Estudio y evaluación de formatos de almacenamiento para matrices dispersas en arquitecturas multi-core

Marc Pasarín, Beatriz Otero[1], Josep R. Herrero[1]
[1]Departamento de Arquitectura de Computadores
Universitat Politècnica de Catalunya

Índice

	Introducción……	4
1	Formatos de almacenamiento……	4
	1.1 COO……	4
	1.2 CSR……	4
	1.3 CSC……	5
	1.4 CSRP……	5
	1.5 ELLPACK……	5
	1.6 Ladder ELL……	6
	1.7 CSRPELL……	6
	1.8 HYB……	6
	1.9 JDS……	6
	1.10 TJDS……	7
	1.11 DIA……	7
	1.12 BCSR……	8
	1.13 IBCSR……	8
	1.14 FEBA……	8
2	Pruebas……	10
3	Resultados……	12
	3.1 Test SpMV serie……	12
	3.2 Test SpMV Nvidia……	14
	3.3 Test SpMV HYB……	20
	3.4 Test SpMV FEBA……	23
4	Conclusiones……	25
5	Bibliografia……	27
6	Anexo: Resultados……	28
	6.1 Matriz bcsstk29……	28
	6.2 Matriz e40r0100……	30
	6.3 Matriz garon2……	32
	6.4 Matriz memplus……	34
	6.5 Matriz mcs10848……	36
	6.6 Matriz Na5……	38
	6.7 Matriz ncvxbqp1……	40

6.8 Matriz nmos3………………………………………………………………………… 42
6.9 Matriz psmigr_1……………………………………………………………………... 44
6.10 Matriz raefsky4……………………………………………………………………... 46
6.11 Matriz s3dkq4m2…………………………………………………………………… 48
6.12 Matriz sme3Da……………………………………………………………………... 50
6.13 Matriz tandem_vtx…………………………………………………………………. 52

Introducción

Este trabajo estudia la influencia que tienen los formatos de almacenamiento de matrices dispersas en arquitecturas multi-core al evaluar el rendimiento de la operación producto matriz-vector.

Para realizar este trabajo, inicialmente se estudiaron e implementaron los siguientes formatos de almacenamiento: COO, CSR, CSRP, ELLPACK/ITPACK, Ladder ELL, CSRPELL, HYB, JDS, TJDS, DIA, BCSR, IBCSR y FEBA.

Una vez implementadas las versiones secuenciales de la operación producto matriz-vector en cada uno de éstos formatos, se implementaron nuevas versiones para realizar la operación producto matriz-vector (SpMV) en una arquitectura multi-core haciendo uso de CUDA.

Para realizar las pruebas de rendimiento (evaluación de cada formato) se utilizaron diferentes matrices obtenidas de otros artículos relacionados con la misma temática de investigación.

En el caso de las versiones implementadas en CUDA, cada formato contempla diferentes versiones de códigos que consideran cambios en: el acceso a los datos en memoria, o la asignación de datos a cada thread. Para algunos formatos, los códigos utilizados han sido extraídos de Nvidia.

1. Formatos de almacenamiento

Existen multitud de formatos de almacenamiento para matrices dispersas, cada uno de ellos con sus ventajas y desventajas. Se pueden dividir como formatos generales o específicos, siendo éstos últimos típicamente a bloques. Cada formato tendrá un rendimiento diferente dependiendo de las características de la matriz que se almacene y la máquina en la que se ejecuten las operaciones, ya que arquitecturas paralelas, vectorizadas y escalares pueden utilizar diferentes algoritmos para cada formato.

1.1. Formato a Coordenadas (COO)

El formato a coordenadas, o COO, es la implementación más natural de una matriz dispersa.
Para cada valor diferente de cero almacena el valor, la fila y la columna. De esta forma, se tienen todos los valores de una forma fácilmente accesible.

Cada matriz se compone de 3 vectores: row, col y val, todos ellos de tamaño número de no ceros. Cada no-cero queda identificado según el índice en los vectores: los vectores row y col contienen la fila y columna del valor, respectivamente, y el vector val almacena el valor.

1.2. Formato Compressed Storage Row (CSR)

El formato Compressed Storage Row, o CSR, es una de los más populares para el uso de matrices dispersas. Al ser un formato muy simple y compacto es muy adecuado para los casos generales, debido a que no tiene ninguna característica que merme el rendimiento especialmente.
Las matrices en el formato CSR se componen principalmente de 3 vectores: row_ptr, col_ind y val.
El vector row_ptr indica, para cada posición, el índice de los otros vectores en los que empieza la fila. En la matriz de ejemplo, el primer valor es cero, puesto que la primera fila empieza en la posición cero de los vectores. Para la segunda fila el valor es 2, ya que la primera fila contiene 2 valores y el primer elemento de la segunda fila se encuentra en ese índice de los otros dos vectores.

El último valor de row_ptr siempre apunta al primer valor fuera de rango. Los vectores col_ind y val indican cada uno de ellos la columna y el valor de cada no-cero, de la misma forma que se realiza en COO.

Esta distribución es muy útil porque permite reducir el tamaño del vector referente a las filas respecto COO, haciendo que la matriz ocupe menos espacio y también permite un acceso directo y secuencial a cada una de las filas.

1.3. Formato Compressed Storage Column (CSC)

El formato Compressed Storage Column (CSC) es exactamente igual al formato CSR, con la única diferencia que, en vez de ordenar por filas, ordena por columnas. De este modo, se tienen los vectores col_ptr, row_ind y val, completamente análogos a CSR.

El vector col_ptr indica el índice de los otros 2 vectores en los que empieza cada columna, siendo el último elemento un índice a la primera posición fuera de rango. El vector row_ind indica, para cada elemento, la fila en la que está situado, y val tiene el valor.

1.4. Formato Compressed Storage with Permutation (CSRP)

El formato CSR con permutación, o CSRP, es una variación del formato CSR cuya principal característica es que reordena las filas en número de no-ceros de forma decreciente, de forma que las primeras filas son las más densas y las últimas las menos densas.
Esto permite, en muchos casos, aprovechar la caché del vector x en diversas filas, puesto que cuantas más posiciones se usen, más posiciones habrá en común, con una mínima desventaja respecto al CSR original, al tener que almacenar los valores con una indirección sobre el vector y, debido a la permutación.
La estructura de este formato es prácticamente igual a CSR, en la forma que los vectores row_ptr, col_ind y val tienen la misma estructura y funcionamiento. Adicionalmente dispone del vector permute que, para cada elemento del vector, indica a que fila real corresponde la fila permutada.

1.5. Formato ELLPACK/ITPACK (ELL)

El formato ELLPACK/ITPACK, de forma reducida ELL, cambia la forma de almacenar los datos de una matriz dispersa, ya que mientras todos los formatos parten de vectores unidimensionales este formato utiliza vectores bidimensionales, dando más estructura a la matriz, con sus ventajas e inconvenientes.
Por una parte, al tratar con los vectores bidimensionales se logra reducir el número de estructuras necesarias para el formato, puesto que el valor de fila es parte intrínseca de la estructura y no debe ser declarado en un vector aparte, como en los casos de CSR o COO.
Por otra parte, estos vectores bidimensionales son densos, de forma que cualquier índice que no deba almacenar valores no-cero mantiene el coste en memoria (y en tiempo de procesado de la matriz).
El formato ELL consiste en 2 vectores bidimensionales, val e índices, de tamaño número de filas por número máximo de no-ceros por fila. Esto es así para poder almacenar todos los no-ceros en la estructura. El vector val almacena los valores no-cero de la matriz mientras que el vector de índices almacena los índices de columna de cada no-cero.
Cada fila de la matriz es almacenada en una fila de los dos vectores, de forma que su acceso es inmediato al mantener el mismo orden. Asimismo, dentro de cada fila los no-ceros se almacenan en orden, siendo el primer elemento de la fila el primer no-cero y así sucesivamente hasta almacenar todos los no-ceros de la fila.
En el caso que el número de no-ceros en la fila sea inferior al máximo se debe realizar un padding para mantener la estructura. De este modo, se rellena el vector val con ceros mientras que el vector indices se rellena con un valor que no tiene relevancia.

1.6. Formato Ladder ELL

El formato ELLPACK/ITPACK with Ladder Scheme, o de forma más abreviada, Ladder ELL, es una variación del formato ELL para aprovechar mejor el espacio.

Para su construcción se reordena la matriz por número de no ceros por fila, en orden descendente, de forma análoga a CSRP, y una vez reordenada, se almacena en formato ELL a trozos, de forma que el los primeros trozos tendrán más no ceros por fila y los últimos menos, pero la variación del número en cada subestructura ELL es muy inferior. Requiere de un vector de permutación y de una organización en las subestructuras como puede ser un vector de punteros a cada una.

1.7. Formato CSR with Permutation in ELLPACK (CSRPELL)

El formato CSR with Permutation in ELLPACK format, o CSRPELL, propuesto por [5] está pensado para procesadores vectorizados, de modo que, aprovechando el poco espacio que ocupa CSR, se logran algoritmos de procesado más eficientes gracias al uso de una estructura auxiliar.

Puede parecer una evolución directa del formato CSRP, pero en realidad esto no es así. CSRPELL mantiene intacta la estructura de CSR, de forma que no hay que transformar los datos, sino simplemente añadir 3 estructuras más: los vectores permute, ell_ptr y ell_nnz.

El vector permute, de tamaño número de filas, almacena el índice de filas en orden descendiente de número de no ceros por fila, siendo el primer elemento el índice de la fila con más elementos y el último el de la fila con menos elementos.

Este formato aprovecha que en las matrices hay diferentes filas con el mismo número de no ceros y, mediante la permutación, los agrupa. De este modo, elementos consecutivos del vector permute se pueden agrupar a partir del mismo número de no ceros por fila. Los vectores ell_ptr y ell_nnz realizan esta función. El vector ell_ptr indica el índice en permute donde empieza cada grupo, y el vector ell_nnz indica, para cada grupo, el número de no ceros por fila de ése grupo.

La referencia a ITPACK/ELLPACK en el nombre no está basada en la estructura con dos vectores bidimensionales como el formato si no que, para cada grupo generado, se pueden recorrer las diferentes filas del mismo modo que en ELL, ya que el trozo correspondiente en col_ind y val en CSRPELL, se correspondería al equivalente de índices y val en ELL, para esa fila. Con una pequeña transformación en los algoritmos, se consigue un funcionamiento análogo a ELL con una estructura de datos CSR.

1.8. Formato HYB

El formato híbrido o HYB ha sido propuesto por NVIDIA [1] para un mayor aprovechamiento de la arquitectura CUDA para tarjetas gráficas. Este formato, compuesto por ELL y COO, es una variación de ELL para aprovechar mejor este formato, sin el problema de los valores desaprovechados.

Haciendo uso de resultados previos del rendimiento de los formatos COO y ELL, o haciendo una estimación sobre ellos, se realiza un cálculo para determinar hasta qué punto se deben almacenar los elementos distintos de cero en el formato ELL, de este modo, al reducir el número máximo de no ceros hay menos ceros en la estructura y el procesado de la matriz es más rápido. Para las filas con más no ceros, aquellos que no estén en formato ELL se almacenan en formato COO, que es un formato muy simple, directo y permite que no haya demasiadas filas (a diferencia, por ejemplo, de CSR, que requiere de todas las filas en el indexado).

1.9. Formato Jagged Diagonal Storage (JDS)

El formato Jagged Diagonal Storage (JDS), también nombrado JAD, abandona el ordenamiento por

filas en pro de un formato parecido a Ladder ELL.

El algoritmo para generar el formato JDS es el siguiente: A partir de una matriz A, se desplazan todos los valores no-cero a la izquierda, resultando una matriz A_{crs} parecida al formato ELL. En este punto, se reordena la matriz por número de no-ceros por fila, quedando una matriz A_{jds} cuyo número de no-ceros por columnas está en orden descendiente. Cada columna de la matriz A_{jds} es una jagged diagonal.

Para almacenar esta matriz resultante se requieren 4 vectores: *jd_ptr*, *col_ind*, *perm* y *val*. El vector *jd_ptr* tiene la misma función que *row_ptr* en CSR, e indica en qué posición de los vector *col_ind* y *val* empieza cada jagged diagonal. Los vectores *col_ind* y *val* indican, para cada elemento, la columna y el valor, respectivamente. El vector *perm*, de tamaño número de filas, indica, para cada fila de la matriz A_{jds}, y por tanto, para cada elemento de una jagged diagonal, la fila que corresponde en la matriz A original.

1.10. Formato Transposed Jagged Diagonal Storage (TJDS)

A partir del formato JDS se ha desarrollado una modificación llamada Transposed Jagged Diagonal Storage (TJDS) cuya mayor diferencia respecto al JDS es que, en vez de realizar un reordenamiento de las filas, lo realiza para las columnas.

En este caso, para una matriz A, desplaza todos los no-ceros hacia las primeras filas, de forma vertical, dando lugar a una matriz Accs. Esta matriz resultante se reordena por número de no-ceros por columna, siendo la primera columna la que tenga más no-ceros y la última la que menos tenga. Esta matriz Atjds contiene, por filas, las trasposed jagged diagonals.

El formato requiere sólo 3 vectores: tjd_ptr, row_ind y val. El funcionamiento es análogo al JDS. tjd_ptr indica en qué posición de row_ind y val empieza cada transposed jagged diagonal, y row_ind y val indican, para cada posición, la fila a la que pertenecen y el valor correspondiente.

En este formato no se incluye un vector de permutación como en el JDS original ya que se parte de que, al convertir la matriz al formato TJDS para una operación del tipo $y=A\cdot x$, también se convierte el vector x al formato (con el mismo reordenamiento con el que se obtiene la matriz A_{tjds}). Esto supone que una matriz en TJDS no se puede usar para más de un vector de entrada.

En la implementación realizada sí se incluye un vector de permutación, de forma que antes de realizar la operación matricial, se reordena el vector x para un correcto funcionamiento.

1.11. Formato DIA o Compressed Diagonal Storage

El formato de almacenamiento por diagonales DIA, también llamado Compressed Diagonal Storage (CDS) en cierta literatura, es un formato específico para matrices cuyos no-ceros residen en unas pocas diagonales, siendo muy poco recomendable para matrices que no cumplan con esto.

El formato consiste en dos vectores: offsets y val.

El vector offsets indica, para cada diagonal que se almacena, el índice de esa diagonal, asumiendo que la diagonal cero es la diagonal de la matriz, siendo las diagonales positivas hacia la derecha, empezando en la misma fila que la cero, y las diagonales negativas hacia la izquierda, empezando en la misma columna que la diagonal cero.

El vector bidimensional val tiene un tamaño del número de diagonales almacenadas por tamaño de la diagonal máxima. Para la diagonal cero almacenará todos los valores, para las diagonales negativas tendrá tantos ceros a la izquierda como distancia hasta la diagonal cero (puesto que en las primeras filas no tiene ningún valor) y para las diagonales positivas tendrá ceros al final, puesto que la diagonal es incompleta.

1.12. Formato Block Compressed Storage Row (BCSR)

El formato Block Compressed Storage Row (BCSR) es la variante de bloques del formato CSR.

La diferencia esencial reside en la distribución a bloques, lo cual quiere decir, que para un tamaño de bloque de m filas y n columnas, el almacenamiento de valores (en este caso el vector val) no almacena un valor no cero, sino que almacena m·n valores, sean o no cero, siempre que alguno de ellos sea diferente de cero.

Los bloques están alineados por su tamaño, tanto en filas como en columnas, así que cada bloque empezará en una fila múltiplo de m y una columna múltiplo de y.

Al ser los bloques de un tamaño fijo, puede suponer un problema que el tamaño de la matriz no sea de un tamaño adecuado para el bloque, lo cual supone que, en las últimas filas y columnas puede llegar a almacenar valores que realmente no existen. Es tarea de los algoritmos para procesar la matriz detectar esos valores y no usarlos en la operación.

En términos de implementación, se puede plantear val como un vector bidimensional, siendo el primer índice el del bloque, y el segundo el de índice de valor dentro del bloque. De otro modo, y a efectos prácticos igual, se puede plantear que todos los bloques están consecutivos. En ambos casos se plantea una ordenación dentro del bloque siendo por fila y luego por columna, siendo los n primeros valores de un bloque la primera fila, y así sucesivamente.

1.13. Formato Indirect BCSR (IBCSR)

Este formato es mencionado en ocasiones como BCSR, pero debido a la diferencia con el BCSR que se usa de forma generalizada, se propone Indirect BCSR (IBCSR) como nombre para referirse a él.

La única diferencia con BCSR está en que val, en vez de tener los valores, contiene el índice de un nuevo vector data que sí contiene los valores. La combinación de val y data es equivalente a val de BCSR. Al tener un bloque de tamaño fijo se puede considerar la inclusión de los índices en val como información redundante, y a decir verdad, en términos de rendimiento, es prácticamente equivalente a BCSR, excepto en el incremento de memoria, de forma que este formato resulta poco recomendable como alternativa.

1.14. Formato Feature-Extraction Based Algorithm (FEBA)

Este formato específico, a diferencia de otros como BCSR, no toma en consideración solamente la distribución de los no-ceros de la matriz en bloques, sino que busca diferentes propiedades de la matriz y almacena la matriz en diferentes estructuras para aprovechar estas características. De ahí el nombre del formato: Feature-Extraction Based Algorithm. Su nombre indica que no es un formato de almacenamiento sino un algoritmo, pero para operar con esos datos posteriormente se debe definir el formato, que proponen en la referencia [2].

La construcción de la matriz, o la fase de preprocesado según la notación de los autores, consta de los siguientes pasos: Extracción de bloques casi-densos, extracción de diagonales casi-densas, extracción de filas y por último, almacenamiento de los datos restantes. A continuación se definen los algoritmos de extracción junto a su formato

1.14.1. Extracción de bloques

Este algoritmo analiza la matriz en submatrices de mb filas, en los que, de cada submatriz, se extrae un bloque denso, seleccionando los valores para extraer el mejor bloque posible, basado en unos parámetros de umbral de densidad.

En primera instancia se mira la submatriz columna por columna, seleccionando únicamente esas columnas que tengan una densidad α o superior, entendiendo como densidad el número de no-ceros de una columna entre el tamaño de la columna. De este modo, α tiene valores del tipo $0 < \alpha \leq 1$, como cualquier valor probabilístico. De esta selección se extrae un conjunto de columnas C.

Una vez seleccionadas las columnas se obtiene una submatriz de mb filas y $|C|$ columas. En esta submatriz se seleccionan las filas de densidad β o superior. El parámetro β tiene las mismas características que α, tanto en definición como, lógicamente, en rango de valores. De la selección de las filas pertinente se extrae un conjunto de filas R.

Una vez obtenidos los conjuntos R y C se puede crear un bloque denso RxC que incluya los valores de las filas y columnas respectivas, tanto si son no-ceros como si son ceros. De este modo, aunque la estructura es densa, la submatriz extraída no necesariamente lo es, de ahí que se trate de bloques casi-densos.

De forma independiente a los parámetros mb, α y β, se recomienda realizar tratar las columnas en diferentes pasadas, cogiendo un tamaño adecuado a la línea de caché, para optimizar el preprocesado.

Para el almacenamiento de la estructura es importante tener en cuenta que los bloques no son de tamaño fijo, sino que cada submatriz de mb filas dará lugar a bloques de diferente tamaño según el patrón de la matriz. Es igualmente importante el hecho que aunque el bloque sea denso, las filas y columnas no son consecutivas, lo cual se corresponde con una irregularidad tanto en los accesos al vector x como las escrituras en el vector y.

Con estos dos hechos en consideración, no hay ninguna relación entre los formatos de almacenamiento a bloques como BCSR y la extracción de bloques de FEBA.

1.14.2. Extracción de diagonales

Una vez extraídos los bloques casi-densos, se procede con la extracción de diagonales casi-densas. Del mismo modo que en el paso anterior, en este paso se vuelve a tratar la matriz a trozos, dividiéndola en submatrices de md filas. Para cada submatriz escoge las diagonales según su densidad y las almacena por separado.

Suponiendo una submatriz de dimensiones md · n, asumiendo que la diagonal que empieza en el primer elemento es la diagonal 0, se puede ver que existen diagonales de -(md-1) hasta (n-1). De estas diagonales, no todas tienen las mismas dimensiones, puesto que las inferiores a 0 y las superiores a (n-md) tienen un tamaño inferior a md. Para una correcta implementación del formato, sólo se tratan las diagonales completas, de tamaño md. Así, sólo se tratan las diagonales de 0 a (n-md).

Para determinar qué diagonales deben extraerse se comprueba su densidad, y si es superior a un parámetro γ, se selecciona. De forma análoga a la extracción de bloques, la densidad de una diagonal es el número de elementos en la diagonal respecto a su tamaño, y γ tiene valores $0 < \gamma \leq 1$.

El algoritmo permite la extracción de diagonales truncadas, debido al analizar un subconjunto de la matriz cada vez, aunque sería inútil en caso que las diagonales truncadas se dividieran entre dos submatrices, ya que en ninguna de las dos submatrices habría suficientes elementos para considerar una diagonal casi-densa.

1.14.3. Extracción de filas

Al tratar con filas con suficientes elementos, se puede aprovechar el producto escalar entre vectores. De este modo, se seleccionan las filas con más de nzmin elementos y se almacenan en una estructura auxiliar que permite una implementación adecuada del producto escalar disperso.

1.14.4. Almacenamiento final

La propuesta de la referencia [1] hace referencia al uso de Ladder ELL como formato de almacenamiento. En realidad esto no es necesario, y el algoritmo de extracción se puede usar con cualquier formato como almacenamiento para los no-ceros restantes. Habiendo extraído la mayoría de no-ceros en los pasos anteriores, si hay una buena elección de parámetros y la matriz dispone de una estructura propicia, se puede usar cualquier formato que tenga un buen rendimiento con pocos elementos dispersos, o simplemente, un formato que dé buenos resultados de forma general.

2. Pruebas

Las pruebas se han realizado con diversos ejecutables:

Test SpMV Serie: Realiza 100 iteraciones de la operación SpMV para los siguientes formatos: CSR, CSRP, BCSR con tamaño de bloque 3x3, IBCSR con tamaño de bloque 3x3, COO, ELL, Ladder ELL, CSRPELL, HYB, FEBA-CSR (con parámetros α, β, γ = 0.5, tamaño de bloque 16, y coeficiente ddoti 0.5).

Test SpMV Nvidia: Realiza 100 iteraciones de la operación SpMV con la implementación proporcionada por NVIDIA [1] para los formatos CSR, COO, ELL y HYB; y una implementación propia para los formatos CSRP, JDS, BCSR 2x2 y Ladder ELL.

Test SpMV HYB: Realiza 100 iteraciones de la operación SpMV en formato HYB variando el parámetro del formato HYB (supuesta mejora de velocidad de ELL sobre COO) entre 1 y 5, con saltos de 0.5.

Test SpMV FEBA: Realiza 100 iteraciones de la operación SpMV en formato FEBA-CSR variando la densidad, con valores 0.125, 0.25 y 0.5; y tamaño de bloques, con valores 4, 8, 16, 32, 64, 128, 256.

Con estos programas se ha usado como conjunto de pruebas las siguientes matrices, las mismas que se han usado en [3]:

Nombre	Tamaño (nº filas/cols)	Nz	Nombre	Tamaño (nº filas/cols)	Nz
bcsstk29	13992	619448	ncvxbpq1	50000	349968
R40r0100	17281	553562	nmos3	18588	386594
garon2	13535	390607	psmigr_1	3140	543162
memplus	17758	126150	sme3Da	12504	874887
msc10848	10848	1229778	tandem_vtx	18454	253350
garon2	5832	305630			

Adicionalmente, se han usado un par de matrices cuya estructura es claramente de bloques, de cara a evaluar hasta qué punto supone una mejora el uso de formatos específicos de bloques, seleccionadas de una lista de [6]:

Nombre	Tamaño (nº filas/cols)	Nz
raefsky4	19779	674195
s3dkq4m2	90449	2455670

3. Resultados

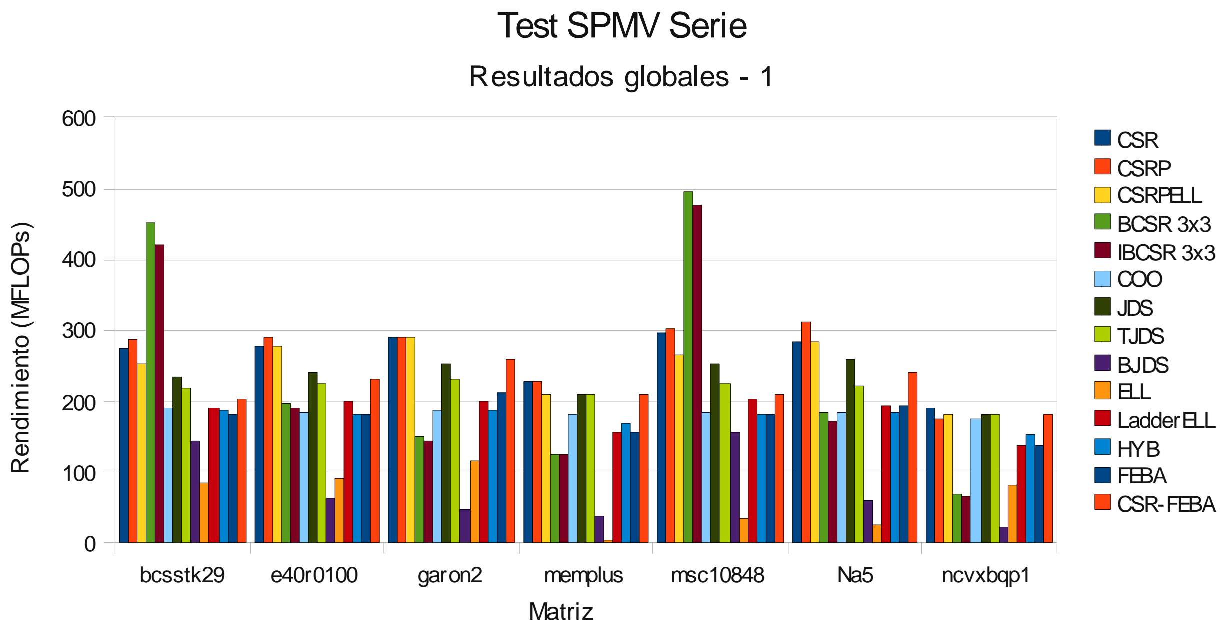
En este apartado se muestran los resultados de las diferentes pruebas realizadas, para todas las matrices, analizando el comportamiento de las pruebas respecto a los formatos y a las matrices.

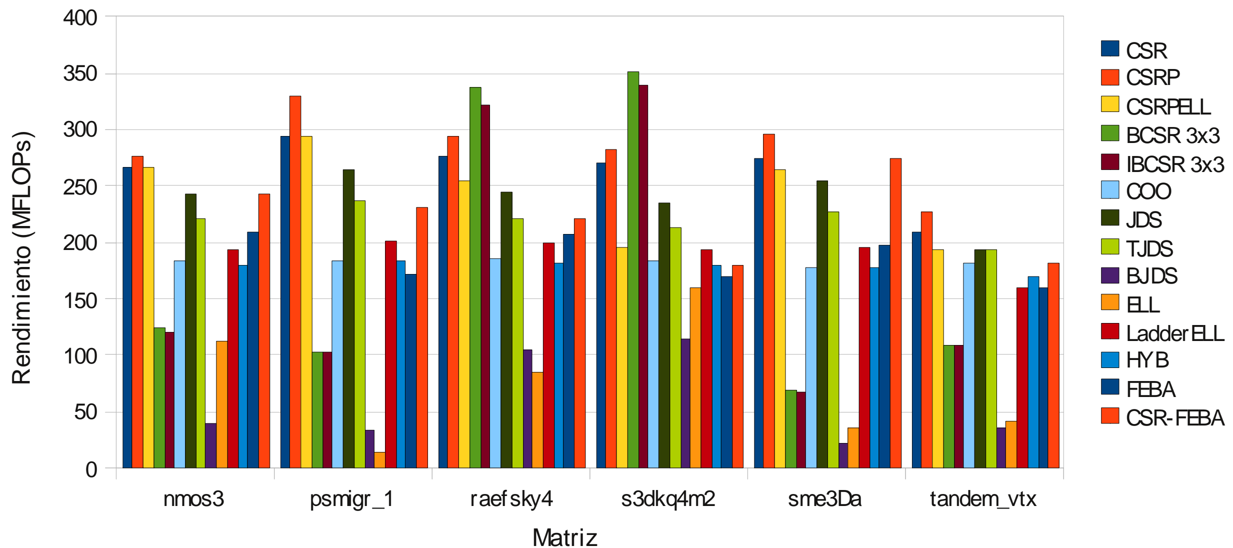
Las pruebas más relevantes consisten en la prueba de formatos en su versión secuencial y la versión para GPUs. Posteriormente se incluyen pruebas más específicas para cada formato.

3.1. Test SpMV Serie (versión secuencial)

En estas gráficas se presentan los resultados de la ejecución del programa Test SpMV Serie, analizando los diferentes formatos: CSR, CSRP, BCSR con bloques 3x3, COO, JDS, TJDS, BJDS, ELL, Ladder ELL, HYB (usando como valor del parámetro la diferencia de rendimiento real entre ELL y COO, en la misma ejecución), y FEBA, en su concepción original y con CSR.

Debido al elevado número de formatos y matrices con los que se ha realizado la prueba se muestran dos gráficas, cada una de ellas con diferentes matrices. En el anexo se puede ver, para cada matriz, más claramente los valores.





Una vez obtenidos los resultados en un procesador serie, se puede ver que de los formatos generales el que mejor rendimiento da es CSR y su variante CSRP, siendo el primero uno de los formatos más extendidos y simples.

El motivo por el cual CSRPELL tiene un rendimiento similar o inferior a CSR es debido a que la variante no modifica la matriz original, sino que añade más estructuras para optimizar la operación en procesadores vectoriales. Estas optimizaciones no son útiles en el procesador de forma que cualquier uso repercute negativamente en el rendimiento.

El formato COO, debido a su estructura dependiente únicamente del número de no-ceros, tiene un rendimiento prácticamente constante en todas las matrices. El uso de índices tanto para la fila como para la columna repercute en un acceso a memoria para cada no-cero a tratar, de forma que la velocidad queda más limitada que en otros formatos.

El formato JDS ofrece un rendimiento que, si bien es inferior al de CSR y variaciones, no es desdeñable, siendo de los formatos estudiados, la segunda mejor opción. La variante TJDS, en la que se hacía hincapié en que era igual que JDS reduciendo el espacio de memoria que ocupa, ofrece un rendimiento inferior y, como se menciona en el apartado 1.10, para poder ser versátil, tampoco existe una mejora en el almacenamiento.

El formato ELL, debido a que su rendimiento depende en gran parte a la distribución de no-ceros entre las filas y que en las matrices reales esta distribución no es constante, ofrece un pobre rendimiento en el procesador serie. También es notorio que, pese a que en ningún caso es un formato aprovechable en la plataforma, los resultados son muy variables respecto la matriz, por la propia estructura del formato, siendo el caso opuesto a COO. En este caso la dependencia de las operaciones a realizar respecto el número de no-ceros es prácticamente nula, pues depende únicamente del número de filas, y el número máximo de no-ceros por fila.

Como se preveía un resultado negativo en ELL debido a sus limitaciones, la variación del formato Ladder ELL solventa ese problema y ofrece un rendimiento superior a COO en la mayoría de casos, estando por debajo de otros formatos como CSR o JDS. Al distribuir la matriz en diferentes ELL se logra que la variación de no-ceros por cada trozo sea inferior. El rendimiento de Ladder ELL, de todos modos, depende del tamaño de trozo usado, puesto que por una parte al hacer los trozos

grandes se genera más *padding* por trozo, pero al generar trozos más pequeños se generan más trozos, y la indirección de cada trozo tiene un coste adicional en la operación.

Respecto a los formatos específicos, vemos como BCSR logra resultados muy buenos en unas pocas matrices, y resultados mediocres en el resto, como era previsible. Aquellas matrices estructuradas consiguen reducir el espacio de memoria ocupado por el formato y las optimizaciones en el algoritmo mejoran su rendimiento, pero las que no cumplen las condiciones previstas realizan un almacenamiento y procesado de ceros importante, reduciendo su rendimiento.

El formato IBCSR, de mecánica muy similar a BCSR aunque con más almacenamiento tiene un rendimiento con el mismo comportamiento que BCSR respecto a las matrices, siendo ligeramente inferior al haber un acceso de memoria más para la obtención de los valores de la matriz.

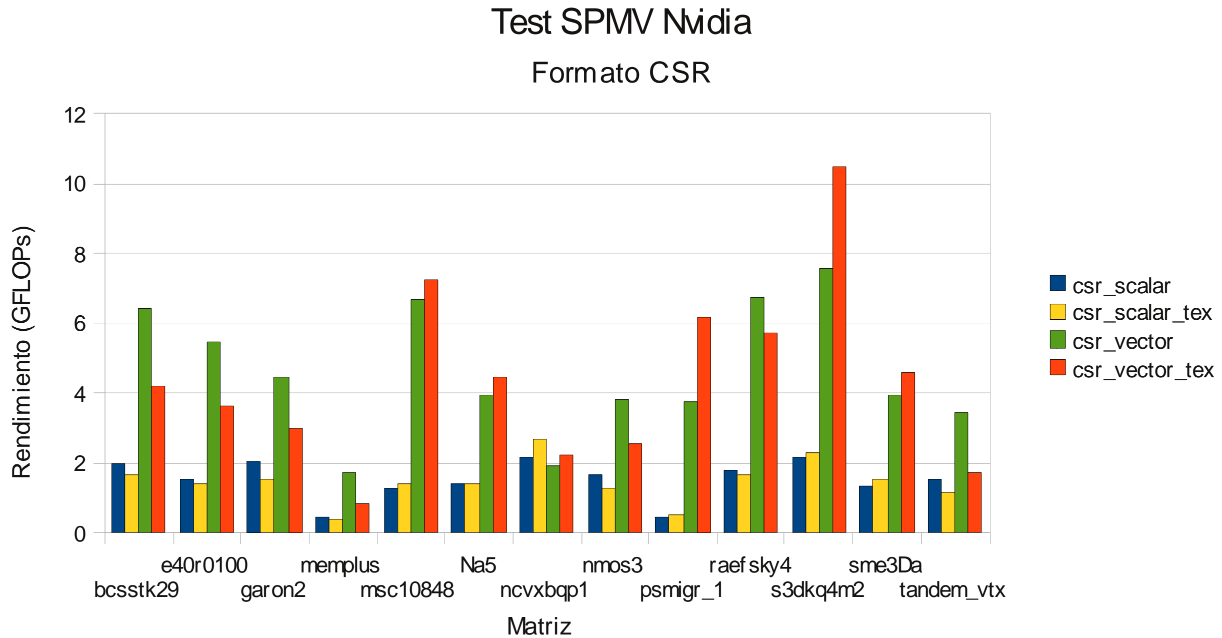
El formato BJDS, implementado de forma genérica sin desenrollado de bucles para cada tamaño de bloque, tiene la misma relación de rendimiento que los BCSR puesto que lo más importante en estos casos es que la matriz cumpla con las características específicas, pero su rendimiento es inferior al que obtendría si se realizaran implementaciones exclusivas para cada tamaño de fichero.

El formato FEBA, que se detalla en el apartado 3.4 con más profundidad, ofrece resultados negativos debido a problemas intrínsecos del formato. Al realizar el almacenamiento final en CSR en vez de usar Ladder ELL aumenta el rendimiento, debido a que el primer formato es mejor que el segundo.

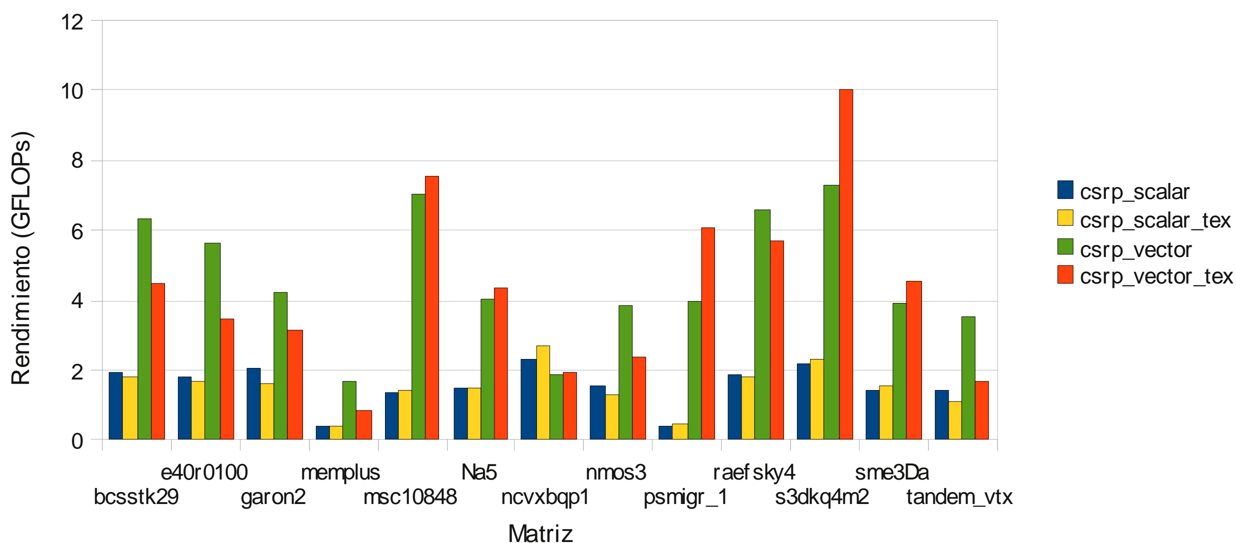
3.2. Test SpMV Nvidia (versión multi-core)

En las gráficas de este apartado se muestran los resultados de distintas variantes para un mismo formato en todas las matrices. Las variantes "tex" son las que usan la memoria de texturas para acceder a los valores del vector x.

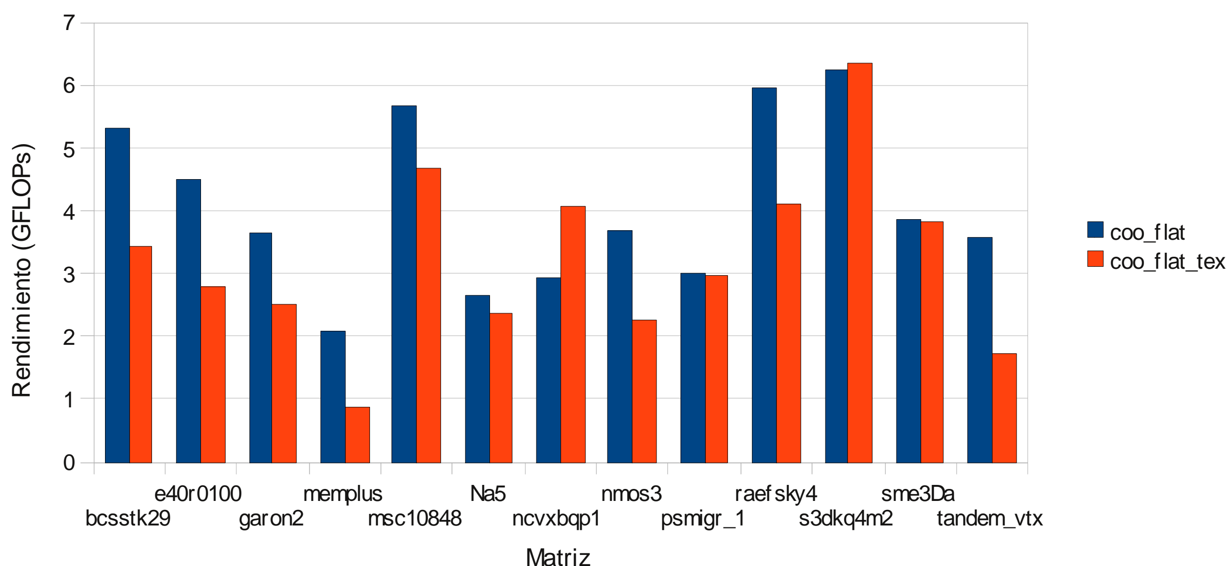
Los mismos resultados, separados por matrices, se pueden ver en el sub-apartado Matrices, dentro de este mismo apartado, para poder comparar diferentes formatos con los mismos datos de entrada.



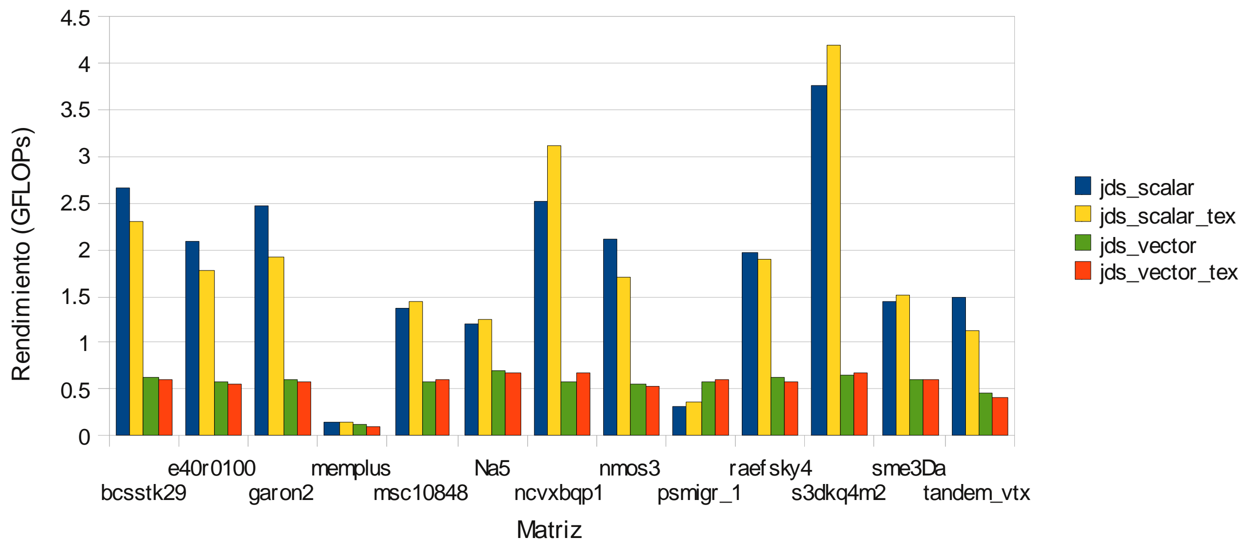
En el formato CSR, se ve claramente como la implementación vectorial es mejor que la implementación escalar, exceptuando en la matriz ncvxbqp1. Esta mejora está fundamentada en que la implementación vectorial aprovecha todos los threads del warp trabajando juntos para realizar finalmente una reducción hasta obtener el valor correspondiente a una fila. En el caso en que haya más no-ceros en una fila que threads en el warp se aprovecha esta característica de la implementación mejorando el rendimiento como se puede observar.



El formato CSRP, una variación de CSR que reorganiza los no-ceros de forma que las filas estén ordenadas por número de no-ceros por filas, pierde su principal característica debido a que todas las filas se tratan en paralelo en la GPU, de forma que los resultados obtenidos son muy similares a los obtenidos en CSR, siendo en ocasiones superiores y en ocasiones inferiores en términos de rendimiento, pero en ningún caso de un modo significativo.

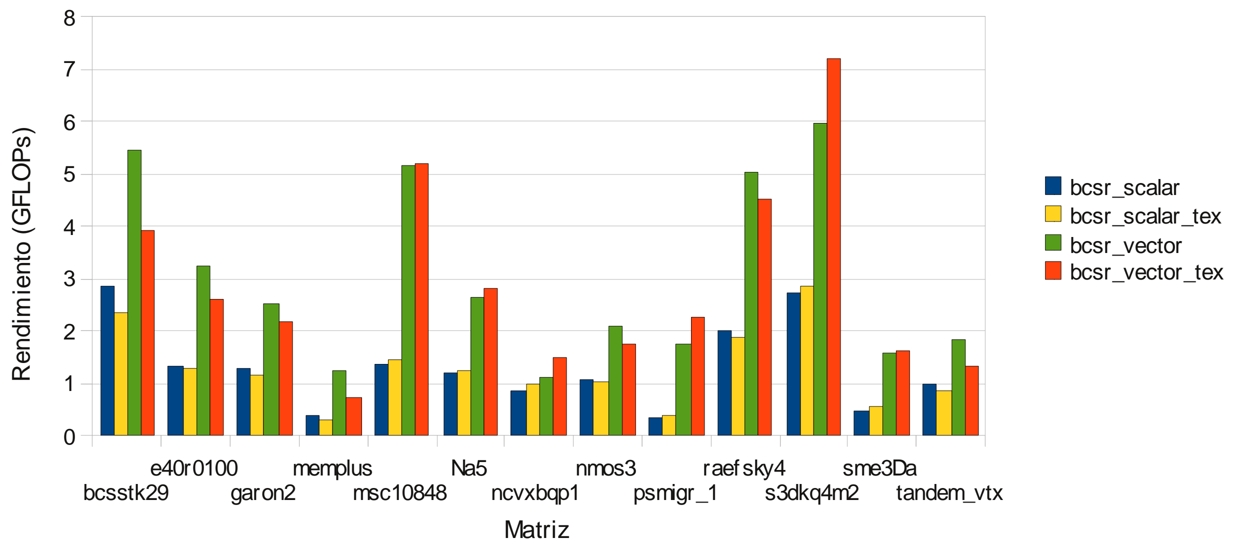


Sobre la GPU el formato COO tiene un rendimiento menos constante que en el procesador serie, y más parecido a CSR.



En la implementación vectorial se distribuyen los warps en filas y los threads en jagged diagonals, de forma que cada uno de ellos debe tener en cuenta si la jagged diagonal en la que se encuentra tiene suficientes elementos en esa fila para cada valor que trata. Este acceso a memoria añadido limita sobremanera el rendimiento del algoritmo, de forma que el algoritmo escalar, en que cada fila es tratada por un thread, que recorre las jagged diagonals hasta que acaba la fila, mucho más simple y eficiente. De todos modos el rendimiento que ofrece es inferior a CSR en su variante vectorial, por ejemplo.

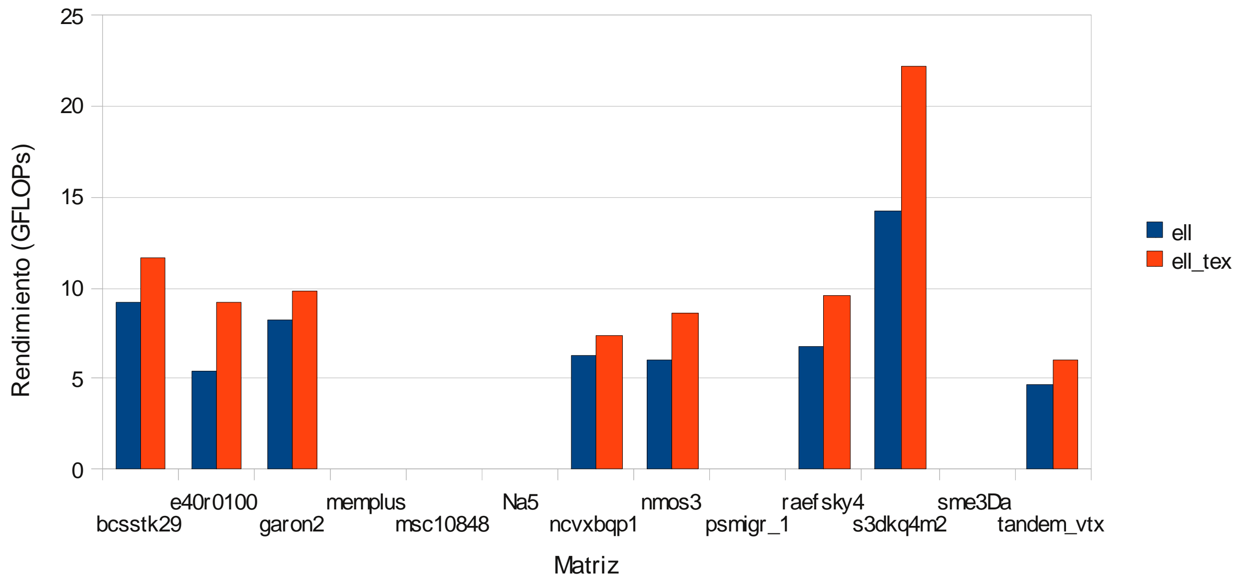
El formato TJDS se intentó implementar en CUDA, pero debido a los reordenamientos que realiza, no es posible distribuir los threads de forma que no haya conflictos de escritura sobre escritura, de forma que, en algún momento, los resultados de la operación pueden resultar incorrectos dependiendo únicamente de la estructura de la matriz, de forma que el formato no es viable de forma general.



El formato BCSR no se puede aprovechar en una arquitectura en la que se dispone de tantos threads, respecto CSR. En la implementación escalar, en la que cada thread se ocupa de una fila sí que se puede aprovechar la estructura de bloques ya que cada fila es tratada como si fuera un procesador escalar, y en esas matrices BCSR escalar da un rendimiento superior a CSR escalar.

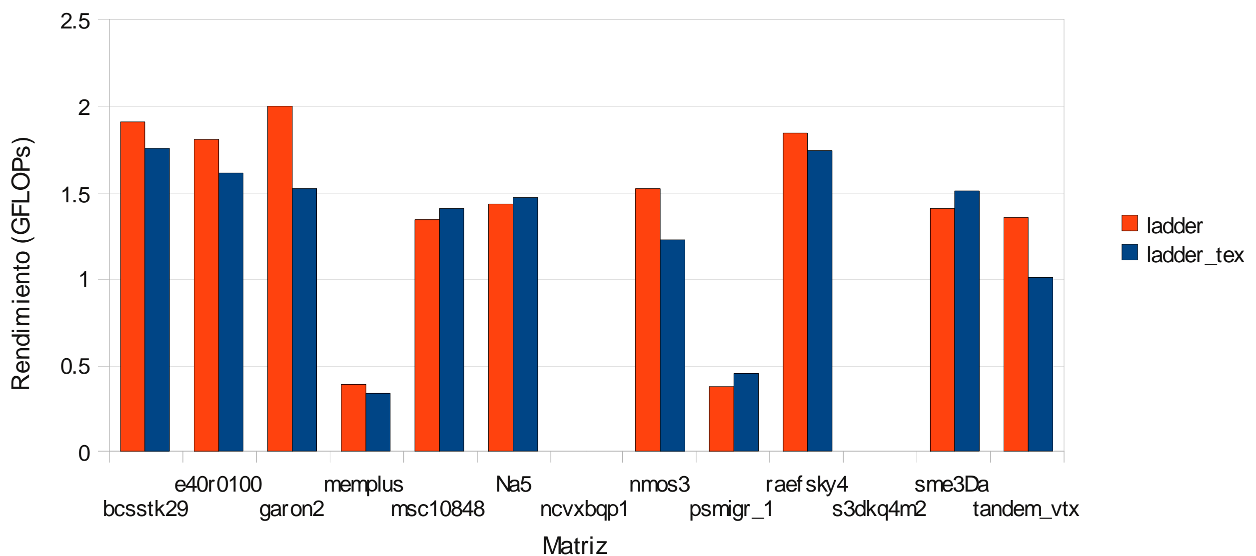
De cualquier modo, BCSR vectorial da mejores resultados que BCSR escalar, debido a que la organización de threads se aprovecha mejor, y en ese caso la única diferencia es que un thread, en vez de tratar un valor, trata un bloque. En este caso se usan menos threads, pero cada uno de ellos realiza más trabajo, de forma que se pierde rendimiento al respecto de CSR vectorial.

La estructura de la matriz en bloques no es aprovechable en esta arquitectura, o por lo menos, no es posible con los mismos formatos que en un procesador serie.



El formato ELL, en la implementación de Nvidia, sólo funciona bajo un determinado umbral de no-ceros por fila, de forma que ciertas matrices, por la propia construcción del algoritmo, no pueden funcionar.

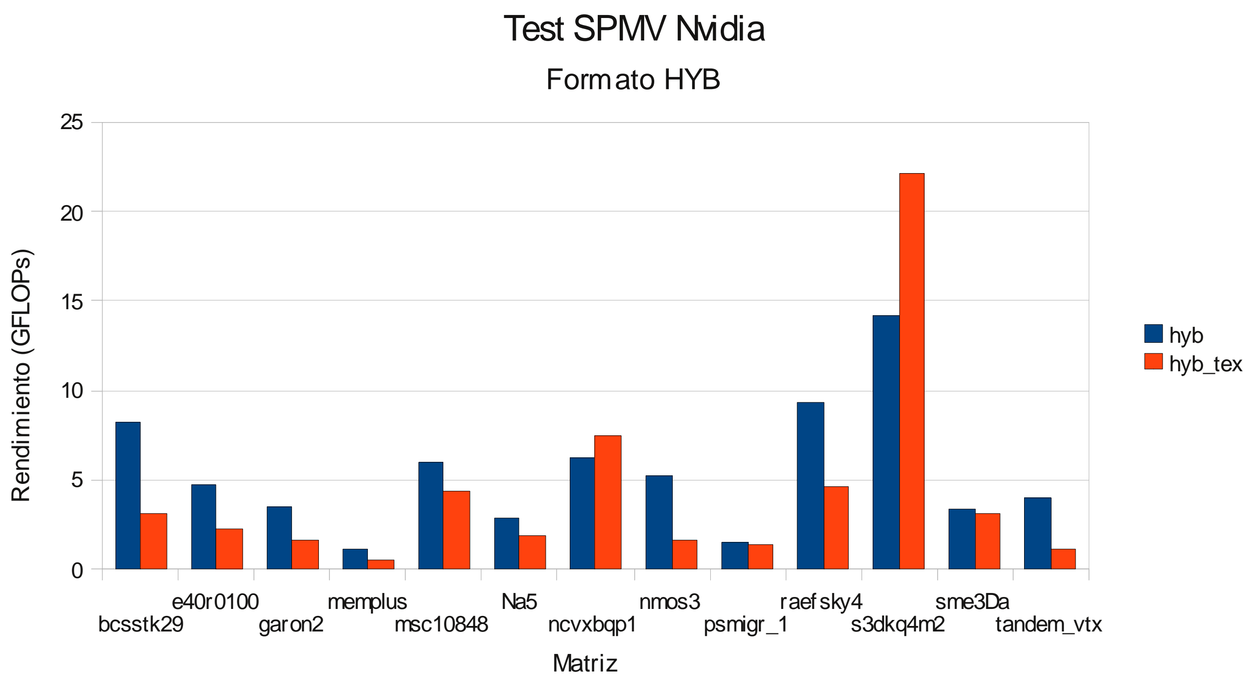
En los casos en los que ELL funciona, los resultados son muy favorables hacia el formato, especialmente en el uso de la memoria de texturas. Esto se debe a que el funcionamiento del algoritmo asigna un thread por fila, y todos los threads ejecutan exactamente el mismo código, de forma que, pese a que muchos threads al final acaben realizando operaciones no necesarias, como es tratar valores que son cero, de forma global el rendimiento es superior a otras alternativas. El problema en este formato es el coste en almacenamiento de las matrices.



El formato Ladder ELL, que mejoraba al formato ELL del que proviene en un procesador serie, debido a que ya no se tratan tantos ceros, no hace lo mismo en una arquitectura multi-core como la GPU.

Al tener múltiples estructuras ELL la primera implementación fue realizar llamadas a la operación SpMV de ELL repetidas veces, pero al sólo poder ejecutar una operación simultáneamente en la GPU, la mejora en la relación entre no-ceros y ceros se pierde al no poder usar todos los threads y el rendimiento era muy bajo.

Posteriormente se ha implementado la operación de forma que los diferentes warps trabajen con los diferentes trozos ELL, que mejora el rendimiento hasta un punto similar a CSR escalar, pero de ningún modo se acerca al rendimiento de ELL. Un factor clave es la obligatoria indirección que se genera en esta implementación debido a mantener la estructura que almacena todos los trozos ELL en la GPU.



El formato HYB, pese a estar pensado para mejorar el rendimiento de ELL, ya bueno de por sí, no logra superar sus resultados.

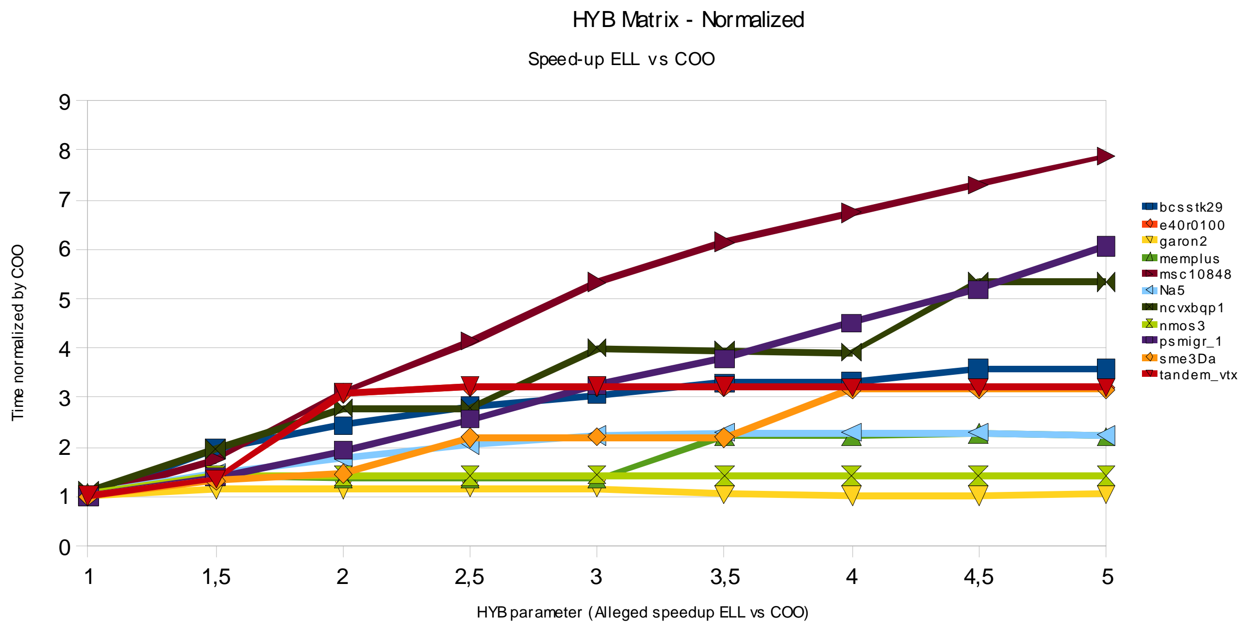
Por otra parte en este caso puede trabajar con cualquier tamaño de matrices sin más complicación, lo cual hace que sea un formato más apto que ELL para trabajar con matrices de forma general, sin conocer sus características previamente.

3.3. Test SpMV HYB

Las pruebas realizadas con el programa Test SpMV HYB arrojan, por una parte, resultados respecto los formatos estándar CSR, usado habitualmente como referencia, COO y ELL, que son los que forman, combinados, el formato HYB; y por otra parte los resultados con el propio formato HYB, variando el parámetro usado de 1 a 5, ya que 1 es el valor mínimo que se puede usar, y al llegar al valor 5 se puede observar una tendencia, sin necesidad de realizar muchas más iteraciones en la ejecución.

La tabla de resultados muestra los tiempos de todas las ejecuciones en segundos, mientras que la gráfica presenta los valores normalizados respecto COO, lo cual permite ver la evolución de todas las matrices desde una misma perspectiva.

Basic format	bcsstk29	e40r0100	garon2	memplus	msc10848	Na5	ncvxbqp1	nmos3	psmigr_1	sme3Da	tandem_vtx
CSR	0.28	0.48	0.33	0.13	0.52	0.14	0.24	0.36	0.45	0.77	0.48
COO	0.34	0.59	0.41	0.14	0.66	0.17	0.24	0.43	0.59	0.99	0.59
ELL	0.74	1.19	0.68	11.16	3.57	1.21	0.49	0.7	7.92	4.83	1.19
HYB parameter											
1	0.34	0.6	0.42	0.15	0.67	0.18	0.27	0.45	0.59	0.99	0.6
1.5	0.67	0.8	0.47	0.2	1.16	0.25	0.47	0.61	0.82	1.32	0.8
2	0.83	1.82	0.47	0.19	2.05	0.3	0.67	0.61	1.14	1.45	1.82
2.5	0.96	1.9	0.48	0.19	2.73	0.35	0.67	0.61	1.51	2.17	1.9
3	1.04	1.9	0.48	0.19	3.52	0.38	0.96	0.62	1.92	2.18	1.9
3.5	1.12	1.9	0.43	0.31	4.05	0.39	0.95	0.61	2.24	2.17	1.9
4	1.13	1.89	0.42	0.31	4.45	0.39	0.94	0.61	2.66	3.13	1.89
4.5	1.22	1.89	0.42	0.32	4.82	0.39	1.28	0.61	3.06	3.13	1.89
5	1.22	1.89	0.43	0.31	5.2	0.38	1.28	0.61	3.57	3.13	1.89



Como se puede ver claramente, los valores mínimos del formato HYB se dan usando parámetro 1, lo cual hace que todos los valores no-cero se almacenen en formato COO, ignorando la parte ELL. Esto es debido a que el formato COO es más rápido que ELL, de forma que al añadir elementos a ELL se perderá rendimiento. El diferente pendiente para cada matriz en la gráfica se debe a que al cambiar el parámetro del formato HYB, cambia el número de columnas que se usará en ELL, pero si todas las columnas mantienen el mismo número de no-ceros, en el momento en que se realiza la

operación SpMV no hay tanta pérdida de rendimiento como si, en cierto número de filas se empiezan a añadir ceros.

Para entender la utilidad del formato HYB hay que tener en cuenta que en la implementación de SpMV sobre GPU, ELL es un formato que tiene un rendimiento elevado, y por ello una optimización sobre es muy útil.

3.4. Test SpMV FEBA

En este apartado se muestran todos los resultados obtenidos para la prueba de Test SpMV FEBA. El planteamiento está pensado suponiendo que el tratamiento de los bloques sería el que más importancia tendría y, por tanto, el que más merece ser estudiado [2].

Block size	Block dens.		bcsstk29	e40r0100	garon2	memplus	msc10848	Na5	ncvxbqp1	nmos3	psmigr_1	sme3Da	tandem_vtx
---	---	CSR	0.28	0.48	0.33	0.12	0.52	0.13	0.22	0.35	0.44	0.77	0.15
		Blocks	0	4320	3384	4425	2711	1443	12500	4647	0	3126	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
4	0.125	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	8.53	17.49	9.66	5.32	12.58	1.8	34.43	14.38	2.78	19.48	7.32
		Execution	0.19	4.69	3.53	0.77	2.17	1.18	3.25	2.97	0.07	12.27	0.06
		Blocks	0	4316	3383	4410	2712	1458	12	4249	0	2445	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
4	0.250	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	8.46	17.28	9.56	5.18	12.51	1.76	32.54	14.19	2.78	18.79	7.22
		Execution	0.19	1.75	1.47	0.45	1.42	0.25	0.23	0.87	0.07	1.35	0.05
		Blocks	0	4194	3303	2276	2504	1405	2	3279	0	714	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
4	0.500	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	8.46	17.25	9.52	5.17	12.48	1.75	32.52	14.18	2.78	18.74	7.21
		Execution	0.19	0.78	0.46	0.12	0.97	0.21	0.23	0.39	0.07	0.8	0.06
		Blocks	0	2160	1692	2216	1356	729	19	2291	0	1509	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
8	0.125	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	4.32	8.82	4.9	2.73	6.37	0.91	17.25	7.26	1.42	9.58	3.75
		Execution	0.19	4.17	3.69	1.06	2.85	0.48	0.24	2.43	0.06	3.33	0.05
		Blocks	0	2160	1674	2091	1356	729	2	1951	0	942	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
8	0.250	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	4.32	8.79	4.87	2.72	6.35	0.91	17.27	7.24	1.42	9.48	3.75
		Execution	0.2	2.33	1.26	0.2	2.17	0.32	0.23	0.94	0.06	1.02	0.06
		Blocks	0	2103	1084	592	1182	699	0	7	0	135	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
8	0.500	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	4.32	8.76	4.85	2.71	6.32	0.9	17.26	7.24	1.42	9.45	3.75
		Execution	0.19	0.83	0.42	0.12	1.14	0.21	0.24	0.34	0.07	0.79	0.06
		Blocks	0	1080	838	1101	678	365	2	1096	0	680	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
16	0.125	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	2.25	4.56	2.54	1.48	3.28	0.48	9.58	3.79	0.73	4.87	2.01
		Execution	0.19	4.16	2.53	0.48	2.17	0.47	0.23	2.15	0.07	1.62	0.06
		Blocks	0	1080	562	348	663	365	0	21	0	237	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
16	0.250	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	2.25	4.54	2.51	1.48	3.26	0.48	9.58	3.77	0.73	4.84	2.01
		Execution	0.19	2.33	0.82	0.15	2	0.34	0.24	0.35	0.07	0.83	0.05
		Blocks	0	1019	22	24	539	37	0	0	0	7	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
16	0.500	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	2.25	4.53	2.51	1.48	3.24	0.47	9.57	3.77	0.73	4.83	2.02
		Execution	0.19	0.76	0.36	0.11	1.01	0.16	0.24	0.34	0.07	0.78	0.06
		Blocks	0	540	282	350	338	183	1	34	0	229	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
32	0.125	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	1.21	2.44	1.35	0.87	1.73	0.26	5.76	2.04	0.39	2.53	1.14
		Execution	0.19	1.73	0.69	0.16	1.64	0.32	0.23	0.37	0.07	0.92	0.06
		Blocks	0	540	12	31	304	115	0	0	0	41	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
32	0.250	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	1.22	2.41	1.34	0.86	1.72	0.26	5.77	2.04	0.39	2.52	1.14
		Execution	0.19	2.31	0.36	0.11	1.25	0.19	0.23	0.34	0.06	0.78	0.06
		Blocks	0	476	0	9	202	4	0	0	0	0	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
32	0.500	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	1.22	2.4	1.34	0.87	1.69	0.25	5.77	2.04	0.4	2.52	1.14
		Execution	0.19	0.65	0.36	0.11	0.78	0.15	0.23	0.34	0.06	0.78	0.06
		Blocks	0	270	7	143	158	92	1	4	0	50	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
64	0.125	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.69	1.37	0.76	0.56	0.96	0.16	3.85	1.17	0.23	1.37	0.74
		Execution	0.19	1.47	0.38	0.14	1.8	0.24	0.23	0.35	0.06	0.8	0.06
		Blocks	0	270	0	15	124	41	0	0	0	1	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
64	0.250	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.7	1.35	0.75	0.56	0.93	0.15	3.9	1.17	0.22	1.37	0.7
		Execution	0.19	0.88	0.36	0.11	1.03	0.17	0.23	0.35	0.07	0.78	0.06
		Blocks	0	133	0	3	79	1	0	0	0	0	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
64	0.500	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.7	1.34	0.76	0.56	0.92	0.15	3.84	1.17	0.22	1.36	0.74
		Execution	0.19	0.58	0.36	0.11	0.68	0.15	0.24	0.34	0.06	0.78	0.06

Block size	Block dens.		bcsstk29	e40r0100	garon2	memplus	msc10848	Na5	ncvxbqp1	nmos3	psmigr_1	sme3Da	tandem_vtx
		Blocks	0	135	0	81	73	35	0	1	0	2	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
128	0.125	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.44	0.82	0.46	0.41	0.56	0.1	2.88	0.74	0.14	0.79	0.49
		Execution	0.19	1.2	0.36	0.13	1.51	0.19	0.24	0.35	0.06	0.78	0.06
		Blocks	0	128	0	7	55	7	0	0	0	0	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
128	0.250	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.44	0.81	0.46	0.41	0.54	0.09	2.89	0.73	0.14	0.78	0.49
		Execution	0.19	0.65	0.36	0.11	0.8	0.16	0.23	0.35	0.06	0.78	0.06
		Blocks	0	0	0	2	11	0	0	0	0	0	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
128	0.500	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.44	0.81	0.47	0.4	0.53	0.09	2.89	0.74	0.14	0.79	0.49
		Execution	0.19	0.57	0.35	0.11	0.62	0.15	0.23	0.34	0.06	0.78	0.06
		Blocks	0	68	0	40	35	14	0	0	0	0	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
256	0.125	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.31	0.54	0.32	0.33	0.35	0.07	2.41	0.52	0.1	0.49	0.38
		Execution	0.19	0.79	0.36	0.13	1.1	0.19	0.23	0.35	0.06	0.78	0.06
		Blocks	0	1	0	4	16	0	0	0	0	0	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
256	0.250	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.31	0.54	0.32	0.33	0.34	0.07	2.41	0.52	0.09	0.49	0.38
		Execution	0.19	0.57	0.35	0.11	0.65	0.15	0.23	0.34	0.07	0.78	0.06
		Blocks	0	0	0	0	3	0	0	0	0	0	0
		Diagonals	11956	9885	8785	1199	17171	3999	3124	15506	3775	2483	2517
256	0.500	DDOTI	0	0	0	0	0	0	0	0	3	0	0
		Preprocess	0.31	0.53	0.32	0.33	0.33	0.06	2.41	0.52	0.09	0.5	0.38
		Execution	0.19	0.57	0.36	0.1	0.62	0.16	0.24	0.35	0.07	0.77	0.05

4. Conclusiones

El formato CSR y sus variantes CSRP y CSRPELL, para procesadores en serie, presentan los mejores resultados en términos generales. CSR es uno de los formatos más estándares que hay y es el más adecuado para todas las matrices, dentro de los formatos usuales. CSRP intenta optimizar un poco más CSR y, en cierto número de matrices, logra mejorar los resultados. Por otra parte, CSRPELL, con un funcionamiento algo distinto, tiene un rendimiento peor que CSR, debido que es una extensión del formato pensado para mejorar el rendimiento en procesadores vectoriales, y la extensión no es aprovechable por otros tipos de procesador, como el serie.

El formato BCSR, como formato específico que es, arroja muy buenos resultados en matrices concretas, y unos resultados sensiblemente inferiores al CSR de referencia en otros casos. Un análisis de la distribución en bloques de la matriz previo a la conversión al formato favorecería la decisión de usar BCSR o un formato escalar, en un procesador serie. En el caso de la operación sobre GPU, al ser el bloque secuencial no se aprovecha el paralelismo para el procesado de cada bloque y el rendimiento es más pobre que el de CSR.

El formato JDS, ampliamente extendido, ofrece un rendimiento peor que el obtenido con el formato CSR. La variación TJDS tiene un rendimiento inferior a JDS, de forma que no parece una solución mejor, puesto que, al final, no se mejora ni en tamaño de almacenamiento ni el rendimiento de la operación. BJDS se implementó como una prueba de concepto al respecto del bloqueo sobre JDS, sin estar convenientemente optimizado para un tamaño concreto, a diferencia de BCSR. De este modo, los resultados, que se podrían ver como muy malos, no lo son al tener en cuenta que, para un caso concreto de BJDS se podría optimizar aumentando su rendimiento considerablemente (hasta alcanzar las mismas cotas que BCSR).

El formato ELL, que funciona de forma muy satisfactoria con matrices que tienen igual número de no-ceros por fila, al ser probado con matrices reales, sale muy mal parado en un procesador serie. Esto es debido al alto número de ceros que almacena para el padding del formato. En el Test SpMV Nvidia, en cambio, arroja unos resultados buenos debido a que el formato, al tener todas las filas iguales, es más fácilmente paralelizable y, debido al modo de trabajo de la GPU, que ejecuta una misma orden en muchos hilos, permite que todos los hilos tarden lo mismo (haya o no valores) simplificando el paralelismo. El formato HYB, eso sí, mejora los resultados de ELL, al ser una evolución pensada precisamente para aprovechar más el paralelismo de las GPU.

El formato Ladder ELL tiene un rendimiento comparable al formato COO en procesadores serie, que no es especialmente bueno pero no arroja resultados demasiado negativos. Probablemente con otras implementaciones, conservando las mismas aspiraciones del formato, se podría lograr un rendimiento similar a CSR. En la implementación para CUDA, debido a problemas en la gestión de memoria para los diferentes steps del formato el rendimiento no es, ni de lejos, el esperado. Posiblemente un cambio importante de implementación, con una gestión más simple de la memoria daría mejor rendimiento, aunque ELL tiene un rendimiento elevado gracias a su simplicidad, así que probablemente no le llegaría a alcanzar.

Respecto al formato HYB, con las diferentes pruebas que se han hecho se ve como, sea cual sea el valor del parámetro usado (la mejora de velocidad teórica de ELL sobre COO) el rendimiento es siempre peor que COO, debido a que COO en realidad funciona más rápido que ELL en un procesador serie. Esto se debe a que cuanto más alto sea el valor del parámetro, más no-ceros se añadirán al formato ELL, en detrimento de COO, que sólo mantiene el excedente.

En el Test SpMV Nvidia HYB tiene un rendimiento excelente, mejorando los resultados de ELL, que ya son de por sí buenos. Al transferir el excedente al formato COO logra homogeneizar la parte ELL, de forma que se aprovecha el alto paralelismo, y el formato COO ya está pensado para

paralelizar datos menos estructurados, con lo cual se aprovecha más el elevado número de procesos simultáneos en ambos casos.

El formato FEBA-CSR, al ser un compendio de diversos formatos de almacenamiento, da resultados diversos, que hay que analizar cuidadosamente para llegar a conclusiones adecuadas.

En las matrices bcsstk29, psmigr_1 y tandem_vtx los resultados de FEBA son claramente mejores que los de CSR, tomado como referencia por su rendimiento correcto y constante.

Al analizar los resultados de Test SpMV FEBA para estas matrices, vemos como el factor que más aprovechan es el procesado de las diagonales, ya que no se encuentran bloques, y en el caso de psmigr_1 se consigue además un aprovechamiento de ddoti, para 3 filas.

Con semejantes resultados podemos decir que la extracción de diagonales y filas casi-densas mejora considerablemente el rendimiento de SpMV, a costa de un tiempo de preprocesado.

Al respecto de este tiempo de preprocesado, que en ocasiones puede resultar muy elevado, se ha comprobado como es inversamente proporcional al tamaño de los bloques. De este modo se concluye que tanto la extracción de diagonales y filas puede ser aprovechada por los diferentes formatos de almacenamiento sin demasiado temor a una pérdida de rendimiento.

Respecto a los bloques, se han realizado diversas pruebas, para ver las diferencias que supone cambiar la densidad y el tamaño de bloques y, al aumentar cualquiera de los dos parámetros el rendimiento. El hecho de que no se aprovechen instrucciones vectorizadas y que los bloques tengan un subconjunto de filas y columnas (y por tanto el acceso a los vectores x e y no sea regular) supone un problema para el rendimiento de esta supuesta mejora.

5. Bibliografía

[1] Efficient Sparse Matrix-Vector Multiplication on CUDA – N. Bell, M. Garland – 2008 (NVIDIA Corp)

[2] A High Performance Algoritm Using Pre-Processing for the Sparse Matrix-Vector Multiplication – R.C. Agarwal, F.G. Gustavson, M. Zubair – 1992 (IBM T.J. Watson Research Center)

[3] Increasing data reuse of sparse algebra codes on simultaneous multithreading architectures – J.C. Pichel, D.B. Heras, J.C. Cabeleiro, F.F. Rivera – 2007 (Univ. Carlos III de Madrid, Univ. de Santiago de Compostela)

[4] Optimization of sparse matrix-vector multiplication on emerging multicore platforms – S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel – 2008 (Univ. of California at Berkeley)

[5] Vectorized sparse matrix multiply for compressed row storage format – F. D'azevedo, Mark R. Fahey, Richard T. Mills

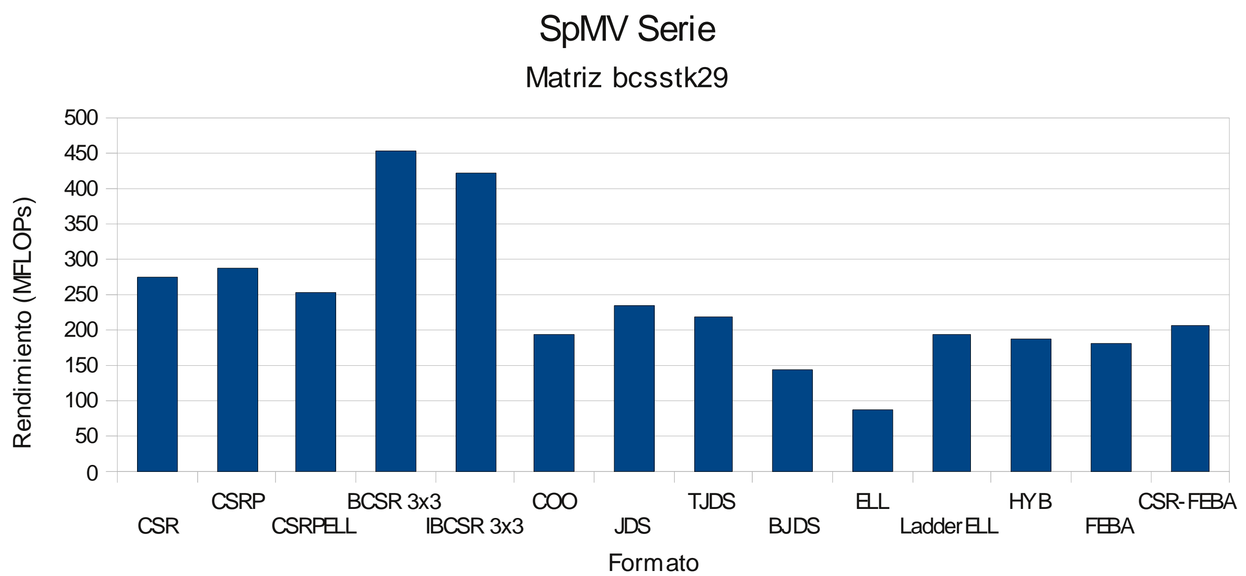
[6] Fast sparse matrix-vector multiplication by exploiting variable block structure – R. Vuduc, H. Moon

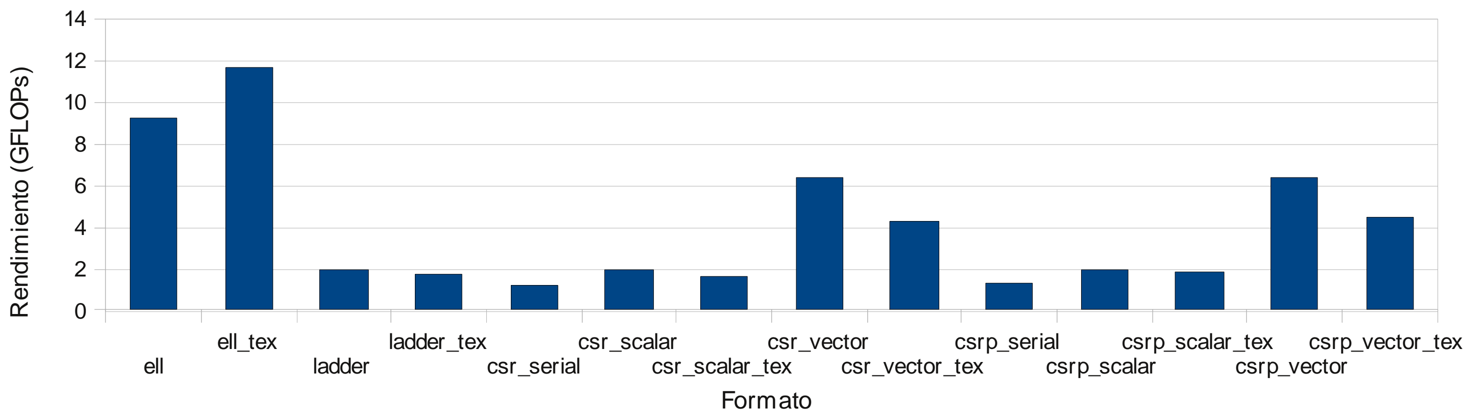
6. Anexo: Resultados

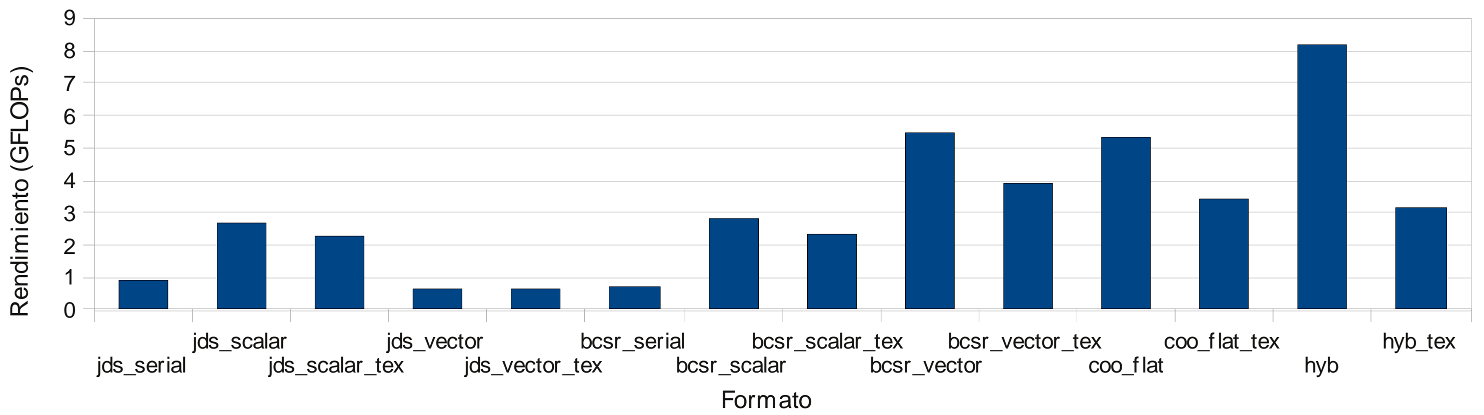
En este apartado se presentan los mismos resultados separados para cada matriz, de forma que se puede ver más claramente como, según las características de la matriz, los resultados varían.

6.1. Matriz bcsstk29

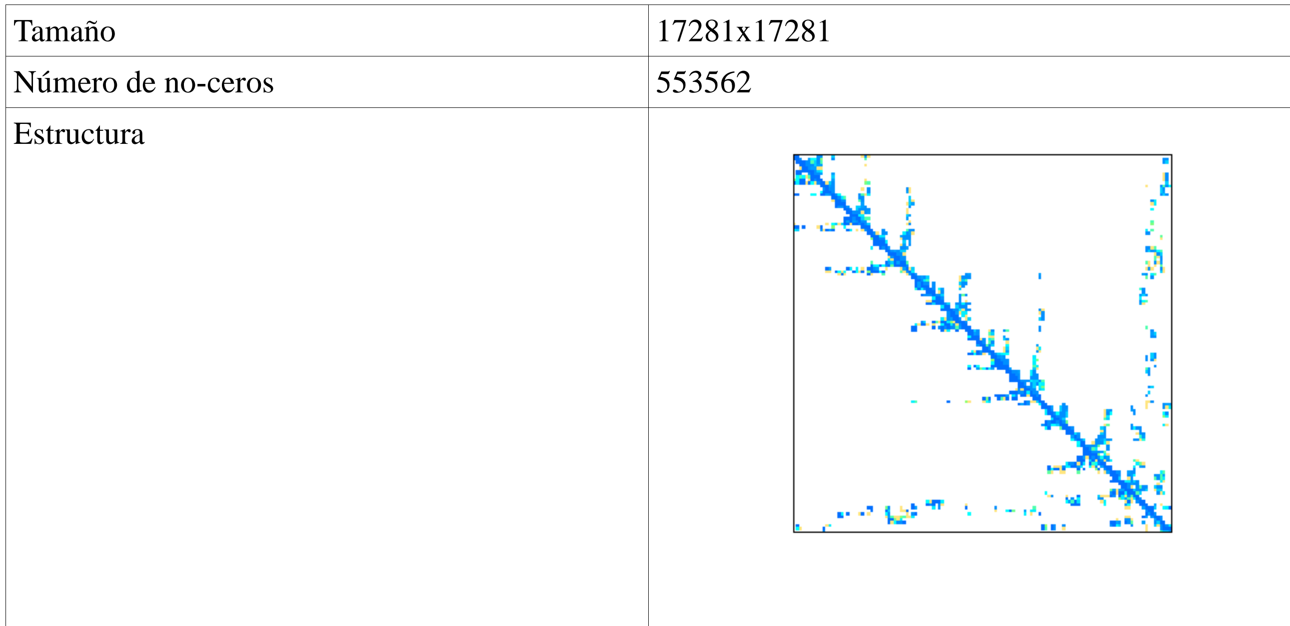
Tamaño	13992x13992
Número de no-ceros	316740
Estructura	

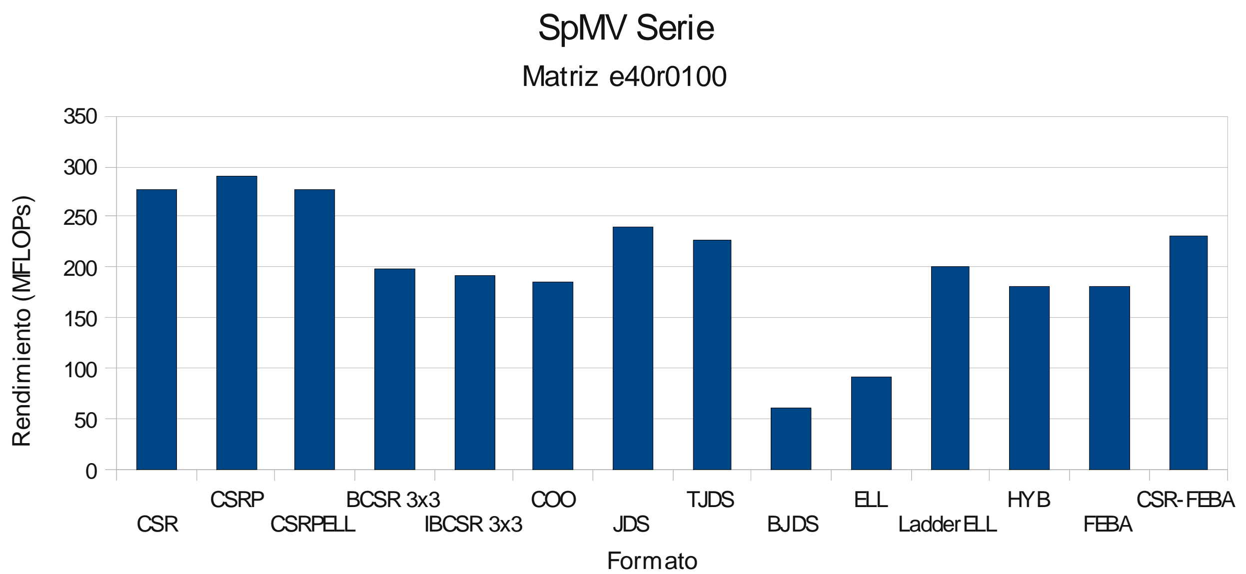


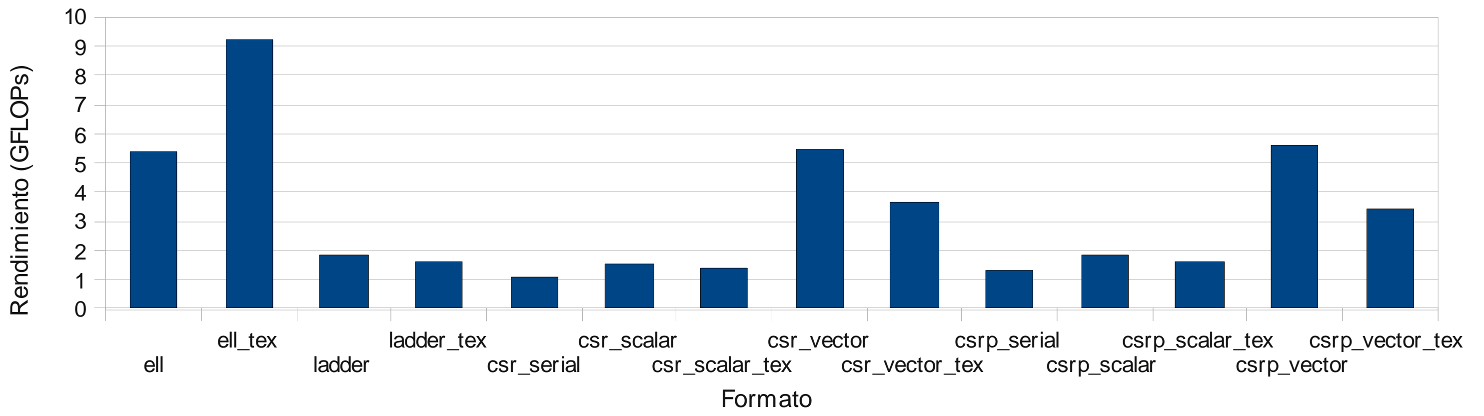

SpMV Nvidia
Matriz bcsstk29 - 1
Rendimiento (GFLOPs)
14
12
10
8
6
4
2
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

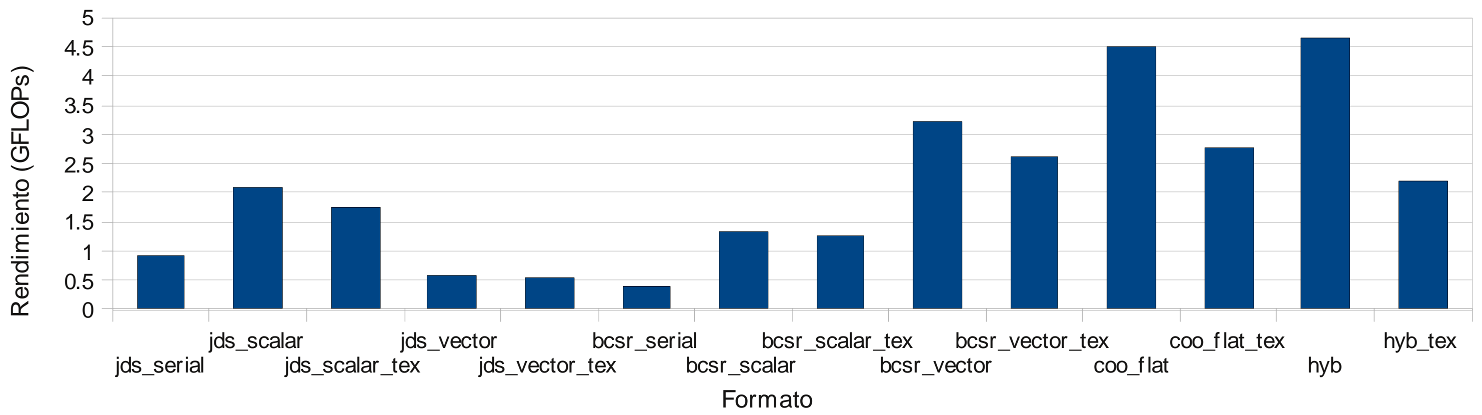

SpMV Nvidia
Matriz bcsstk29 - 2
Rendimiento (GFLOPs)
9
8
7
6
5
4
3
2
1
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.2. Matriz e40r0100

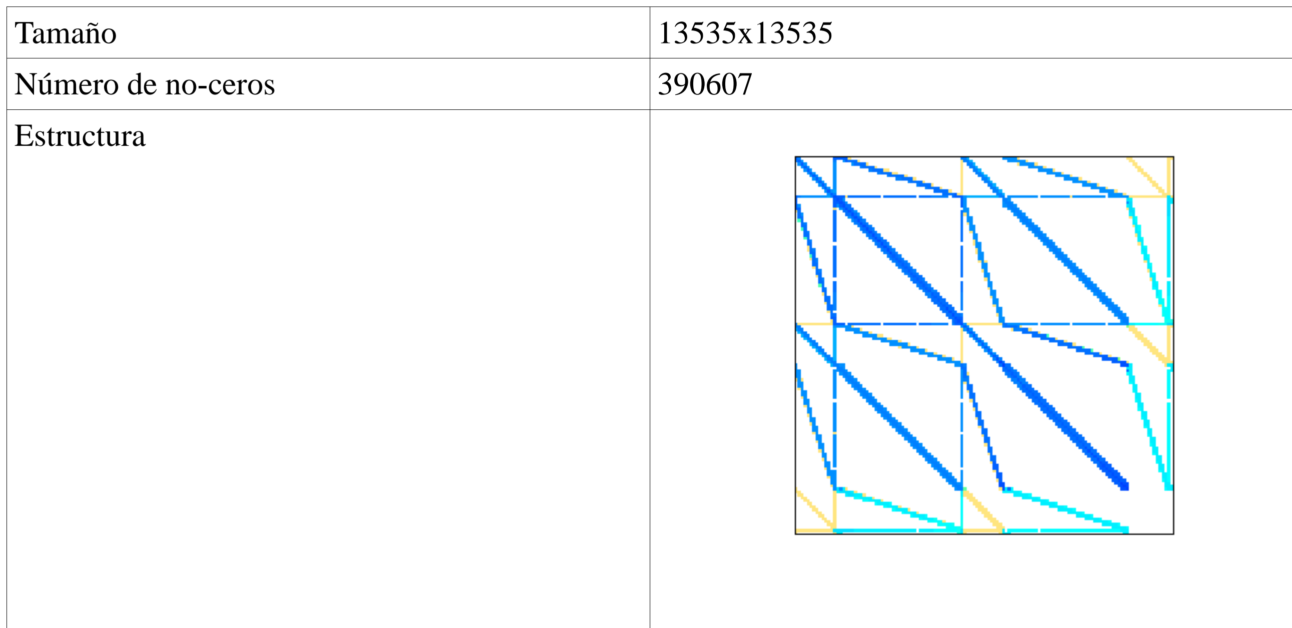
Tamaño	17281x17281
Número de no-ceros	553562
Estructura	

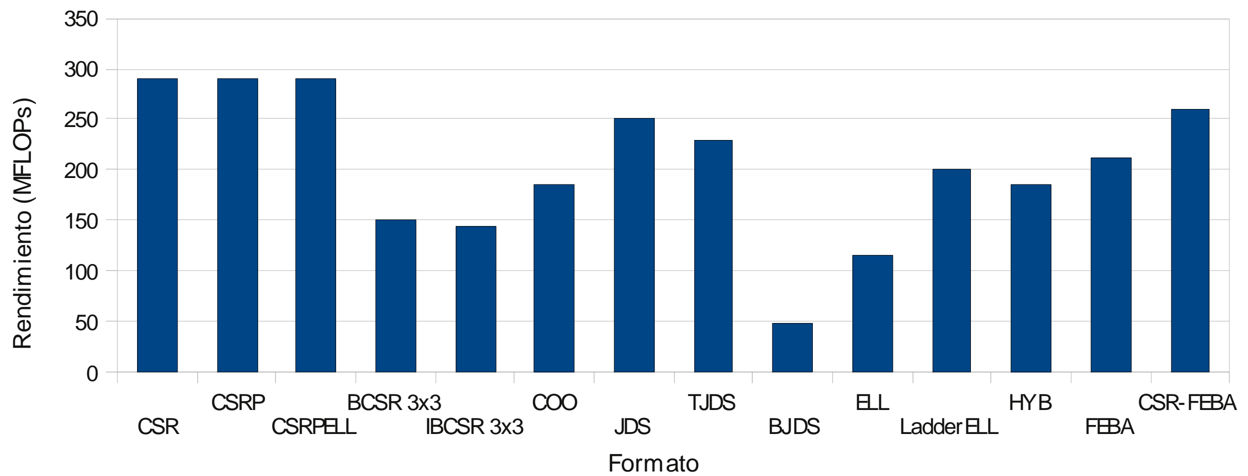


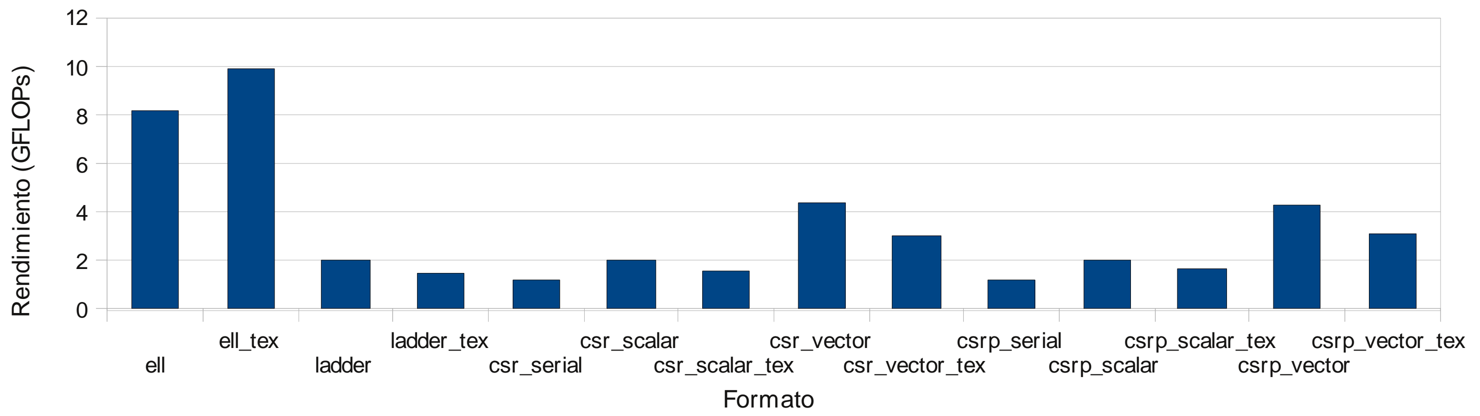

SpMV Nvidia
Matriz e40r0100 - 1
Rendimiento (GFLOPs)
10
9
8
7
6
5
4
3
2
1
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

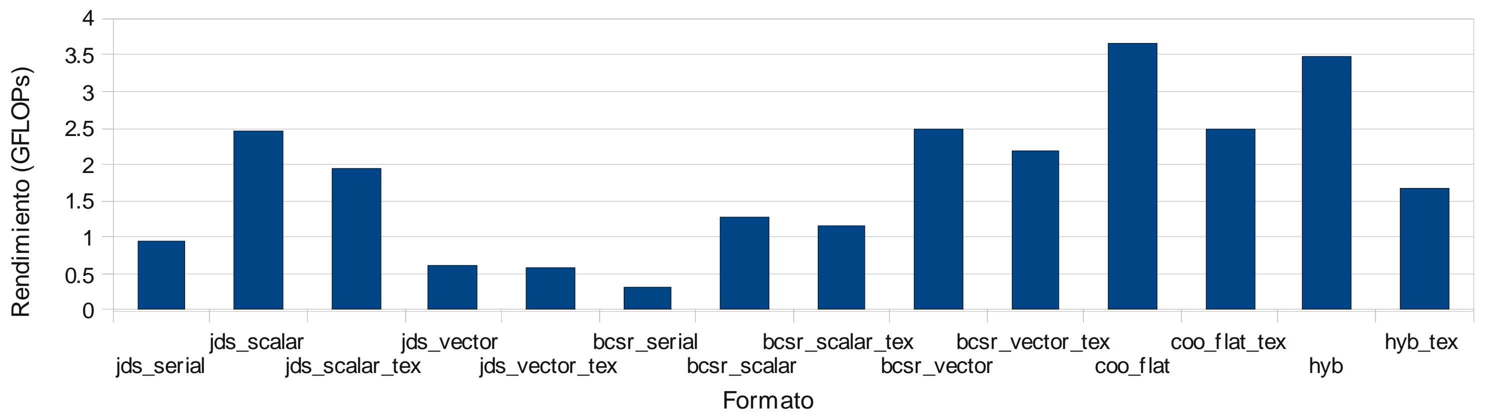

SpMV Nvidia
Matriz e40r0100 - 2
Rendimiento (GFLOPs)
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.3. Matriz garon2

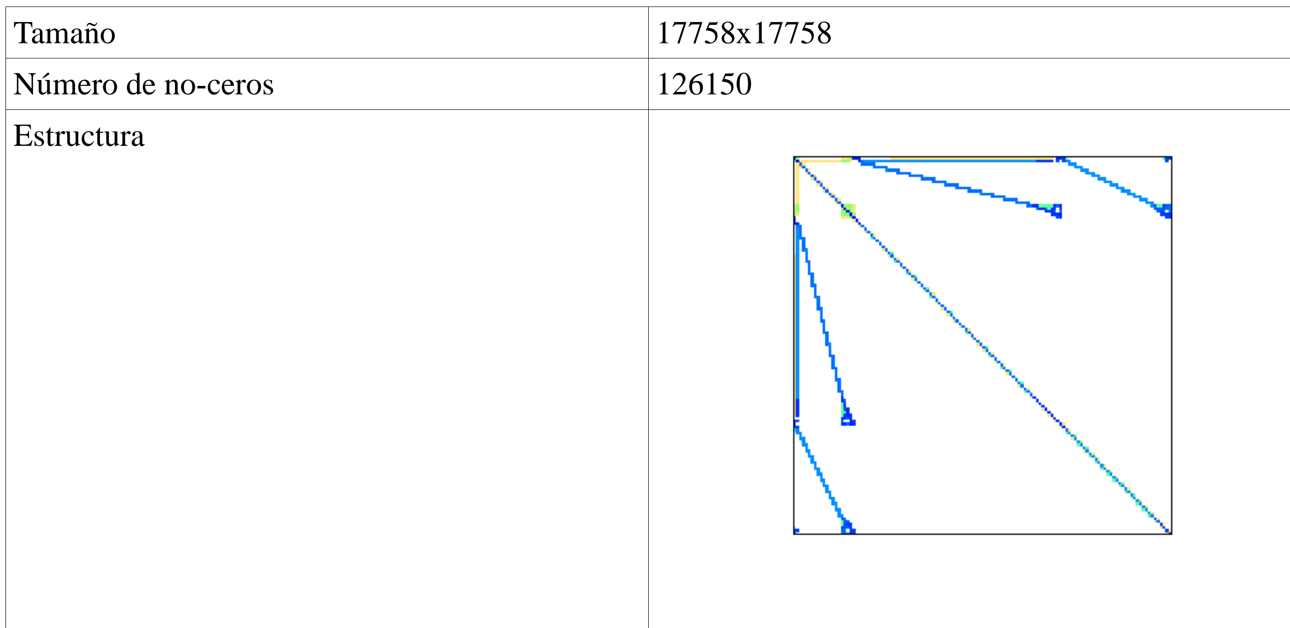
Tamaño	13535x13535
Número de no-ceros	390607
Estructura	

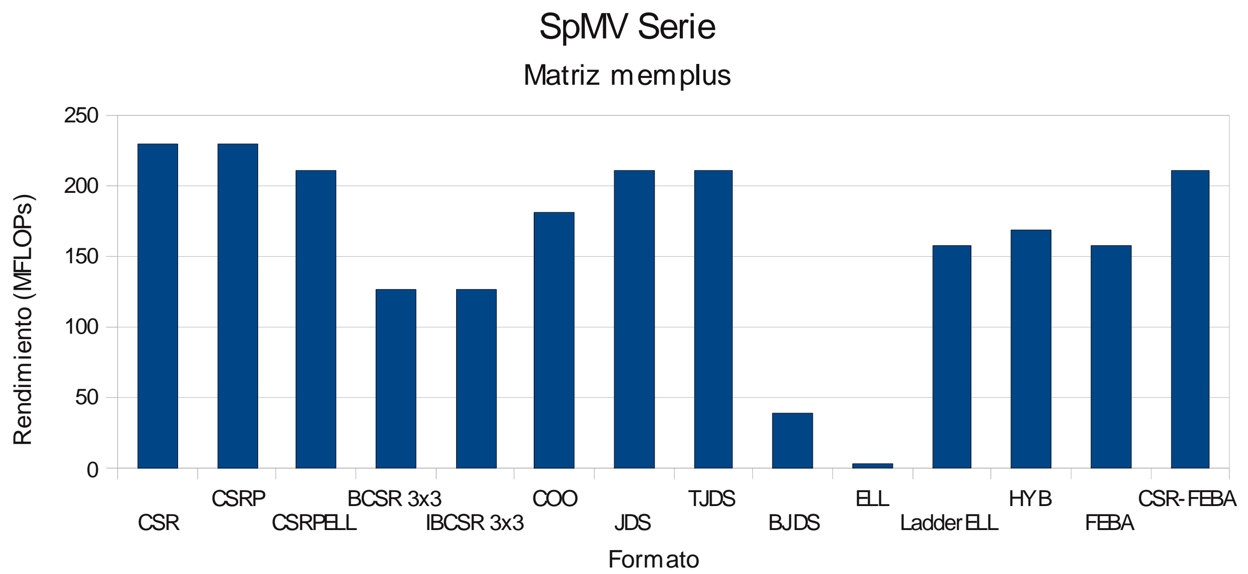


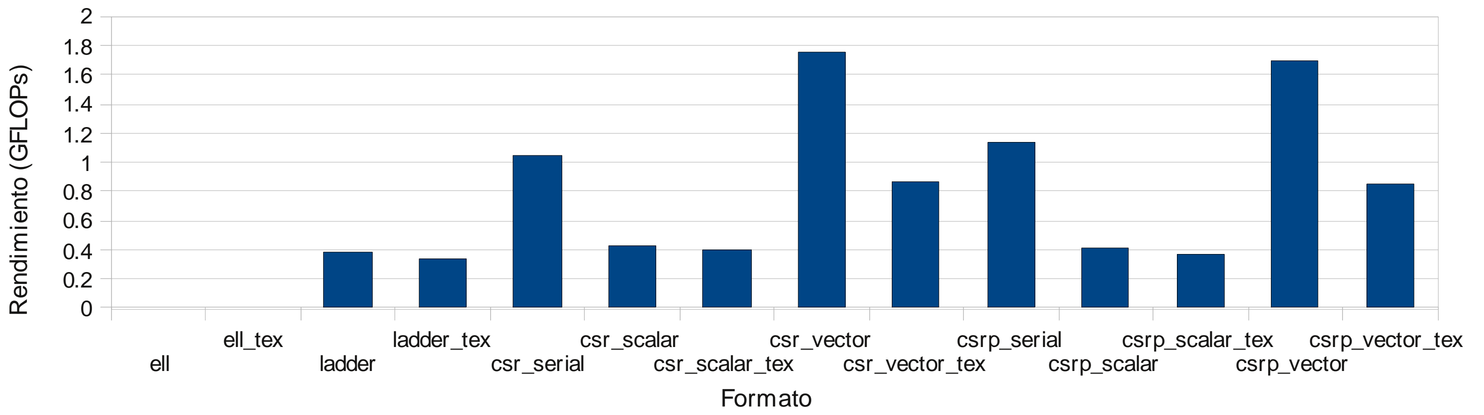

SpMV Nvidia
Matriz garon2 - 1
Rendimiento (GFLOPs)
12
10
8
6
4
2
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

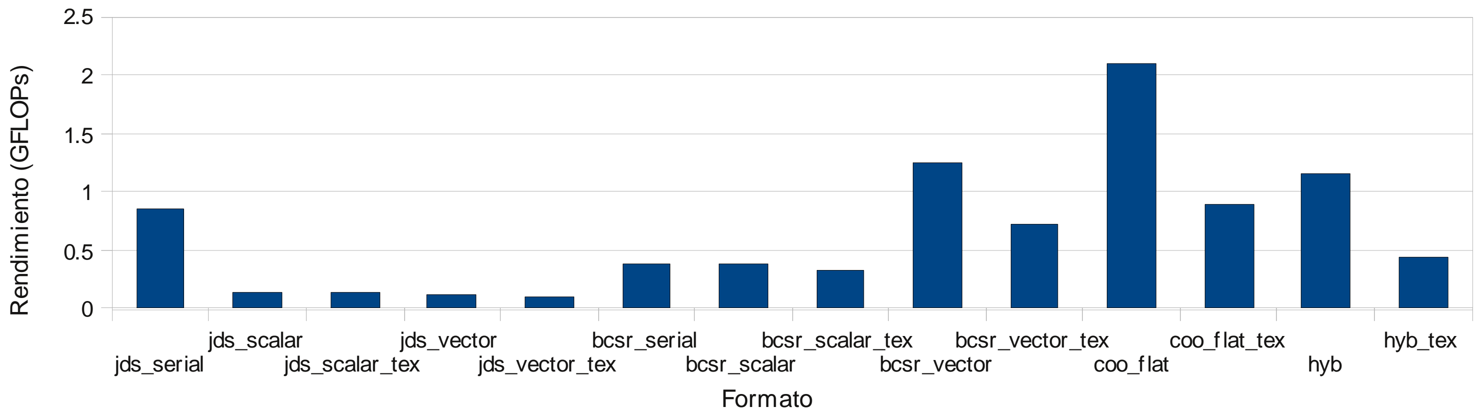

SpMV Nvidia
Matriz garon2 - 2
Rendimiento (GFLOPs)
4
3.5
3
2.5
2
1.5
1
0.5
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.4. Matriz memplus

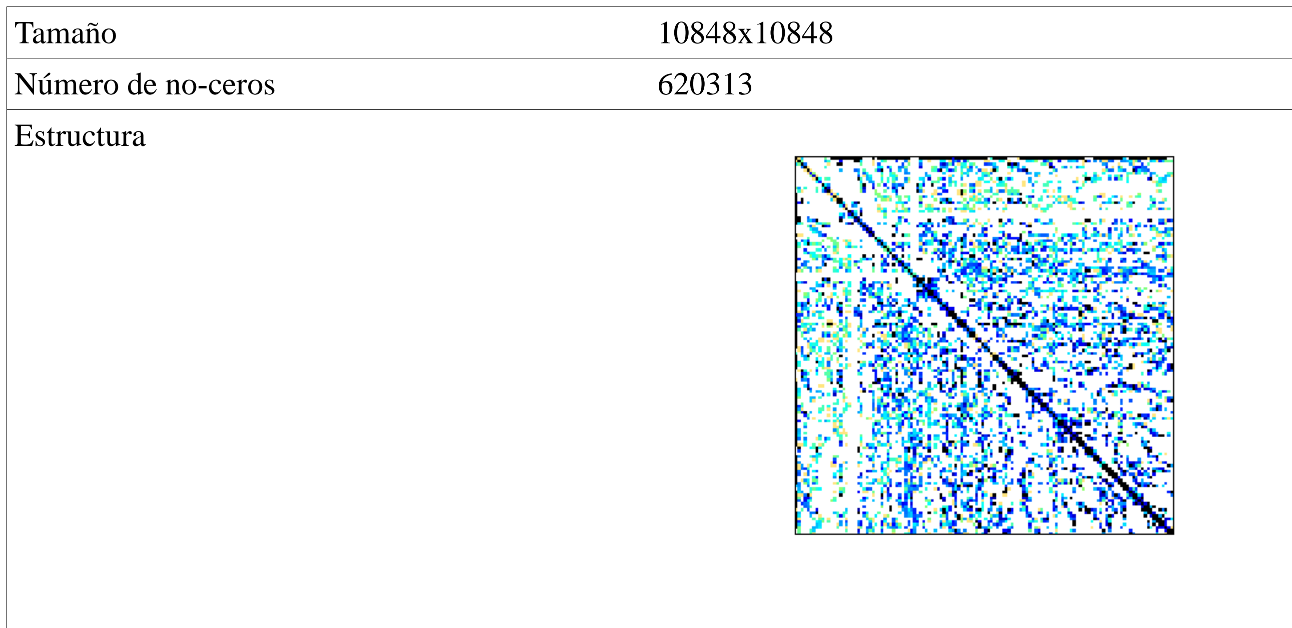
Tamaño	17758x17758
Número de no-ceros	126150
Estructura	

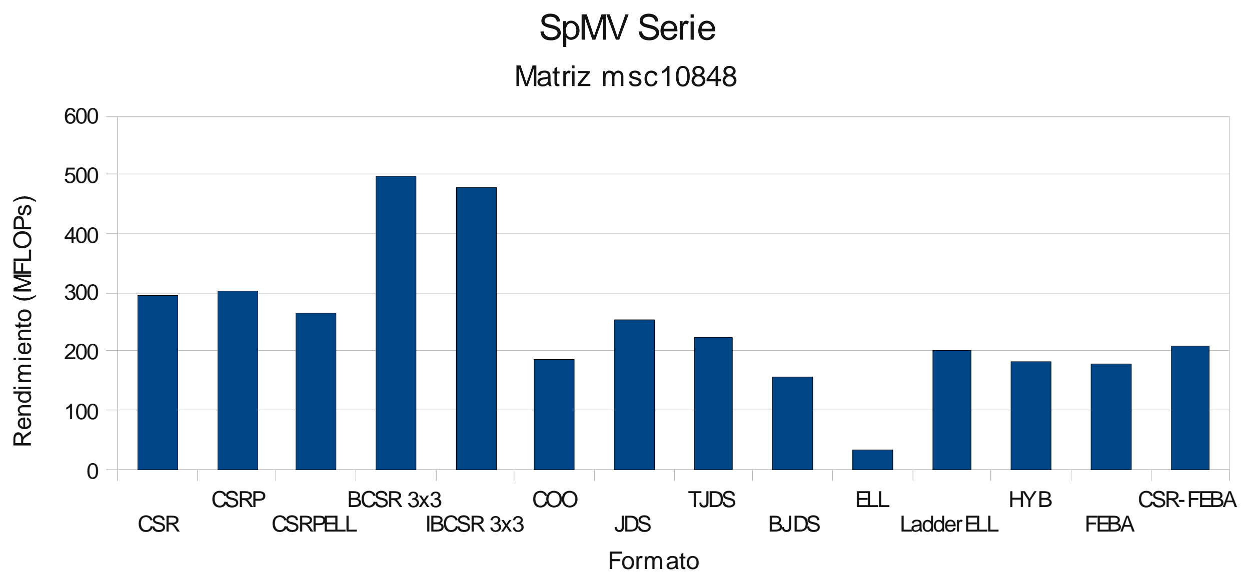


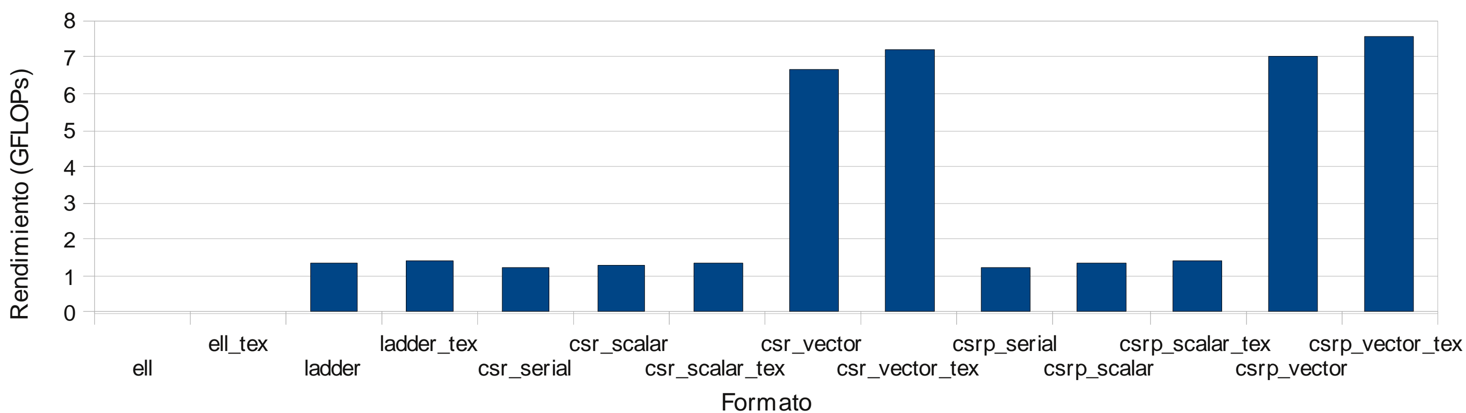

SpMV Nvidia
Matriz memplus - 1
Rendimiento (GFLOPs)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

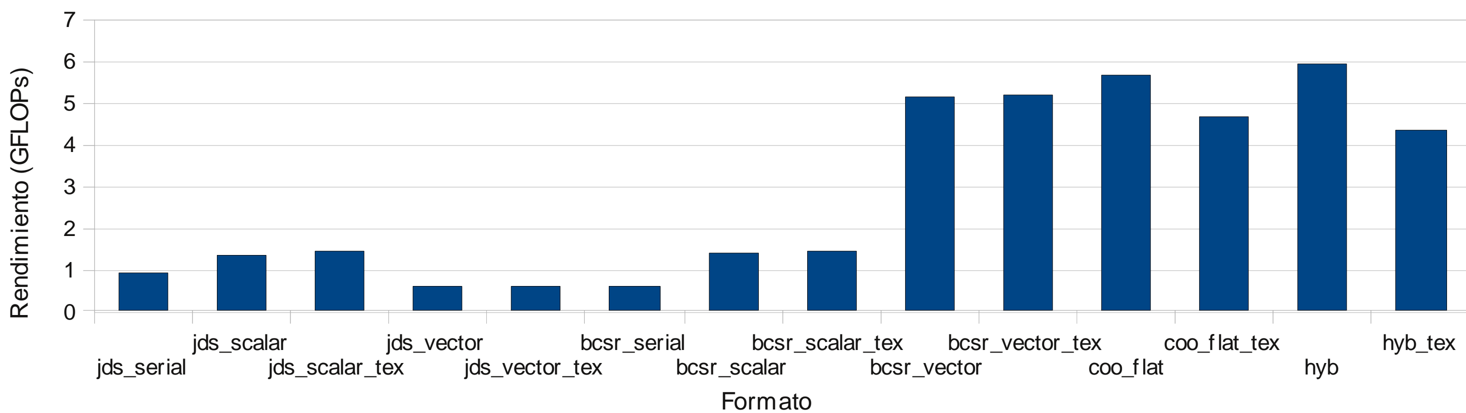

SpMV Nvidia
Matriz memplus - 2
Rendimiento (GFLOPs)
2.5
2
1.5
1
0.5
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.5. Matriz msc10848

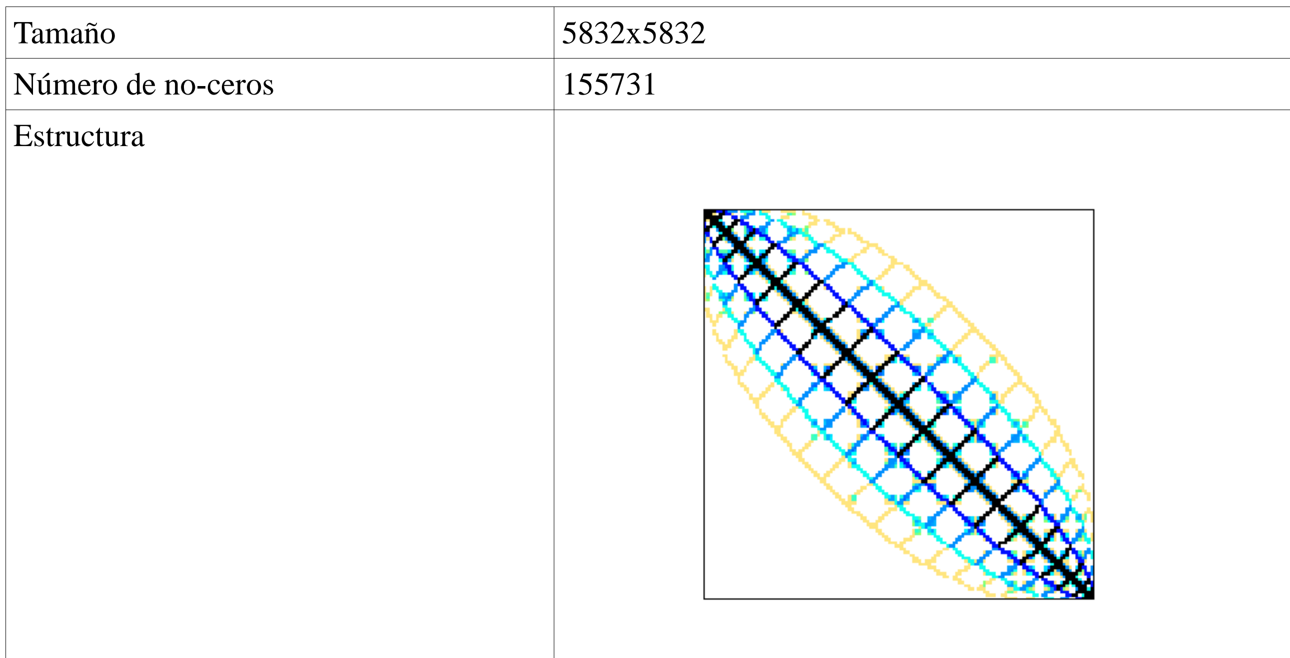
Tamaño	10848x10848
Número de no-ceros	620313
Estructura	



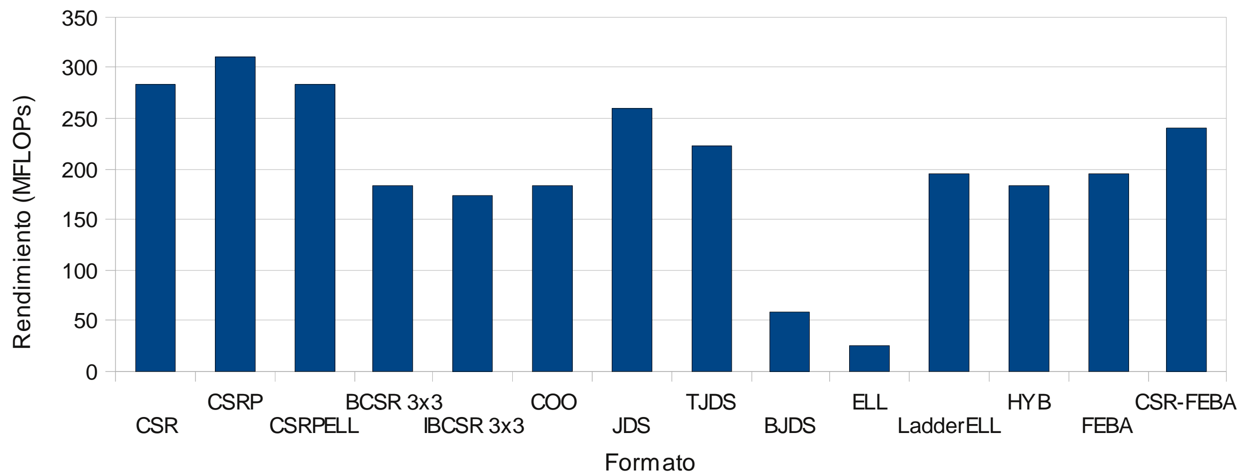

SpMV Nvidia
Matriz msc10848 - 1
Rendimiento (GFLOPs)
8
7
6
5
4
3
2
1
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

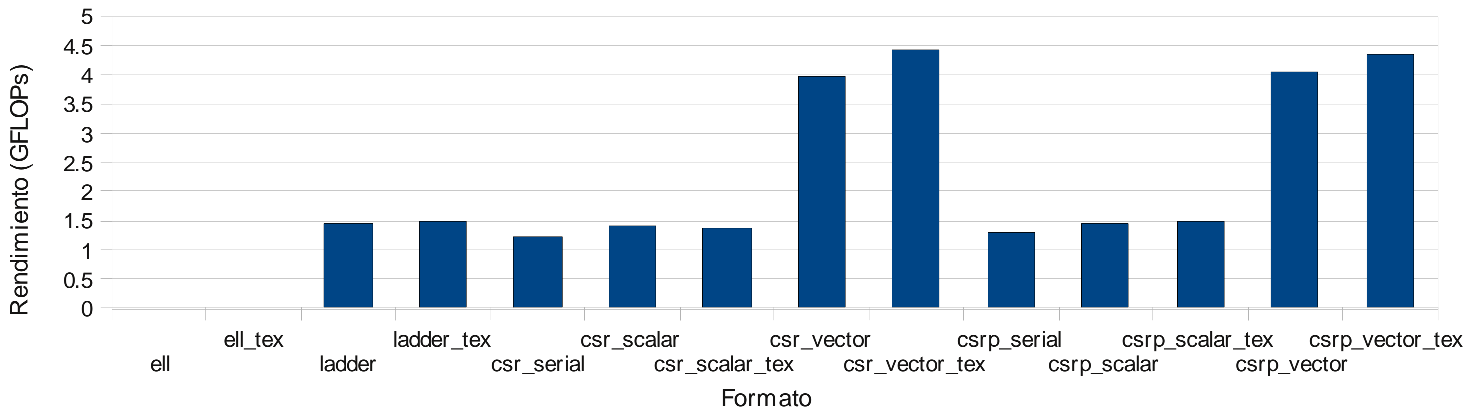

SpMV Nvidia
Matriz msc10848 - 2
Rendimiento (GFLOPs)
7
6
5
4
3
2
1
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

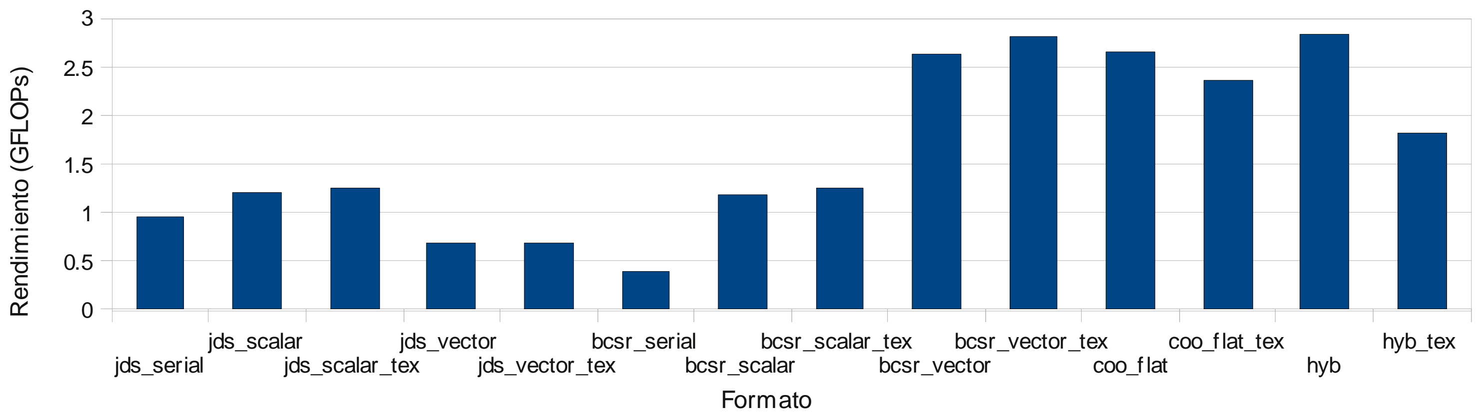
6.6. Matriz Na5



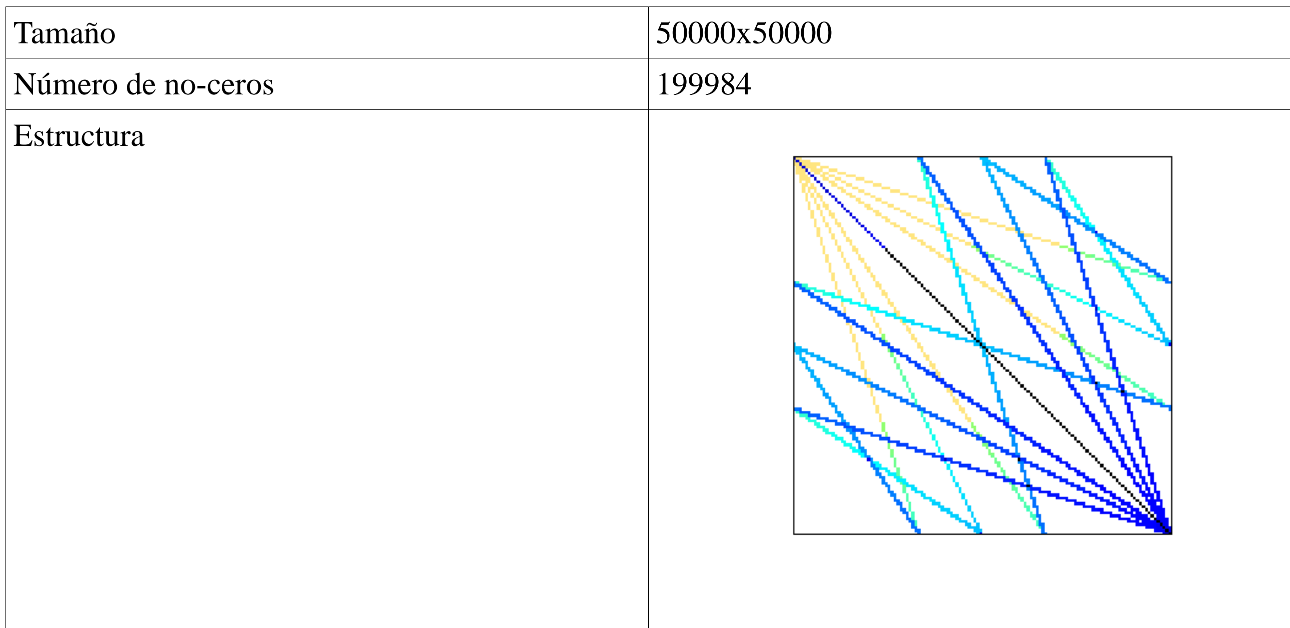
Tamaño	5832x5832
Número de no-ceros	155731
Estructura	

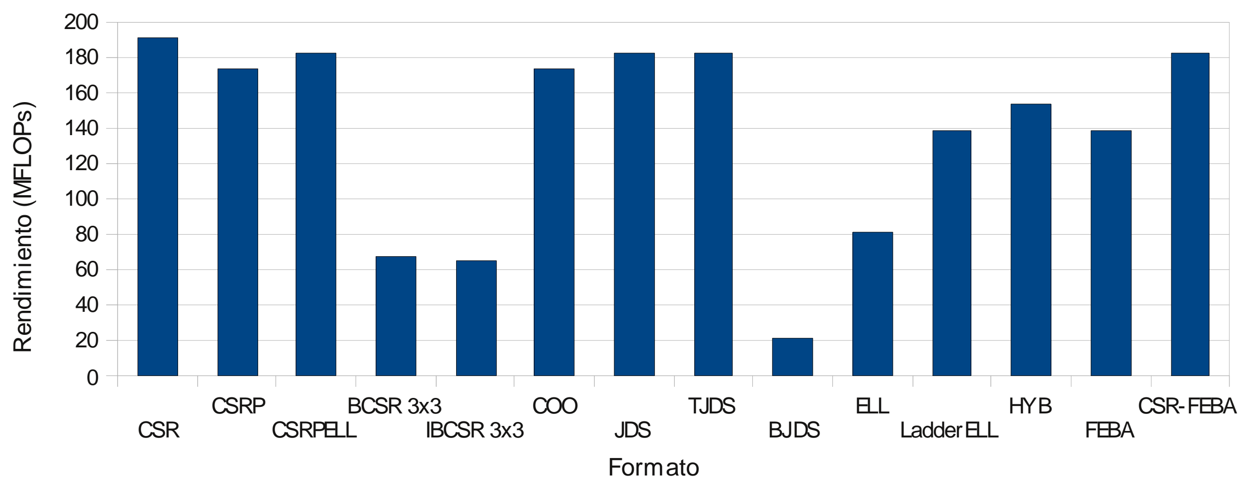


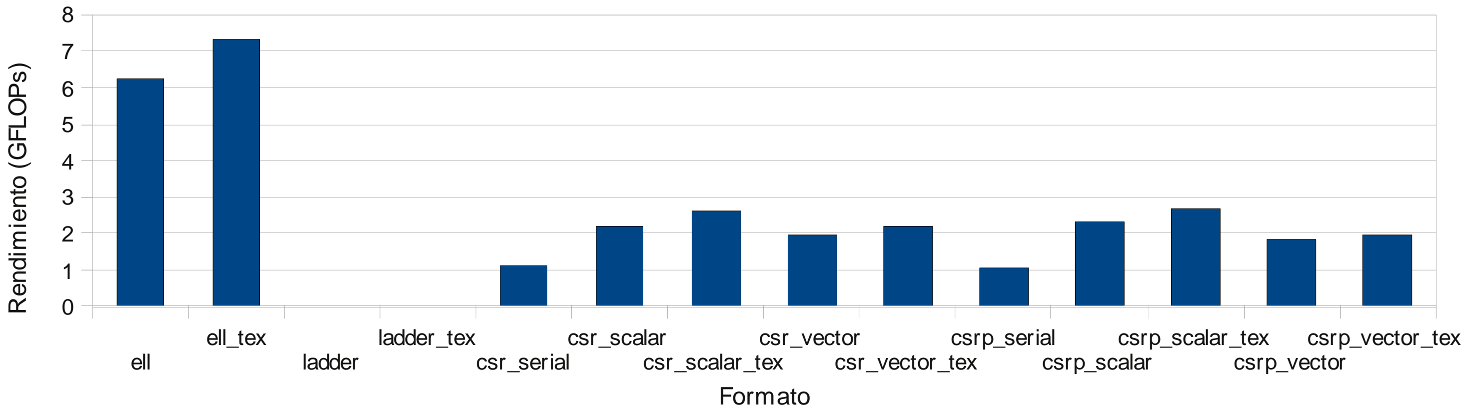

SpMV Nvidia
Matriz Na5 - 1
Rendimiento (GFLOPs)
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

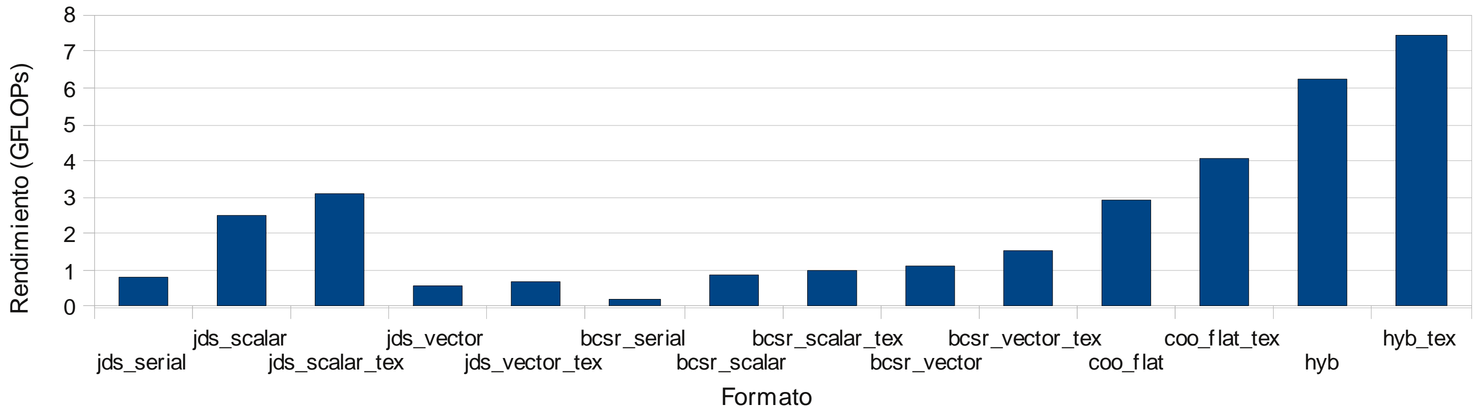

SpMV Nvidia
Matriz Na5 - 2
Rendimiento (GFLOPs)
3
2.5
2
1.5
1
0.5
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.7. Matriz ncvxbqp1

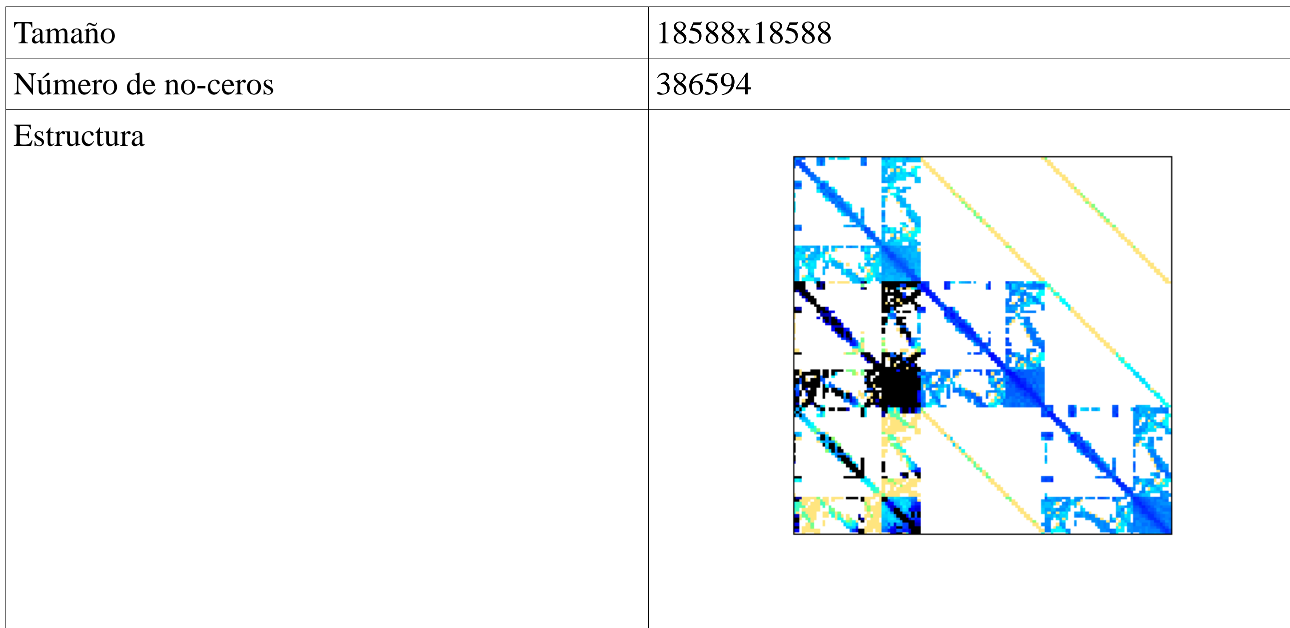
Tamaño	50000x50000
Número de no-ceros	199984
Estructura	

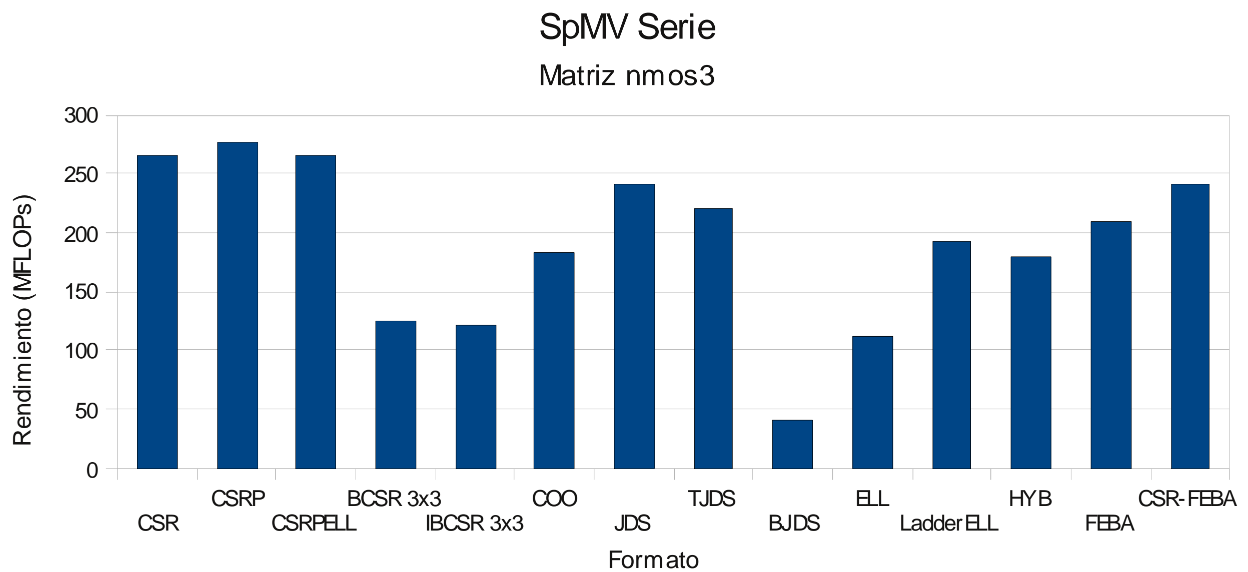


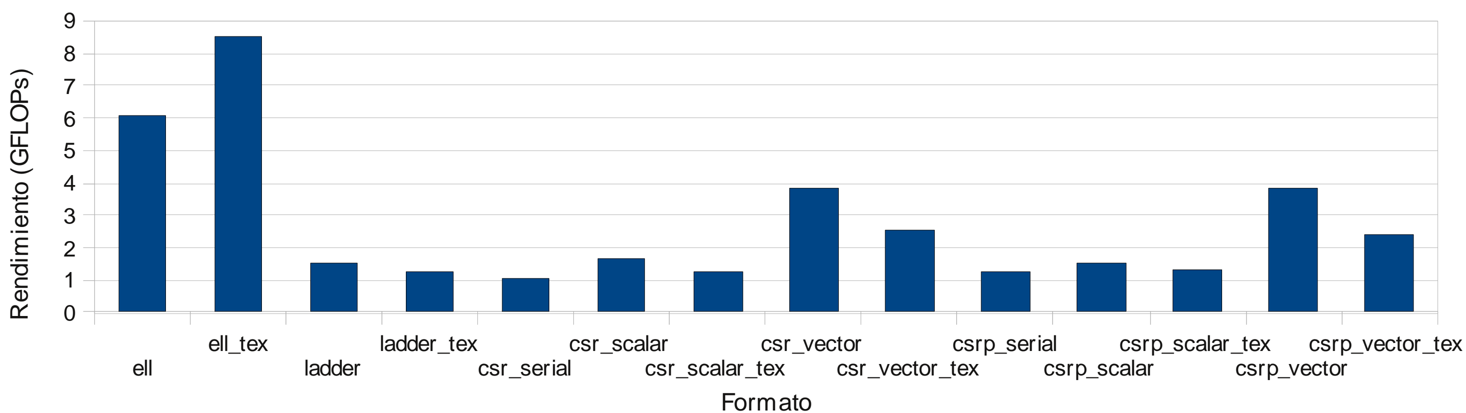

SpMV Nvidia
Matriz ncvxbqp1 - 1
Rendimiento (GFLOPs)
0
1
2
3
4
5
6
7
8
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

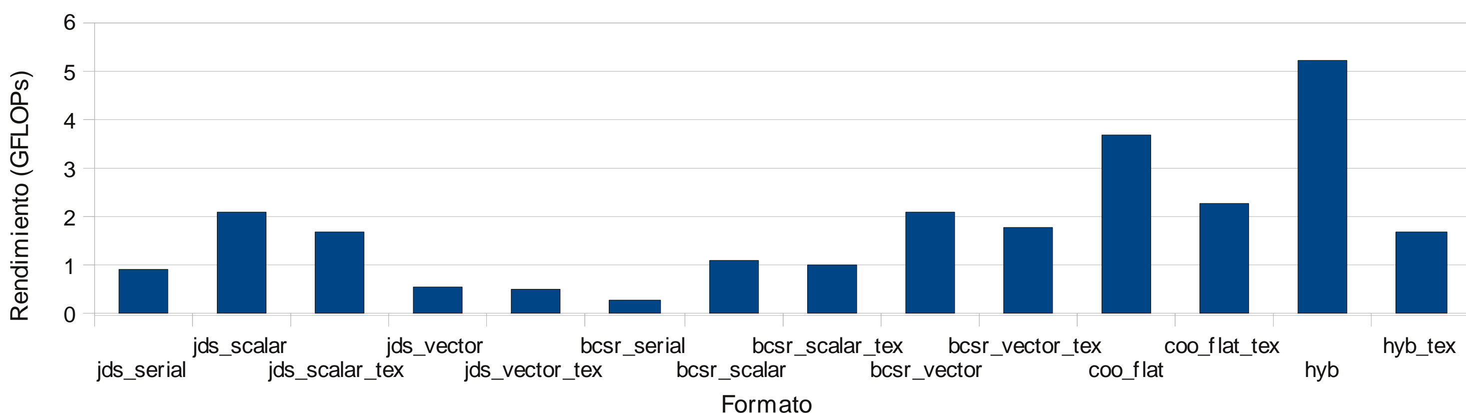

SpMV Nvidia
Matriz ncvxbqp1 - 2
Rendimiento (GFLOPs)
0
1
2
3
4
5
6
7
8
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.8. Matriz nmos3

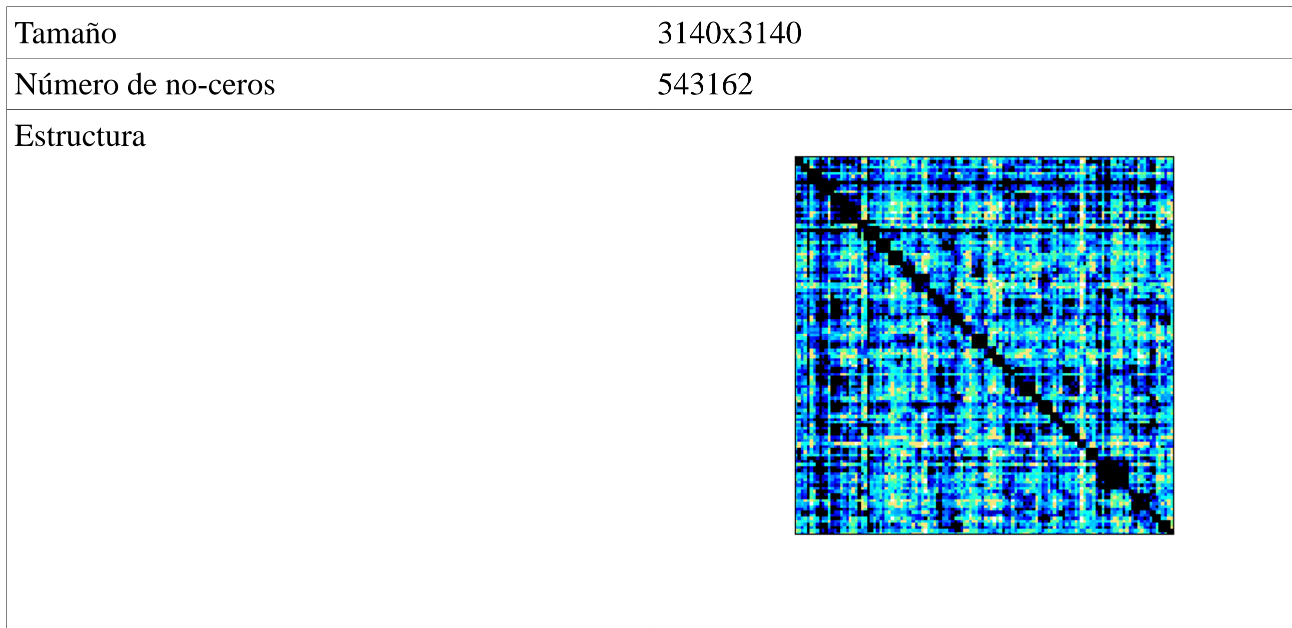
Tamaño	18588x18588
Número de no-ceros	386594
Estructura	

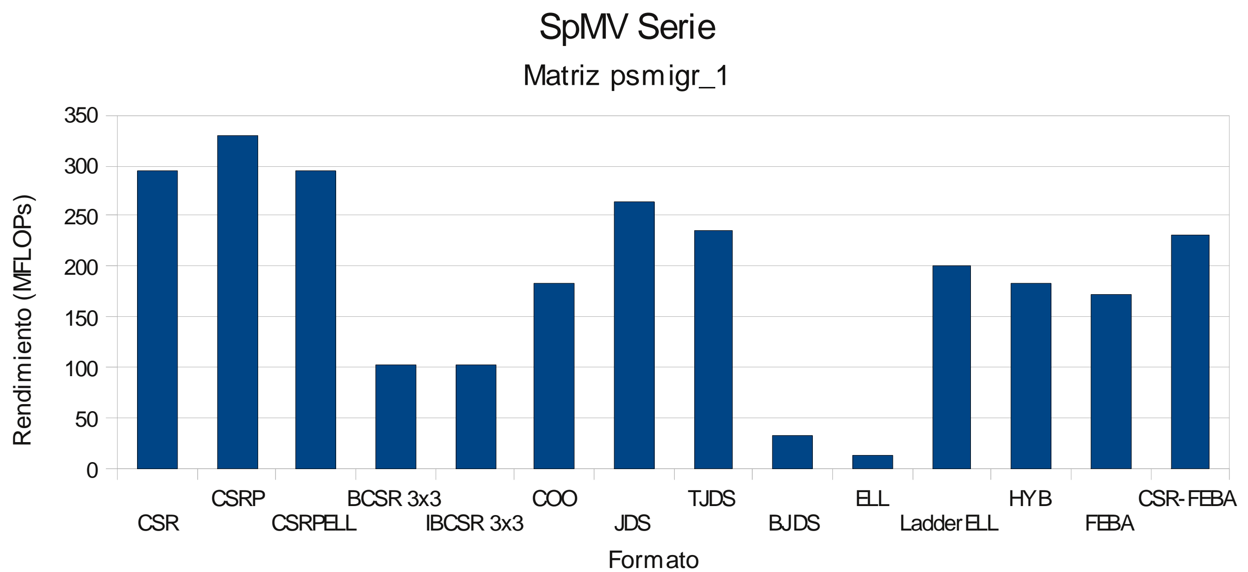


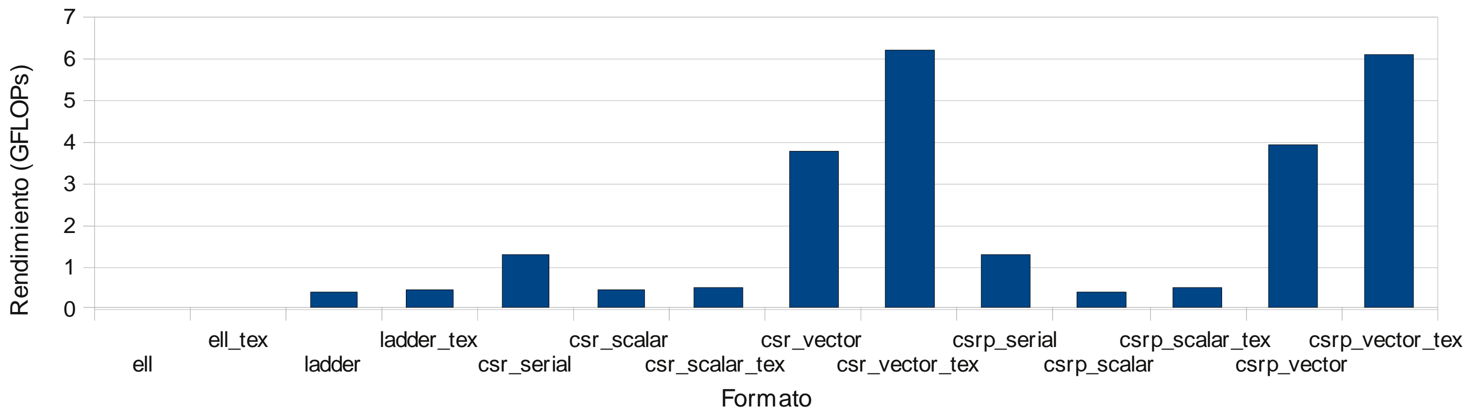

SpMV Nvidia
Matriz nmos3 - 1
Rendimiento (GFLOPs)
9
8
7
6
5
4
3
2
1
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

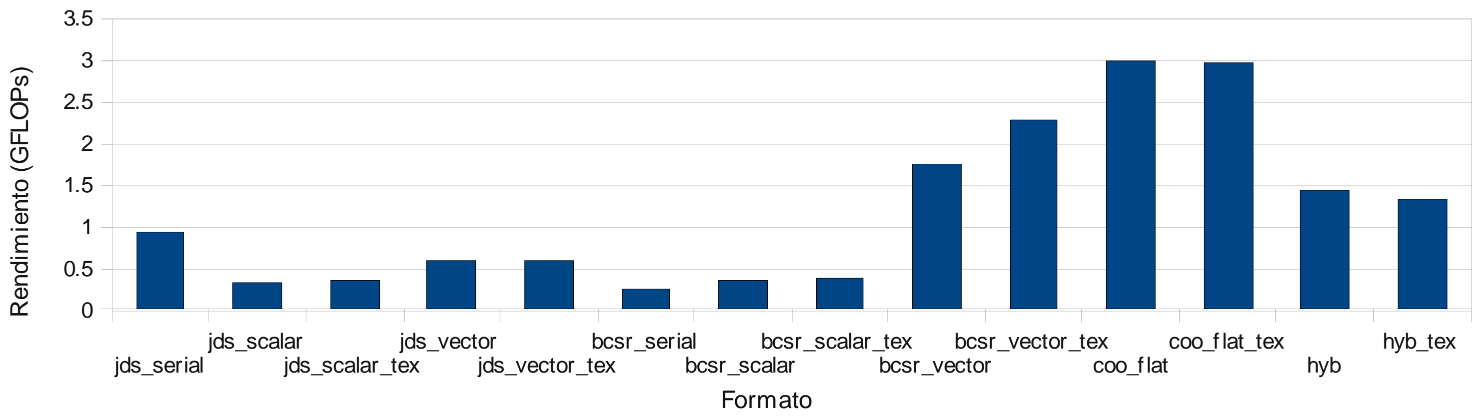

SpMV Nvidia
Matriz nmos3 - 2
Rendimiento (GFLOPs)
6
5
4
3
2
1
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.9. Matriz psmigr_1

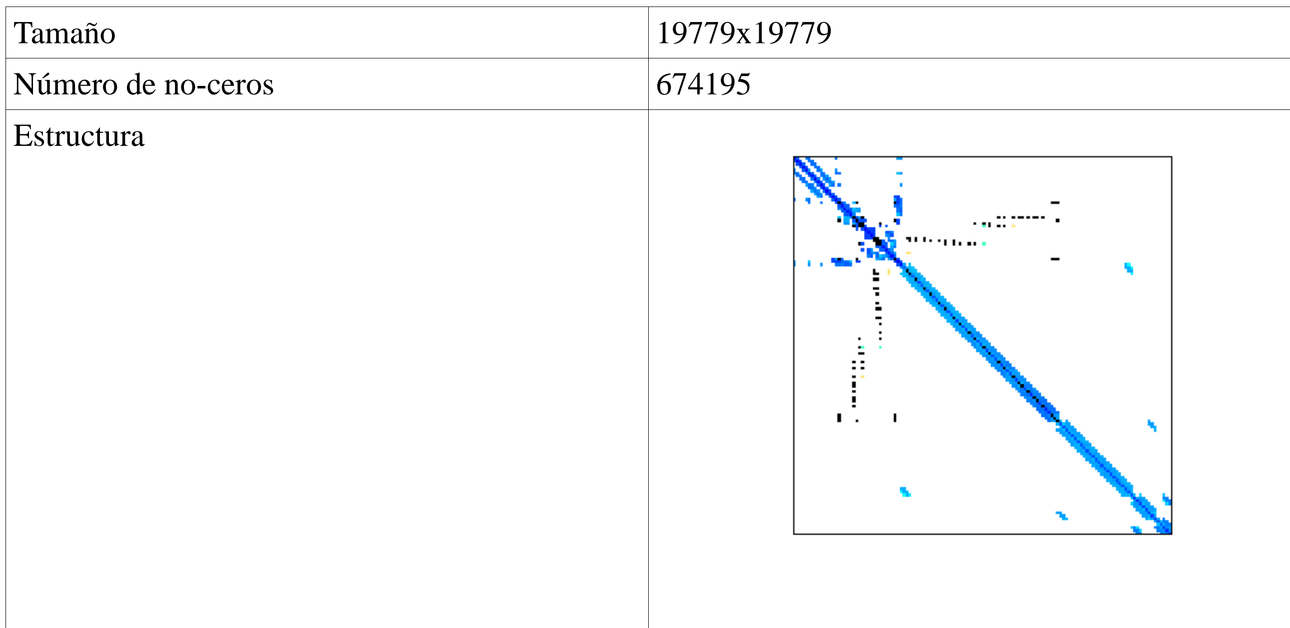
Tamaño	3140x3140
Número de no-ceros	543162
Estructura	

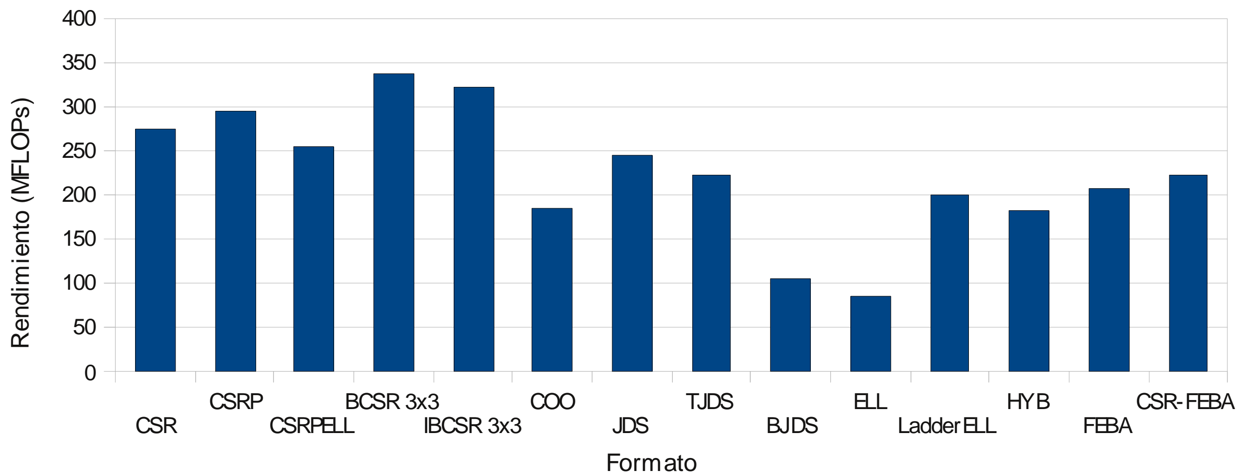


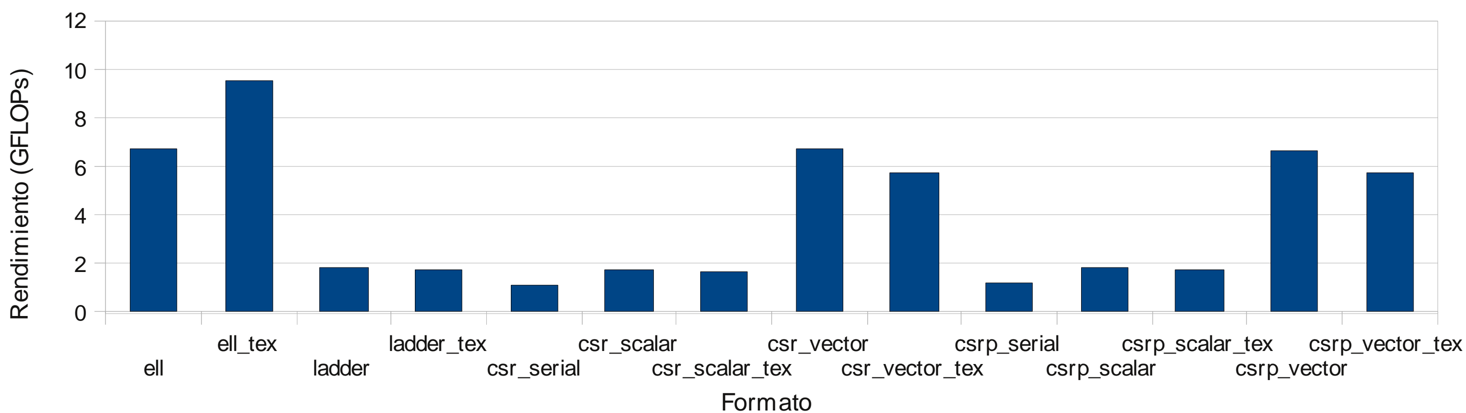

SpMV Nvidia
Matriz psmigr_1 - 1
Rendimiento (GFLOPs)
7
6
5
4
3
2
1
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

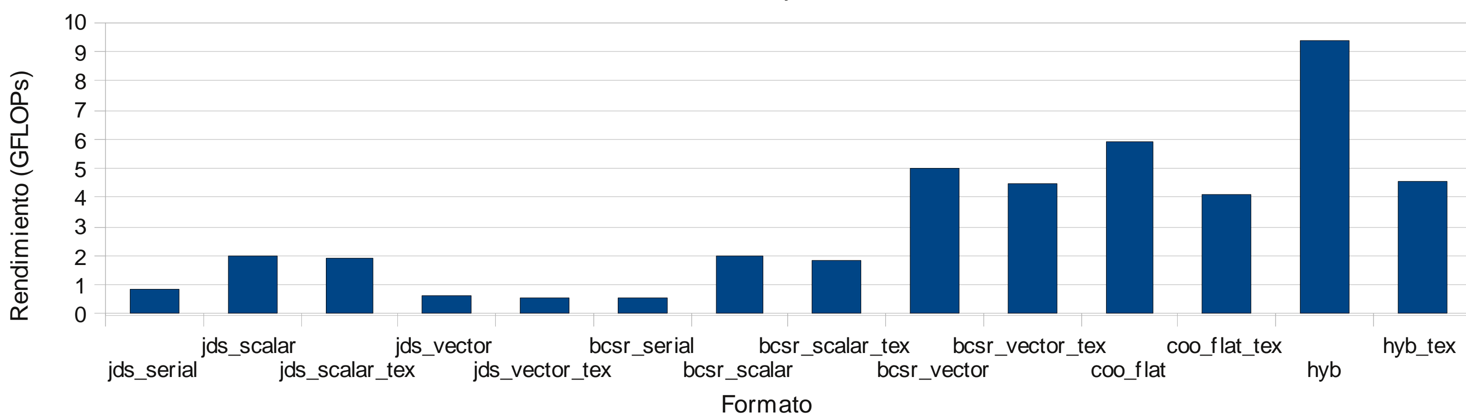

SpMV Nvidia
Matriz psmigr_1 - 2
Rendimiento (GFLOPs)
3.5
3
2.5
2
1.5
1
0.5
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.10. Matriz raefsky4

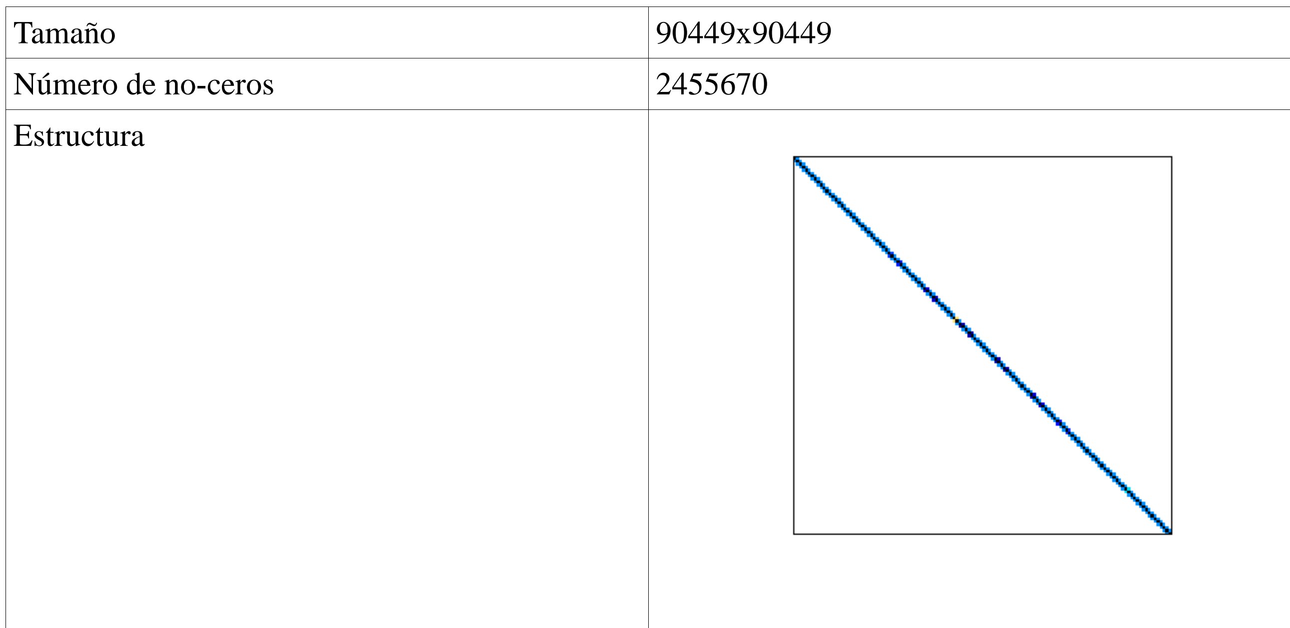
Tamaño	19779x19779
Número de no-ceros	674195
Estructura	

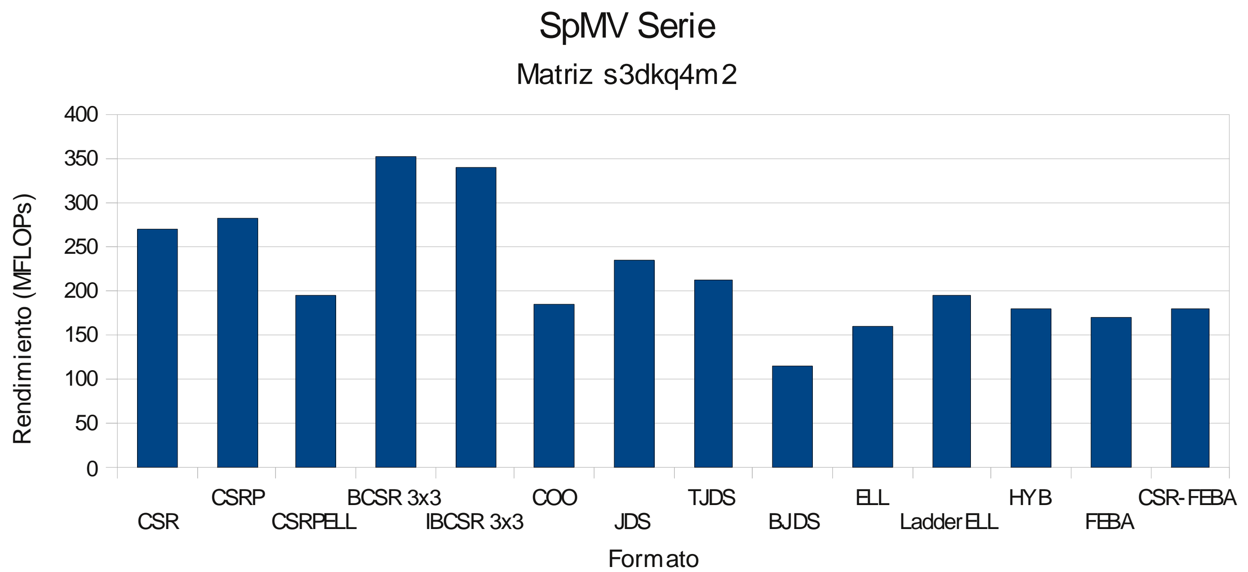


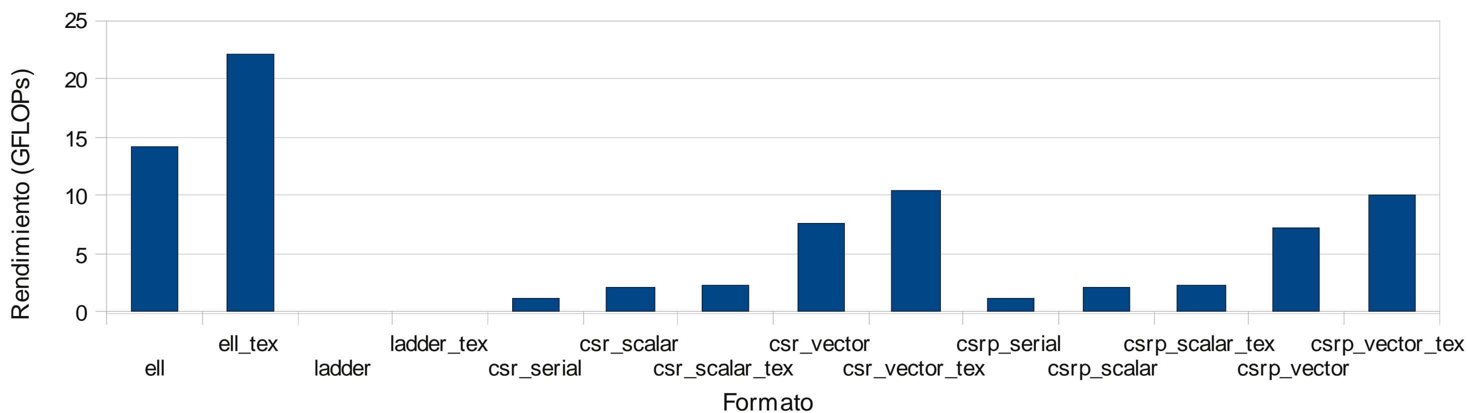

SpMV Nvidia
Matriz raefsky4 - 1
Rendimiento (GFLOPs)
12
10
8
6
4
2
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

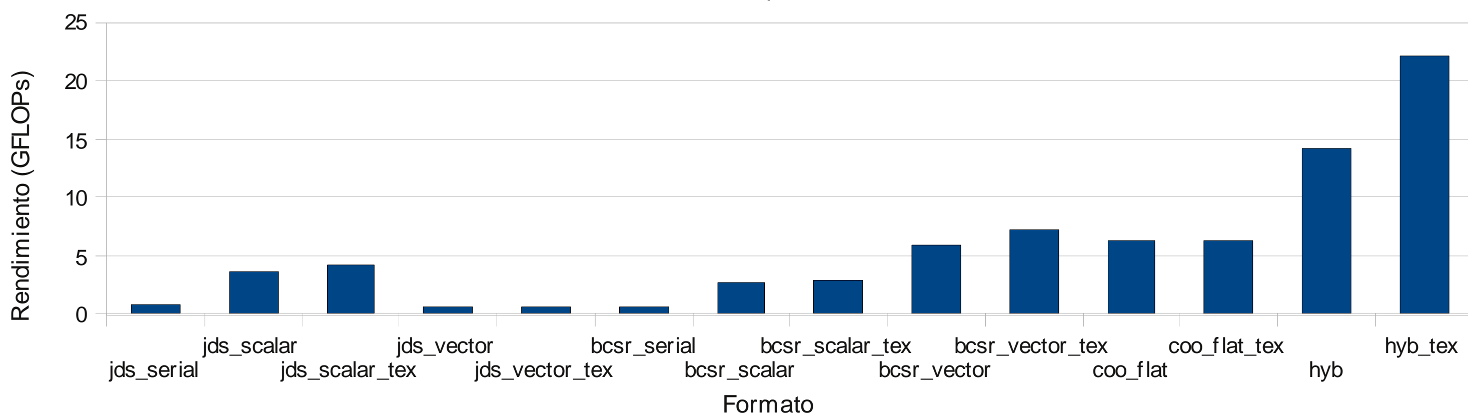

SpMV Nvidia
Matriz raefsky4 - 2
Rendimiento (GFLOPs)
10
9
8
7
6
5
4
3
2
1
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.11. *Matriz s3dkq4m2*

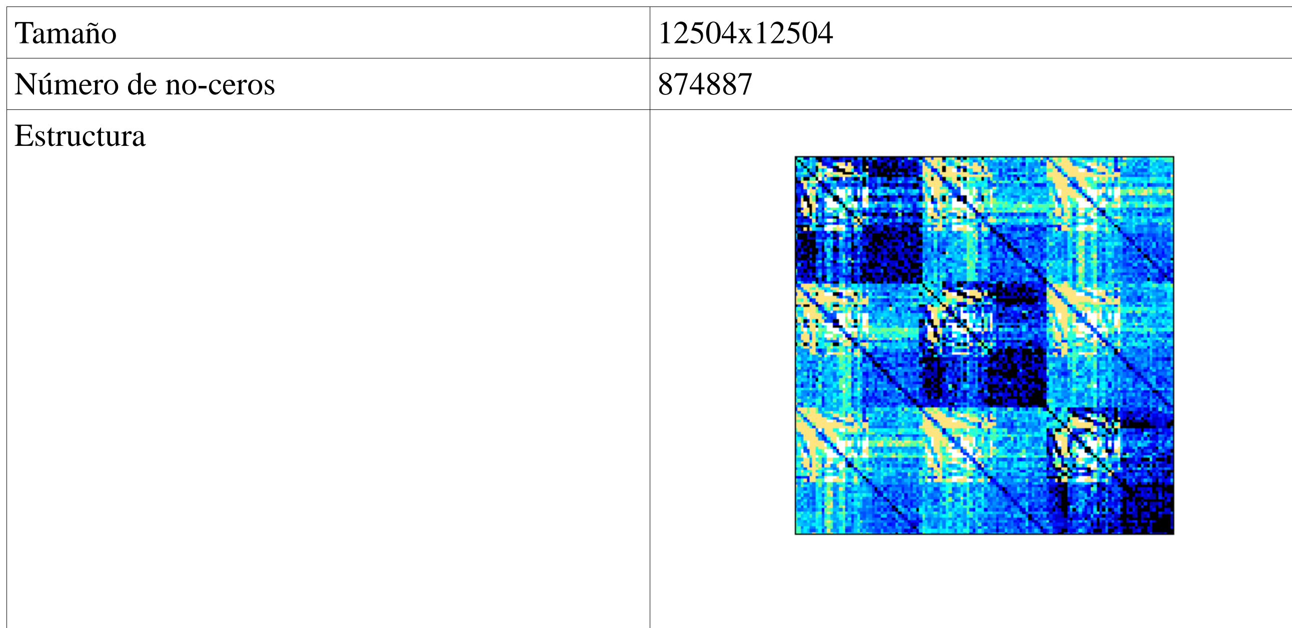
Tamaño	90449x90449
Número de no-ceros	2455670
Estructura	

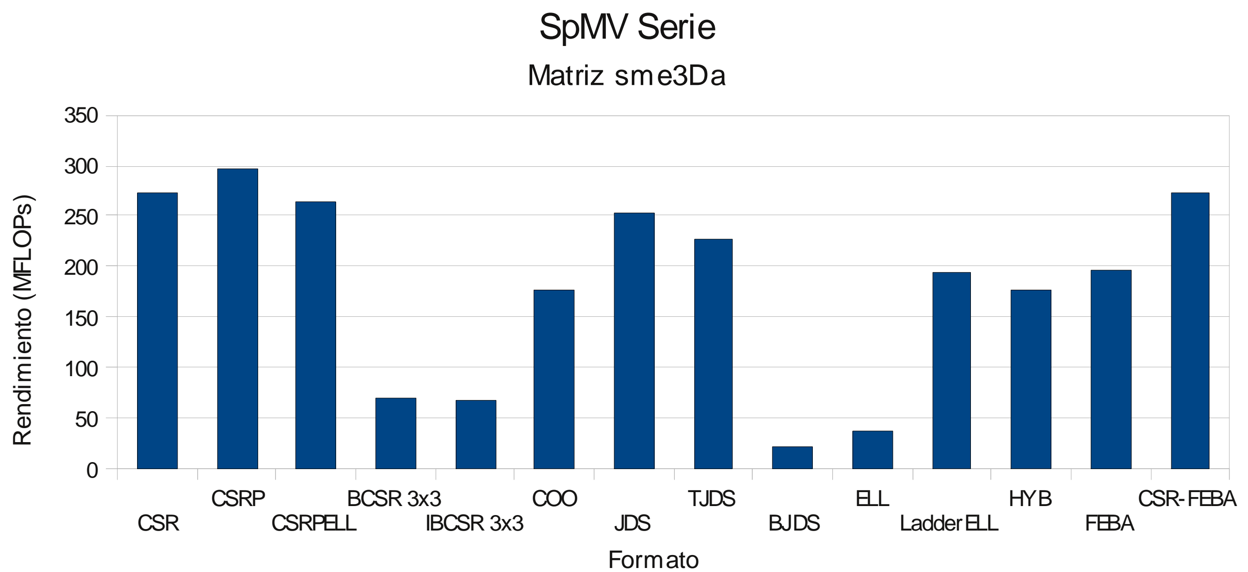


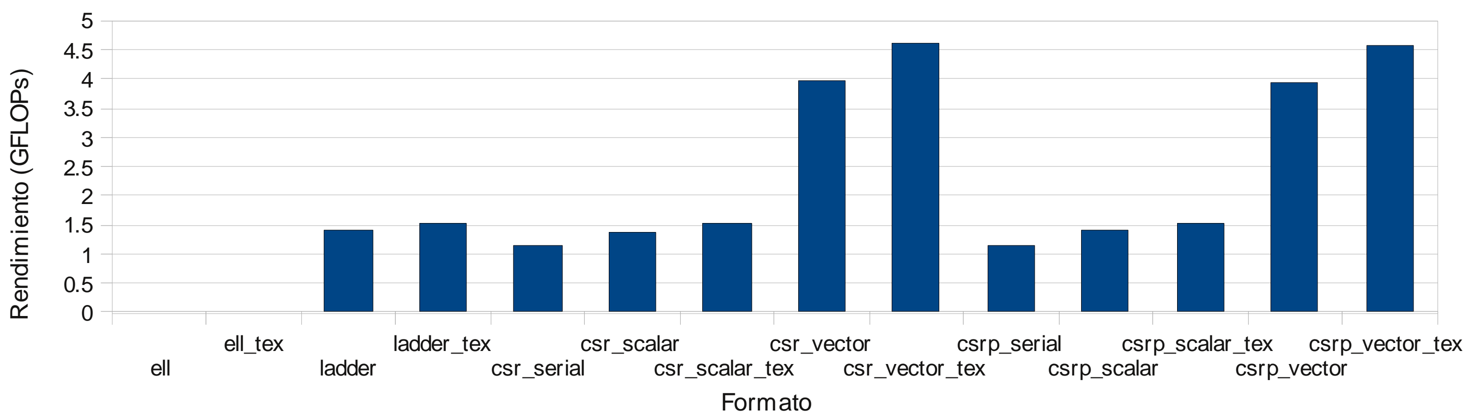

SpMV Nvidia
Matriz s3dkq4m2 - 1
Rendimiento (GFLOPs)
25
20
15
10
5
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

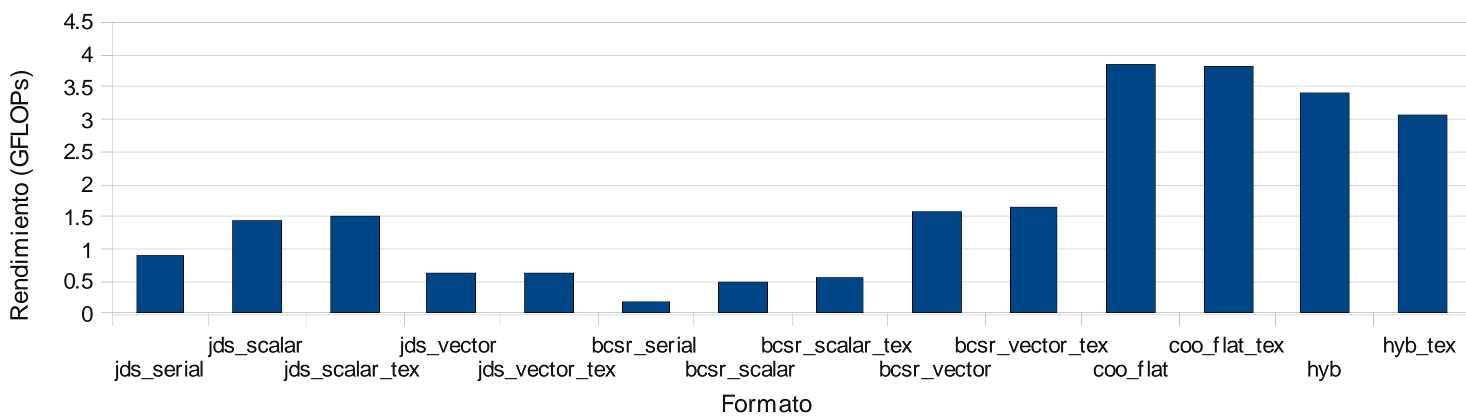

SpMV Nvidia
Matriz s3dkq4m2 - 2
Rendimiento (GFLOPs)
25
20
15
10
5
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.12. Matriz sme3Da

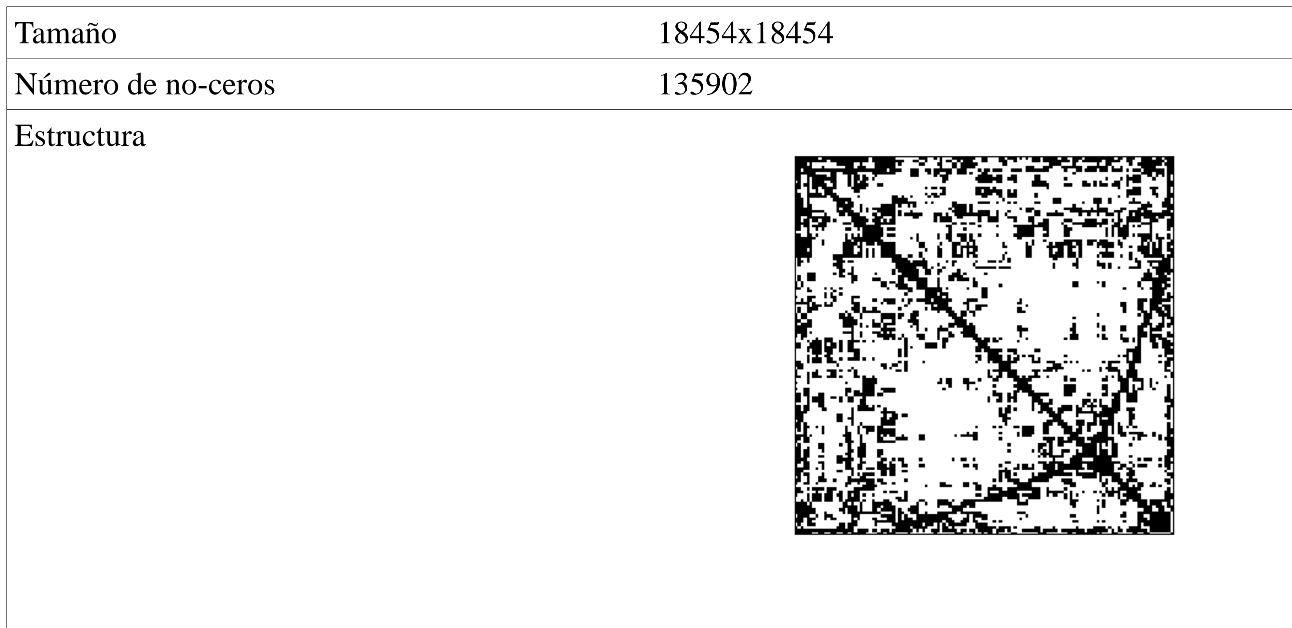
Tamaño	12504x12504
Número de no-ceros	874887
Estructura	

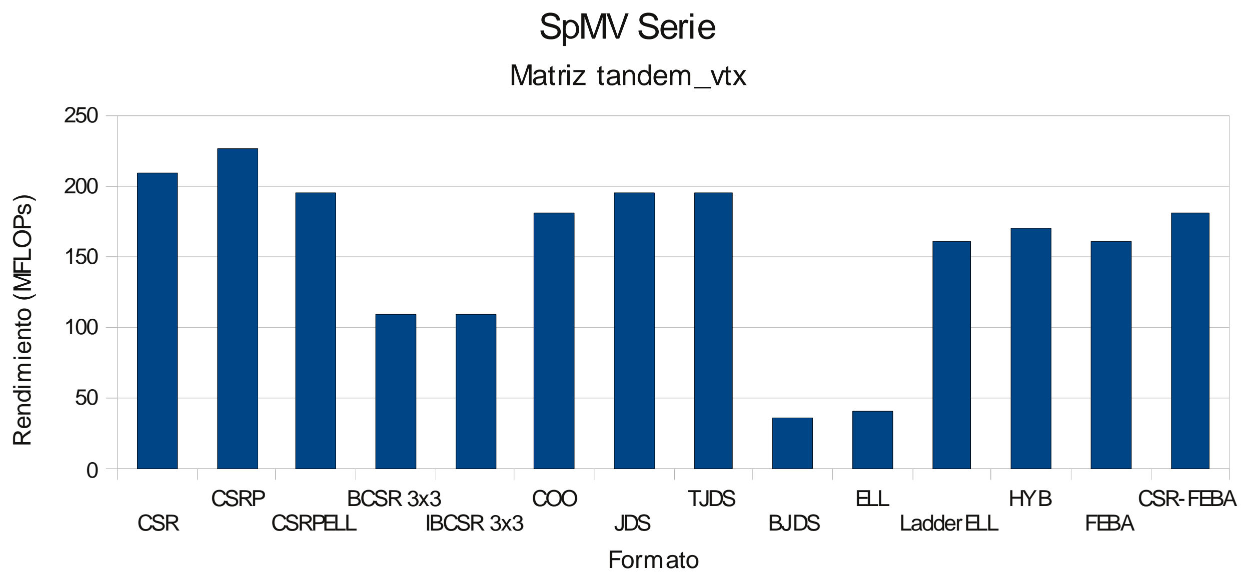


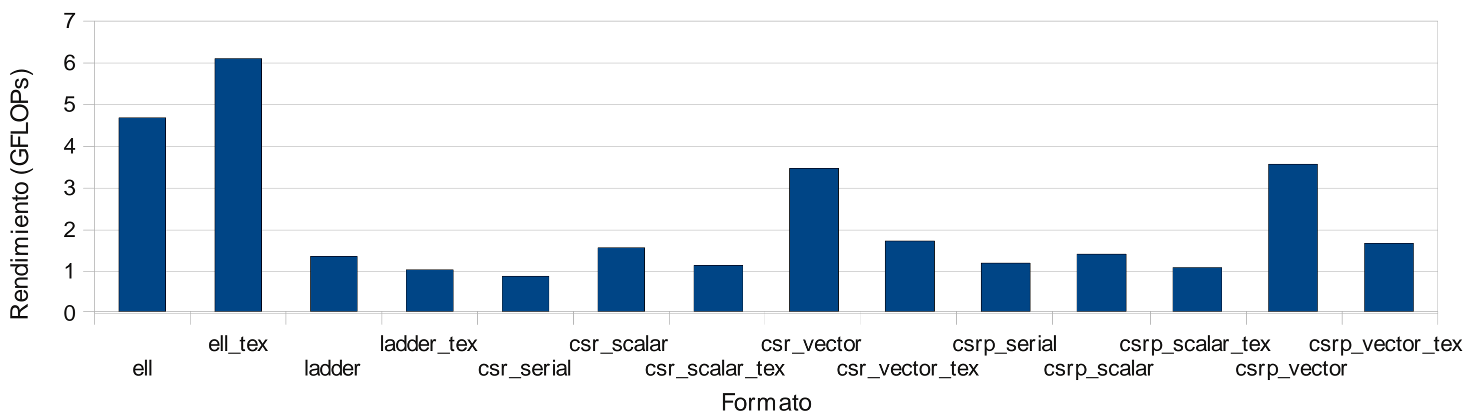

SpMV Nvidia
Matriz sme3Da - 1
Rendimiento (GFLOPs)
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato

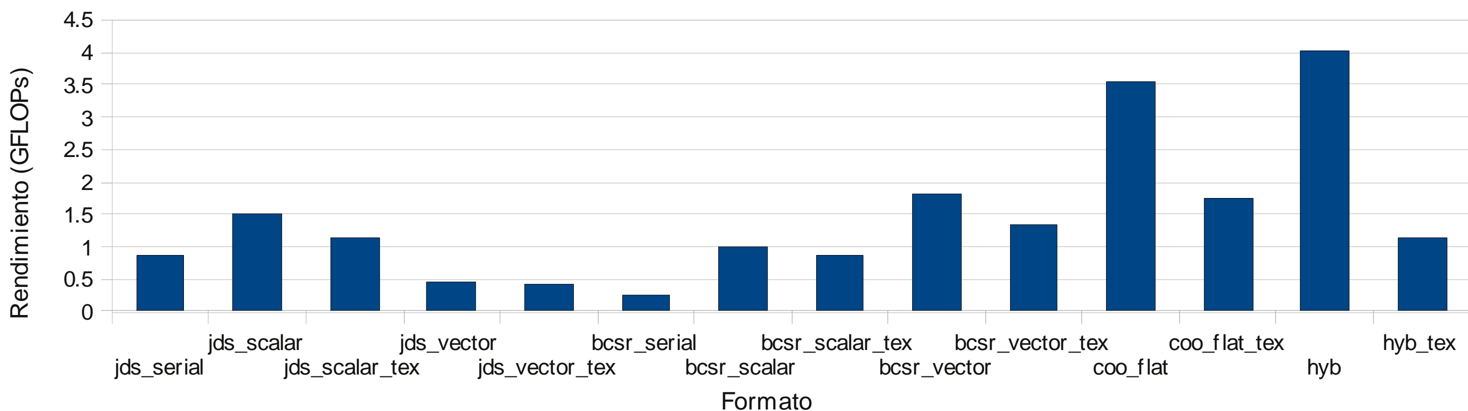

SpMV Nvidia
Matriz sme3Da - 2
Rendimiento (GFLOPs)
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato

6.13. Matriz tandem_vtx

Tamaño	18454x18454
Número de no-ceros	135902
Estructura	




SpMV Nvidia
Matriz tandem_vtx - 1
Rendimiento (GFLOPs)
0
1
2
3
4
5
6
7
ell
ell_tex
ladder
ladder_tex
csr_serial
csr_scalar
csr_scalar_tex
csr_vector
csr_vector_tex
csrp_serial
csrp_scalar
csrp_scalar_tex
csrp_vector
csrp_vector_tex
Formato


SpMV Nvidia
Matriz tandem_vtx - 2
Rendimiento (GFLOPs)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
jds_serial
jds_scalar
jds_scalar_tex
jds_vector
jds_vector_tex
bcsr_serial
bcsr_scalar
bcsr_scalar_tex
bcsr_vector
bcsr_vector_tex
coo_flat
coo_flat_tex
hyb
hyb_tex
Formato