- ORIGINAL ARTICLE -

Leveraging index compression techniques to optimize the use of co-processors

Aprovechamiento de las técnicas de compresión de índices para optimizar el uso de coprocesadores

Manuel Freire[1], Raul Marichal[1], Agustin Martinez[1], Daniel Padron[1], Ernesto Dufrechou[1], and Pablo Ezzatti[1]

[1] *Instituto de Computación (InCo), Facultad de Ingeniería, Universidad de la República, Uruguay*
{mfreire,rmarichal,amartinez,daniel.padron,edufrechou,pezzatti}@fing.edu.uy

Abstract

The significant presence that many-core devices like GPUs have these days, and their enormous computational power, motivates the study of sparse matrix operations in this hardware. The essential sparse kernels in scientific computing, such as the sparse matrix-vector multiplication (SpMV), usually have many different high-performance GPU implementations. Sparse matrix problems typically imply memory-bound operations, and this characteristic is particularly limiting in massively parallel processors. This work revisits the main ideas about reducing the volume of data required by sparse storage formats and advances in understanding some compression techniques. In particular, we study the use of index compression combined with sparse matrix reordering techniques in CSR and explore other approaches using a blocked format. The systematic experimental evaluation on a large set of real-world matrices confirms that this approach achieves meaningful data storage reductions. Additionally, we find promising results of the impact of the storage reduction on the execution time when using accelerators to perform the mathematical kernels.

Keywords: blocked formats, matrix storage reduction, memory access, reordering techniques, sparse matrices

Resumen

La importante presencia que tienen hoy en día los dispositivos multinúcleos como las GPU, y su enorme poder computacional, motivan el estudio de las operaciones matriciales dispersas en dicho hardware. Las rutinas matemáticas esenciales para la computación científica en álgebra dispersa, como la multiplicación matriz dispera vector (SpMV), suelen tener muchas implementaciones de alto rendimiento diferentes en GPUs. Los problemas de matrices dispersas normalmente implican operaciones acotadas por la memoria, y esta característica es particularmente limitante en procesadores masivamente paralelos. Este trabajo revisa las ideas principales sobre cómo reducir el volumen de datos requerido por los formatos de almacenamiento dispersos y avanza en la comprensión de algunas técnicas de compresión. En particular, estudiamos el uso de compresión de índices combinada con técnicas de reordenamiento de matrices dispersas en CSR y exploramos otros enfoques utilizando un formato a bloques. La evaluación experimental sistemática en un gran conjunto de matrices del mundo real confirma que este enfoque logra reducciones significativas en el almacenamiento de datos. Además, encontramos resultados prometedores del impacto de la reducción del almacenamiento en el tiempo de ejecución cuando se utilizan aceleradores para realizar las operaciones matemáticas.

Palabras claves: acceso a memoria, formatos a bloques, reduccion de almacenamiento de matrices, matrices dispersas,técnicas de reordenamiento

1 Introduction

The sparse algebra field has constantly evolved since the fifties when the pioneer works by R. Willoughby et al. and other authors [1] made the first steps in sparse matrix research. This early and rapid development is already reflected in I. Duff's review of the state of the art of sparse matrices in 1977 [2].

Nowadays, the sparse matrices are a key building block in many scientific computing problems in diverse fields such as circuit simulation [3] or analysis of social networks graphs [4]. The most relevant operation on sparse algebra is the product of sparse matrices with dense vectors (SPMV) since it is the main kernel of iterative methods to solve sparse linear systems of equations.

The importance of the routine motivated a great number of works dedicated to improve its performance on the different hardware platforms. In modern times, this work has been centered around platforms that integrate many lightweight cores and provide high

memory bandwidth like GPUs, FPGAs or even some multi-core CPUs. The GPUs, in particular, have dominated the HPC field in numerical linear algebra in the recent past, since the introduction of CUDA, because of its impressive peak performance in floating-point operations (FPOs). However, this peak performance is archived only when a high memory throughput can be maintained since the memory latency is the main bottleneck in HPC platforms.

Apart from the problems generated by the memory-bounded nature of the sparse matrix problems, they present other performance restrictions like load imbalance, (the number of nonzero entries to be processed by each computing unit varies significantly), indirect access to memory, (indexing data structures to access the floating-point values), and low data locality. However, despite only archiving a fraction of the peak performance in this devices, it is still possible to exploit this devices superior memory bandwidth. In this line there is a line of work in developing techniques that allow to transfer less data [5, 6] or to adapt the routines to the specific matrices [7, 8].

This work is a revised and extended version of [5], the new contributions are as follows:

- We further expand the analysis of the compression techniques previously proposed by implementing an optimized version of CSR which uses the most promising idea. This format divides the matrix in three submatrices in order to improve the storage.
- We compare both, the original CSR and our optimized version against a well known blocked format. In this line, we evaluate the compression of this format in a large set of matrices in order to explore different non-zero patterns. We also explore the usage of the RCM reordering heuristic on this format.
- We present a detailed theoretical study of the index overhead of CSR, optimized-CSR and *bmSparse* formats.
- Finally, we evaluate the impact of the reductions of storage obtained in the execution time by performing a small proof of concept.

The rest of the article is structured as follows. In Section 2 we summarize some basic concepts about sparse matrices, in particular, concepts related to sparse matrix storage and reordering techniques. Next, in Section 3, we revisit several works that are strongly related to our objectives. Later, in Section 4 we present a systematic evaluation of two highlighted strategies for the reduction of index storage. In Section 5 we perform both a theoretical and empirical analysis of a blocked format, *bmSparse*. Section 6 present the execution times obtained by performing a small test. Finally, at the end of our article, in Section 7, we offer some concluding remarks and identify promising future lines of work.

2 Basic concepts

In this section we include a brief introduction to sparse matrix storage and the most important concepts about the techniques of sparse matrix reordering.

2.1 Sparse matrix storage formats

Sparse storage formats are strategies used to store sparse matrices avoiding the cost of explicitly storing all zeros. In general, they store the non-zero values of the matrices plus data structures that allow to recreate the column and row indices of the values since they are not implicit anymore. For example, the coordinate format (COO) stores the nonzero values of the matrix and their coordinates as three arrays. If the COO elements are ordered row-wise, then it can be said that the *row_ind* (i.e. the row indices arrray) array stores redundant information since it would be enough to store the beginning of each row. A similar alternative to COO that addresses this problem is Compressed Sparse Row (CSR), [9]. It shares the array of values and columns of COO, but compresses the array of row coordinates, storing only the index where each row starts in the other two arrays. The main advantages of CSR are its compactness, and that it allows accesssing all the elements of a row directly. On the other hand, it requires additional operations to access each value.

There are a lot of formats that use a similar idea as CSR like, for example, Compressed Sparse Column (CSC) which is analogous to CSR but compresses the column array. Another example is Compressed Diagonal Storage (CDS) which is specially designed for matrices with band structure. When the matrices have a blocked structure (i.e. they have the non-zeros clustered) a good strategy is the Block Compressed Row Format, this strategy divides the matrix in equal size dense blocks and stores them analogous as CSR but considering block rows instead of scalars. This format requires the elements of the same block to be contiguous in the array of values and uses *padding* in that array to fill the blocks with zeros since it assumes they are the same size. In this line, there are formats that use variable-size blocks like 1D-VBL (*Varibable Block Length*) [10], that uses one-dimensional blocks, and VBR (*Variable Block Row*) [11], that uses two-dimensional blocks.

Due to the importance of SpGEMM on data science and graph problems, this operation has gained importance in recent years. SpGEMM performance depends on two matrices and the intersection of both sparsity patterns and cannot be known in advance thus making irregularity the main problem. BmSparse, the format proposed by Zhang et al. [12], and later optimized by

Berger et al. [13, 14], attacks the irregularity with a blocked approach of fixed size combined with the use of bitmaps inside each block.

2.2 Reordering techniques

The Cuthill-McKee algorithm (CM) [15] is known reordering heuristic used to cluster the non-zeros into a band. This technique applies a Breadth-First-Search (BFS) traversal strategy in the graph associated with the matrix and number the node. The starting point in the heuristic is defined as level 0 and, while there are unvisited nodes, it adds all neighbours of a level that have not been visited into the list of the next level. The order in which the nodes in each level are numbered defines the permutation obtained, in CM the nodes in the same level are sorted from low to high degrees. A variant of the CM algorithm is called the Reverse-Cuthill-McKee (RCM) and is the commonly used algorithm to find a permutation that reduces the bandwidth. This strategy reverses the ordering found in CM which results in the same bandwidth but yields a lower profile. In the original version of the RCM proposed by A. George, those nodes in the graph with minimum degree are also selected as initial vertices. The bandwidth and profile reductions of the resulting matrix, obtained by the CM and RCM heuristics depend heavily on the choice of the initial vertex. For this reason, several studies have been carried out on how to choose the initial vertex.

3 Related work

In this section, we briefly present the most remarkable ideas addressed in different investigations to achieve more efficient sparse matrix formats.

Perhaps one of the simplest strategies to improve the memory access pattern in sparse methods is to save the diagonal of the matrix separately. This strategy is advantageous when applying preconditioners to the diagonal in iterative methods of solving linear systems [16]. Sun et al. proposed the Compressed Row Segment with Diagonal-pattern (CRSD) format [17] which uses that idea. The format is useful mainly on matrices that have diagonal patterns on groups or segments of adjacent rows. It stores the components of each diagonal in a segment, in vectors whose index corresponds to the offset with respect to the main diagonal.

Another simple idea proposed by Bell and Garland is the HYB format [18] which mixes ELL and COO This format seeks to mitigate specific weaknesses of the ELL scheme, which, while offering advantages in terms of data locality, it is not very efficient in cases where the number of nonzero elements in each row varies considerably. The idea is to divide the matrix into two parts, each one with one strategy. The part stored in ELLEPACK, A_{ELL} is an array of size $n \times k$ where the number of elements per row is close to k and A_{COO} for the rest of the elements. The election of the number k determines the effectiveness of the format, this can be decided, as in *CUSP* [19] library, with an heuristic. Other strategy that combines ELL and Vectored CSR is EVC-HYB [20], this format is mainly focused on improving the performance of the SpMV in GPUs. This format first sorts the rows from smallest to largest and later they partition the rows into two groups: long and short. Finally, the rows in the first category are stored in VCSR while the second ones are stored in ELL. This combines the strength of ELL in small and regular rows with the good results that VCSR gets in matrices with row of sufficient length and, if possible, multiples of 32 to exploit coalesced access on GPUs.

In recent years, various efforts in sparse ALN worked with reduced precisions or modifications of standard formats. Some examples are [21, 22], where the authors evaluate how the use of reduced precisions (such as half and single) to store some coefficients of the preconditioners obtained with the Jacobi method, improves the performance when using these preconditioners in iterative methods to solve systems of linear equations. These formats seek to reduce the overhead produced by data transfer. Similar ideas are applied in [23], but decoupling the floating-point format used for arithmetic operations from the format used to store data in memory.

A different but complementary approach is to reduce the memory weight of the indices by reducing their precision. In this line, Shiming Xu et al. [24] proposed an optimization of the SPMV based on the ELL format which reduces the number of bits needed to represent the indices. The authors used the distance to the diagonal instead of the actual column index. They also use reorderings to try to reduce the distance to the diagonal in order to require less bits, to that purpose they use RCM method.

Another idea is the CoAdELL format [25], , which extends the work [26] of the same authors. The work focused on the division by warps of the computations with matrices stored in ELL-based formats. This is archived by compressing the storage associated with column indices using an encoding based on the difference between the indices of two consecutive non-zero elements in the same rows, this strategy is called delta encoding.

This is achieved using a compression technique to reduce the storage associated with column indexes. The idea is to use an encoding based on the difference between the indices of two consecutive nonzero elements in the same row, a technique that most authors call *delta encoding*. As these differences or deltas will have lower values than the indices, they can be represented with fewer bits.

Tang et al. [27] proposed a family of efficient compression schemes, which they call *bit-representation*

optimized (BRO). The approach was to reduce the number of bits required to represent indices. For the design of the BRO storage schemes, the authors focused on particular aspects of the target architectures. For example, to be worth to use on the GPU the cost of decompression must be relatively light compared to the addition and multiplication performed by the SPMV thus using the majority of the GPU cycles into useful work. They proposed two index compression techniques called BRO-ELL and BRO-COO based on the formats ELL and COO and compressing the indices using delta encoding. For example, in BRO-ELL, once the column index vectors have been transformed into delta encoding, the authors suggest dividing each of these into segments (called slices) of height h. Subsequently, each slice is compressed by an independent thread according to the number of bits needed for each delta index. In addition to BRO-COO and BRO-ELL, the authors also present BRO-HYB, which is a hybrid format of the two previous proposals. This format is similar to HYB, storing the regular part in BRO-ELL and the irregular one in BRO-COO.

Willcock and Lumsdaine proposed *Delta-Coded Sparse Row* (DCSR), a compression scheme the uses delta encoding to reduce the memory cost of the indices. They store tge differences between the column positions of nonzero elements in a row, using the minimum number of bytes possible. For this, a set of six command codes is used to encode the index data. In another work, Monakov et al. [28] proposed a new format that they called sliced ELLPACK, to improve the performance of SpMV in GPUs. This format uses a simple heuristic for reordering to reduce the zero padding required by the previous format. It has the main parameter S, which is the size or number of rows in each *slice*, each one of its are stored in ELL. The general idea is that, since the rows are divided in slices of similar sizes, the zero fill-in is only to reach the longer row in the slice and not in the whole matrix limiting the imbalance. This format improves the performance of the SpMV.

The authors of [27] extended their previous work using compression methods. They proposed the use of an heuristic for reordering that they called *BRO-aware reordering* (BAR). This strategy groups together the columns with similar patterns in therm of the bits required to encode with the goal of reducing the total space and thus the number of memory transactions when operating the matrix. The authors formulate the obtention of the permutation P as a clustering problem.

Finally, the increasing importance of Machine Learning in many areas, plus the computational capacity these methods require, motivated multiple studies to optimize sparse operations that are important in this context. The main operations are SpMM (Sparse-dense Matrix Multiplication) and SDDMM (Sampled Dense Dense Matrix Multiplication) [29, 30, 31]. In the work of Hong et al. [29], they proposed and ordering strategy based on adaptive *tiling*, the heuristic is called Adaptive Sparse Tiling (ASpT). It consists of grouping elements of the matrix into blocks or tiles, usually 2D, with which certain operation is carried out, for example, multiplications and convolutions. This technique is widely used in high-performance implementations of dense matrix-matrix multiplications, both for CPU and GPU. This strategy is used to optimize the two operations, SpMM and SDDMM.

4 Systematic theoretical evaluation

In this section we study the effectiveness of different techniques for the sparse matrix compression focusing on the reduction of index storage. For the testing of this section we used a subgroup of the *SuiteSparse Matrix Collection* with more than 1400 matrices with different sizes and non-zero structures. The only common characteristic is that all the studied matrices have a symmetric non-zero pattern. The main goal is to select the best techniques in order to reduce the overhead generated by the index storage. The matrices in CSR, which we call direct index compression, are used as the baseline for the comparison of the other strategies. In this section we focus on two well known compression techniques called delta-to-diagonal and delta encoding. We center our study in the effect of applying a previous reordering to each one.

An important observation is that we are focused in the compression of the index which is the one not compressed in CSR. For that reason, we do not consider the compression of the row vector in this work.

4.1 Direct index compression

This first part studies the original indices (i.e., in CSR) in order to use them as a baseline to evaluate the rest of the strategies. We divide the indices in three categories depending on the minimum number of bits required to store the indices. The categories are three, the indices which require less than 8 bits for column indices, those requiring 9 to 16 bits, and those requiring 17 to 32 bits. In this approach there are not negative values for the indices. In the same way, we classify the matrices. To define the categories for the matrix we use the largest index (i.e. the index that requires more bits). Since we have only square matrices and they do not have empty rows or columns at the end, applying this strategy to a matrix is the same to take the matrix dimension which is equal to the largest column index vvalue. Therefore, we need to evaluate if the matrix dimension is less or equal to $2^8 - 1$, $2^{16} - 1$ or $2^{32} - 1$.

The top image in Figure 1 shows in a bar chart the result of the direct-index classification. From the 1407 matrices studied we get that 116 require 8 bits, 931 require 16 bits, and 360 require 32 bits. This means that only a bit more than 25% of the total matrices require 32 bits to store their indices.

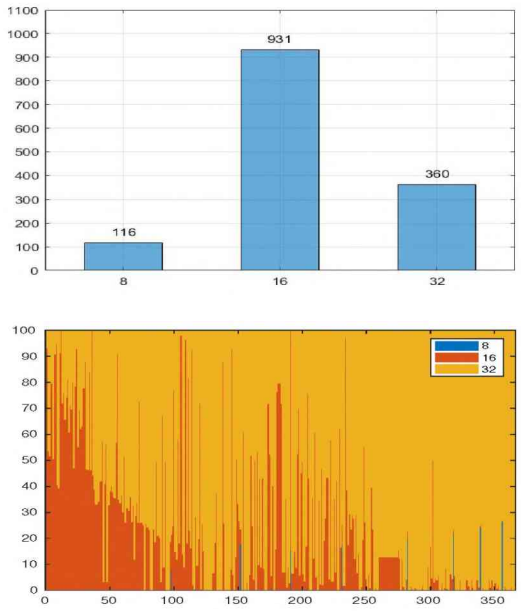

Figure 1: Index size distribution of the studied matrices for direct-indexing.

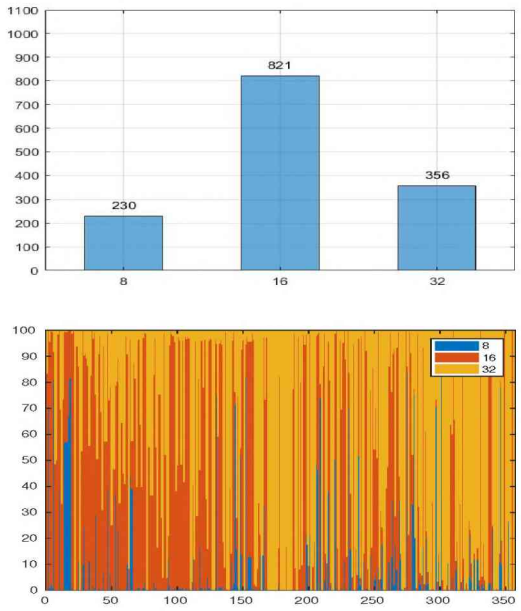

Figure 2: Index size distribution of the studied matrices for delta-to-diagonal compression.

In this and the following studies, we expand the evaluation over the matrices that require 32 bits. The specific idea is to evaluate how many rows of each of that matrices fall in the three categories. The bottom image of Figure 1 presents these results. It shows the percentage of the total rows that imply 8, 16, and 32 bits for each matrix. It is clear that most rows are in the two top categories, the 16 or 32-bit classes. In this type of figures each value of the x-axis represents an entire matrix and the proportion of colors shows the percentage of each class of rows. It is easy to see that there are only a few matrices with a large number of rows storable with 8-bit indices. It is important since in this matrices we could use a hybrid strategy that uses different integer sizes according to the rows classification.

4.2 Delta-to-diagonal encoding

This strategy uses the distance of the column to the diagonal instead of the column index itself and stores this value in the CSR representation. For that reason, the category of the row will be given by the distance of the first (or last) non-zero to the diagonal. This value is equivalent to the bandwidth of each row, $\beta(A_i)$. We present the same results as before, in the top part of Figure 2 we show the number of matrices in each family (8, 16 or 32 bits). It is important to see that, unlike previously, where we only needed to work with positive indices, we require signed integers in this technique. For this work, we assume that the ranges are symmetric relatively to zero (i.e. the column index of the diagonal values). This means that we get the next ranges $[-2^7, 2^7]$, $[-2^{15}, 2^{15}]$ y $[-2^{31}, 2^{31}]$.

As in the previous case, we expand the study taking the matrices of the 32-bit class focusing on the percentage of rows that could be stored in fewer bits. The bottom side of the Figure 2 shows that the number of rows that require fewer bits increased when we compare with the previous representation. Additionally, the number of matrices that can be represented with 8-bits is grater than the previous approach.

4.3 Delta-to-diagonal encoding with reordering

This variant uses a reordering before performing the diagonal encoding. We follow the proposal of Xu et al. [24], using the RCM heuristic on each sparse matrix and, after that, perform the replacement of each index by the difference with the diagonal (i.e. the encoding presented in Section 4.2) in the reordered matrix.

The number of matrices that can be represented with 8, 16 and 32 bits, with and without reordering, can be compared by observing Figures 2 and 3. The charts show that the use of the RCM heuristic offers substantial benefits by significantly reducing the number of matrices that require 32 bits to store their column indices from 353 to 63. In other words, there are 290 matrices (i.e. 82%) that required 32-bit indices in the direct compression but with this strategies do not need anymore. Many of this matrices move to the 16-bit class as illustrated by the increment of the number of sparse matrices in this class paired with the decrement mentioned previously. In fact, 154 matrices initially classified in the 16-bit class move to the 8-bit class when the RCM is applied. The detailed explanation of how many matrices moved from one class to another is

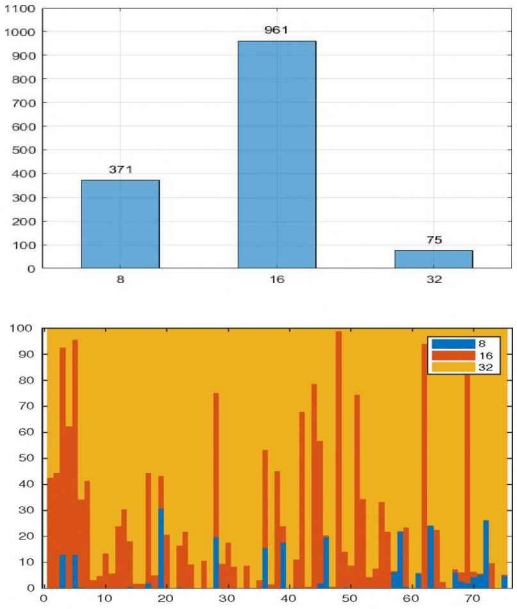

Figure 3: Index size distribution of the studied matrices for delta-to-diagonal compression after applying RCM reordering.

presented in the Figure 4. We name this a *composition matrix* and it is useful to analyze the benefit of the reordering technique. Each row represents the number of matrices that originally could be represented using the number of bits that are specified in the row. Also, each row is subdivided into the three categories according to the bits required after the RCM application. On the other hand, each column represents the number of original matrices that after the reordering require this value of bits. The composition matrix shows that, although the numbers of the upper triangle are usually low in all but one we have nonzero values. This means that in a few matrices, the effect of applying the reordering is negative making that the bits needed for column indices increases. Specifically, 15 sparse matrices move from 8 to 16 bits, and 2 matrices move from 16 to 32 bits. While these numbers are significantly smaller than the number of matrices that benefit from the reordering this shows that there are problems with the approach and, in some cases, it could handicap the compression.

Figure 5 shows an example of the problem mentioned above, the matrix `FIDAP/ex25` moves from the 8-bit to the 16-bit category after the application of the RCM heuristic. In the original scheme, its 848 rows can be represented with 8 bits but, after the reordering, only 475 rows can be represented with 8 bits and 376 rows require 16 bits. The original matrix has a block-diagonal (also called periodic nonzero pattern) which is lost after RCM is applied.

At this point, it is not possible to define a general rule to predict if there will be (or how much) storage savings after using the RCM reordering heuristic com-

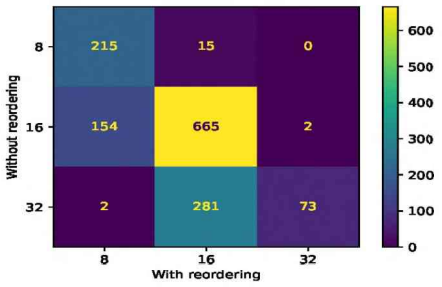

Figure 4: The Composition Matrix shows the number of matrices that move form one category to the other after reordering with RCM, applying delta-to-diagonal encoding.

bined with delta-to-diagonal encoding. However, it is important to consider the nonzero structure of the matrix before the transformation. In some cases, using a reordering technique breaks a convenient pattern, even increasing the number of bits required to store the indices.

As in the case without reordering, we present an expanded study focusing on the matrices of the 32-bit category, this study focus on the evaluation of the percentages of rows that can be stored with less bits. The results can be observed in the bottom part of Figure 3. The figure shows that, in general, after applying the RCM heuristic, the matrices in the 32-bit category have only a few rows storable with a smaller integer size.

4.4 Delta encoding

This strategy modifies the value with which we substitute the column index and uses the difference between two consecutive indices which is called delta encoding. This idea is similar to those presented in works such as Maggioni et al. [25], where the authors proposed the CoAdELL format, Kourtis et al. [32] for the CSR-DU format, and other efforts [27, 33]. Unlike the previous strategy, which considered both positive and negative distances, the distances in this encoding are all positive values. This gives us an extra bit to store the indices and allows the representation of larger differences.

As in the previous parts, the upper image of Figure 6 shows the categorization of sparse matrices according to the number of bits required to store their largest column index. The results of delta encoding can be compared with delta-to-diagonal by focusing on Figure 2). The benefits achieved by the former are clear. Specifically, the number of matrices in the 32-bit category is reduced, and, in the same line, the number of matrices in lower categories (specially in 8 bits) grows impressively. Similar to the previous experiments, we focus on the matrices in the 32-bit class aiming to understand the percentage of rows of each matrix that could be stored using 8 or 16 bits. The results of this

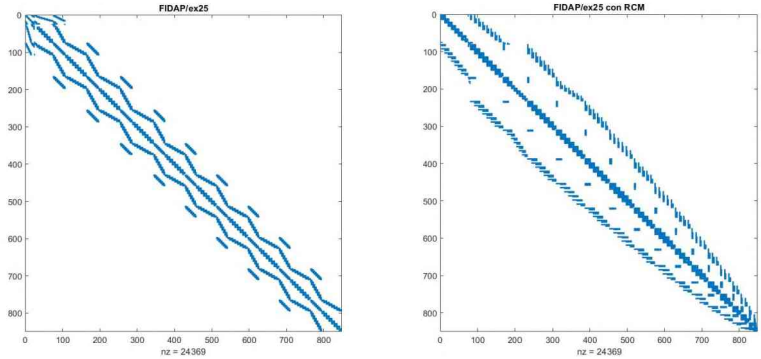

Figure 5: An example of a matrix that moves from the 8-bit to the 16-bit category after applying RCM using delta-to-diagonal encoding.

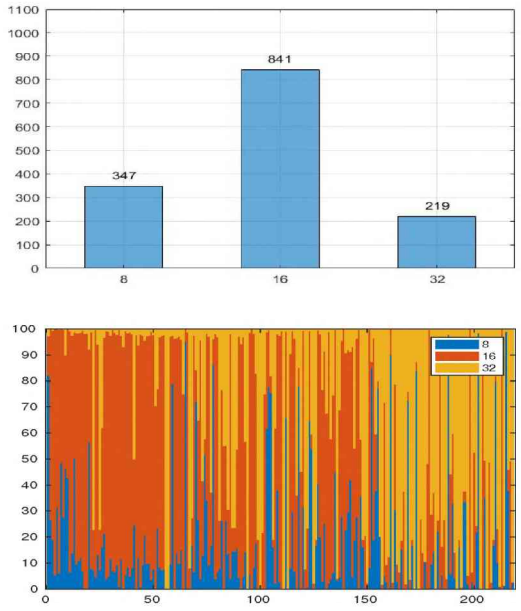

Figure 6: Index size distribution of the studied matrices for delta encoding.

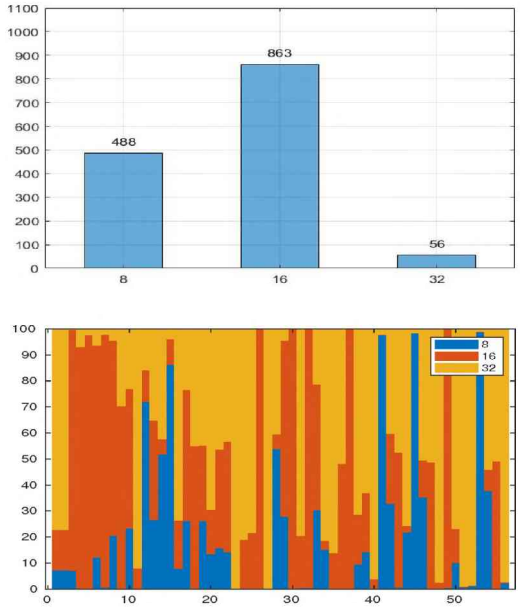

Figure 7: Index size distribution of the studied matrices for delta encoding after applying RCM.

study are summarized in lower part of Figure 6. As in delta-to-diagonal encoding without reordering, the results do not allow considering a hybrid strategy to store the column indices for most matrices.

4.5 Delta encoding with reordering

Considering the benefits obtained by performing the RCM reordering heuristic in delta-to-diagonal encoding studied previously it is natural to try the same approach with the delta encoding strategy. This idea is similar to the ones explored in [27]. Again, figure 7 summarizes the experimental results obtained by this strategy. The figure presents a similar prospect that the one obtained with delta-to-diagonal. The usage of the RCM heuristic significantly improves the compression and archives even better results than in delta-to-diagonal Comparing this strategy with and and without using the reordering, we can see that in the former, the number of matrices that require 32 bits is reduced in the order of 75%.

By observing the composition matrix presented in Figure 8 it is possible to observe some matrices that increment the number of bits needed for its representation after the RCM. The numbers in this case are a bit higher than the previous approach. In this, the upper triangle corresponds to 22 matrices that move from the 8 to the 16-bit class and 4 that move from the 16 to the 32-bit class.

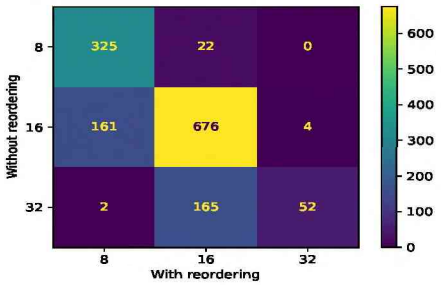

Figure 8: Composition Matrix for the delta encoding strategy, showing the number of matrices that form each category with and without reordering

Finally, same as previous approaches we end our analysis by focusing the study on the 32-bit category, which is the most important from the compression perspective. The bottom side of figure 7 presents the percentages of rows that require each number of bits for the 56 matrices in this category. The results are no different than those obtained in previous experiments.

4.6 Summary of the experimental evaluation

Variant	8	16	32
4.3.1. CSR column index	116	931	360
4.3.2. Delta-to-diagonal	230	821	356
4.3.3. Delta-to-diagonal with RCM	371	961	75
4.3.4. Delta encoding	347	841	219
4.3.5. Delta encoding with RCM	488	863	56

Table 1: Summary of the classification results for the evaluated strategies

The Table1 summarizes the results and lists the number of matrices in each category for all the approaches. This table presents some noteworthy observations. First, all techniques reduce the number of matrices that require 32 bits. Although in some techniques, like delta-to-diagonal encoding the gap is small, in some others like delta encoding with RCM the improvement is substantial. While RCM does not have the specific objective of improving the strategies like for example [34], it notably improves the matrix classification. This affirmation is clear when we focus on the number of matrices in the 32-bit category. These results motivate the use of methods for compression that use different integer sizes mentioned in the next sections.

5 Block sparse storage format

In the previous section we explored opportunities of index memory reductions over compress sparse storage (i.e. CSR or CSC), we complement it with a block sparse storage paradigm.

5.1 *bmSparse*: both a benchmark and a case of study

The *bmSparse* format [12] is a blocked format that groups the values into 8×8 blocks, which are represented by a key-bitmap pair (each one a 64 bits integer), storing only the blocks with at least one non-zero element. The key is a 64-bits integer that is used to store the row and column index of the block. On the other hand in the bitmap each bit represents an element of the block represented in row-major order in which if the bit is 0 the element is zero and it is non-zero otherwise.

As said previously, *bmSparse* uses the coordinates to refer to blocks. This means that A_{ij} references to the block ij, this is the block between $(8 \times i, 8 \times j)$ and $(8 \times (i+1), 8 \times (j+1))$. To store a matrix the *bmSparse* format uses four vectors: *keys*, *bmps*, *values* and $offsets$. The first two were explained previously while the third and fourth work similarly to values and *row_ptr* vectors of CSR but for blocks instead of rows.

Since a storage format is, in a way, a compression scheme, *bmSparse* can be used as a benchmark for comparison. As it is a block oriented, if our format is any good it has to beat *bmSparse* at least in some of the matrices with a non-blocked structure.

To approach this evaluation we first present a theoretical study and later perform an evaluation in a sizeable set of matrices.

5.2 Theoretical analysis of *bmSparse*

The aim of this section is to analyze the storage cost of three formats, CSR, *bmSparse* and our proposal, optimized CSR. To this end it is important to identify the variables that affect the final storage size. In CSR there are only two variables that affect the storage, they are the number of non-zeros (nnz) and the number of rows (n). On the other hand, the optimized CSR depends of the same variables of CSR plus three new variables: the number of rows in which the column indices can be stored in 8, 16 and 32 bits. Finally, for *bmSparse* there is other metric in place, the number of blocks. While this variable depends only of the non-zero pattern, it is strongly related with the number of non-zeros and more slightly with the dimension. This correlations over the matrices of the SuiteSparse collection are presented in Figures 9 and 10.

It is clear that a theoretical calculation based on the characteristics of the matrix is only possible in CSR. This format uses a pointer to the start of each row (size N) plus two vectors to store the values and the column indices (size NNZ). The values can be stored in different data types (half, simple, double) and thus the cost is not fixed. Since this work is focused on index compression and all the formats use the same structure to store values we focus the analysis in the index storage. The above results in the equation (1).

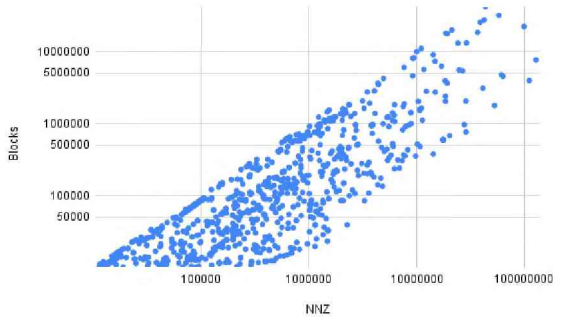

Figure 9: Number of blocks against NNZ of evaluated matrices.

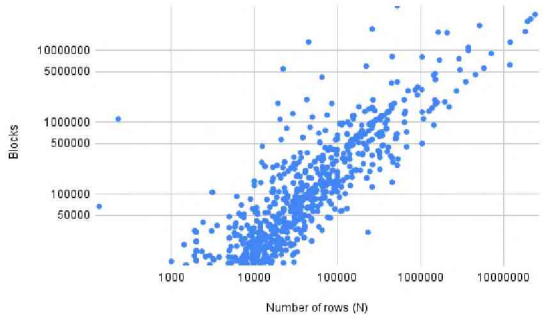

Figure 10: Number of blocks against number of rows of evaluated matrices.

$$Size(M) = nnz(M) \times 4 + (rows(M) + 1) \times 4 \quad (1)$$

For the optimized CSR we can also find an expression but it is dependent on the non-zero pattern since it affects how we store the column indices. The variation with CSR is that we can store the column indices in less bits thus the factor related with this metric is the only one affected. This is presented in equation (2)

$$\begin{aligned} Size(M) =& \#nnz_{32} \times 4 + \#nnz_{16} \times 2 + \#nnz_8 \\ &+ (rows(M) + 1) \times 4 + 4 \end{aligned} \quad (2)$$

where the final 4 comes from the vector which points to the start of each section of the matrix and nnz_i is the number of non-zeros which their column index is stored in i bits. It is easy to see that the storage related to the column will be always less or equal than in CSR since $\sum_{i=8,16,32}(\#nnz_i)$

Finally, *bmSparse* has a simple equation. For each block there are two int64, one for the indices and one for the bitmap, plus the values storage. Equation (3) shows the storage required.

$$Size(M) = nnz(M)/avg(nnzpb) \times 16 \quad (3)$$

where $avg(nnzpb)$ is the average of non-zeros on each block. From this equation it follows that for *bmSparse* to be better than CSR we need that the following equation is fulfilled.

$$avg(nnzpb) > \frac{16 \times nnz(M)}{4 \times (rows(M) + 1 + nnz(M))} \quad (4)$$

which is true with $avg(nnz\_per\_block)$ between 3 and 4 depending of the relation of $nnz(M)$ and $rows(M)$.

5.3 Empirical results

Taking into account the analysis presented above we performed an experimental evaluation of the storage required for each of the three formats. As presented in the title of the section we use *bmSparse* bot as a benchmark for comparison and as a case of study since the ideas can be applied to the format with little modifications To this end we use the open implementation of *bmSparse* provided in [14].

First, we evaluated the storage required by the three formats, CSR, optimized-CSR and *bmSparse* in a set of above 500 instances. In all the formats we only evaluated the index-related cost (i.e. the size presented in the previous section). In Figure 11 we present the ratio of reduction obtained with both optimized-csr and *bmSparse*.

Figure 11: Factor of reduction when comparing the storage against CSR (i.e. $size_{csr}/size_{optimized}$ and $size_{csr}/size_{bmsparse}$)

The first obvious conclusion is that our format get a way better compression both that *bmSparse* and the original CSR. In general, our new optimized-CSR reduces the storage in around 20% and up to 3.99×. On the other hand, at first impression, *bmSparse* appears to be a bad compression scheme. However, comparing *bmSparse* with a CSR-based format is not a completely fair comparison since this implementation of the format uses a similar approach to COO in the keys. For example, a further (and relatively easy) optimization of *bmSparse* could be to move the keys to a 32-bits integer and store the row index in a similar way to CSR.

Since *bmSparse* benefits from the clustering of the non-zeros it is reasonable to think that RCM will have

a good impact in the same way it has in the delta-encoding compression. To test this hypothesis we performed the storage evaluation of *bmSparse* after applying RCM. The comparison with previous results is presented in Figure 12

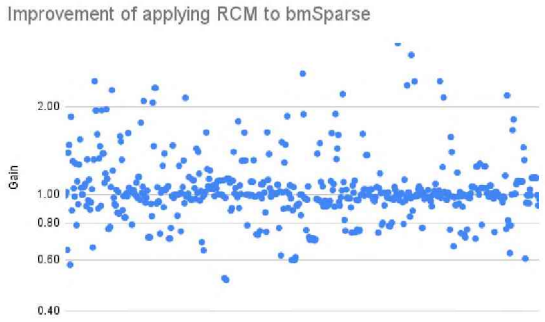

Figure 12: Factor of reduction when comparing the storage of *bmSparse* after applying RCM.

The results presented in Figure 12 show that *bmSparse* can improve its compression by applying RCM. Moreover, since the storage cost of this format depends heavily of the distribution of the non-zeros a hybrid approach to store the denser parts of the matrix in *bmSparse* and the scattered non-zeros in other format like CSR or COO. This strategy would be ideal for operations like SpMV in which both parts of the matrix could be processed separately and the results summed afterwards. This is an alternative approach to the one used in [35] and we leave it as future work.

6 Experimental evaluation

In this section, we present the results of execution time of the proposed storage strategy. Since the delta encoding had the best results on Section 4 we used this approach to the new proposed format.

To study the impact of the new proposed storage format on the execution time we implemented the format. The basic idea of this format is to reorder the rows depending of the bits needed to store the column in delta encoding so rows in the same category (8, 16 and 32 bits) are grouped together, then we store in a vector which row each category starts. To test the impact we evaluated the execution time required to make the transfer from CPU to GPU. While this is not a direct test of the kernels it works as a small proof of concept since, to execute a routine, it is needed to move the matrix from the CPU were is loaded to the GPU where the mathematical kernel is executed.

We selected a group of different matrices with distinct characteristics to test our implementation against CSR. These matrices are presented in Table 2. Figure 13 presents the speedups of execution time obtained by these same matrices.

The experimental evaluation was performed in a server that has an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with 64GB of RAM, 64kB of L1 cache, 256kB of L2 cache, and 8MB of L3 cache. The GPU is a NVIDIA RTX 3090 TI. The version of the CUDA Toolkit is 11.4. The experiments are performed using double-precision floating-point data.

Matrix	Dim	NNZ	NNZ avg
bcsstm25	15439	15439	1
t3dl_e	20360	20360	1.00
poli_large	15575	33074	2.12
poli3	16955	37849	2.23
fd15	11532	44206	3.83
bips98_1450	11305	44678	3.95
rajat06	10922	46983	4.30
circuit_3	12127	48137	3.97
cryg10000	10000	49699	4.97
std1_Jac2_db	21982	498771	22.69
std1_Jac3_db	21982	531826	24.19
Zd_Jac3_db	22835	713907	31.26
c-big	345241	2341011	6.78
Goodwin_095	100037	3226066	32.25
rajat31	4690002	20316253	4.33

Table 2: Set of matrices used in the experimental evaluation.

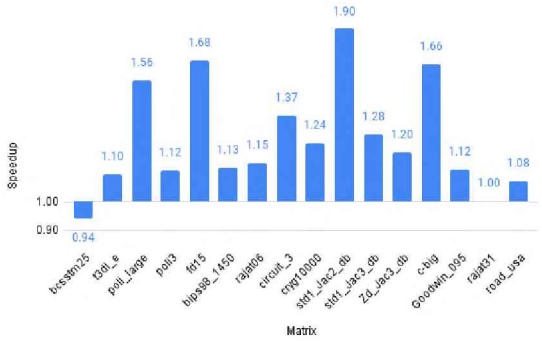

Figure 13: Speedup of the execution time in the memory transfers against CSR.

The results show a general improvement in the cost of the transference with only one example which is not statistically significant with only a loss of 6%. On the other hand, there are three matrices, fd15, std1_Jac2_db and c_big, with get speedups above 60% with factors of 1.68 and 1.9 and 1.66 respectively. This suggests that the storage reductions could have a direct impact in the execution cost of executing mathematical routines on GPU, specially on matrices which can move most of their rows to lower categories.

7 Final remarks and future work

We performed an extensive evaluation of different strategies to reduce the storage cost of sparse matrices. Our approach included more general strategies

focused in the CSR format but also explored more specific approaches like *bmSparse*.

Our work focused on systematic evaluation of the different approaches, specifically in the compression ratios. We found two promising strategies to reduce the overhead introduced by the explicit storage of the indices by reducing the storage required for the columns. This approaches included the use of the RCM heuristic to reorder the matrices plus the compression by storing distances instead of the indices. We also addressed other format, *bmSparse*, to study the impact of the compression strategies outside of CSR. Both studies were performed in large enough sets with hundreds of matrices from Suite Sparse Matrix Collection. Finally, we performed a small experimental evaluation focused on execution time which gave promising results.

Considering the experimental results we can make three important affirmations. First, the approaches to compress the indices by storing distances strongly reduce the storage required by most sparse matrices. Moreover, the previous application of reordering techniques is, in general, a good idea that improves the result. Secondly, the reduction of storage comes with a reduction of the execution time of the memory transfers thus impacting in mathematical kernels that use accelerators like, for example, GPUs. Finally, this strategies are not CSR-specific and can be used in conjunction with other formats and result in hybrid approaches with great reductions on storage. This compression strategies will have even more impact in contexts in which the data is stored in smaller formats such as half precision.

As part of future work, we have different exiting lines. An important line of work is to expand our implementation to develop GPU kernels that address the sparse matrix-vector multiplication. It would be also interesting to evaluate delta-to-diagonal implementations which we did not address in this work. The RCM method is an heuristic designed to improve the bandwidth of a matrix and not specifically the techniques. It is interesting to explore other heuristics directly focused on improving, for example, the delta encoding. An interesting line of work is to explore trajectory-based heuristics as the ones mentioned at the end of Section 4 but focused on this objectives instead of the concentration of non-zeros around the diagonal. Another interesting line is the creation of a hybrid format that uses this techniques to improve the performance of more specific formats like *bmSparse*. The general idea would be to use the format in the areas of the matrix in which it gets good performance and our optimized-CSR in the rest. In *bmSparse* for example this would be to store the dense blocks with that format and the other with the new proposal. In this approach the SpMV would be performed independently and the result of each submatrix summed at the end. Finally, we want to develop a public software library that uses our optimized-CSR and experimentally evaluate some mathematical kernels.

Competing interests

The authors have declared that no competing interests exist.

Funding

Manuel Freire received funding from the UDELAR CSIC-INI project *CompactDisp: Formatos dispersos eficientes para arquitecturas de hardware modernas.*

Authors' contribution

- Conceptualization (MF, RM, ED, PE)
- Software (AM, DP)
- Supervision (ED, PE)
- Redaction - original manuscript (MF, RM, ED, PE)
- Redaction - review and editing (MF, ED, PE)

All the authors have read and approved the final version

Acknowledgements

This work is partially funded by the UDELAR CSIC-INI project *CompactDisp: Formatos dispersos eficientes para arquitecturas de hardware modernas.* The authors also thank PEDECIBA Informática and the University of the Republic, Uruguay.

References

[1] F. G. Gustavson, W. Liniger, and R. Willoughby, "Symbolic generation of an optimal crout algorithm for sparse systems of linear equations," *J. ACM*, vol. 17, no. 1, pp. 87–109, 1970.

[2] I. Duff, "A survey of sparse matrix research," *Proceedings of the IEEE*, vol. 65, no. 4, pp. 500–535, 1977.

[3] T. Davis and E. P. Natarajan, "Algorithm 907: Klu, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, pp. 36:1–36:17, 2010.

[4] A. Chakraborty, T. Dutta, S. Mondal, and A. Nath, "Application of graph theory in social media," *International Journal of Computer Sciences and Engineering*, vol. 6, pp. 722–729, October 2018.

[5] M. Freire, R. Marichal, E. Dufrechou, and P. Ezzatti, "Towards reducing communications in sparse matrix kernels," in *Conference on Cloud Computing, Big Data & Emerging Topics*, pp. 17–30, Springer, 2023.

[6] M. Freire, R. Marichal, S. Gonzaga, E. Dufrechou, and P. Ezzatti, "Enhancing the sparse matrix storage using reordering techniques," in *Latin America High Performance Computing Conference (CARLA 23)*, pp. 66–76, 2023.

[7] E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "Selecting optimal SpMV realizations for GPUs via machine learning," *Int. J. High Perform. Comput. Appl.*, vol. 35, no. 3, 2021.

[8] E. Dufrechou, P. Ezzatti, M. Freire, and E. S. Quintana-Ortí, "Machine learning for optimal selection of sparse triangular system solvers on gpus," *J. Parallel Distributed Comput.*, vol. 158, pp. 47–55, 2021.

[9] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second ed., 2003.

[10] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, (New York, NY, USA), p. 30–es, Association for Computing Machinery, 1999.

[11] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations - version 2," 1994.

[12] J. Zhang and L. Gruenwald, "Regularizing irregularity: Bitmap-based and portable sparse matrix multiplication for graph data on gpus," in *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, (New York, NY, USA), pp. 1–8, Association for Computing Machinery, 2018.

[13] G. Berger, M. Freire, R. Marini, E. Dufrechou, and P. Ezzatti, "Unleashing the performance of bmsparse for the sparse matrix multiplication in GPUs," in *Proceedings of the2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pp. 19–26, November 2021.

[14] G. Berger, M. Freire, R. Marini, E. Dufrechou, and P. Ezzatti, "Advancing on an efficient sparse matrix multiplication kernel for modern gpus," *Concurrency and Computation: Practice and Experience*, p. e7271, 2022.

[15] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*, pp. 157–172, ACM Press, 1969.

[16] M. Bollhoefer and Y. Saad, "On the relations between ILUs and factored approximate inverses," *SIAM J. Matrix Anal. Appl.*, vol. 24, no. 1, pp. 219–237, 2002.

[17] X. Sun, Y. Zhang, T. Wang, X. Zhang, L. Yuan, and L. Rao, "Optimizing SpMV for diagonal sparse matrices on GPU," in *2011 International Conference on Parallel Processing*, pp. 492–501, IEEE, September 2011.

[18] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11, 2009.

[19] N. Bell and M. Garland, *Cusp library*, 2012.

[20] D. Guo, W. Gropp, and L. N. Olson, "A hybrid format for better performance of sparse matrix-vector multiplication on a GPU," *The International Journal of High Performance Computing Applications*, vol. 30, pp. 103–120, July 2015.

[21] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí, "Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers," *Concurrency and Computation: Practice and Experience*, vol. 31, p. e4460, March 2018.

[22] F. Goebel, H. Anzt, T. Cojean, G. Flegar, and E. S. Quintana-Ortí, "Multiprecision block-jacobi for iterative triangular solves," in *Euro-Par 2020: Parallel Processing*, pp. 546–560, Springer International Publishing, 2020.

[23] T. Grützmacher, T. Cojean, G. Flegar, F. Göbel, and H. Anzt, "A customized precision format based on mantissa segmentation for accelerating sparse linear algebra," *Concurrency and Computation: Practice and Experience*, vol. 32, July 2019.

[24] S. Xu, H. X. Lin, and W. Xue, "Sparse matrix-vector multiplication optimizations based on matrix bandwidth reduction using NVIDIA CUDA," in *2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, IEEE, August 2010.

[25] M. Maggioni and T. Berger-Wolf, "CoAdELL: Adaptivity and compression for improving sparse matrix-vector multiplication on GPUs," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IEEE, May 2014.

[26] M. Maggioni and T. Berger-Wolf, "AdELL: An adaptive warp-balancing ELL format for efficient sparse matrix-vector multiplication on GPUs," in *2013 42nd International Conference on Parallel Processing*, IEEE, October 2013.

[27] W. T. Tang, W. J. Tan, R. Ray, Y. W. Wong, W. Chen, S. hao Kuo, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "Accelerating sparse matrix-vector multiplication on GPUs using bit-representation-optimized schemes," in *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ACM, 2013.

[28] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *High Performance Embedded Architectures and Compilers*, pp. 111–125, Springer Berlin Heidelberg, 2010.

[29] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ACM, February 2019.

[30] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the gpu," in *Euro-Par 2018: Parallel Processing* (M. Aldinucci, L. Padovani, and M. Torquati, eds.), (Cham), pp. 672–687, Springer International Publishing, 2018.

[31] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse GPU kernels for deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, IEEE Press, 2020.

[32] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proc. of the 2008 conference on Computing frontiers*, pp. 87–96, ACM Press, 2008.

[33] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Proceedings of the 20th annual international conference on Supercomputing - ICS '06*, pp. 307–3016, ACM Press, 2006.

[34] M. Freire, R. Marichal, E. Dufrechou, P. Ezzatti, and M. Pedemonte, "Trajectory-based metaheuristics for improving sparse matrix storage," in *Proceedings of the 9th Latin American Conference on Computational Inteligence (LACCI 2023)*, (Recife, Brasil), pp. 66–76, November 2023.

[35] G. Berger, E. Dufrechou, and P. Ezzatti, "Sparse matrix-vector product for the bmsparse matrix format in gpus," in *Proceedings of the 21ST International workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2023)*, LNCS, (Limassol, Cyprus), Springer, August 2023.

Citation: M. Freire, R. Marichal, A. Martinez, D. Padron, E. Dufrechou and P. Ezzatti. *Leveraging index compression techniques to optimize the use of co-processors.* Journal of Computer Science & Technology, vol. 24, no. 1, pp. 1–13, 2024.
DOI: 10.24215/16666038.24.e01
Received: April 15, 2023 **Accepted:** January 29, 2024.