Towards reducing communications in sparse matrix kernels

Manuel Freire$^{1\,[0000-0002-4887-8918]}$, Raul Marichal$^{1\,[0000-0003-4974-0803]}$, Ernesto Dufrechou$^{1\,[0000-0003-4971-340X]}$, and Pablo Ezzatti$^{1\,[0000-0002-2368-8907]}$

Instituto de Computación, INCO Facultad de Ingeniería Universidad de la República
Montevideo, Uruguay {mfreire,rmarichal,edufrechou,pezzatt}@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. The systematic experimental evaluation on a large set of real-world matrices confirms that this approach is promising, achieving meaningful data storage reductions.

Keywords: sparse matrices, · memory access · reordering technique · matrix storage reduction

1 Introduction

The evolution of the sparse matrix research has evolved constantly since the '50s when the pioneer works by Gustavson and others [11] made the first steps in this field. Nowadays, sparse matrices are employed in different contexts in science and engineering, and the resolution of many scientific computing problems implies an intensive use of sparse matrix operations. Maybe the most relevant of these is the product of sparse matrices with dense vectors (SpMV) since it is the heart of iterative methods to solve sparse linear systems. This has motivated several works to improve its performance on different hardware platforms in the last 40 years. A related aspect that has a strong impact on sparse matrix research, and scientific computations, is the evolution of hardware platforms in the HPC field. The main design trend of modern computer architectures is to integrate many lightweight cores and provide high memory bandwidths. These are referred as throughput-oriented architectures. GPUs are key examples that, since the appearance of CUDA in 2007, dominated the HPC landscape of numerical linear algebra. Like other throughput-oriented hardware platforms, such as FPGAs or some multi-core CPUs, these devices offer impressive peak performance in floating-point operations (FPOs) as long as a high memory throughput can be maintained. However, the latency of memory accesses has not improved as much as the computing power for HPC platforms. This situation poses important challenges in the context of memory-bound problems, where few FPOs are performed for each memory access [7, 6]. Besides being inherently memory-bound, sparse matrix problems generally present other performance restrictions such as load imbalance, indirect access to memory, and low data locality. However, despite being capable of reaching only a mild fraction of the peak performance in these devices, it is still possible to achieve important gains by exploiting their superior memory bandwidth. Therefore, it is mandatory to apply techniques that allow transferring less data between the memory and the cores for each FPO, increasing the achieved computational intensity.

In this work, we review the state-of-the-art techniques devoted to improving sparse matrix storage, such as efforts to improve the data locality and reduce the volume of data movements between levels of the memory hierarchy. Also, we identify some promising techniques and study them in detail, performing an extensive experimental evaluation. **The main contribution of our work is an extensive evaluation performed on the SuiteSparse matrix collection**[1] **of the impact of index compression and reordering on the storage volume of sparse matrices.** The obtained results show that important storage reductions can be reached in the sparse matrices evaluated. This reductions are highly relevant because they can lead to improvements in the relation between communications and FPOs, **transferring less data from memory**.

The rest of the article is structured as follows. In Section 2 we summarize some basic concepts about sparse matrices. Next, in Section 3, we revisit several works that are strongly related to our objectives. A systematic experimental evaluation of some highlighted strategies for these purposes follows in Section 4. At the end of our article, in Section 5, we offer some concluding remarks and identify promising future lines of work.

2 Basic concepts

Sparse matrix storage formats They are strategies that avoid storing redundant matrix information by storing the nonzero values of a matrix together with a combination of data structures that allow determining the coordinates associated with each element. For example, the coordinate format (COO) stores the nonzero values and their coordinates as three arrays. Another standard format, Compressed Sparse Row (CSR), 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

[1] https://sparse.tamu.edu/

the other hand, it requires additional operations to access each value. Various formats similar to CSR have been proposed, such as the Compressed Sparse Column (CSC) format, which is analogous to CSR but compresses the column array. Another alternative is *Compressed Diagonal Storage* (CDS), which takes advantage of the band matrix structure to represent the matrix by eliminating the row and column arrays. *Block Compressed Row Format* (BSR) divides the original matrix into dense blocks of equal size and makes the array of columns represent block positions. The block compressed row array is analogous to the corresponding CSR array but considers rows of blocks 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. Other formats use variable-size blocks. Among them, two of the most prominent are 1D-VBL (*Varibable Block Length*) [18], which uses one-dimensional blocks, and VBR (*Variable Block Row*) [19], in which the blocks are two-dimensional.

There are multiple possible criteria to evaluate formats. In [14] the authors review the available literature on the subject and note that commonly used evaluation criteria depend on a variety of parameters, such as architecture dependencies, implementation quality, and representation used for the values.

The Cuthill-McKee algorithm [5] It is a reordering technique based on the application of a Breadth-First-Search (BFS) traversal strategy, in which the graph associated with the matrix is traversed by levels. Starting from a root node (level 0), the unvisited adjacent nodes of the nodes on each level are sorted and added to the list of nodes of the next level. Then the ones in the current level are numbered and the procedure repeats for the next level until all nodes are visited. The order in which each level is numbered gives rise to different orderings or permutations of rows and columns. In the Cuthill-McKee algorithm, the adjacent nodes to a visited node are always traversed from low to high degrees. A popular variant of the CM algorithm is the Reverse-CM, in which the order obtained for the rows/columns is "reversed". The RCM heuristic results in arrays with the same bandwidth as CM, but with 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 strongly depend 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

Perhaps one of the simplest strategies to improve the memory access pattern in sparse methods is to save the diagonal of the matrix separately. In this line, Sun et al. presented the Compressed Row Segment with Diagonal-pattern (CRSD) format [20]. The format is focused mainly on matrices with diagonal patterns on groups or segments of adjacent rows. It consists of storing the components of each diagonal in a segment, in vectors whose index corresponds to the offset with respect to the main diagonal. Another example can be the HYB format that combines the ELL and COO formats, proposed by Bell and Garland [2]. The idea is to divide the matrix into a A_{ELL} 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. Choosing the column k that determines the partition can be solved, as in the implementation of *CUSP* [3] library, with a heuristic. Other hybrid format, that combines ELL and Vectored CSR is EVC-HYB [10], focused on improving the performance of the SpMV in GPUs. First, the authors sort the rows based on their lengths, from smallest to largest, and then they partition the rows into two groups, long and short. Centered in this partition, the matrix entries are stored in ELL or VCSR as appropriate, seeking to exploit the strenghts of both formats. While ELL works well with arrays whose number of nonzero elements per row is low and regular, VCSR works better with matrices whose rows are of sufficient length and, if possible, multiples of 32.

In recent years, various efforts in sparse ALN worked with reduced precisions or modifications of standard formats. An example is [1], where the authors evaluate the use of reduced precisions (half and single) to store some coefficients of preconditioners. This format aims to reduce the overhead produced by data transfers. Similar ideas are applied in [9], but decoupling the floating-point format used for arithmetic operations from the format used to store data in memory. A complementary approach is to reduce the precision of the indices associated with the coefficients. In this sense, in the work of Shiming Xu et al. [23] an optimization of the SpMV is proposed, based on the ELL format, whose objective is to reduce the number of bits needed to represent the indices. The authors explore the possibility of using the distance to the diagonal instead of the actual value of the coordinate as the column index. They target a set of square matrices where they also seek to reduce the distance of the nonzero elements to the diagonal through rearrangements or permutations such as the RCM method. Other authors also tried other heuristics instead of RCM [16].Another idea is the CoAdELL format [15], focused on the division by warps of the computations with matrices stored in ELL-based formats. 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. [21] propose using a family of efficient compression schemes, which they call *bit-representation optimized* (BRO), to reduce the number of bits required to represent indices. For the design of the BRO storage schemes, the authors considered essential aspects related to the target architectures. Without going into implementation details, the authors study the impact of the steps necessary to apply these techniques. For example, to be able to perform decompression on the GPU, it must be relatively light compared to the addition and multiplication operations of the SpMV so that most of the GPU cycles are allocated to useful work and not used only to decompress the index data. In this sense, two index compression techniques are presented, BRO-ELL and BRO-COO, based on the formats ELL and COO, compressing the column 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, useful in cases when the number of nonzero elements per row varies substantially. This format is analogous to HYB, storing the regular component in BRO-ELL and the irregular component in BRO-COO.

Willcock and Lumsdaine proposed *Delta-Coded Sparse Row* (DCSR), a compression scheme based on the delta encoding of the indices is proposed, encoding the indices as the 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 investigation, Monakov et al. [17] propose a new format that they called sliced ELLPACK, to improve the performance of SpMV in GPUs. This format has the main parameter S, which is the size or number of rows in each *slice*. Each of these slices is stored in ELL format. While the storage overhead in the sliced ELLPACK format is limited to slices with an imbalance in the number of nonzero elements per row, this can still cause noticeable performance degradation. The authors then propose a simple reordering heuristic that can substantially improve the performance of the SpMV implementation for that format, based on grouping rows with the same number of nonzero elements.

To take advantage of compression methods, the same authors of [21] extended their first proposals with the use of an ordering strategy that they called *BRO-aware reordering* (BAR). This strategy consists of bringing together those columns with similar patterns in terms of the number of bits necessary for their encoding to reduce the total space and, consequently, possibly reduce the number of transactions when operating with the matrix. The authors formulate the obtention of the permutation P as a clustering problem. The increasing use of Machine Learning in multiple areas, and the computational capacity these methods need, motivated several studies that seek to optimize sparse operations that arise in these contexts, such as the SpMM (Sparse-dense Matrix Multiplication) and SDDMM (Sampled Dense Dense Matrix Multiplication) [12, 24, 8]. For example, in Hong et al. [12], an ordering strategy based on adaptive *tiling* is designed, called Adaptive Sparse Tiling (ASpT). The technique is applied to improve the performance of the two primitives (SpMM and SDDMM). *Tiling* is a fundamental technique for data locality optimization. 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.

Other operation that has gained popularity in recent years is the SpGEMM (Sparse Matrix-Sparse Matrix multiplication) since it is an important part on many graph problems as well as in the field of data science. Since SpGeMM involves two sparse matrices the irregularity is one of the main problems, in the format presented in Berger et al. the authors use a blocked format with bitmaps to address irregularity [4, **?**].

4 Systematic experimental evaluation

This section studies the effectiveness of techniques for sparse matrix compression, focusing on reducing index storage volume. The dataset used for our study is the group of matrices with symmetric non-zero patterns of the *SuiteSparse Matrix Collection*, which comprises 1407 matrices with different sizes and nonzero structures. Our purpose is to identify the most promising techniques to reduce index storage. In this context, we employ the sparse matrices represented with the CSR format as a baseline, which we call direct index compression. Then, we evaluate two different strategies: delta-to-diagonal encoding and delta encoding, assessing the effect of applying a previous reordering on each one.

4.1 Direct index compression

The first effort explores the original indices (i.e., in CSR), which will serve as the baseline to assess the rest of the strategies. Our evaluation classifies each index into three categories based on the minimum number of bits required to represent it. The categories comprise the indices requiring less than 8 bits, those requiring 9 to 16 bits, and those requiring 17 to 32 bits, respectively. In this case, all the indices are non-negative values. We also classify each matrix into the same three categories according to its largest index (or the largest index of each row or column). As we deal with square matrices and there are no empty rows or columns at the end, applying this procedure to the baseline matrix representation is equivalent to taking the matrix dimension, equal to the largest column index value. Therefore, we need to determine if the matrix dimension is less or equal to $2^8 - 1$, $2^{16} - 1$ or $2^{32} - 1$.

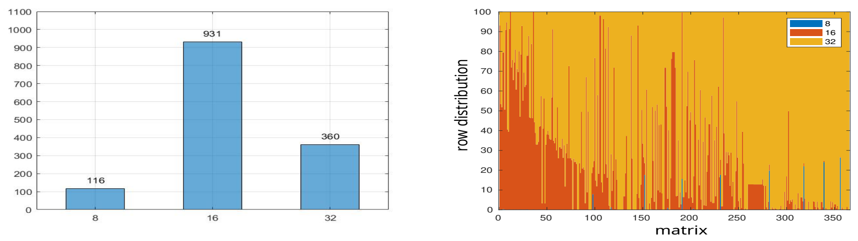

Fig. 1. Index size distribution of the studied matrices for direct-indexing.

The left image in Figure 1 presents a bar chart with the results of the direct-index classification. We can see that from the 1407 studied matrices, 116 require 8 bits, 931 require 16 bits, and 360 require 32 bits. In other words, approximately 25% of the total matrices require 32 bits to store their indices.

In this and the following studies, we expand the evaluation over the matrices that require 32 bits. Specifically, we analyze the number of rows falling in each of the three categories for each of these matrices. The right image of Figure 1 summarizes these results, showing the percentage of the total rows that imply 8, 16, and 32 bits for each matrix. It is easy to see that most rows lie in the 16 or 32-bit classes. In this figures each value in the x-axis represents one matrix. In other words, only a few matrices present a large number of rows storable with 8-bit indices, where a hybrid strategy that uses different integer sizes according to the row classification could be applied.

4.2 Delta-to-diagonal encoding

This encoding replaces the column index value in the CSR representation by the difference between the corresponding column and the diagonal. Therefore, the integer size required to represent each row is determined by the distance of the first nonzero value to the diagonal (a value equivalent to the bandwidth of each row, $\beta(A_i)$). As before, left part of Figure 2 presents the number of matrices in each family (8, 16 or 32 bits). Notice that unlike the previous analysis, where we need to manipulate only positive indices, we need to employ integer numbers in this technique. Expressly, we assume symmetric ranges relative to zero (the column index of the diagonal values in the new encoding). Thus, we obtain the next ranges $[-2^7, 2^7]$, $[-2^{15}, 2^{15}]$ y $[-2^{31}, 2^{31}]$.

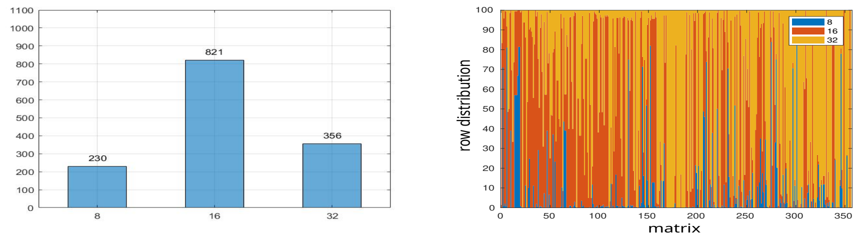

Fig. 2. Index size distribution of the studied matrices for delta-to-diagonal compression.

Expanding the analysis as in the previous case, we take the matrices of the 32-bit class and evaluate the percentage of rows that could be stored with fewer bits. The right-hand side of figure 2 shows a significant increase in the fraction of rows that require fewer bits compared to the previous representation. Furthermore, several large matrices with a high percentage of rows representable with 8-bits can be observed.

4.3 Delta-to-diagonal encoding with reordering

This variant applies a reordering to the sparse matrix previous to the diagonal encoded. Specifically, and following the proposal of Xu et al. [23], we use the RCM heuristic on each sparse matrix, and later we replace the column indices by the difference with the diagonal (the encoding presented in Section 4.2) in the reordered matrix.

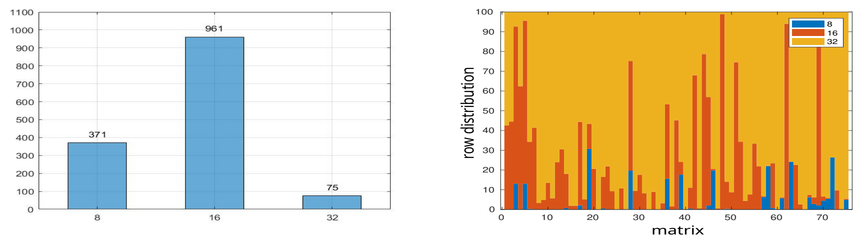

Fig. 3. Index size distribution of the studied matrices for delta-to-diagonal compression after applying RCM reordering.

The number of matrices that are representable using the delta-to-diagonal technique with 8, 16, and 32 bits, with and without the reordering, can be compared observing Figures 2 and 3. From these charts, we can deduce that using the RCM heuristic offers substantial benefits, significantly diminishing the number of matrices that require 32 bits to store their indices from 353 to 63. In other words, 290 matrices (82%) that initially needed 32-bit indices can benefit from applying these two techniques. On the other hand, the number of sparse matrices in the 16-bit class increases. It is also evident that the majority of the matrices that form the 16-bit class after the transformation originally belonged to the 32-bit class. In fact, 154 matrices initially classified in the 16-bit class move to the 8-bit class when the RCM is applied. More details are included in the Figure 4. This chart, which we name *composition matrix*, is useful to compare the benefit of applying a reordering technique. Each row and column is labeled as a matrix class (8, 16, or 32-bits). The value in row x and column y is the number of matrices of class x that move to class y after the reordering. In the composition matrix, it is easy to see that, although the numbers in the upper triangle are generally low, we have several nonzero values (only one is zero). This means that, in a few matrices, the reordering technique affects the number of bits needed for column index storage negatively. In particular, 15 sparse matrices move from 8 to 16 bits, and two matrices move from 16 to 32 bits. Although these numbers are significantly smaller than those of the matrices that benefit, this situation evidences that there are problems where applying these reordering techniques handicaps the compression.

Figure 5 shows the case of the matrix `FIDAP/ex25`, which moves from the 8-bit to the 16-bit category after the application of the RCM heuristic. In the original

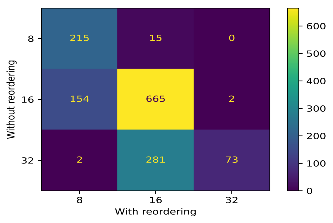

Fig. 4. The Composition Matrix shows the number of matrices that move form one category to the other after reordering with RCM, applying delta-to-diagonal encoding.

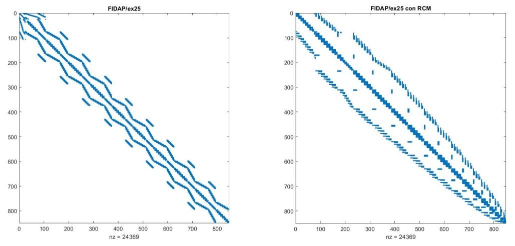

Fig. 5. An example of a matrix that moves from the 8-bit to the 16-bit category after applying RCM using delta-to-diagonal encoding.

form, its 848 rows can be represented with 8 bits while, after the reordering, only 475 rows can be represented with 8 bits and 376 rows requiere 16 bits. This matrix shows a block-diagonal or periodic nonzero pattern, which is lost when RCM is applied. Similar to the previous example, in Figure 6 we show the case of `Sinclair/3Dspectralwave2` matrix, where the application of the reordering technique does not improve the index representation. In this case, the matrix moves from the 16-bit to the 32-bit class. The causes of this situation are similar to the previous one.

At this point, it is difficult to establish a general rule that predicts the volume of storage saved using RCM reordering heuristic with delta-to-diagonal encoding. However, the nonzero structure of the matrix needs to be carefully considered before the transformation. In some cases, the reordering breaks a convenient pattern, even increasing the number of bits required to store the indices.

As in the case without reordering, we expand our study focusing on the matrices of the 32-bit category and we evaluate which percentage of rows can be stored with less bits. These results can be observed in the right part of Figure 3. In general, the figure shows that after the application of the RCM, the matrices in this category have relatively few rows storable with a smaller integer size. Additionally, it does not seem to be a strong correlation between the number of nonzero coefficients and the number of rows that require fewer bits.

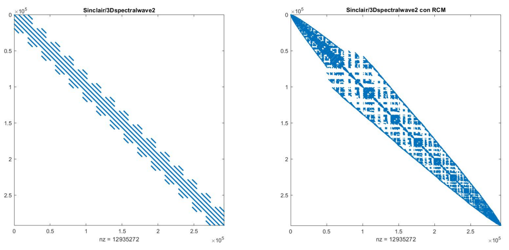

Fig. 6. Example of a matrix that moves from the 16-bit to the 32-bit category after applying RCM using delta-to-diagonal encoding.

4.4 Delta encoding

This study modifies the rule to substitute the column index value by the difference between two consecutive indices, i.e., delta encoding. This idea is similar to those presented in works as Maggioni et al. [15], where the authors propose the CoAdELL format, Kourtis et al. [13] for the CSR-DU format, and other efforts [21, 22]. Unlike the previous case, which considered positive and negative distances, in this encoding all values are positive. This provides an extra bit to store the indices and allows the representation of larger deltas.

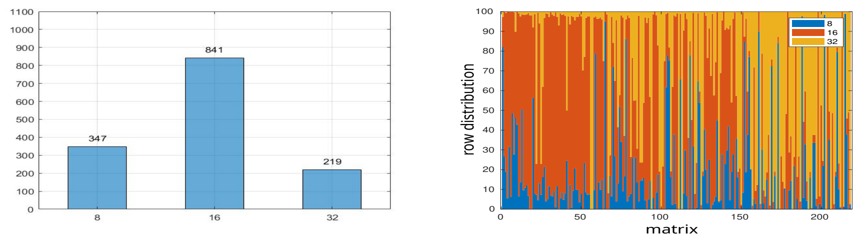

Fig. 7. Index size distribution of the studied matrices for delta encoding.

As before, the left part of Figure 7 shows the categorization of sparse matrices according to the number of bits required to store the largest column index. Comparing delta encoding with the delta-to-diagonal variant (results in Figure 2), the benefits achieved by the former are clear. Specifically, the number of matrices in the 32-bit category is heavily reduced, and, aligned with this, the number of matrices in the 8 bits category grows impressively. Similar to the previous experiments, we evaluate the matrices in the 32-bit class to understand how many rows of each matrix could be stored using 8 or 16 bits. The results of this study are summarized in rigth part of Figure 7. As in delta-to-diagonal encoding without reordering, the results do not allow considering a hybrid strategy to store the column indices.

4.5 Delta encoding with reordering

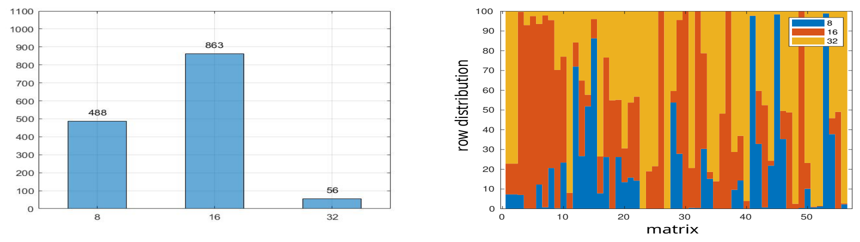

Fig. 8. Index size distribution of the studied matrices for delta encoding after applying RCM.

Considering the benefits reached using the RCM reordering heuristic in the delta-to-diagonal encoding studied previously, in this work, we propose the application of the ideas of Maggioni et al. to the delta encoding strategy. Similar ideas were explored in [21]. Figure 8 summarizes the experimental results reached in this case. The figure shows that the effect of reordering produces similar results using delta encoding and delta-to-diagonal. The application of the RCM heuristic drastically improves the compression, actually achieving more significant improvements in the case of delta encoding. Comparing the use of delta encoding with and without the previous reordering, we can see that with reordering, the number of matrices that require 32 bits is reduced in the order of 75%.

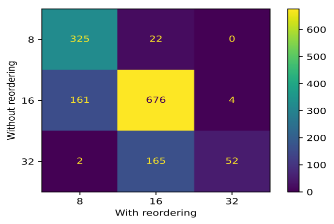

Fig. 9. Composition Matrix for the delta encoding strategy, showing the number of matrices that form each category with and without reordering

Observing the composition matrix that is presented in Figure 9 it is possible to notice some matrices that increment the number of bits needed for its representation after the RCM. In this case, the upper triangle corresponds to 22 matrices that move from the 8 to the 16-bit class and four that move from the 16 to the 32-bit class. These numbers are a bit higher than the previous matrix.

Following the same procedure as the previous approaches, we analyze the matrices in the 32-bit category, which is the most interesting from the compression perspective. The right side of figure 8 shows the percentage of rows that require 8, 16, and 32 bits for the 56 matrices in this category, sorted by their number of nonzeros. The results are similar to those reached 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

To summarize and analyze the obtained results, we include Table 1 that lists the classification of the total matrices evaluated with each approach. From this table, we can highlight several results. First, the results show that the reduction of the number of matrices that require 32 bits is considerable for all the evaluated techniques. Although the gap is of only four matrices in the delta-to-diagonal encoding, the benefits are substantial in other cases. As second observation, the addressed reordering technique (RCM) improves the matrix classification. This affirmation can be corroborated, by the reduction of the number of matrices in the 32-bit category. These results motivate the exploration of methods for matrix index compression and storage formats that consider different integer sizes.

5 Final remarks and future work

We evaluated the strategies proposed to reduce the storage volume of sparse matrix formats. Our effort includes a systematic experimental evaluation of some of the most promising ones. We thoroughly review the different techniques presented in previous efforts that align with our objective. Specifically, we identify two methods based on index compression techniques, i.e., the delta-to-diagonal encoding and delta encoding, and assess their use with sparse matrix reordering procedures such as the RCM heuristic. We perform our experimental evaluation on 1407 matrices from the Suite Sparse Matrix Collection. Our purpose is to avoid the potential bias in the analysis that can arise in small sets, caused by specific patterns. Considering the experimental results, we can affirm that these techniques strongly reduce the storage required by sparse matrices. In particular, the combination of delta encoding with reordering techniques allows storing the column indices of 96% of the evaluated matrices using at most 16-bit integers.

As part of future work, we plan to advance in four distinct directions. One interesting line of work is developing a computational method to address the sparse matrix-vector product using sparse formats that incorporate these index compression ideas. Although the RCM method is a heuristic specifically designed to optimize the sparse matrix profile, other heuristics could be explored with the specific focus of harnessing techniques such as delta encoding. Another interesting line is the creation of a new storage format that mixes 8, 16 and 32 bits representation for indices depending of the matrix. The general idea would be to divide the matrix into three blocks in which have the rows use each number of bits. Finally, developing a publicly available software library is essential to use and experimentally evaluate these techniques.

Acknowledgments

We acknowledge support of the ANII MPG Independent Research Group: Efficient Heterogeneous Computing at UdelaR, a partner group of the Max Planck Institute in Magdeburg. This work is partially funded by the UDELAR CSIC-INI project *CompactDisp: Formatos dispersos eficientes para arquitecturas de hardware modernas.* We also thank PEDECIBA Informática and the University of the Republic, Uruguay.

References

1. Anzt, H., Dongarra, J., Flegar, G., Higham, N.J., Quintana-Ortí, E.S.: Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers. Concurrency and Computation: Practice and Experience **31**(6), e4460 (March 2018). https://doi.org/10.1002/cpe.4460
2. Bell, N., Garland, M.: 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)
3. Bell, N., Garland, M.: Cusp library (2012), https://github.com/cusplibrary/cusplibrary
4. Berger, G., Freire, M., Marini, R., Dufrechou, E., Ezzatti, P.: 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)
5. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th national conference. pp. 157–172. ACM Press (1969). https://doi.org/10.1145/800195.805928
6. Dufrechou, E., Ezzatti, P., Freire, M., Quintana-Ortí, E.S.: Machine learning for optimal selection of sparse triangular system solvers on gpus. J. Parallel Distributed Comput. **158**, 47–55 (2021), https://doi.org/10.1016/j.jpdc.2021.07.013

7. Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Selecting optimal SpMV realizations for GPUs via machine learning. Int. J. High Perform. Comput. Appl. **35**(3) (2021), https://doi.org/10.1177/1094342021990738
8. Gale, T., Zaharia, M., Young, C., Elsen, E.: 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)
9. Grützmacher, T., Cojean, T., Flegar, G., Göbel, F., Anzt, H.: A customized precision format based on mantissa segmentation for accelerating sparse linear algebra. Concurrency and Computation: Practice and Experience **32**(15) (July 2019). https://doi.org/10.1002/cpe.5418
10. Guo, D., Gropp, W., Olson, L.N.: A hybrid format for better performance of sparse matrix-vector multiplication on a GPU. The International Journal of High Performance Computing Applications **30**(1), 103–120 (July 2015). https://doi.org/10.1177/1094342015593156
11. Gustavson, F.G., Liniger, W., Willoughby, R.: Symbolic generation of an optimal crout algorithm for sparse systems of linear equations. J. ACM **17**(1), 87–109 (1970)
12. Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., Sadayappan, P.: Adaptive sparse tiling for sparse matrix multiplication. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. ACM (February 2019). https://doi.org/10.1145/3293883.3295712
13. Kourtis, K., Goumas, G., Koziris, N.: Optimizing sparse matrix-vector multiplication using index and value compression. In: Proc. of the 2008 conference on Computing frontiers. ACM Press (2008). https://doi.org/10.1145/1366230.1366244
14. Langr, D., Tvrdík, P.: Evaluation criteria for sparse matrix storage formats. IEEE Transactions on Parallel and Distributed Systems **27**(2), 428–440 (2016). https://doi.org/10.1109/TPDS.2015.2401575
15. Maggioni, M., Berger-Wolf, T.: CoAdELL: Adaptivity and compression for improving sparse matrix-vector multiplication on GPUs. In: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops. IEEE (May 2014). https://doi.org/10.1109/ipdpsw.2014.106
16. Marichal, R., Dufrechou, E., Ezzatti, P.: Optimizing sparse matrix storage for the big data era. In: Cloud Computing, Big Data & Emerging Topics - 9th Conference, JCC-BD&ET, La Plata, Argentina, June 22-25, 2021, Proceedings. Communications in Computer and Information Science, vol. 1444, pp. 121–135. Springer (2021), https://doi.org/10.1007/978-3-030-84825-5_9
17. Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures. In: High Performance Embedded Architectures and Compilers, pp. 111–125. Springer Berlin Heidelberg (2010)
18. Pinar, A., Heath, M.T.: Improving performance of sparse matrix-vector multiplication. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing. p. 30–es. SC '99, Association for Computing Machinery, New York, NY, USA (1999)
19. Saad, Y.: Sparskit: a basic tool kit for sparse matrix computations - version 2 (1994)
20. Sun, X., Zhang, Y., Wang, T., Zhang, X., Yuan, L., Rao, L.: Optimizing SpMV for diagonal sparse matrices on GPU. In: 2011 International Conference on Parallel Processing. IEEE (September 2011). https://doi.org/10.1109/icpp.2011.53
21. Tang, W.T., Tan, W.J., Ray, R., Wong, Y.W., Chen, W., hao Kuo, S., Goh, R.S.M., Turner, S.J., Wong, W.F.: 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). https://doi.org/10.1145/2503210.2503234

22. Willcock, J., Lumsdaine, A.: Accelerating sparse matrix computations via data compression. In: Proceedings of the 20th annual international conference on Supercomputing - ICS '06. ACM Press (2006). https://doi.org/10.1145/1183401.1183444
23. Xu, S., Lin, H.X., Xue, W.: 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)
24. Yang, C., Buluç, A., Owens, J.D.: Design principles for sparse matrix multiplication on the gpu. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018: Parallel Processing. pp. 672–687. Springer International Publishing, Cham (2018)