MÉTODOS NUMÉRICOS PARA LA COMPUTACIÓN

Tema 6: Matrices dispersas

2020/21

---

16 de diciembre de 2020

Grupo 03: José María Amusquívar Poppe y Prashant Jeswani Tejwani

Universidad de Las Palmas de Gran Canaria

Escuela de Ingeniería en Informática

Índice

Actividad práctica 1 3
Ejercicio 1 3
Ejercicio 2 4
Ejercicio 3 (optativo) 6
Ejercicio 4 (optativo) 9
Actividad práctica 2 10
Ejercicio 1 10
CSR 10
COO 11
Ejercicio 2 (optativo) 12
SKY LINE 12
CSC 14
Actividad práctica 3 15
Ejercicio 1 (optativo) 15
Referencias 20

Actividad práctica 1

Se realiza la codificación COO y CSR en Matlab.

Ejercicio 1

Se implementa dos funciones en Matlab que devuelvan la codificación COO y CSR para una matriz que se le pase como entrada.

```
function [row, col, val] = COO(A)
    n = nnz(A); % number of nonzero
    row = zeros(1,n);
    col = zeros(1,n);
    val = zeros(1,n);

    k = 1;
    for i = 1:size(A,1)
        for j = 1:size(A,2)
            if A(i,j) ~= 0
                row(k) = i;
                col(k) = j;
                val(k) = A(i,j);
                k = k + 1;
            end
        end
    end
end
```

Figura 1. Codificación COO

```
function [rowOff, col, val] = CSR(A)
    n = nnz(A); % number of nonzero
    nz = 0;
    rowOff = zeros(1,size(A,1)+1);
    col = zeros(1,n);
    val = zeros(1,n);

    k = 1;
    l = 1;
    if A(1,1) ~= 0
        rowOff(1) = 0;
        l = 2;
    end

    for i = 1:size(A,1)
        for j = 1:size(A,2)
            if A(i,j) ~= 0
                col(k) = j;
                val(k) = A(i,j);
                k = k + 1;
                nz = nz + 1;
            end
        end
        rowOff(l) = nz;
        l = l + 1;
    end
end
```

Figura 2. Codificación CSR

Para el método *COO* se obtiene el número de valores no nulos de la matriz mediante *nnz()* y se inicializan los vectores a cero. A continuación, se recorre la matriz con el fin de buscar los valores no nulos, a medida que se vayan encontrando se almacena el índice de fila en el vector de filas, el índice de columnas en el vector de columnas y el valor no nulo en el vector de valores.

Para el método *CSR* se obtiene el número de valores no nulos de la matriz mediante *nnz()* y se inicializan los vectores a cero (en este caso, la dimensión del vector de filas será el número de filas + 1 y se comprueba si el primer elemento de la matriz no es nulo). A continuación, se recorre la matriz con el fin de buscar los valores no nulos, a medida que se vayan encontrando se almacena el índice de columnas en el vector de columnas y el valor no nulo en el vector de valores. Al terminar de recorrer una fila se almacena en el vector de filas el número de elementos no nulos de la fila.

Como ejemplo utilizaremos la siguiente matriz → A = $\begin{matrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{matrix}$

```
disp("Matriz dispersa: ");
A = [1,7,0,0;0,2,8,0,;5,0,3,9;0,6,0,4]
disp("---------------------------------");

[row, col, val] = COO(A);
disp("COO a la matriz A:");
disp("Vector de filas: ");
disp(row);
disp("Vector de columnas: ");
disp(col);
disp("Vector de valores: ");
disp(val);
disp("---------------------------------");

[row, col, val] = CSR(A);
disp("CSR a la matriz A:");
disp("Vector de filas: ");
disp(row);
disp("Vector de columnas: ");
disp(col);
disp("Vector de valores: ");
disp(val);
```

Figura 3. Ejemplo de codificación

```
Matriz dispersa:

A =

     1     7     0     0
     0     2     8     0
     5     0     3     9
     0     6     0     4

---------------------------------
COO a la matriz A:
Vector de filas:
     1     1     2     2     3     3     3     4     4

Vector de columnas:
     1     2     2     3     1     3     4     2     4

Vector de valores:
     1     7     2     8     5     3     9     6     4

---------------------------------
CSR a la matriz A:
Vector de filas:
     0     2     4     7     9

Vector de columnas:
     1     2     2     3     1     3     4     2     4

Vector de valores:
     1     7     2     8     5     3     9     6     4
```

Figura 4. Resultado obtenido

Ejercicio 2

Se programa un generador de matrices aleatorias escasas en Matlab, tomando como entrada la densidad deseada de valores nulos para la matriz y el rango de valores.

```
M = generateSparse(4,4,8,1,50)
```

Figura 5. Ejemplo de matriz dispersa

```
function [A] = generateSparse(nRow, nCol, zDensity, vMin, vMax)
    if zDensity > nRow * nCol
        disp("La densidad de ceros debe ser menor que nFilas * nColumnas");
        A = zeros(1,1);
        return
    end

    if zDensity <= 0
        disp("La densidad de ceros debe ser mayor que 0");
        A = zeros(1,1);
        return
    end

    if nRow <= 0 || nCol <= 0
        disp("La dimensión debe ser al menos 1x1");
        A = zeros(1,1);
        return
    end

    if vMin <= 0 || vMax <= vMin
        disp("Los valores deben ser mayores que 0");
        A = zeros(1,1);
        return
    end

    A = randi([vMin vMax],nRow,nCol);

    for i = 1:zDensity
        posRow = randi([1 nRow],1,1); % position row in matrix
        posCol = randi([1 nCol],1,1); % position col in matrix

        if A(posRow, posCol) ~= 0
            A(posRow, posCol) = 0;
        else
            while A(posRow, posCol) == 0
                posRow = randi([1 nRow],1,1);
                posCol = randi([1 nCol],1,1);
            end
            A(posRow, posCol) = 0;
        end
    end
end
```

Figura 6. Generador de matrices dispersas

El método toma los siguientes parámetros de entrada: *generateSparse* (nº de filas, nº de columnas, nº de valores nulos, rango de valores inferior, rango de valores superior). Primero se comprueba los parámetros que sean válidos y a continuación se genera una matriz de valores aleatorios del rango pasado por parámetro. Luego se itera tantas veces como números de ceros indicado, se genera 2

```
M =

    39     0    14     6
    40    33    34     0
     0     0     0    48
     0     0     9     0
```

Figura 7. Resultado obtenido

números aleatorios entre el rango de la dimensión de la matriz (los cuales será una posición aleatoria de la matriz) y se almacena un 0 en la posición.

Ejercicio 3 (optativo)

A continuación, se caracteriza el ahorro promedio que se produce en almacenamiento en función de los diferentes valores de densidad del ejercicio 2. Para ello se crean matrices de 10x10 (y más adelante de 100x100) con densidades ascendentes hasta el número de elementos de la matriz menos uno (ya que sino en la última iteración se obtendría una matriz nula). Se pasa la misma codificar la misma matriz mediante COO y CSR, y a continuación se obtiene el número de bytes que ocupa la matriz (sin codificar), la matriz codificada en COO y CSR (se suman los bytes que ocupan los tres vectores) mediante la función *whos* de MATLAB.

```
fid = fopen('sizes10x10.txt','wt');
for i=1:(10*10)-1
    disp("Matriz 10x10 con nº de ceros = " + i);
    A = generateSparse(10,10,i,1,50);
    [rowCOO, colCOO, valCOO] = COO(A);
    [rowCSR, colCSR, valCSR] = CSR(A);
    a = whos('A').bytes;
    coo = whos('rowCOO').bytes + whos('colCOO').bytes + whos('valCOO').bytes;
    csr = whos('rowCSR').bytes + whos('colCSR').bytes + whos('valCSR').bytes;

    % write in file
    fprintf(fid, '%d,%d,%d\n', a, coo, csr);

    disp("Bytes A = " + a);
    disp("Bytes COO = " + coo);
    disp("Bytes CSR = " + csr);
    disp("-----------------------------------");
end
fclose(fid);
```

Figura 8. Cálculo de los bytes para distintas densidades según la codificación

Estos valores se escriben en un fichero en el siguiente formato separados por comas:

```
1   800,2376,1672
2   800,2352,1656
3   800,2328,1640
4   800,2304,1624
5   800,2280,1608
6   800,2256,1592
7   800,2232,1576
8   800,2208,1560
9   800,2184,1544
10  800,2160,1528
11  800,2136,1512
12  800,2112,1496
```

Figura 9. Formato del fichero de salida

A continuación, se grafican los resultados obtenidos para analizar el ahorro que se produce en almacenamiento en función de los diferentes valores de densidad.

```
clc;
clear;

data = load('sizes10x10.txt');
dim = size(data);
a = data(:, 1);
coo = data(:, 2);
csr = data(:, 3);

figure(1)
subplot(1,2,1);
plot([1:99], a);
hold on
plot([1:99], coo);
plot([1:99], csr);
title("Dimensión de la matriz de 10x10");
ylabel("Tamaño (bytes)");
xlabel("Nº de ceros");
legend('Matriz sin codificar','COO', 'CSR', 'Location', 'northeast');
hold off

data = load('sizes100x100.txt');
dim = size(data);
a = data(:, 1);
coo = data(:, 2);
csr = data(:, 3);

figure(1)
subplot(1,2,2);
plot([1:9999], a);
hold on
plot([1:9999], coo);
plot([1:9999], csr);
title("Dimensión de la matriz de 100x100");
ylabel("Tamaño (bytes)");
xlabel("Nº de ceros");
legend('Matriz sin codificar','COO', 'CSR', 'Location', 'northeast');
hold off
```

Figura 10. Fichero para obtener las gráficas

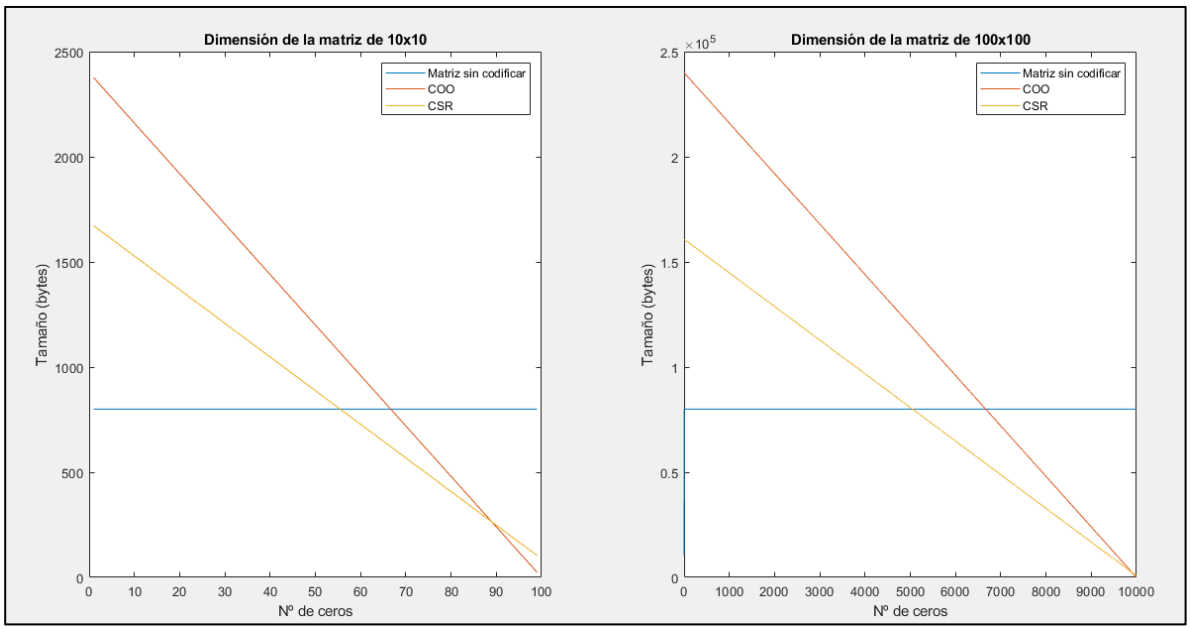

Figura 11. Gráficas obtenidas para matrices de tamaño 10x10 y 100x100

Observamos que ambas gráficas para distintos tamaños de matrices son iguales, pero en distinta escala. Se observa que para matrices de dimensión 10x10 el número de bytes sin codificar es constante (800 bytes y para dimensión de 100x100 el almacenamiento es de 80.000 bytes) mientras que para COO y CSR es decreciente a medida que aumenta el número de ceros en la matriz. Además, después de que la matriz esté formada por un 90% de ceros la codificación COO necesita menos almacenamiento que la CSR. Esto es así ya que, cuando hay un número muy pequeño de valores no nulos, la dimensión del vector de filas de la codificación COO será menor al vector de filas de la codificación CSR la cual es siempre es constante (número de filas + 1).

Por lo que podemos decir que a partir de un umbral merecerá la pena codificar una matriz que esté formada por un número de ceros significativo. Por ejemplo, para el caso de la matriz de dimensión 10x10, cuando más de 55% de la matriz esté formada por ceros se ahorrará almacenamiento si se codifica mediante CSR y cuando sea mayor que 70% mediante COO.

Ejercicio 4 (optativo)

Se implementa un generador de matrices aleatorias escasas simétricas. Para ello se sigue prácticamente la misma manera de implementar el ejercicio 2 aunque los parámetros son ligeramente distintos. Las matrices simétricas deben ser cuadradas por lo que se le pasa como parámetros de entrada: *generateSymmetricSparse* (dimensión de la matriz, nº de valores únicos no nulos, rango de valores inferior, rango de valores superior). Se comprueban estos parámetros al principio del método.

```
clc;
clear;

% method(dimension, number of nonzeros values, lower range, upper range)
A = generateSymmetricSparse(4,3,1,10)


function [A] = generateSymmetricSparse(n, zDensity, vMin, vMax)
    if zDensity > n * n
        disp("La densidad de ceros debe ser menor que nFilas * nColumnas");
        A = zeros(1,1);
        return
    end

    if zDensity <= 0
        disp("La densidad de ceros debe ser mayor que 0");
        A = zeros(1,1);
        return
    end

    if n <= 0
        disp("La dimensión debe ser al menos 1x1");
        A = zeros(1,1);
        return
    end

    A = zeros(n,n);

    for i = 1:zDensity
        value = randi([vMin vMax],1,1); % value
        posRow = randi([1 n],1,1); % position row in matrix
        posCol = randi([1 n],1,1); % position col in matrix

        if A(posRow, posCol) == 0
            A(posRow, posCol) = value;
            A(posCol, posRow) = value;
        else
            while A(posRow, posCol) ~= 0
                posRow = randi([1 n],1,1);
                posCol = randi([1 n],1,1);
            end
            A(posRow, posCol) = value;
            A(posCol, posRow) = value;
        end
    end
end
```

Figura 12. Generador de matrices dispersas simétricas

Se inicializa una matriz de ceros y se itera tantas veces como el número de valores no nulos se pasa como parámtero. Se genera una posición aleatoria para almacenar un valor (también aleatorio) en dos posiciones que no hayan valores nulos. Se muestra el resultado obtenido con una dimensión de matriz 4x4, tres valores únicos no nulos con valores en un rango de 1-10:

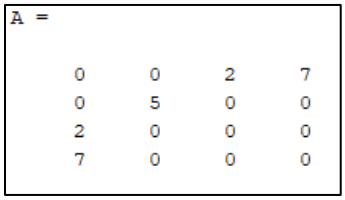

Figura 13. Resultado obtenido

Actividad práctica 2

Se realiza la codificación dispersa con MKL en C++. La matriz a utilizar en esta práctica es la misma que se ha utilizado en el primer apartado (Figura 13).

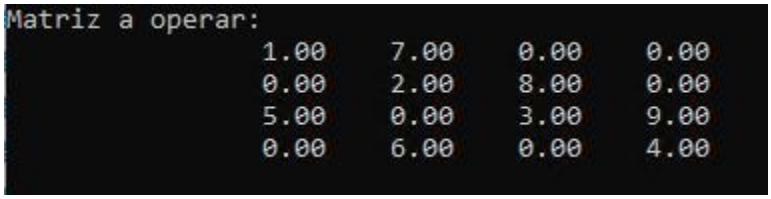

Figura 14. Matriz a comprimir usando los distintos formatos.

Ejercicio 1

Para poder realizar las operaciones de compresión “CSR” y “COO”, se ha empleado funciones de la librería “MKL”.

CSR

El primero que se ha calculado es la compresión “CSR”, empleando para ello la función:

```
#pragma warning(suppress : 4996)
mkl_ddnscsr(job, &m, &n, matrixOver, &lda, valuesOutCSR, ja, ia, &info);
```

Figura 15. Función que comprime una matriz escasa en formato "CSR".

De la figura anterior se puede apreciar el uso de la instrucción “pragma warning()”. Esta función se ha utilizado para desactivar una advertencia de “Visual Studio”, que informaba que la función usada debajo estaba en desuso, por lo que era imposible ejecutar el código sin desactivarlo.

Para que esta función se ejecute correctamente, es necesario pasarle como parámetros una serie de datos como las dimensiones de la matriz “m” y “n” (usando “&” para pasarle su dirección de memoria), “matrixOver” corresponde con la propia matriz escasa acompañado de su “lda”, los siguientes tres parámetros se corresponden con variables donde se almacenarán los datos resultados, “valuesOutCSR” es el vector donde se guardarán los valores distintos de cero de la matriz, “ja” e “ia” también son vectores que almacenarán los índices de las columnas donde se sitúan los elementos distinto de cero, y el número de elementos distinto de cero que contiene cada fila, respectivamente; y el último parámetro de la función es un entero que guarda el resultado de la llamada, tanto si ha sido exitosa como si ha sido errónea.

El parámetro “job” es el más especial de todos estos, pues éste es un vector de enteros de 6 elementos, en el que cada uno especifica una configuración de compresión. El primer elemento especifica si se desea convertir de matriz escasa a formato CSR (un 0) o viceversa (un 1); el segundo y tercer elemento especifica si los índices de las matrices empiezan en 0 (un 0) o en 1 (un 1); el cuarto elemento configura la porción de matriz almacenada, en este caso un 2 especifica que se desea almacenar toda la matriz; el quinto elemento recibe el número de elementos distinto de cero de la matriz (“nnz” fue calculado con una función específica); y el último elemento configura si se desea almacenar los resultados en los vectores “ja” e “ia”.

```
funcion2.initJOB(new int[6] {0, 0, 0, 2, nnz, 1});
```

Figura 16. Inicialización del vector "job" para el formato "CSR".

Destacar que, dado que el vector “job” varía según el tipo de compresión que se realice, se ha utilizado un método que se encarga de modificar el vector interno “job” con los nuevos valores, justo antes de llamar a cualquier método de compresión (Figura 15).

Una vez se ha llamado al método de la figura 14, se procede a imprimir los valores de los tres vectores resultado para comprobar que son correctos:

```
CSR
        El vector de valores es:
                1.00, 7.00, 2.00, 8.00, 5.00, 3.00, 9.00, 6.00, 4.00

        El vector de los indices de columnas es:
                0, 1, 1, 2, 0, 2, 3, 1, 3

        El vector de suma de los elementos no-nulos(offset) es:
                0, 2, 4, 7, 9
```

Figura 17. Vectores resultados de la compresión "CSR".

COO

El segundo tipo de compresión que se ha realizado es el “COO”, utilizando para ello la función:

```
#pragma warning(suppress : 4996)
mkl_dcsrcoo(job, &n, valuesOutCSR, ja, ia, &nnz, valuesOutCOO, rows, cols, &info);
```

Figura 18. Función que comprime una matriz escasa en formato "COO".

Algo que destacar de esta función es que, para poder llamarla, primero se ha de realizar la compresión “CSR”, obteniendo así sus tres vectores resultados. Pues, al igual que esta función”, ésta recibe una serie de parámetros similares tales como la dimensión de la matriz (“n”); los tres vectores obtenidos mediante la compresión “CSR”; el número de elementos no nulos de la matriz (“nnz”); un vector que almacenará los valores distintos de cero (“valuesOutCOO”), por lo que este vector debe ser análogo al vector “valuesOutCSR”; también recibe dos vectores que almacenarán las coordenadas de estos últimos valores, “rows” las filas y “cols” las columnas, por lo que el vector “cols” debe ser igual que el vector “ja”; y también recibe un entero con el código de salida.

En este tipo de compresión, el vector “job” es parecido al vector utilizado en “CSR”, con la única diferencia que en la última posición del vector debe situarse un 3, con esto se consigue que los tres vectores sean escritos con los datos obtenidos.

```
funcion2.initJOB(new int[6]{0, 0, 0, 2, nnz, 3});
```

Figura 19. Inicialización del vector "job" para el formato "COO".

Una vez procesada esta llamada, se procede a imprimir los resultados y verificar que estos sean adecuados y correctos:

```
COO
        El vector de valores es:
                1.00, 7.00, 2.00, 8.00, 5.00, 3.00, 9.00, 6.00, 4.00

        El vector de los indices de columnas es:
                0, 1, 1, 2, 0, 2, 3, 1, 3

        El vector de los indices de filas es:
                0, 0, 1, 1, 2, 2, 2, 3, 3
```

Figura 20. Vectores resultados de la compresión "COO". Similares a los obtenidos en "CSR".

Ejercicio 2 (optativo)

Para este apartado se ha realizado la conversión a dos tipos distintos, empleando la misma metodología que el apartado anterior.

SKY LINE

Para realizar este apartado, se ha empleado la función de “MKL”:

```
#pragma warning(suppress : 4996)
mkl_dcsrsky(job, &n, valuesOutCSR, ja, ia, asky, pointers, &info);
```

Figura 21. Función que comprime una matriz escasa en un formato “SKY LINE”.

Al igual la compresión “COO”, este tipo de compresión requiere haber realizado una compresión a formato “CSR” antes. Los parámetros son similares a los explicados con anterioridad, con la diferencia que los resultados de esta conversión se almacenan en dos vectores, “asky” y “pointers”. Antes de explicar estos dos vectores, se debe explicar la configuración que contiene el vector “job”, el cual tiene la siguiente forma:

```
printf("SKY LINE");
printf("\nMatriz inferior convertida");
funcion2.initJOB(new int[6]{ 0, 0, 0, 0, nnz, 0 }); //Upper matrix.
funcion2.SKY();
printf("\nMatriz superior convertida");
funcion2.initJOB(new int[6]{ 0, 0, 0, 1, nnz, 0 }); //Lower matrix.
funcion2.SKY();
```

Figura 22. Inicialización del vector "job" para conversión a formato "SKY LINE".

El vector “job” tiene la misma estructura que los mencionados anteriormente con la única diferencia que el cuarto elemento especifica si se quiere comprimir el triángulo superior de la matriz (un 0) o el inferior (un 1); y, además, para almacenar todos los resultados en los dos vectores, el último elemento debe ser igual a 0.

Pues bien, esta función devuelve dos vectores, si el cuarto elemento de “job” es igual a 0, el vector “asky” guardará aquellos valores distintos de cero que se encuentren en, y por encima de, la diagonal principal, recorriéndolos por columnas, de arriba hacia abajo, y respetando aquellos ceros que se encuentren entre valores no nulos. Entonces, el vector “pointers” representa todos los índices de aquel primer elemento de cada nueva columna almacenada en el vector “asky”. Por tanto, si el cuarto elemento del vector “job” es igual a 1, entonces se almacena en el vector “asky” aquellos valores no nulos que se encuentren en, y por debajo, de la diagonal principal, además de su respectivo vector “pointers”; por ello se realizan dos llamadas a la misma función (Figura 19).

Por ejemplo, teniendo la siguiente matriz de ejemplo:

$$c = \begin{bmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix}$$

Figura 23. Matriz escasa de ejemplo para compresión "SKY LINE".

Se obtiene que sus vectores para el triángulo superior son (con indexación igual 1):

```
values  = ( 1  -2   5   4  -4   0   2   7   8   0   0  -5 )
pointers = ( 1   2   4   5   9   13 )
```

Y para el triángulo inferior (con indexación igual 1):

```
values   = ( 1  -1   5  -3   0   4   6   7  4   0   -5 )
pointers = ( 1   2   4   7   9   12 )
```

Una vez llamada la función, se procede a imprimir los resultados y corroborar su validez:

```
SKY LINE
Matriz inferior convertida
        El vector de valores es:
                1.00, 2.00, 5.00, 0.00, 3.00, 6.00, 0.00, 4.00

        El vector de punteros es:
                0, 1, 2, 5, 8

Matriz superior convertida
        El vector de valores es:
                1.00, 7.00, 2.00, 8.00, 3.00, 9.00, 4.00

        El vector de punteros es:
                0, 1, 3, 5, 7
```

Figura 24. Vectores resultados de la compresión inferior y superior mediante "SKY LINE".

CSC

Este tipo de conversión emplea el mismo mecanismo que la conversión CSR, ya que simplemente, en lugar de recorrer la matriz por filas, se recorre por columnas, por lo que el orden de almacenamiento es distinto. La función que se emplea es:

```
#pragma warning(suppress : 4996)
mkl_dcsrcsc(job, &n, valuesOutCSR, ja, ia, valuesOutCSC, ja1, ia1, &info);
```

Figura 25. Función que comprime una matriz escasa en formato "CSC".

Tal como se puede apreciar, los parámetros que recibe esta función son idénticos a los de la función “CSR”, entendiendo que para la función “CSR” se requiere la matriz escasa, mientras que para esta función simplemente los tres vectores resultados de la compresión “CSR”. También, su vector “job” es mucho más simple que cualquier otro, pues requiere de pocas configuraciones:

```
funcion2.initJOB(new int[6]{ 0, 0, 0, 0, 0, 1 });
```

Figura 26. Inicialización del vector "job" para el formaro "CSC".

De este vector, el único elemento distinto a los anteriores vistos es el último, pues éste configura si se desea guardar los tres vectores resultados (un 1) o no (un 0).

A continuación, se representan los resultados de esta conversión para su respectiva verificación:

```
CSC
        El vector de valores es:
                1.00, 5.00, 7.00, 2.00, 6.00, 8.00, 3.00, 9.00, 4.00

        El vector de los indices de filas es:
                0, 2, 0, 1, 3, 1, 2, 2, 3

        El vector de suma de los elementos no-nulos(offset) es:
                0, 2, 5, 7, 9
```

Figura 27. Vectores resultados de la compresión "CSC".

Actividad práctica 3

Ejercicio 1 (optativo)

Se realiza una operación de multiplicación de matrices escasas grandes utilizando BLAS, repitiendo y utilizando *SparseBLAS*. Y, posteriormente, se compara el tiempo para distintas densidades.

En este apartado, se ha desarrollado un método generador de matrices dispersas en C++, al que se le pasa como parámetros el número de filas y columnas, el valor mínimo y máximo que cogerán los elementos no nulos (siempre mayor que cero), y un quinto parámetro que especifica el número de elemento no nulos que tiene la matriz:

```
int m = 1000; int n = 1000;

int zDensityA = x; int vMinA = 10; int vMaxA = 100;
double* matA = utils.genSparseMatrix(m, n, vMinA, vMaxA, zDensityA);
```

Figura 28. Generación de una matriz dispersa con tamaño y rango de valores prefijado.

Por tanto, en la figura anterior, se generará una matriz de 1000 filas por 1000 columnas (cuadrada), con uno rango de valores que variará entre 10 y 100, y una densidad de elementos no nulos "zDensity" variable, empezando en un valor que esté siempre por debajo de "m*n".

La función que genera las matrices dispersas primero comprueba que todos los parámetros sean adecuados, ya que las dimensiones deben ser mayores que 1, la densidad debe ser menor que "m*n", y entre otras comprobaciones. Después de ello, se inicializa el generador de números aleatorios, proporcionado en prácticas anteriores, con el rango deseado, y, entonces, se crea una matriz (vector pero que funciona como matriz) de tamaño "m*n" con todos sus elementos iguales a cero. Y, finalmente, se rellena la matriz con una cantidad de números aleatorios especificada por el parámetro "zDensity".

```
double* genSparseMatrix(int m, int n, int vMin, int vMax, int zDensity) {
        if (m <= 1 || n <= 1) {
                m = 2; n = 2;
        }
        if (zDensity >= m * n) zDensity = (m * n) - 1;
        if (zDensity <= 0) zDensity = 1;
        if (vMin <= 0 || vMax <= vMin) {
                vMin = 1; vMax = 10;
        }

        std::default_random_engine generator;
        std::uniform_real_distribution<double> aleatorio(vMin, vMax);

        double* sparseMat = new double[m * n]{ 0 };
        while (zDensity > 0) {
                int pos = (rand() % (m*n));
                if (sparseMat[pos] == 0.0) {
                        sparseMat[pos] = aleatorio(generator);
                        zDensity--;
                }
        }
        return sparseMat;
}
```

Figura 29. Método que genera una matriz dispersa configurada.

Puesto que se especifica realizar una multiplicación de matrices escasas, se han generado dos matrices de este tipo, cuyo producto será obtenido empleando la función “cblas_dgemm()” perteneciente a la librería “CBLAS”; y también empleando dos funciones de “SparseBLAS”, ya que se comprime tanto en formato “CSR” como en “COO”, estas dos funciones son “mkl_dcsrmm()” y “mkl_dcoomm()”. Las tres funciones mencionadas realizan la operación:

$$Res = alfa * A * B + beta * C$$

Donde “alfa” y “beta” son escalares (1 y 0, respectivamente para realizar “A*B”), las matrices a operar se corresponden con “A” y “B”, la matriz “C”, en este caso”, no es relevante, por lo que se inicializa a cero, además de multiplicarla por un “beta” igual a cero. Remarcar que el resultado de la operación se sobrescribe en la matriz “C”, por tanto “Res” es igual a “C” después de realizar la llamada. Sin embargo, las funciones que operan con matrices comprimidas, sólo pueden trabajar con una matriz comprimida y las otras sin compresión, así que, se ha decidido que la matriz que se comprimirá será la matriz “A”.

Entonces, se procede a calcular los tiempos, para esto se ha desarrollado un bucle que se encargará de generar valores de “zDensity” que van en disminución, empezando en un valor que se encuentre siempre por debajo de “m*n”, y al que se le restará un valor “jump”.

```
int min = 1; int max = 30000; int jump = 100;
Utils utils;
int sizeVects = (max / jump);
funcion3 funcion3(sizeVects); //Inicializa vectores de tiempos.

for (int x = (max*min); x > min; x = x-jump) {
        int m = 1000; int n = 1000;
```

Figura 30. Inicialización de parámetros, y cabecera del bucle inverso.

Como se puede apreciar en la figura anterior, para la realización de estas pruebas, se ha establecido un tamaño fijo para las matrices de 1000 por 1000, por lo que existirán 100000 elementos en la matriz; además se ha establecido que se empiece con una cantidad de 30000 elementos no nulos (suficientes para notar una diferencia), esta cantidad ("zDensity") irá en disminución en cada iteración, restando 100 elementos en cada una.

Una vez generadas las dos matrices dispersas con las configuraciones mencionadas anteriormente, se procede a calcular el número de elementos distintos de cero (se realiza esto más que todo para corroborar que se han generado correctamente), una vez calculado esto, se actualizan los valores de la clase que contiene los métodos y atributos que se encargan de realizar las operaciones.

```
int nnzA = utils.countNNZ(matA, m * n);
int nnzB = utils.countNNZ(matB, m * n);
funcion3.setValues(matA, matB, m, n, nnzA, nnzB);
```

Figura 31. Actualización de los valores de la clase "funcion3" encargada de realizar los cálculos.

Una vez actualizados los atributos, se procede a realizar las respectivas multiplicaciones, empezando con la función de "CBLAS" y seguido de "SparseBLAS" con las compresiones "CSR" y "COO".

```
bool change = true; //Intercambia A*B a B*A (Entonces B es comprimida en lugar de A)
funcion3.blas(change);
funcion3.CSR(); //Comprime las matrices dispersas para realizar las operaciones (CSR).
funcion3.sparseBlasCSR(change);
funcion3.COO(); //Comprime las matrices dispersas para realizar las operaciones (COO).
funcion3.sparseBlasCOO(change);
printf("-----------------------------------------\n");
funcion3.setCont();
```

Figura 32. Llamada a cada una de las funciones de multiplicación y compresión.

Tal como se aprecia en la figura anterior, primero se llama a la función "blas()" que realiza la multiplicación de las dos matrices dispersas sin compresión, posteriormente, se llama a la función "CSR()" que comprime ambas matrices en dicho formato, para luego llamar a la función "sparseBlasCSR()" que realiza la respectiva multiplicación, y lo mismo con la compresión "COO". Algo que destacar de todas estas llamadas, es que algunas de ellas reciben un parámetro booleano

“change”, si este está a verdadero, entonces se realiza la multiplicación “A*B”, si está a falso, entonces se realiza “B*A” (por tanto, se comprimirá sólo “A” o “B”, dado que sólo se comprime la primera matriz). Al final de cada iteración, se llama al método “setCont()” que actualiza el contador que se encarga de indexar el vector de tiempos.

Ahora, se analizará las llamadas realizadas en la figura 31, empezando por la función “blas()”. Esta función realiza una multiplicación de matrices común, por tanto, trata las matrices dispersas como si se tratase de cualquier matriz. La función utilizada es la misma que se explicó en un informe anterior, pues esta recibe parámetros como si se traspone o no las matrices, si se recorre por filas o columnas, sus dimensiones, ambos escalares, y las tres matrices a operar.

```
start = dsecnd();
cblas_dgemm(layout, trans, trans, m, n, k, alpha, matrixA, lda, matrixB, ldb, beta, matrixC, ldc);
fin = (dsecnd() - start);
```

Figura 33. Función de "CBLAS" que realiza una multiplicación de matrices.

Como se puede observar, el cálculo del tiempo empieza antes y termina justo después de la llamada, al igual que en las funciones de multiplicación posteriores. Al final de cada llamada, se guarda el tiempo calculado en un vector de tiempos específico para esta función: `timesBlas[cont] = fin;`

Ahora, se explicará la siguiente llamada que corresponde con “CSR()”, la cual simplemente comprime ambas matrices a este formato, empleando para ello la siguiente instrucción que ya fue explicado en un apartado anterior:

```
void CSR() {
        //Fill three vectors of matrix A.
        #pragma warning(suppress : 4996)
        mkl_ddnscsr(jobA, &m, &n, matrixA, &lda, valuesOutCSRA, jaA, iaA, &info);

        //Fill three vectors of matrix B.
        #pragma warning(suppress : 4996)
        mkl_ddnscsr(jobB, &m, &n, matrixB, &lda, valuesOutCSRB, jaB, iaB, &info);
}
```

Figura 34. Función que codifica la matriz dispersa en tres vectores resultados “CSR”.

El siguiente paso es, una vez se tiene los tres vectores resultados, llamar a la función “sparseBlasCSR()”, la cual realiza la multiplicación de, si se le pasa un verdadero, la matriz “A” comprimida por la matriz B sin comprimir, o la matriz “B” comprimida por la matriz “A” sin comprimir si se le ha pasado un falso.

```
start = dsecnd();
#pragma warning(suppress : 4996)
mkl_dcsrmm(&trans, &m, &n, &k, &alpha, matdescra, valuesOutCSRA, jaA, iaA, &(iaA[1]), matrixB, &ldb, &beta, matrixC, &ldc);
fin = (dsecnd() - start);
```

Figura 35. Función que multiplica una matriz "CSR" por otra sin compresión.

Tal como se aprecia en la figura anterior, los parámetros son parecidos a los utilizados en funciones anteriores, a excepción del vector de caracteres “matdescra”, el cual es un vector de 6 elementos, en el que cada uno detalla una configuración para la operación.

```
char* matdescra = new char[6]{'G', '_', 'N', 'C', '_', '_'};
```

Figura 36. Vector de configuración de la función "mkl_dcsrmm()".

Los únicos valores que se configuran en este caso son el primer elemento que especifica que se trata de matrices generales; el tercero que especifica que los valores de la diagonal son no unitarios; y el cuarto que especifica que la indexación empieza en cero.

Finalmente, las últimas dos llamadas que faltan explicar son la “COO()”, que tiene la misma estructura que la función “CSR()”, simplemente que se realiza al formato “COO”; y la función “sparseBlasCOO()”, la cual tiene la misma estructura que la función “sparseBlasCSR()”, simplemente que empleando la función siguiente para el cálculo de la multiplicación:

```
start = dsecnd();
#pragma warning(suppress : 4996)
mkl_dcoomm(&trans, &m, &n, &k, &alpha, matdescra, valuesOutCOOA, rowsA, colsA, &nnzA, matrixB, &ldb, &beta, matrixC, &ldc);
fin = (dsecnd() - start);
```

Figura 37. Función que calcula la multiplicación de matrices en formato "COO".

Para concluir, una vez se ha terminado todas las iteraciones del bucle, se obtienen los tres vectores de tiempos, un vector para cada tipo de multiplicación realizada, y se escriben en un fichero “CSV”

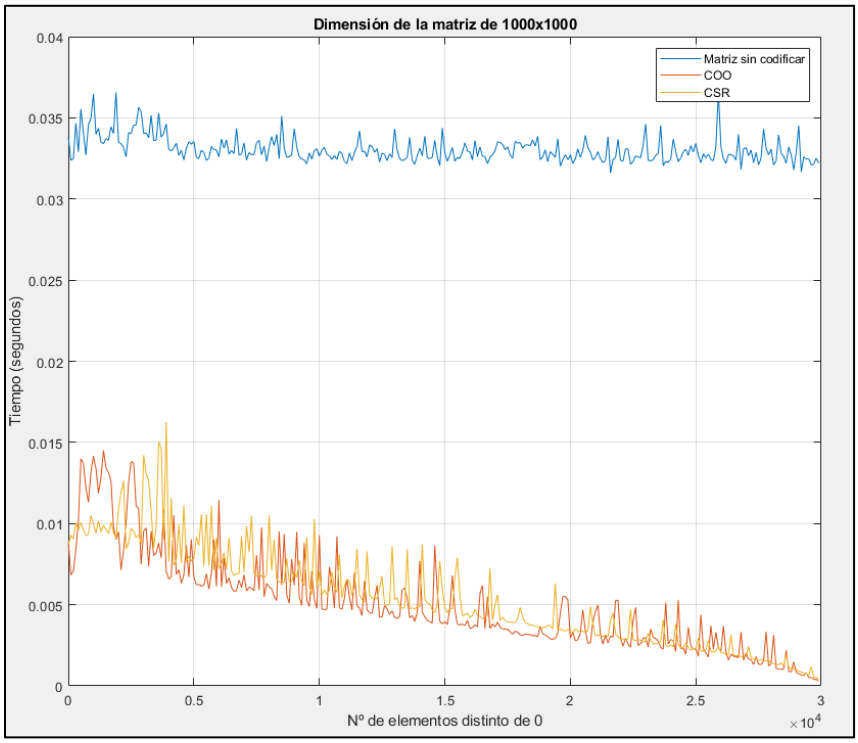

Figura 38. Gráfica que compara los tiempos de ejecución con compresión y sin ella.

Referencias

ULPGC. (s.f.). Obtenido de https://ncvt-aep.ulpgc.es/cv/ulpgctp21/pluginfile.php/412003/mod_resource/content/9/6%20Matrices%20dispersas.pdf