reemplazo permiten definir las características del algoritmo evolutivo. Mediante la evaluación de la población basados en su valor de *fitmess* se puede privilegiar los individuos más adaptados (elitismo), aumentar la presión selectiva, incrementar la diversidad genética, etc. Los operadores evolutivos determinan el mecanismo de exploración del espacio de búsqueda del problema, probablemente los más conocidos son los operadores de mutación y recombinación, la utilización de estos establece el tipo de AE. Finalmente, la condición de parada determina la finalización del AE, que puede estar basada en el número de generaciones, variación de valores de fitness, estimaciones del error cometido, entre otras.