DIV: An Index & Value compression method for SpMV on large matrices

Dimitrios Galanopoulos
Department of Computer Science
National Technical University of Athens
Athens, Greece
dgal@cslab.ece.ntua.gr

Panagiotis Mpakos
Department of Computer Science
National Technical University of Athens
Athens, Greece
pmpakos@cslab.ece.ntua.gr

Petros Anastasiadis
Department of Computer Science
National Technical University of Athens
Athens, Greece
panastas@cslab.ece.ntua.gr

Nectarios Koziris
Department of Computer Science
National Technical University of Athens
Athens, Greece
nkoziris@cslab.ece.ntua.gr

Georgios Goumas
Department of Computer Science
National Technical University of Athens
Athens, Greece
goumas@cslab.ece.ntua.gr

Abstract

SpMV on large matrices is a heavily memory-bound kernel, a characteristic attributed to its extremely low computational intensity. To address this, research has mainly focused on compressing the matrix indices. Nevertheless, the values of a matrix usually occupy up to two thirds of the total size. Research on value compression, on the other hand, has been limited to specific matrix types. In this paper, we propose DIV, a combined index and value lossless compression scheme, based on variations of delta and run-length encoding, that achieves substantially improved SpMV performance for large matrices, i.e., those that exceed the CPU cache. We evaluate its performance against other state-of-the-art matrix formats, on an Intel Xeon and an AMD EPYC platform. Our format achieves 77% and 115% geometric mean speedup respectively versus the Intel MKL library. We finally demonstrate the applicability of DIV on a Biconjugate Gradient Stabilized solver, where we also achieve significant speedups.

CCS Concepts

• **Computing methodologies → Linear algebra algorithms**; • **Theory of computation → Data compression**.

Keywords

Sparse Matrix-Vector Multiplication, Floating-Point Compression, Sparse Matrix Value-Index Compression

ACM Reference Format:
Dimitrios Galanopoulos, Panagiotis Mpakos, Petros Anastasiadis, Nectarios Koziris, and Georgios Goumas. 2025. DIV: An Index & Value compression method for SpMV on large matrices. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3721145.3725767

1 Introduction

Sparse Matrix-Vector (SpMV) multiplication is encountered in many fields including scientific computing and machine learning. It performs the product of a sparse matrix with a dense input vector and returns a dense output vector. SpMV is one of the lowest-performing kernels, a fact that can be attributed to many potential performance issues, like its extremely low operational intensity (*memory bandwidth bound*), irregular memory accesses to the input vector (*memory latency bound*), *load imbalance* and *low instruction-level parallelism (ILP)* [5, 10–12, 24, 31, 34].

Although much work has been done to mitigate these problems [2, 4, 7, 8, 18, 22, 23, 28, 30], as matrices grow larger and the gap between computational power and memory speed widens, the memory bandwidth bottleneck remains the most important obstacle to high performance [31]. The case of large matrices specifically is indeed significant, as they represent contemporary and challenging real problems. For example, the configuration of the HPCG benchmark[1] states that data arrays should not fit inside the cache (remarked as "unrealistic in a real application setting"), but on the contrary occupy a significant portion of the main memory (at least 1/4).

Focusing on the case of SpMV on large matrices in CPUs, the kernel is memory bandwidth bound and all other bottlenecks have a minor impact on performance. To showcase this, we present in Figure 1 an evaluation of SpMV performance on an AMD EPYC CPU, for a dataset of 29 *large matrices*, i.e., substantially larger (2x) than the CPU Last Level Cache - LLC, from the SuiteSparse matrix collection [3] (see Section 5 for details on the setup and dataset). We apply matrix formats designed to address each bottleneck, and compare with the standard CSR (with simple row-granularity balancing) as the baseline and the Intel MKL library as a state-of-practice format. Specifically, we test: a) vectorization, as a prominent computational optimization for ILP, b) CSR5 [28], which breaks short rows and tackles low ILP and imbalance, c) Merge [30], which tackles load imbalance, d) MKL_col_0, an erroneous version of MKL SpMV that zeros out the column index array of the CSR format and thus leads to perfect access to the input vector (as an indicator of the upper bound of any optimization that would target the irregular memory accesses

[1] https://hpcg-benchmark.org/faq/index.html

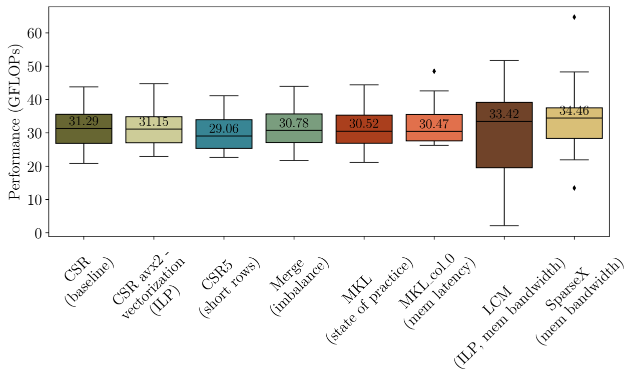


Figure 1: Evaluation of approaches targeting the various SpMV bottlenecks, for a dataset of large matrices (i.e., substantially larger than the CPU cache) on an AMD EPYC 64-core CPU.

to the input vector), e) LCM [2], a format that searches for strided substructures in the matrix and generates an optimized codelet for each one, therefore targeting ILP and memory bandwidth, and f) SparseX [8], a format that aggressively tries to reduce the size of the structural metadata of sparse matrices (i.e., indices), addressing the memory bandwidth bottleneck. Interestingly, mainly SparseX is able to reliably speedup SpMV for large matrices (median speedup of 13%, geometric mean speedup of 7.7%), while LCM also has some advantageous cases but is a lot more irregular, (median speedup of 9.5%, geometric mean speedup of −12%, i.e., slowdown). This implies that, for the case of large matrices, optimizations should be directed to mitigating the memory bandwidth bottleneck, and that the other bottlenecks are not dominant.

Data compression emerges as a natural solution to address the memory bandwidth bottleneck of SpMV. Compression sacrifices unused processing power to reduce the memory footprint and, consequently, the required bandwidth. Most of the research effort so far has focused on compressing the redundancy in the structure of the matrix, i.e., reducing the indexing footprint of sparse matrices [2, 20, 21, 36, 37]. In the generic sparse representation format CSR, indexing metadata include approximately $(m + nnz)$ 32-bit integers, where m and nnz are the number of rows and nonzero elements of the matrix respectively. Many applications however, require double precision arithmetic, which means that the memory footprint of the actual values of the matrix is nnz 64-bit floats, i.e., almost 2/3 of the total matrix footprint. Clearly, matrix values are an excellent target for compression.

Compression of floating-point numbers is a field that has been broadly explored. Some efforts focus on complicated and compute-intensive lossless implementations [1, 9, 26, 32], while others focus on heavy compression using lossy methods [25]. Most of these schemes, though, are incompatible with the extremely low computational intensity of SpMV, since the compute overhead they introduce outweighs the execution time of the kernel itself. This is because they usually target reducing disk I/O access times, and not memory access times. Other schemes target specific applications that do not require high precision arithmetic [25], or assume very smooth variations in the values in order to be effective and accurate [9]. For the specific case of SpMV, the most common compression scheme that has been proposed is dictionary-based, and tries to reduce the replication of the unique values of the matrix [13, 20, 33, 38, 39]. This method though, is highly sensitive to the number of the unique values, and rapidly loses its compression effectiveness as this number increases.

In this paper we aim to aggressively compress the memory footprint of a sparse matrix and thus we target both its metadata (indexing structure) and actual data. To that end, we propose DIV, a combined index and value compression scheme. Our approach is currently applied to CPUs and is designed to be effective for large-scale matrices that exceed the size of the LLC, where the memory bandwidth limitation is much more severe. DIV combines index and value compression techniques to effectively reduce the matrix memory footprint (Sections 3,4). It achieves lossless matrix compression and is capable of drastically improving the SpMV performance of large matrices. The compression process has a rather low overhead, in the order of a few dozens of SpMV operations on average, making it effective both for offline and online processing, (e.g., in solvers with multiple iterations). In addition, since exploiting matrix symmetry can also be viewed as a way to mitigate the memory bandwidth bottleneck, we propose a symmetric extension of our DIV format.

We evaluate our method on two different platforms, an AMD EPYC 64-core CPU and an Intel Xeon 56-core CPU, and against various matrix formats, including formats specialized for symmetric matrices (Section 5). Our implementation offers significant speedups compared to state-of-the-art index and value compression formats, and on the AMD platform we achieve on average above single-precision floating-point performance. In particular, against the best performing state-of-the-art double-precision format tested, SparseX [8], we achieve speedups of 87% on Intel and 100% on AMD, while against the state-of-practice MKL format we achieve 77% and 115% respectively. Furthermore, we incorporate DIV into a simple Biconjugate Gradient Stabilized solver, where it again achieves notable speedups (Section 5.4). We demonstrate that it has relatively low preprocessing overhead and more importantly, that there are no functional or performance shortcomings that would prohibit its integration in an end-to-end application. Finally (Section 6), we explore the performance scalability, extended matrix datasets and showcase that our format remains effective in lower floating point precisions (32-bit). DIV is distributed under an open-source license and the code is publicly available[2].

2 Background

2.1 Floating-point Arithmetic

In this paper we work with floating-point numbers as defined in the IEEE 754-2008 standard [17]. The double-precision 64-bit numbers comprise 1 bit for the sign, 11 bits for the exponent and 52 bits for the fraction or mantissa, while the single-precision 32-bit numbers have 1, 8 and 23 bits of the respective parts. The nature of this representation makes it denser around zero, growing sparser as the exponent of the number increases.

[2]GitHub repository: https://github.com/cslab-ntua/SpMV-Research
(relevant folder : benchmark_code/CPU/AMD/spmv_code_bench/DIV)

```
// Serial
for (j=0;j<nnz;j++)
    y[RowIdx[j]] += Values[j] * x[ColIdx[j]]

// Parallel
parallel for (j=0;j<nnz;j++)
    y[RowIdx[j]] = atomic_add( y[RowIdx[j]],
                   Values[j] * x[ColIdx[j]] )
```

Listing 1: SpMV kernels for the COO format.

```
// Serial
for (i=0;i<num_rows;i++)
    for (j=RowPtr[i];j<RowPtr[i+1];j++)
        y[i] += Values[j] * x[ColIdx[j]]

// Parallel
parallel for (i=0;i<num_rows;i++)
    for (j=RowPtr[i];j<RowPtr[i+1];j++)
        y[i] += Values[j] * x[ColIdx[j]]
```

Listing 2: SpMV kernels for the CSR format.

2.2 SpMV Kernel and Matrix Formats

The SpMV operation involves the multiplication of a sparse matrix with a dense input vector, resulting in a dense output vector, i.e., $y = A \cdot x$. The sparse matrix is stored in an appropriate format that reduces its memory footprint by keeping only the nonzero values. Therefore, a sparse matrix format needs to additionally encode the structure of the nonzeros distribution inside the matrix. There is a wide variety of formats in literature, but in this paper we work with the state-of-practice (i.e., those provided by vendors) and the most relevant state-of-the-art formats discussed next.

2.2.1 Coordinate Format (COO). The COO format is a straightforward approach that stores both row and column indices for each nonzero element. COO is very frequently used for file storage of the matrices. COO's flat nature makes nonzero reordering easier, helps with load balancing and avoids the loop overheads of the double loop of CSR. These benefits, however, come at the cost of larger memory footprints, while the random accesses to the output vector make parallelization less efficient, as atomic writes to the output vector are required. In Listing 1 we show both a serial and a parallel implementation of the SpMV kernel with the COO format.

2.2.2 Compressed Sparse Row Format (CSR). CSR is the de facto format and base to many derivatives, due to its solid performance, ease of implementation and straightforward parallelization. In CSR the nonzeros are sorted according to their row indices, storing the values and column indices in two arrays, *Values* and *ColIdx* respectively. An additional *RowPtr* array contains the position of the first element of each row in the previous two arrays. Likewise, in Listing 2 we show serial and parallel implementations for CSR.

2.2.3 CSR5 [28]. CSR5 is a popular format that tackles the load imbalance of CSR, by dividing rows into subproblems of almost equal size. This modification is particularly tailored for GPUs and in general devices with large core counts, where conventional CSR fails to distribute workload effectively among processing elements.

2.2.4 SparseX [8]. SparseX partitions the original matrix into distinct substructures, in order to minimize the overall memory footprint. This objective is realized through the identification of dense, horizontal, vertical, diagonal or block substructures within the sparse matrix. We include this format in our evaluation as a representative of drastic index compression schemes that are advantageous for larger matrices.

2.2.5 LCM [2]. The LCM format is an inspector-executor framework that searches for strided and partially strided regions / memory accesses in sparse codes (i.e., sparse matrix substructures), much like SparseX, and generates optimized and vectorized codelets for each one.

2.2.6 CSR&RV [38]. This format uses a dictionary-based value compression scheme, one of the most commonly found value compression techniques in literature, and excels when matrices have few unique values. However, it cannot be considered a generic approach, since its performance dramatically drops as the number of unique values increases.

2.2.7 CFS [6]. This targets SpMV on symmetric matrices. It avoids both fine-grained synchronization and non-scalable reduction operations by breaking the computation into phases, wherein a subset of the rows is processed in parallel so that no race conditions during the update of the output vector occur.

3 Design Considerations

When designing a compression scheme for the SpMV kernel, we need to adhere to a number of requirements and encounter a variety of alternative options regarding the algorithm, the granularity of data and others, that fine tune the selected approaches. In this section, we discuss the compression and decompression requirements in our case, show why existing floating-point compression schemes that operate at large data granularities are not suitable, and explain why we need to resort to simpler solutions. We then provide some more details for each one of them.

3.1 Compression and Decompression Overhead

As in several other research works on the field [2, 6, 8] and in production libraries like the MKL Inspector-Executor scheme, we assume that we have the luxury to preprocess the sparse matrix, or, in other words, that the compression cost is not in the critical path of the application. This is because in typical scenarios the matrix does not change, so it can be preprocessed offline once, saved on disk and reused multiple times in different application executions. In other cases, the matrix is used in iterative algorithms (like the Krylov iterative solvers) where the preprocessing cost can be amortized during the iterative execution. In Section 5.4 we test our scheme for online compression in the Biconjugate Gradient Stabilized iterative solver, where we demonstrate that it can be a viable option for this case as well. The decompression cost, however, needs to meet much stricter requirements, since decompression takes place every time the matrix is used in the SpMV operation during application execution. A heavyweight scheme for decompression is doomed to fail in the use cases under consideration. Consequently, in our design, we allow for some tolerance in the cost of compressing the matrix, but require minimal tolerance in the cost of decompressing the matrix.

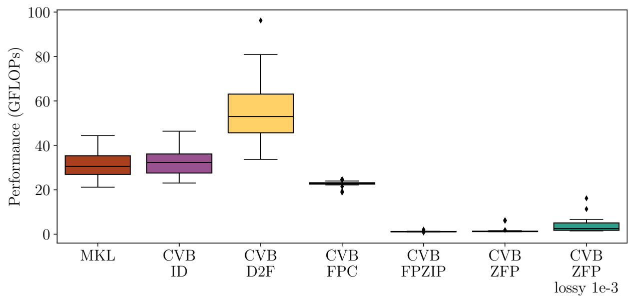


Figure 2: Performance evaluation of existing compression schemes applied to SpMV on an AMD EPYC 64-core CPU.

3.2 Existing Compression Methods and Compression Granularity

The goal of compression for SpMV is to reduce the matrix memory footprint and in turn alleviate the CPU-memory traffic. To that end, data should be decompressed in parts that fit inside the CPU cache, otherwise the approach would induce memory traffic and cancel any benefits of compression. Our first attempt was to employ existing floating-point compression techniques that operate at a coarse-grained, chunk granularity. During execution, we load each compressed packet from memory to the CPU cache, decompress it fully inside the cache and then immediately apply the SpMV operations to the decompressed data. A big advantage of this method is that it can be combined easily with any pre-existing compression method.

We implemented a version of this strategy for testing (we refer to it as CVB - Compressed Value Blocks), by integrating various lossless algorithms designed to compress floating-point data (FPC [1], FPZIP [26], ZFP [25]). In Figure 2 we present a performance evaluation (details about the testbed and dataset are in Section 5). 'CVB ID' is a baseline check, where we do not apply any compression to the data. Since this matches the performance of the Intel MKL library, we can assume an efficient implementation of this baseline. On the other hand, 'CVB D2F' is a simple cast of the double values to floats and back. This is lossy, but it is an indicator that this strategy could potentially offer performance benefits. Nevertheless, no other actual compression method that we tried achieved any speedup over MKL, not even the lossy variant of ZFP (tolerance 1e-3).

The primary pain point lies in the overhead associated with decompression, therefore a design goal of our approach is to minimize the decompression cost. To achieve this, we need to operate in a more fine-grained manner, even if sacrificing some additional potential for compression. More specifically, our method works on single matrix elements in a continuous, streaming manner, and we resort to simpler approaches to exploit data redundancy, like delta encoding for adjacent values and run-length encoding for replicated values. In this way, basic operations (fetch element, decompress element, compute) may better utilize the micro-architectural characteristics of modern out-of-order (OoO) CPUs. We also implement a variant of our method that exploits matrix symmetry.

3.3 Adapting Floating-point Delta Encoding for SpMV

The main goal of delta encoding is to produce a residual of reduced significant bits, for example by subtracting two numbers, 'a' and 'b', and getting rid of leading or trailing zero bits. This leads to the observation that different permutations of the same dataset can have varying degrees of delta compression effectiveness. Moreover, due to the nature of floating-point numbers, operations can induce errors (e.g., $a - b + b = a + error$, if a, b are not close enough) that increase with the distance of the operands. Sorting data by value is a permutation that minimizes the distance between consecutive values and thus seems a viable approach to address both previous considerations.

At the same time, previous work on floating-point compression [9, 26, 32] bypasses floating-point errors by handling their bit representations as integers and applying integer subtraction on them. We also explored other techniques, like Kahan summation [19], however these have high decompression overheads. We thus have two options to consider, straightforward floating-point subtraction on the values ('float-delta' encoding), or integer subtraction on their bit representations ('integer-delta' encoding). We ultimately selected the integer-delta encoding due to its lossless properties. In the following paragraphs, we address some complications of this method. First, note that while floating-point subtraction creates trailing zero bits, integer subtraction creates leading zero bits. For example:

$$\begin{aligned} x = 0.923486 &: \texttt{3fed8d32830a0b1c} \\ y = 0.923487 &: \texttt{3fed8d349be8ff32} \\ y - x = 0.000001 &: \texttt{3eb0c6f7a0b}\mathbf{\texttt{00000}} \\ I(y) - I(x) &: \mathbf{\texttt{0000000}}\texttt{218def416} \end{aligned}$$

where with '$I()$' we denote the integer representation of the bits of a floating-point number. On top of that, it is a common occurrence to have values with trailing zero bits in the dataset (e.g., integer or quantized values). This pairs well with float-deltas, but for integer-deltas it compels us to track zeros in both directions. This can make the code somewhat more complex and increase the decompression overhead:

$$\begin{aligned} x = 9.000000 &: \texttt{4022000000000000} \\ y = 13.000000 &: \texttt{402a000000000000} \\ y - x = 4.000000 &: \texttt{401}\mathbf{\texttt{0000000000000}} \\ I(y) - I(x) &: \mathbf{\texttt{000}}\texttt{8}\mathbf{\texttt{000000000000}} \end{aligned}$$

Furthermore, integer-deltas are highly dependent on the order of the operands. Integer subtraction of a larger *absolute value* floating-point number from a smaller one results in a negative integer, with no leading zeros:

$$\begin{aligned} x = -0.923487 &: \texttt{bfed8d349be8ff32} \\ y = -0.923486 &: \texttt{bfed8d32830a0b1c} \\ y - x = 0.000001 &: \texttt{3eb0c6f7a0b}\mathbf{\texttt{00000}} \\ I(y) - I(x) &: \underline{\texttt{fffffff}}\texttt{de7210bea} \\ \text{but,} \quad x - y = -0.000001 &: \texttt{beb0c6f7a0b}\mathbf{\texttt{00000}} \\ I(x) - I(y) &: \mathbf{\texttt{0000000}}\texttt{218def416} \end{aligned}$$

Finally, integer-deltas for values of opposite signs give negative values, i.e., the first bit is 1:

```
int greater_than(a, b) {
    if (a*b <= 0)
        return (a > b) ? 1 : (a < b) ? -1 : 0;
    return (fabs(a) > fabs(b)) ? 1
         : (fabs(a) < fabs(b)) ? -1 : 0;
}
```

Listing 3: Value sorting comparison function for integer-deltas.

$$
\begin{aligned}
x = 0.923486 &: \texttt{3fed8d32830a0b1c}\\
y = -0.923487 &: \texttt{bfed8d349be8ff32}\\
y - x = -1.846973 &: \texttt{bffd8d338f798527}\\
I(y) - I(x) &: \texttt{\textbf{8}\underline{000000}218def416}
\end{aligned}
$$

By taking into account the previous remarks, we can infer that an optimal arrangement of the values, for good compression results using integer-deltas, is to first separate the negative and positive ones and then sort each part by absolute value. This method can be expressed with the comparison function shown in Listing 3.

3.4 Exploiting Replicated Values

We aim to combine delta encoding with techniques that take advantage of the multiplicity of the matrix values. Dictionary methods, as mentioned in Section 1, are not scaling efficiently with the matrix size and fail when there is a large number of unique values. We aim to exploit this form of redundancy with the use of a form of run-length encoding. We will split the values into groups of the same multiplicity or 'replication factor'. For example, each value of a group with replication factor 3 has two more replicates. Therefore, for each such group we only need to keep the unique values, and two more numbers as metadata: the replication factor and the number of the unique values.

This method is efficient with regard to the extra metadata needed. The worst case with the maximum metadata is when we have the maximum possible groups, which means each group would have only one unique value and the groups would also have the smallest possible replication factors. Therefore, the worst case is when we have groups with multiplicities 1, 2, 3, ... with only one element each. The total number of elements grows with the square of the number of groups ($1 \cdot 1 + 2 \cdot 1 + 3 \cdot 1 + \ldots + rf_{max} \cdot 1 = \frac{(rf_{max}+1) \cdot rf_{max}}{2}$). This means that, in the worst case, the number of groups, and as a consequence the size of the extra metadata needed, only grows according to the square root of the total number of elements.

3.5 Adaptation for Vectorization Support

When the matrix size exceeds the size of the CPU cache, vectorization generally cannot improve the performance of SpMV (see Figure 1). Nevertheless, by reducing the memory footprint with compression and with the additional decompression overhead, we end up with a more execution-heavy load that can actually take advantage of vectorization. To adapt the method for vectorization support, we distribute the sorted values into consecutive parts, equal in number to the utilized vectorization width. We refer to these parts as *lanes*, and each such lane can be processed independently.

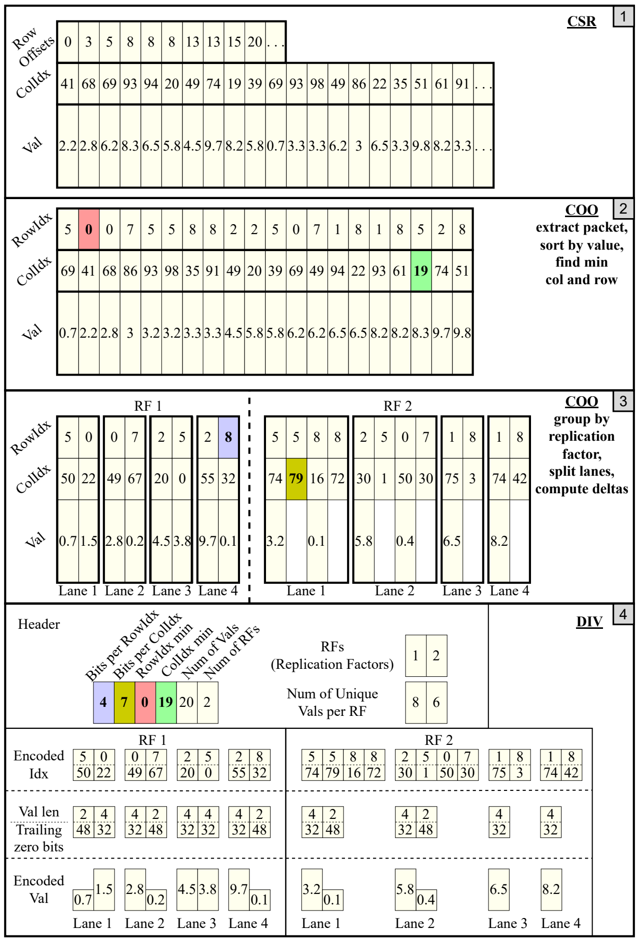


Figure 3: Overview of the compression method. The sizes are simplified for demonstration purposes.

4 Methodology

As discussed in Section 3, we aim to reduce the memory footprint by employing simple compression algorithms with fast decompression. More specifically, we utilize delta encoding to compress close adjacent values and run-length encoding to merge multiple replicas of the same value. For the matrix indices, we apply a variation of delta encoding from a local minimum. We also implement a symmetric variant to further reduce the footprint of symmetric matrices. We then decompress the matrix on the fly and perform the required operations to compute the entire SpMV operation. In this section, we provide an overview and then more details of our method, called DIV.

4.1 DIV Design and Overview

Figure 3 presents an overview of DIV with the help of a simplified example. We refer to the respective compression stage numbers in the text. We assume an input matrix in the CSR format $\boxed{1}$ and describe how the matrix is progressively partitioned to realize our scheme.

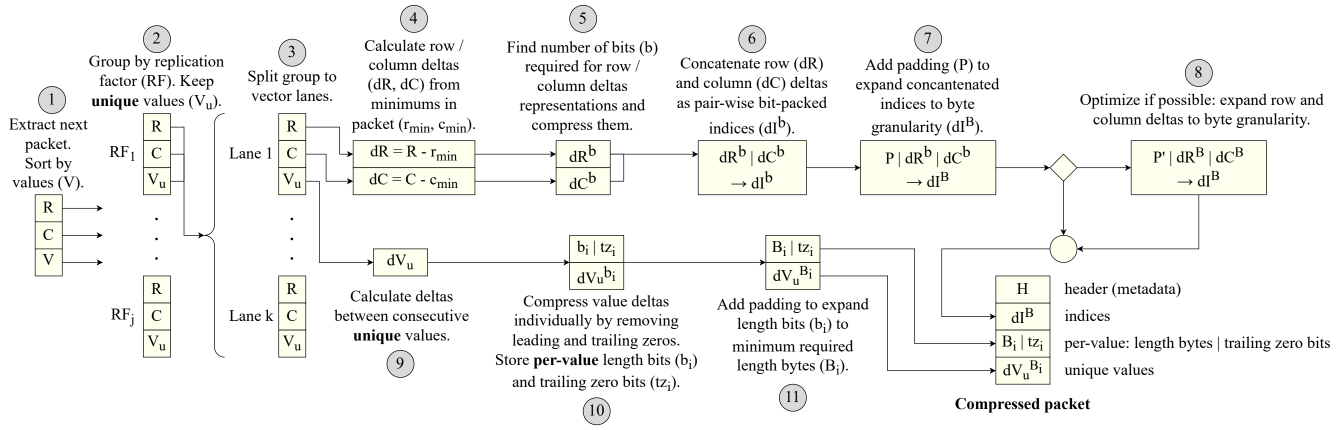

1
Extract next packet. Sort by values (V).
R
C
V
2
Group by replication factor (RF). Keep unique values (Vu).
RF1
RFj
3
Split group to vector lanes.
Lane 1
Lane k
4
Calculate row / column deltas (dR, dC) from minimums in packet (rmin, cmin).
dR = R - rmin
dC = C - cmin
5
Find number of bits (b) required for row / column deltas representations and compress them.
dRb
dCb
6
Concatenate row (dR) and column (dC) deltas as pair-wise bit-packed indices (dIb).
dRb | dCb → dIb
7
Add padding (P) to expand concantenated indices to byte granularity (dIB).
P | dRb | dCb → dIB
8
Optimize if possible: expand row and column deltas to byte granularity.
P' | dRB | dCB → dIB
dVu
Calculate deltas between consecutive unique values.
9
bi | tzi
dVubi
Compress value deltas individually by removing leading and trailing zeros. Store per-value length bits (bi) and trailing zero bits (tzi).
10
Bi | tzi
dVuBi
Add padding to expand length bits (bi) to minimum required length bytes (Bi).
11
H
dIB
Bi | tzi
dVuBi
header (metadata)
indices
per-value: length bytes | trailing zero bits
unique values
Compressed packet

Figure 4: DIV compression of an individual packet. The steps shown are for one lane, of one replication factor group of the packet. The same steps are repeated for the rest, until the complete compressed packet is formed.

Thread-level partitioning: To enable parallelization and avoid the need for synchronization on the output vector we initially split the matrix into *submatrices*, consisting of complete consecutive rows, and distribute each one to a CPU thread. We balance the workload by distributing as close to the same number of nonzeros as possible, limited of course by the row granularity.

Packet-level partitioning: $\boxed{2}$ The more values we process together, the more opportunities we have to eliminate redundancy between them. On the other hand, it also widens the range of the indices and makes index compression less effective. Therefore, each thread partitions its designated submatrix into *packets* of again consecutive rows (first and last row of each packet can be partial) and works on each packet independently. It first sorts each packet by value, as described in Section 3.3 and in Listing 3, in order to increase the efficiency of delta encoding for the values.

Since the sorted-by-value packets have a random indexing scheme, we use COO as their representation, which does not expect any special order of the indices. To mitigate the additional storage overhead of COO, we employ a delta encoding scheme that compresses the indices of each packet by subtracting the minimum respective index in the packet (Section 4.2.1). While there is no limit in the span of column indices, we can control the span of row indices by constraining the maximum number of rows in every packet, and by extension the packet size. This way, the packet size determines the trade-off between value and index compression efficiency.

Replication-level partitioning: $\boxed{3}$ To take advantage of possible value replication in the packet, we split the packet into groups based on their replication factor as described in Section 3.4. This is referred to as 'RF' in Figure 3. For example RF 2 means each value has 1 more replicate, while RF 1 means all values are unique.

Vectorization-level partitioning: Moreover, to utilize the CPU vectorization capabilities, we further split each replication factor group into independent sub-groups, or *lanes* (Section 3.5), to feed the vector units of the processor and decrease in this way the decompression overhead. The number of lanes is equal to the width of the CPU's vector units that are utilized.

$\boxed{4}$ Finally, we end up with a series of packets per thread, each one consisting of a header with metadata and its replication factor groups. Each group contains all the group value and index deltas. For each value delta only the significant (nonzero) part of its 64-bit representation is stored, along with its length in bytes ('Val len' in Figure 3) and its number of trailing zeros (Section 3.3). We note that the numbers shown in Figure 3 are simplified for the example. The header contains information regarding the number of bits that are used for the encoding of row and column indices and the minimum row and column indices. Note that, based on the way the packets are formed (packet-level partitioning), we typically need more bits to encode column indices than row indices, like in our example we need 7 and 4 bits respectively. The header further contains the number of elements, the number of replication factor areas, and for each area its RF and the number of unique values. The header imposes a memory footprint overhead of $O(1)$ for the scalar metadata plus a worst case of $O(\sqrt{packet\_nonzeros})$ (as discussed in Section 3.4) for the replication factor metadata, most commonly again $O(1)$.

4.2 Packet Compression in Detail

In Figure 4 we present the compression steps for each packet (we reference each step by its number). The first step of our compression scheme is to sort the data of the packet as described in Section 3.3 ①. Then we group the sorted values by replication factor (Section 3.4) ②. Lastly, for each such group we split the data into vectorization lanes (Section 3.5) ③ and for each lane we take the following steps.

4.2.1 Steps of index compression. For each packet we find the minimum row and minimum column index ④, and we encode the nonzero coordinates as deltas from these minimums ⑤. We use a ceiling of 256 rows to keep a maximum of 8 bits for the row delta values. On the other hand, we cannot control the span of the column indices, so the column delta sizes are matrix dependent, but usually still compressible to below 4 bytes. Our experiments also showed that working at bit granularity (as opposed to byte granularity) is inducing too high a performance overhead during decompression. Therefore, we concatenate the row and column delta values pairwise ⑥, add padding to the resulting bits to expand to the minimum required bytes ⑦ and, if possible, distribute this padding so that the row and column delta bits are also at byte granularity ⑧. We have implemented optimized code paths for the convenient index sizes, selected at decompression time (run-time) for each packet.

4.2.2 Steps of value compression. Value compression follows a similar logic. We calculate the (integer-)deltas between consecutive values in the sorted vectorization lane ⑨ and then compress each value delta individually ⑩ by discarding the leading and trailing zero bits. Since there is significant variance in compressibility, one size cannot fit all values of a packet effectively. For each delta we have to encode information about both the leading and the trailing zero bits. Therefore, after expanding the compressed deltas to byte granularity ⑪, we additionally store 4 bits for the length in bytes and 4 bits for the trailing zeros, (i.e., a shift at decompression), concatenated as 1 byte.

4.3 Decompression and Execution

We end up with a compressed packet composed of four parts, as shown in Figure 3 and Figure 4: a header with metadata, the deltas of the indices, the deltas of the values and for each one its size and trailing zero bits. Consecutive packets are distributed to the CPU threads, ensuring that no two threads share the same matrix row. Each thread can then sequentially process each packet in a streaming manner: a) decompress a value and its coordinates, b) gather the respective input/output vector values $x[column]$ and $y[row]$, c) execute the SpMV operation $y[row] += value * x[column]$ and d) proceed to the next value.

4.4 Variant for Symmetric Matrices

We also implement a symmetric extension of DIV. From each packet, we extract the nonzeros whose symmetric elements belong to the same owning thread, so that no synchronization between threads is needed during the SpMV execution. We form an additional new packet with these nonzeros, following the exact same methodology as described above, but storing only one nonzero for each symmetric pair, and expanding to the pair accordingly during decompression.

5 Evaluation

5.1 Experimental Setup

In our evaluation process, we consider three variations of our format: *DIV*, *DIV_RF* and *DIV_RF_SYM*. The difference between the first two is that in DIV_RF we enable the replication factors technique, as described in Section 3.4, while in DIV we do not. Lastly, DIV_RF_SYM is the symmetric extension of DIV_RF, as described in Section 4.4. For the packets we empirically select a size of 16K (i.e., 16384) values, which in our tests led to a plateau in average performance, due to the ceiling we impose of 256 maximum consecutive rows per packet (Section 4.2.1). The most effective packet size for each matrix is not always the same, but we leave further exploration for future work.

We assess the performance of SpMV across two CPU architectures: an Intel Platinum 8480+ CPU and an AMD EPYC 7763 CPU. More details on the testbed hardware and the utilized formats/libraries can be found in Table 1. The AMD EPYC CPU is configured as NPS4 (4 NUMA nodes per socket). To ensure optimal data locality, the threads are consistently pinned to the cores using

Table 1: Testbed characteristics and formats tested.

	Testbed I	**Testbed II**
CPU	Intel Xeon Platinum 8480+	AMD EPYC 7763
Cores	56 @ 2.0 GHz	64 @ 2.45 GHz
Memory	256 GB	256 GB
	DDR4 180 GB/s	DDR4 100 GB/s
LLC	105 MB (L3)	256 MB (L3)
Formats	DIV	DIV
	DIV_RF (64/32-bit)	DIV_RF (64/32-bit)
	DIV_RF_SYM	DIV_RF_SYM
	MKL-IE 2023.1	MKL-IE 2023.1
	LCM	LCM
	SparseX	SparseX
	Dictionary - custom	Dictionary - custom
	Dictionary - CSR&RV	
	CSR symmetric	CSR symmetric
	CFS	CFS
Compiler	gcc 13.2.0	gcc 12.2.0
	-O3 -march=native	-O3 -march=native

OpenMP environment variables, and the matrices are initialized in a parallel manner, following the Linux first-touch policy. The memory bandwidth of each testbed is calculated using the STREAM benchmark [29].

For the dataset, we form a matrix suite (Table 2) from the SuiteSparse sparse matrix collection [3], composed of 29 out of the 31 matrices of the collection that a) include floating point values and b) are larger than twice the size of the LLC of the AMD CPU, so that at least half the data reside outside the cache. The remaining 2 were too large and slowed the testing. The matrices cover a wide spectrum of sizes and unique value percentages, the most performance-influential factors for compression. The 'UV' column contains the percentages of the unique values for each matrix, calculated as the ratio of unique values to the total number of nonzeros. We note that a UV of around 50% is very common, because many matrices of scientific applications are symmetric. In the 'CF' column we present the compression fraction of DIV_RF (packet size 16384), i.e., the ratio of the compressed to the original total matrix in double-precision CSR format. The geometric mean of the compression fraction we achieve is 49.4%. We can compare this with the CF of the single-precision CSR, which is approximately 66% (same memory footprint for indices, half for values). We indeed observe that in most cases we achieve comparable or better total compression.

We compare DIV against several other formats, namely the Inspector-Executor CSR of the Intel MKL library, CSR5 [28], Merge [30], LCM [2], SparseX [8], two dictionary based value compression methods (CSR&RV [38] and a custom one), a custom symmetric CSR implementation (CSR sym) and CFS [6]. We also tested the symmetric version of MKL, which always had the same or lower performance compared to the general MKL, as well as the AMD AOCL library, which was significantly worse (harmonic mean of only 10 GFLOPs on the AMD platform). Consequently, we exclude them from our following plots and discussion. Moreover, as discussed in Section 1 and shown in Figure 1, the formats that do not target the memory bandwidth bottleneck have very similar performance characteristics for big matrices. Therefore we do not include detailed results from baseline/vectorized CSR, CSR5 and

Table 2: Matrix suite used for the evaluation. We include the matrix symmetry (G/S), the memory footprint of the CSR format (Size) and the percentage of the unique values (UV). In CF we show the compression fraction of DIV_RF (packet size of 16384 values), defined as the ratio of the compressed to the original total matrix size (e.g., CSR in single-precision would have a CF of about 66%). Lastly, we compare DIV_RF (all matrices) and DIV_RF_SYM (symmetric matrices only) versus the best other competitor double-precision format (including symmetric formats) on the AMD EPYC 64-core CPU.

	Size	UV	CF	Best Competitor		DIV variant	
Matrix	(GB)	(%)	(%)	Format	Gflops	_RF	_SYM
(G) spal_004	0.5	6	39.6	SparseX	64.7	37.3	
(S) ldoor	0.5	47	76.9	LCM	51.7	62.7	71.7
(S) dielFilterV2real	0.5	51	83.1	CSR sym	46.1	49.9	57.6
(S) af_shell10	0.6	21	64.3	CFS	160.6	75.4	77.3
(G) nv2	0.6	26	67.3	MKL-64	33.8	43.9	
(S) boneS10	0.6	7e-5	25.2	Dict	163.6	150.0	118.3
(G) circuit5M	0.7	7e-2	30.1	Dict	41.7	64.6	
(S) Hook_1498	0.7	19	58.4	CFS	76.2	80.6	77.9
(S) Geo_1438	0.7	46	62.3	CFS	74.8	67.9	74.8
(S) Serena	0.7	51	65.1	CFS	58.3	63.9	73.0
(G) vas_stokes_2M	0.7	2	41.4	Dict	40.6	91.6	
(S) bone010	0.8	5e-5	25.2	Dict	154.7	148.0	122.0
(S) audikw_1	0.9	48	72.0	CSR sym	45.6	48.7	54.9
(S) Long_Coup_dt0	1.0	41	59.5	CFS	63.4	64.1	67.6
(S) Long_Coup_dt6	1.0	41	59.5	CFS	62.6	64.3	72.0
(S) dielFilterV3real	1.0	51	84.4	CSR sym	44.7	38.3	41.8
(S) nlpkkt120	1.1	2e-3	33.1	Dict	63.6	113.7	113.6
(G) cage15	1.1	5e-4	33.8	Dict	49.0	83.0	
(G) ML_Geer	1.2	99	69.6	SparseX	39.6	45.4	
(S) Flan_1565	1.3	6e-1	30.4	CFS	67.4	122.7	103.7
(S) Cube_Coup_dt0	1.4	15	42.6	CFS	59.4	81.4	82.2
(S) Cube_Coup_dt6	1.4	15	42.7	CFS	58.7	79.9	81.8
(S) Bump_2911	1.4	51	57.6	CFS	58.4	56.7	70.8
(G) vas_stokes_4M	1.5	4	44.9	MKL-64	25.9	56.6	
(S) nlpkkt160	2.6	2e-3	33.1	Dict	47.7	78.1	78.8
(G) HV15R	3.2	84	83.7	SparseX	35.2	35.0	
(S) Queen_4147	3.7	48	62.9	CSR sym	55.0	43.6	64.9
(G) stokes	3.9	3	46.9	MKL-64	24.1	40.4	
(S) nlpkkt200	5.1	1e-3	33.1	Dict	46.3	65.3	65.1
Harmonic Mean				**(all matrices):**	**51.0**	**61.1**	
				(symmetric only matrices):	**62.2**	**68.1**	**73.5**

Merge in the following study, and we point the reader to Figure 1 for a summary of their results. For the Intel platform, we use both the 'Dictionary' formats, but for the AMD platform we only consider the custom implementation, 'Dictionary(Custom)', since lack of support for the AVX512 instruction set prevents us from testing CSR&RV. For the MKL format we utilize the 'mkl_sparse_optimize' library function that optimizes the kernel before the SpMV iterations.

We observe non-negligible CPU performance fluctuations when the runtime length is too small. Therefore, in each configuration (testbed/matrix/format) we run at least 256 SpMV iterations, more if the runtime length is too small, and record the median performance in GFLOPs.

5.2 Performance Evaluation of DIV

In Table 2 (rightmost four columns) and Figure 5 we present per matrix and aggregate performance results respectively for DIV. The Figure 5a and 5b include the entire dataset of Table 2 for the Intel and AMD platforms, while Figure 5c and 5d include only the symmetric matrices and showcase the performance of the symmetric formats for these platforms. The speedups reported are calculated as the *geometric mean* across the matrices of each case. We also note that we report results for the maximum number of cores per platform, as this was found to be the best performing configuration for all formats (see also Section 6.1).

On the Intel platform, the DIV variants are the best formats in 28 out of 29 matrices in the general case and 16 out of 20 in the symmetric. DIV achieves 51% and DIV_RF 77% speedup over MKL. In the symmetric only case, DIV achieves 56%, DIV_RF 83% and DIV_RF_SYM 89% speedup versus MKL. Compared to CFS, a research format targeting symmetric matrices, DIV_RF_SYM achieves 24% speedup. On the AMD platform, the DIV variants have the highest performance in 25 out of 29 matrices in the general case and 17 out of 20 in the symmetric. DIV achieves 91% and DIV_RF 115% speedup over MKL 64-bits, SparseX achieves 7.7%, and MKL 32-bits gains 76% speedup. In the symmetric only case, DIV achieves 102%, DIV_RF 132% and DIV_RF_SYM 143% speedup versus MKL. Compared to CFS, DIV_RF_SYM achieves 47% speedup.

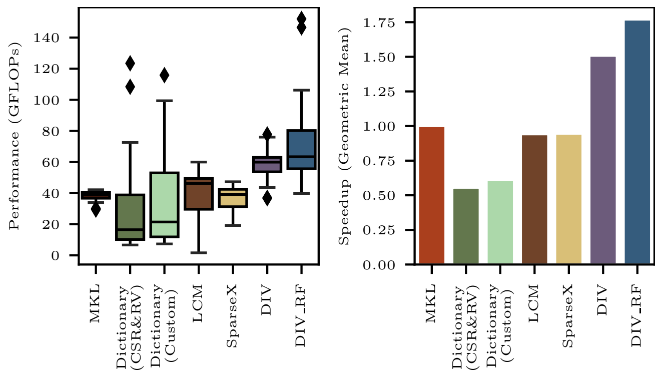
5.3 Correlation Between Performance and Unique Values

We conduct an analysis of the performance based on grouping the matrices by their unique values fractions on the Intel and AMD platforms, and present the results in Figure 6. Firstly, we observe that the index compression formats (MKL, LCM, SparseX) are indeed unaffected by the variance in unique value percentages. We also notice that, as it is expected, the dictionary methods collapse when the number of unique values rises. This is because the value indices for the lookup table need to be able to represent a wider range of numbers, i.e., more bytes for each index, bigger tables and increased random access to them. This becomes more common as the matrix sizes increase, since the same fraction of unique values actually contains a bigger number of them. Our method on the other hand, is able to reliably outperform the MKL 64-bit implementation on average across all ranges of unique values fractions, and on both platforms, since it is able to take advantage of the similarity between the values and not just their replication.

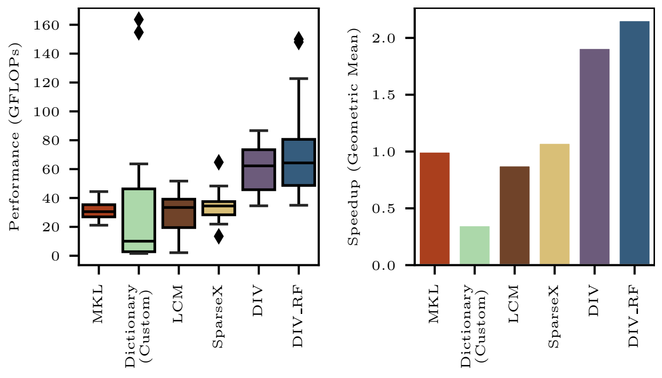
5.4 Application to Biconjugate Gradient Stabilized

In this section we study an end-to-end application of the DIV matrix format and SpMV kernel in the Biconjugate Gradient Stabilized (BiCGSTAB) solver. The purpose of this experiment is to assess the preprocessing overhead of the compression stage of DIV and also to illustrate that the format can effectively be integrated into a real application, delivering notable performance improvements.

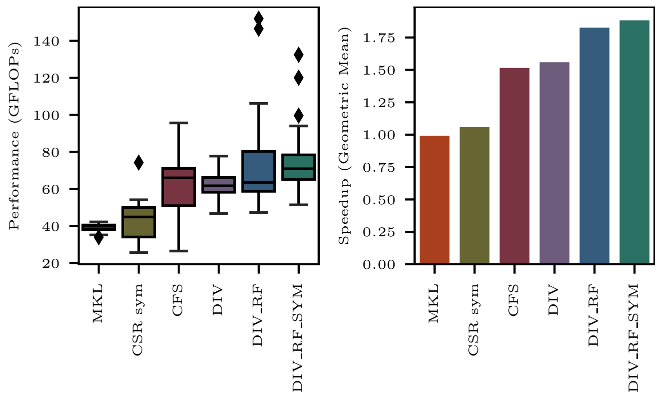
We tested a basic textbook BiCGSTAB implementation with a simple Jacobi preconditioner and present the results in Figure 7. The matrix dataset is the same except for spal_004, which is not a square matrix. We run the solver for a fixed number of iterations: 10, 20, 30, 40, 50, 100, 500 and 1000. As shown in Figure 7b, DIV_RF achieves an end-to-end speedup (i.e., including the compression time for the DIV_RF format) of −27%, −5%, 9%, 18%, 25%, 43%, 63% and 66% respectively (geometric mean) versus MKL 64-bits. The compression



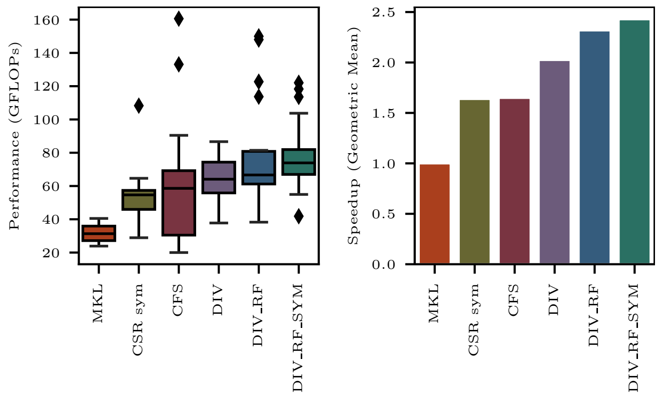
(a) Intel Xeon (56 cores) - all matrices



(b) AMD EPYC (64 cores) - all matrices

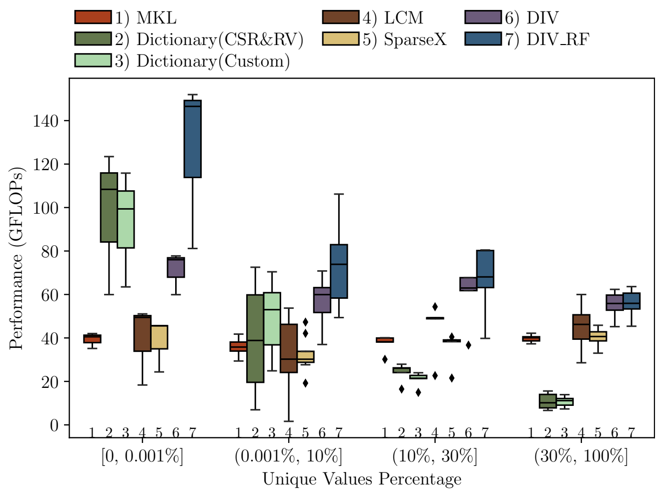


(c) Intel Xeon (56 cores) - symmetric matrices only

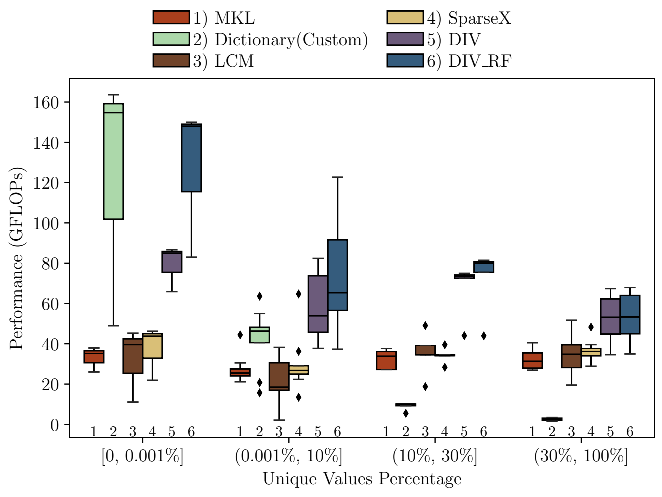


(d) AMD EPYC (64 cores) - symmetric matrices only

Figure 5: Performance evaluation of DIV / DIV_RF / DIV_RF_SYM on an Intel (56 cores) and an AMD (64 cores) CPU.



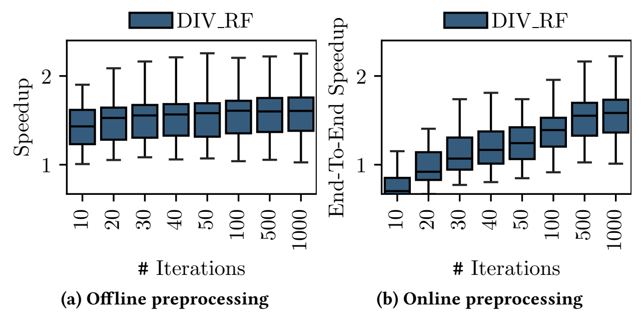
(a) Intel Xeon (56 cores)



(b) AMD EPYC (64 cores)

Figure 6: Performance evaluation of DIV / DIV_RF, grouped by unique values fractions.

time can be amortized in about 20 BiCGSTAB iterations. In terms of equivalent BiCGSTAB iterations with the MKL kernel, the DIV_RF compression time is equal to about 9. This indicates that there are indeed opportunities for online preprocessing.



(a) Offline preprocessing

(b) Online preprocessing

Figure 7: Speedup of BiCGSTAB with DIV_RF versus BiCGSTAB with MKL 64-bits assuming (a) offline compression and (b) online compression.

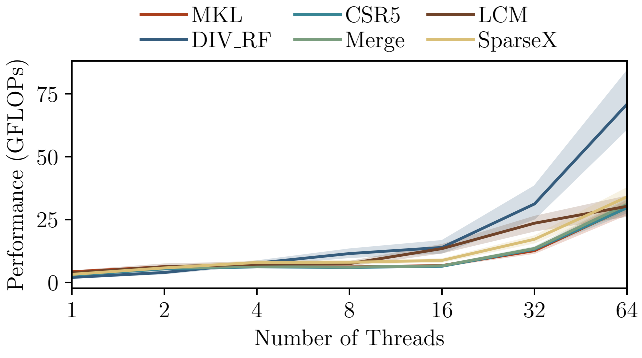


Figure 8: Performance scaling on the AMD EPYC (64 cores).

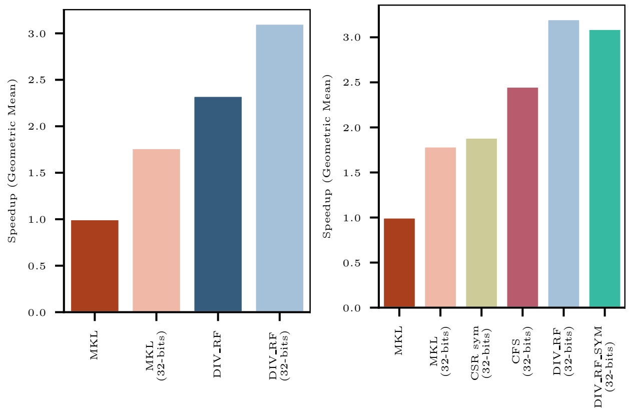
6 Further Topics

6.1 Scalability

In Figure 8 we demonstrate the scaling behavior of DIV and various other formats on the AMD platform with the same matrix dataset (strong scaling). We observe that, due to the memory bandwidth bottleneck, the performance mostly scales with the number of memory nodes available: 1 for cores 1 to 16, 2 for 32 cores and 4 for 64 cores. Likewise, we see that the formats that employ some type of compression (i.e., LCM, SparseX and DIV_RF) are able to scale better. We also note from our measurements that DIV_RF is the only one that scales close to linearly for 1 to 8 cores, showcasing the effect of the extra computational load of decompression, until it too becomes memory-bound.

6.2 Single-precision

Although the main motivation behind DIV was the double-precision SpMV case, in this section we show that our format continues to be effective in lower precision. Specifically, in Figure 9 we present single-precision performance results on the AMD platform. To retain memory boundedness, we use a smaller dataset of the 18 largest matrices (13 for the symmetric case) that remain larger than 2× the LLC with the lower precision values. Note that we were not able to compile LCM and SparseX for single-precision values.



(a) All matrices

(b) Symmetric matrices only

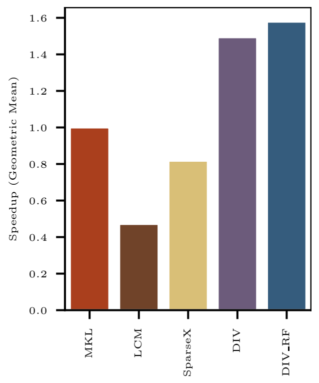
Figure 9: Single-precision (formats noted as 32-bits) performance results on the AMD EPYC (64 cores).

In the general case (Figure 9a), our format DIV_RF 32-bits gains 76% speedup over MKL 32-bits. It is also noteworthy that our double-precision DIV_RF method surpasses the performance of the MKL 32-bits variant, while not being burdened by lower accuracy. In the symmetric only case (Figure 9b) we observe that DIV_RF performs better than the symmetric variant, DIV_RF_SYM. This indicates that, due to the smaller value sizes, the problem has become less memory bound, and the simpler kernel might be a better solution for some of these matrices. DIV_RF 32-bits achieves 79% speedup versus MKL 32-bits and 31% versus the CFS 32-bits format.

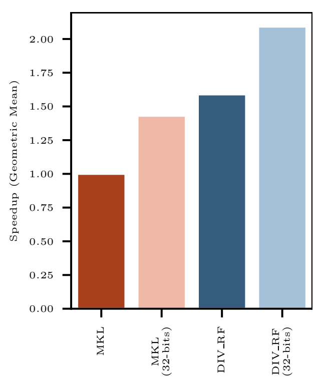
6.3 Extended Matrix Dataset

Large matrices are rather scarce in public datasets, while the CPU caches are steadily increasing in size. From the 1718 matrices with real values in the SuiteSparse sparse matrix collection [3], 951 are smaller that 1MB in double-precision CSR format, while only 340 are larger than 10MB and 108 are larger than 100MB. In this section we expand our matrix dataset by enforcing a memory bandwidth bottleneck on smaller than the CPU cache matrices, which we achieve by clearing the cache after each SpMV iteration. While not a perfect solution, in this way we are able to test DIV with the value distribution of these additional matrices, and also assess performance for a hypothetical system with a smaller cache, or in application scenarios where access to other data would pollute the cache.

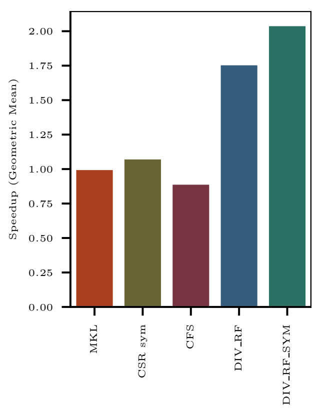
In Figure 10 we present performance results on the AMD platform. The dataset consists of the SuiteSparse real matrices with memory footprint greater than 10MB. Some of them had to be omitted, as they caused either the SparseX or the CFS format to crash at runtime. In the general double-precision case, DIV_RF with 29.3 GFLOPS geomean performance achieves 58% speedup over MKL (248 matrices), while for the symmetric only matrices DIV_RF_SYM with 44.8 GFLOPS achieves 104% (103 matrices). In the general single-precision case, DIV_RF 32-bits with 41 GFLOPS achieves 46% speedup over MKL 32-bits (214 matrices), while for the symmetric only matrices DIV_RF_SYM 32-bits with 61.4 GFLOPS achieves 86% (91 matrices).



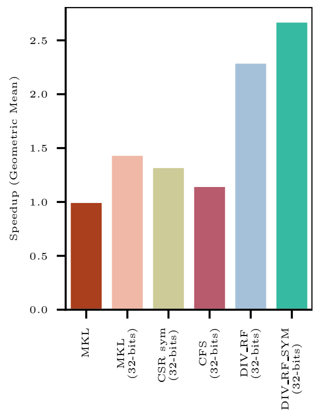
(a) double-precision



(b) single-precision



(c) double-precision - symmetric matrices only



(d) single-precision - symmetric matrices only

Figure 10: Performance speedups for an extended matrix dataset on the AMD EPYC (64 cores).

6.4 GPU Implementation

GPUs offer substantially more computational power and memory bandwidth than CPUs, and have established popularity in many scientific fields. At the same time, studies [31] indicate that they also are heavily memory bound in the SpMV kernel and could therefore potentially benefit from applying index and value compression with DIV. Nevertheless, GPUs have very different architectures from CPUs, with massively more available parallelism, smaller and distributed caches and different synchronization mechanisms. In a straightforward porting scenario of DIV to GPUs, we can assume the GPU thread warp taking the role of the vectorization lanes and the thread block the role of each thread of our CPU method. Since we cannot make the packet size too small to retain compression efficiency, multiple threads (e.g., a thread block) would have to cooperatively process a packet, which would require additional synchronization. Applying the above in a way that fully takes advantage of the GPU capabilities requires non-negligible adaptation of the current DIV implementation, and is the next step in our future work.

7 Related Work

7.1 Floating-point Compression Techniques

Compression of floating-point numbers has a rich research history [35]. Usually though, the main focus is size reduction and I/O acceleration, so the algorithms tend to be more complicated and compute-intensive, which makes them ill-suited for low compute density kernels like SpMV. Engelson et al. [9] explain and provide detailed lossless and lossy variations of delta encoding, the technique also utilized in our work. Ratanaworabhan et al. [32] use a combination of xor-based deltas from recent values and table lookups using them as hashes. FPC [1] compresses floating-point data utilizing xor-based deltas of each value from predicted values, and leading-zero compressing the result. Lindstrom et al. in [26] use a similar strategy with deltas based on integer subtraction from predicted values, and passing the residuals to entropy encoders. In their work on ZFP [25] they present a drastic compression lossy scheme, where 3D data are converted to fixed-point, passed through a 3D orthogonal block transform with a custom provided kernel to decorrelate the values, and then compressed.

7.2 SpMV Data Compression

The implementations of compression schemes on SpMV data are largely grouped into two categories. The most straightforward is the use of mixed precision floating-point values, and more generally truncation or segmentation of the numbers' bits. Liu et al. [27] propose converting values within the $[-1, 1]$ interval to single precision when the individual error is within a given tolerance. Hu et al. [16] combine delta encoding of the exponential part of the double and truncation of the mantissa to the desired number of bits. Grützmacher et al. [14, 15] go a step further and propose segmenting the mantissa of the values, and storing them such that the same segments of all values are consecutive in memory. Using this modular precision scheme they give the option of selecting the desired accuracy by keeping or ignoring trailing segments.

Another popular method is the dictionary based value compression. The main idea is finding the unique values of a matrix and then replacing all the values with integer indices to a lookup table holding the unique only values. Kourtis et al. [20] apply this principle along with compression of the nonzero indices with delta encoding. Yoshifuji et al. [39] adds a variant of delta compression of the value indices to the lookup table and some other very matrix-specific optimizations. Yan et al. [38] is a more vectorization-friendly implementation of the dictionary based compression. Another approach is proposed by Ruiter et al. (VCSC)[33]. They identify the per-column unique values, sort each column according to the nonzero values and use a run-length encoding of the values, instead of value indices to a lookup table. This could alleviate the increased memory footprint of the dictionary indices when there are many unique values, but still relies on the fraction of unique values being small, especially since it needs to replicate unique values across different columns. Unfortunately we could not include this format in our evaluation, as they only provide a serial implementation, most likely a result of using the CSC format as a base.

8 Conclusion

Compression is an obvious means to evercome the memory wall, but for low compute intensity kernels like SpMV, there are practical limitations to the applicable compression methods. In our work we describe the DIV sparse matrix format, a combined index and value compression method for large sparse matrices with double-precision values, designed for accelerating the SpMV kernel on CPUs. It utilizes delta encoding and effectively reduces the matrix memory footprint.

We evaluate our method on an Intel Platinum platform with 56 cores and an AMD EPYC platform with 64 cores, and demonstrate that it achieves 77% and 115% geometric mean speedup respectively, compared to the state-of-practice Inspector-Executor double-precision CSR of the Intel MKL library. We show that our method can achieve performance gains even with reduced precision inputs, and present an end-to-end study when applied to the Biconjugate Gradient Stabilized solver, where we achieve notable end-to-end speedups versus MKL, even while including the preprocessing overhead.

Acknowledgments

This work has been supported by the the European High Performance Computing Joint Undertaking (EuroHPC) Grant DComEX (956201-H2020-JTI-EuroHPC-2019-1), and by the GSRI action "Funding of R&I projects in the frame of the R&I program of the Joint Undertaking for the European High Performance Computing (EuroHPC JU)".

References

[1] Martin Burtscher and Paruj Ratanaworabhan. 2009. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Trans. Comput.* 58, 1 (2009), 18–31. doi:10.1109/TC.2008.131

[2] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. 2022. Vectorizing sparse matrix computations with partially-strided codelets. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 32, 15 pages. https://api.semanticscholar.org/CorpusID:245007162

[3] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[4] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and N. Sun. 2022. AlphaSparse: Generating High Performance SpMV Codes Directly from Sparse Matrices. *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis* (2022), 1–15. https://api.semanticscholar.org/CorpusID:254877267

[5] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2017. Performance analysis and optimization of sparse matrix-vector multiplication on modern multi-and many-core processors. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 292–301.

[6] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2019. Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 48, 15 pages. doi:10.1145/3295500.3356148

[7] Athena Elafrou, Georgios I. Goumas, and Nectarios Koziris. 2019. BASMAT: bottleneck-aware sparse matrix-vector multiplication auto-tuning on GPGPUs. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (2019). https://api.semanticscholar.org/CorpusID:59618194

[8] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2018. SparseX: A library for high-performance sparse matrix-vector multiplication on multicore platforms. *ACM Transactions on Mathematical Software (TOMS)* 44, 3 (2018), 1–32.

[9] V. Engelson, D. Fritzson, and P. Fritzson. 2000. Lossless compression of high-volume numerical data from simulations. In *Proceedings DCC 2000. Data Compression Conference*. 574–. doi:10.1109/DCC.2000.838221

[10] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)* 43, 4 (2017), 1–49.

[11] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N Koziris. 2009. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *The Journal of Supercomputing* 50 (2009), 36–77.

[12] Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. 2008. Understanding the performance of sparse matrix-vector multiplication. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. IEEE, 283–292.

[13] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. 2015. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 64–67. doi:10.1109/FCCM.2015.30

[14] Thomas Grützmacher and Hartwig Anzt. 2018. A Modular Precision Format for Decoupling Arithmetic Format and Storage Format. In *Euro-Par Workshops*. https://api.semanticscholar.org/CorpusID:57662952

[15] Thomas Grützmacher, Terry Cojean, Goran Flegar, Fritz Göbel, and Hartwig Anzt. 2019. A customized precision format based on mantissa segmentation for accelerating sparse linear algebra. *Concurrency and Computation: Practice and Experience* 32 (2019). https://api.semanticscholar.org/CorpusID:198342465

[16] Siyi Hu, Makiko Ito, Takahide Yoshikawa, Yuan He, and Masaaki Kondo. 2022. Memory Bandwidth Conservation for SpMV Kernels Through Adaptive Lossy Data Compression. In *International Conference on Parallel and Distributed Computing: Applications and Technologies*. https://api.semanticscholar.org/CorpusID:258440447

[17] IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (2008), 1–70. doi:10.1109/IEEESTD.2008.4610935

[18] Humayun Kabir, Joshua Dennis Booth, and Padma Raghavan. 2014. A multilevel compressed sparse row format for efficient sparse computations on multicore processors. *2014 21st International Conference on High Performance Computing (HiPC)* (2014), 1–10. https://api.semanticscholar.org/CorpusID:11966175

[19] William Kahan. 1965. Pracniques: further remarks on reducing truncation errors. *Commun. ACM* 8, 1 (1965), 40.

[20] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. 2008. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *Proceedings of the 5th Conference on Computing Frontiers* (Ischia, Italy) *(CF '08)*. Association for Computing Machinery, New York, NY, USA, 87–96. doi:10.1145/1366230.1366244

[21] Kornilios Kourtis, Vasileios P. Karakasis, Georgios I. Goumas, and Nectarios Koziris. 2011. CSX: an extended compression format for spmv on shared memory systems. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. https://api.semanticscholar.org/CorpusID:10688229

[22] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.

[23] Chenyang Li, Tian Xia, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2021. SpV8: Pursuing Optimal Vectorization and Regular Computation Pattern in SpMV. *2021 58th ACM/IEEE Design Automation Conference (DAC)* (2021), 661–666. https://api.semanticscholar.org/CorpusID:243876169

[24] Yishui Li, Peizhen Xie, Xinhai Chen, J. Liu, Bo Yang, Shengguo Li, Chunye Gong, Xinbiao Gan, and Han Xu. 2019. VBSF: a new storage format for SIMD sparse matrix–vector multiplication on modern processors. *The Journal of Supercomputing* 76 (2019), 2063–2081. https://api.semanticscholar.org/CorpusID:106410121

[25] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683. doi:10.1109/TVCG.2014.2346458

[26] Peter Lindstrom and Martin Isenburg. 2006. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 1245–1250. doi:10.1109/TVCG.2006.143

[27] J. Liu. 2022. Accuracy Controllable SpMV Optimization on GPU. *Journal of Physics: Conference Series* 2363 (2022). https://api.semanticscholar.org/CorpusID:253417997

[28] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[29] John D McCalpin et al. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2, 19-25 (1995).

[30] Duane Merrill and Michael Garland. 2016. Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 678–689.

[31] Panagiotis Mpakos, Dimitrios Galanopoulos, Petros Anastasiadis, Nikela Papadopoulou, Nectarios Koziris, and Georgios Goumas. 2023. Feature-based SpMV Performance Analysis on Contemporary Devices. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 668–679. doi:10.1109/IPDPS54959.2023.00072

[32] P. Ratanaworabhan, Jian Ke, and M. Burtscher. 2006. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*. 133–142. doi:10.1109/DCC.2006.35

[33] Skyler Ruiter, Seth Wolfgang, Marc Tunnell, Timothy Triche, Erin Carrier, and Zachary J. DeBruine. 2023. Value-Compressed Sparse Column (VCSC): Sparse Matrix Storage for Redundant Data. *ArXiv* abs/2309.04355 (2023). https://api.semanticscholar.org/CorpusID:261660589

[34] Prabhjot Sandhu, Clark Verbrugge, and Laurie J. Hendren. 2020. A Fully Structure-Driven Performance Analysis of Sparse Matrix-Vector Multiplication. *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (2020). https://api.semanticscholar.org/CorpusID:215752935

[35] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei keng Liao, and Alok N. Choudhary. 2014. Data Compression for the Exascale Computing Era - Survey. *Supercomput. Front. Innov.* 1 (2014), 76–88. https://api.semanticscholar.org/CorpusID:3527510

[36] Zhaonian Tan, Weixing Ji, Jianhua Gao, Yueyan Zhao, Akrem Benatia, Yizhuo Wang, and Feng Shi. 2020. MMSparse: 2D partitioning of sparse matrix based on mathematical morphology. *Future Gener. Comput. Syst.* 108 (2020), 521–532. https://api.semanticscholar.org/CorpusID:216022538

[37] Chenggang Clarence Yan, Hui Yu, Weizhi Xu, Yingping Zhang, Bochuan Chen, Zhu Tian, Yuxuan Wang, and Jian Yin. 2015. Memory bandwidth optimization of SpMV on GPGPUs. *Frontiers of Computer Science* 9 (2015), 431–441. https://api.semanticscholar.org/CorpusID:3404873

[38] Junjun Yan, Xinhai Chen, and Jie Liu. 2022. CSR&RV: An Efficient Value Compression Format for Sparse Matrix-Vector Multiplication. In *Network and Parallel Computing*, Shaoshan Liu and Xiaohui Wei (Eds.). Springer Nature Switzerland, Cham, 54–60. doi:10.1007/978-3-031-21395-3_5

[39] Naoki Yoshifuji, Ryo Sakamoto, Keigo Nitadori, and Junichiro Makino. 2016. Implementation and Evaluation of Data-Compression Algorithms for Irregular-Grid Iterative Methods on the PEZY-SC Processor. *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)* (2016), 58–61. https://api.semanticscholar.org/CorpusID:16110604