Proceedings of the Federated Conference on Computer Science and Information Systems pp. 513–521
DOI: 10.15439/2017F70
ISSN 2300-5963 ACSIS, Vol. 11

On Memory Footprints of Partitioned Sparse Matrices

Daniel Langr*† and Ivan Šimeček*
* Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems
Thákurova 9, 160 00, Praha, Czech Republic
Email: {langrd,xsimecek}@fit.cvut.cz
† Výzkumný a zkušební letecký ústav, a.s.
Beranových 130, 199 05, Praha, Czech Republic

***Abstract*—The presented study analyses 563 representative benchmark sparse matrices with respect to their partitioning into uniformly-sized blocks. The aim is to minimize memory footprints of matrices. Different block sizes and different ways of storing blocks in memory are considered and statistically evaluated. Memory footprints of partitioned matrices are additionally compared with lower bounds and the CSR storage format. The average measured memory savings against CSR in case of single and double precision are 42.3 and 28.7 percents, respectively. The corresponding worst-case savings are 25.5 and 17.1 percents. Moreover, memory footprints of partitioned matrices were in average 5 times closer to their lower bounds than CSR. Based on the obtained results, we provide generic suggestions for efficient partitioning and storage of sparse matrices in a computer memory.**

I. Introduction

The way how sparse matrices are stored in a computer memory may have a significant impact on the required memory space, i.e., on the matrix memory footprints. Reduction of matrix memory footprints may positively influence related computations and executions of corresponding programs. For example:

- Lower matrix memory footprints yield faster processing of matrices by I/O subsystems, e.g., when checkpointing-restart resilience methods are applied within high performance computing (HPC) applications [1], [2].
- Lower matrix memory footprints may increase the efficiency and performance of sparse matrix computations if these are bounded by memory bandwidth. This is, e.g., often the case of sparse matrix vector multiplication (SpMV)[1].
- Lower matrix memory footprints allow larger matrices to fit in the available amount of memory, which, therefore, allows to solve computational problems to higher extent or with higher accuracy.

This work was supported by the Czech Science Foundation under grant no. 16-16772S and by Czech Technical University in Prague under grant SGS17/215/OHK3/3T/18.

[1]Memory bandwidth is not the only bound for SpMV performance; there are others as well [3]. However, in cases where the memory bandwidth is the main bottleneck, by reducing memory footprints of matrices one can reduce the overall timings of SpMV applications such as iterative solvers.

One way of reducing memory footprints of sparse matrices is their partitioning into blocks (which also promotes spatial locality during computations). Much has been written about block processing of sparse matrices, frequently in the context of memory-bounded character of SpMV [4]–[26]. In this article, we address the problem of minimizing memory footprints of sparse matrices by their partitioning into uniformly-sized blocks. Its solution raises two essential questions: How to choose a suitable block size? And, how to store resulting nonzero blocks in a computer memory? These questions form a multi-dimensional optimization problem that needs to be solved prior to the partitioning itself. We refer to both these problems—optimization and partitioning—as (*block*) *preprocessing*.

The above introduced optimization problem raises another question: How to specify the optimization space, i.e., the space of tested configurations? Intuitively, the larger the optimization space is, the lower matrix memory footprint can be found, however, at a price of longer preprocessing runtime. To amortize block processing of a sparse matrix, the optimization space thus need to be chosen wisely in a form of a trade-off: we want it to be small enough to ensure its fast exploration but also large enough to contain the optimal or nearly-optimal configuration generally for any sparse matrix.

We present a study that analyses memory footprints of 563 representative sparse matrices from the University of Florida Sparse Matrix Collection (UFSMC) [27] with respect to their partitioning into uniformly sized blocks. These matrices arose from a large variety of applications of multiple problem types and thus have highly diverse structural and numerical properties. Our goal is to minimize memory footprints of matrices and we consider an optimization space that consists of different block sizes and different ways of storing blocks in memory. Based on the obtained results, we finally provide suggestions for both efficient and effective block preprocessing of sparse matrices in general.

II. Methodology

In Section I, we referred to the *matrix memory footprint* as to the amount of memory space required to store a given

matrix in a computer memory. More precisely, we can define it as a number of bits (or bytes) which is needed to store the values of nonzero elements of a given matrix together with the information about their structure, i.e., their row and column positions.

A. Sparse Matrix Storage Formats

The ways how sparse matrices are stored in a computer memory are generally called *sparse matrix storage formats*; we call them *formats* only if the context is clear. Matrix memory footprint is thus a function of a given matrix and format (memory footprints for the same matrix but distinct formats may differ considerably).

In case of partitioned sparse matrices, their nonzero blocks represent individual submatrices that can be treated separately. In practice, well-proven formats used for nonzero blocks of sparse matrices are:

- The *coordinate* (COO) format, which stores values of block nonzero elements together with their row and column indices [7], [17], [21].
- The *compressed sparse row* (CSR) format, which stores values and column indices of lexicographically ordered block nonzero elements together with the information about which values / column indices belongs to which block row [17], [19]–[21].
- The *bitmap* format, which stores values of block nonzero elements in some prescribed order and encodes their row and column indices in a bit array [8], [15], [17].
- The *dense* format, which stores values of both nonzero and zero block elements in a dense array (row and column indices of nonzero elements are thus effectively determined by positions of their values within this array) [13], [14], [17], [28].

B. Blocking Storage Schemes

Considering these formats, we have 6 options how to store nonzero blocks of a sparse matrix in memory:

1) store all the blocks in the COO format,
2) store all the blocks in the CSR format,
3) store all the blocks in the bitmap format,
4) store all the blocks in the dense format,
5) store *all the blocks* in a format that minimizes the memory footprint of a given matrix (we refer to this option as *min-fixed*),
6) store *each block* in a format that minimizes the contribution of this block to the memory footprint of a given matrix (we refer to this option as *adaptive*).

We call these options *blocking storage schemes*, or shortly *schemes* only. Since the first 4 schemes prescribe a fixed format for all the blocks, we call them *fixed-format schemes*.

For the min-fixed and adaptive schemes, we consider formats for nonzero blocks to be chosen from COO, CSR, bitmap, and dense. In case of the min-fixed scheme, the matrix memory footprint thus contains 2 additional bits for storing the information about the format used for all the nonzero blocks. In case of the adaptive scheme, the matrix memory footprint contains 2 additional bits for each nonzero block to store the information about its format.

C. Block Sizes

To evaluate memory footprints of a given matrix for different schemes and some particular tested block size, we need information about numbers of nonzero elements of all nonzero blocks [17]. In the end, this information must be obtained for each distinct block size from the optimization space, which represents the most demanding part of the whole optimization process [29]. The block preprocessing runtime is thus approximately proportional to the number of distinct tested block sizes. Consequently, the lower is their count, the higher are the chances that the partitioning will be profitable at all.

Generally, there is $O(m \times n)$ ways how to choose a block size for an $m \times n$ matrix, but for fast block preprocessing, we need to consider only few of them.[2] One possible approach is to consider only block sizes

$$2^k \times 2^\ell, \quad \text{where} \quad 1 \leq k \leq K \quad \text{and} \quad 1 \leq \ell \leq L, \qquad (1)$$

which reduces the number of tested block sizes to $K \times L$. Such a choice, among others, results in substantially faster preprocessing in general [29]. Within the presented study, we consider block sizes (1) and set $K = L = 8$. The choice of these upper bounds stemmed from our auxiliary experiments which showed that space-optimal block sizes have mostly less than 64 rows/columns. Taking into account block sizes with up to 256 rows/columns should cover even the remaining corner cases.

D. Optimization Space

In the summary, our optimization space is initially defined by $\mathcal{S}_6 \times \mathcal{B}_{64}$, where $\mathcal{S}_6$ denotes a set of selected blocking storage schemes:

$$\mathcal{S}_6 = \big\{\text{COO}, \text{CSR}, \text{bitmap}, \text{dense}, \text{min-fixed}, \text{adaptive}\big\}$$

and $\mathcal{B}_{64}$ denotes a set of selected block sizes:

$$\mathcal{B}_{64} = \big\{2^k \times 2^\ell : 1 \leq k, \ell \leq 8\big\}.$$

E. Additional Considerations

When measuring matrix memory footprints, we need to decide how to represent information about nonzero blocks and how to represent indices. In the presented study, we assume that:

1) nonzero blocks are stored in memory in the lexicographical order;
2) block column index for each nonzero block is stored explicitly;
3) the number of nonzero blocks for each block row is stored;

[2] In addition to multiplication and Cartesian product, we also use the multiplication sign "$\times$" to specify matrix/block sizes. In such cases, $m \times n$ does not denote multiplication, but a matrix/block size of height m and width n (i.e., having m rows ans n columns).

TABLE I: Counts of tested matrices falling under particular problem types (referred to as "kinds" in the UFSMC).

Problem	Matrices
2D/3D	36
acoustics	4
chemical process simulation	25
circuit simulationi	41
computational fluid dynamics	47
computer graphics/vision	8
counter-example	2
duplicate model reduction	5
economic	24
eigenvalue/model reduction	2
electromagnetics	11
frequency-domain circuit sim.	4
least squares	7
linear programming	51
materials	15
model reduction	11
optimization	66
power network	35
semiconductor device	16
statistical/mathematical	1
structural	82
theoretical/quantum chemistry	42
thermal	11
weighted graph	17

4) a minimum possible number of bits, i.e., $\lceil \log_2 n \rceil$ bits, is used to store an index related to n entities (such an approach is in the literature sometimes referred to as *index compression*).

F. Benchmark Matrices

Sparse matrices are often divided into two main categories—*high performance computing (HPC) matrices* and *graph matrices*. Efficient processing of graph matrices is generally governed by special rules that are different from those being effective for HPC matrices [9], [30], [31] (e.g., higher matrix memory footprints in some cases lead to higher performance of computations and graph matrices are also typically not suitable for simple block processing mainly due to emergence of hypersparse blocks [8], [9]). Within this work, we focused mainly (but not exclusively) on HPC matrices. Namely, we considered all real matrices from the UFSMC that contained more than 10^5 nonzero elements and had a unique structure of nonzero elements. This way, we obtained 563 sparse matrices arising from different application problems (see Table I) and thus having different structural (and numerical) properties; we denote these matrices by $A_1, \ldots, A_{563}$. Of these matrices, 281 were square symmetric and the remaining 282 were either rectangular or square unsymmetric.

G. Matrix Memory Footprint

For symmetric matrices, we always assume storage only of their single triangular parts in memory, which is a common practice. Referring to the *number of nonzero elements* of a matrix, we thus generally need to distinguish between the number of *all* nonzero elements and the number of elements that are assumed to be *stored* in a computer memory. While measuring memory footprints of sparse matrices, we take into account the latter one.

According to the text above, a matrix memory footprint for a sparse matrix A_k partitioned into uniformly-sized blocks is a function of the following parameters:

1) sparse matrix A_k,
2) block storage scheme $s \in \mathcal{S}_6$,
3) block size $h \times w \in \mathcal{B}_{64}$,
4) number of bits b required to store a value of a single matrix nonzero element.

We denote this function by $\mathrm{MMF}_{\boxplus}(A_k, s, w{\times}h, b)$. We further assume storing values of matrix nonzero elements in either single or double precision IEEE floating-point format [32], which implies $b = 32$ or $b = 64$, respectively, in case of real matrices. We refer to such a floating-point precision as *precision* only.

We say that a matrix memory footprint for a given matrix A and a given precision determined by b is *optimal* (with respect to our work) if it equals

$$\min\bigl\{\mathrm{MMF}_{\boxplus}(A, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\bigr\}.$$

We call the corresponding blocking storage scheme and block size optimal as well.

Let $\mathcal{S} \subseteq \mathcal{S}_6$ and $\mathcal{B} \subseteq \mathcal{B}_{64}$. $\mathcal{S} \times \mathcal{B}$ thus define a subspace of the optimization space $\mathcal{S}_6 \times \mathcal{B}_{64}$. Let

$$\Delta^b_{\mathcal{S},\mathcal{B}}(k) = \Bigl(\min\bigl\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}, h \times w \in \mathcal{B}\bigr\} / \min\bigl\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\bigr\} - 1\Bigr) \times 100.$$

This function expresses of how much percent is the minimal memory footprint of A_k from $\mathcal{S} \times \mathcal{B}$ higher (worse) than its optimal memory footprint. To assess the subspace $\mathcal{S} \times \mathcal{B}$, we define the following parametrized set

$$\mathcal{U}^b_{\mathcal{S},\mathcal{B}} = \bigl\{\Delta^b_{\mathcal{S},\mathcal{B}}(k) : 1 \le k \le 563\bigr\}.$$

The minimum, mean (average; μ), and maximum of $\mathcal{U}^b_{\mathcal{S},\mathcal{B}}$ then reflect the best, average, and worst cases, respectively, for $\mathcal{S} \times \mathcal{B}$ across the tested matrices.

If $\mathcal{S}$ or $\mathcal{B}$ consists of a single element only, we omit the curly braces in the subscript of $\mathcal{U}$ for the sake of readability; e.g., we write $\mathcal{U}^b_{s,\mathcal{B}_{64}}$ and $\mathcal{U}^b_{\mathcal{S}_6,h\times w}$ instead of $\mathcal{U}^b_{\{s\},\mathcal{B}_{64}}$ and $\mathcal{U}^b_{\mathcal{S}_6,\{h\times w\}}$.

III. Results and Discussion

A. Blocking Storage Schemes

First, we assessed blocking storage schemes. Table II shows for how many tested matrices were individual schemes optimal. The adaptive scheme clearly dominates this evaluation metric; it was optimal for 464 tested matrices, which corresponds to $82.4\,\%$ of their total count. Note that the min-fixed scheme was never optimal; this is due to the necessity to store additional information about the format used for blocks (if

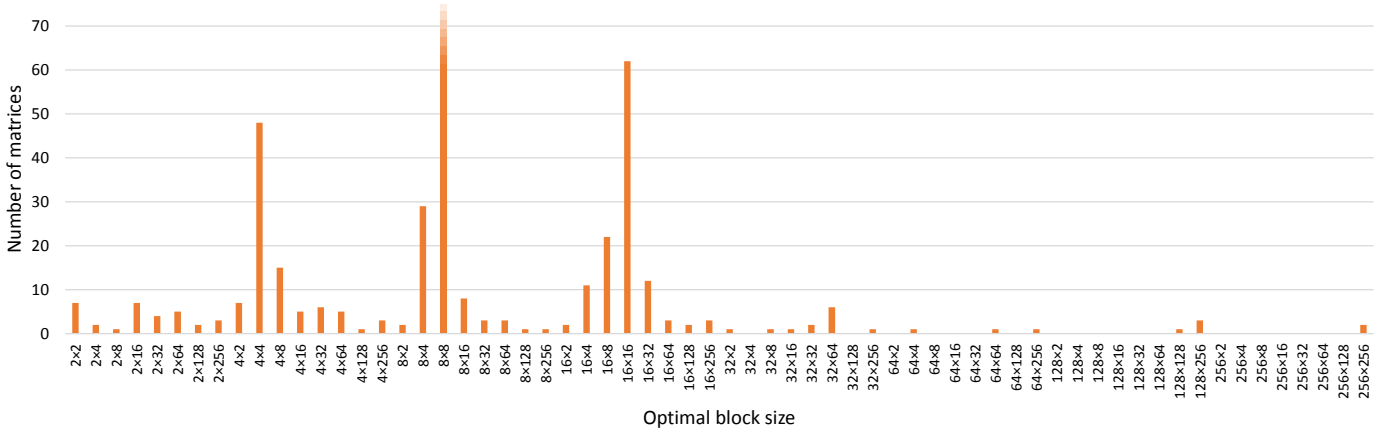

Fig. 1: Numbers of tested matrices for which are block sizes optimal, measured for double precision; block size 8×8 was optimal for 257 matrices.

TABLE II: Counts of tested matrices for which are blocking storage schemes optimal; the numbers are the same for both single and double precision.

Scheme	Matrices
COO	58
CSR	0
bitmap	36
dense	5
min-fixed	0
adaptive	464

we ignored the additional 2 bits required by this scheme, it would be optimal for $58 + 36 + 5 = 99$ matrices). However, the numbers in Table II reflect only best cases, i.e., matrices that were most suitable for particular schemes. To find out how much were particular schemes better than the others in average and for their worst-case (most unsuitable) matrices, we need complete statistics of $\mathcal{U}^b_{s,\mathcal{B}_{64}}$; these are presented in Table III and lead to the following observations:

- No fixed-format scheme minimized matrix memory footprints in comparison with the others. Bitmap was the best in average, however, it was inferior to both COO and CSR in worst cases.
- Dense provided extremely high matrix memory footprints in average and worst cases. Due to the explicit storage of zero elements, this scheme is suitable only for kinds of matrices that contain highly dense blocks; obviously, there were only few such matrices in our tested suite (recall that the dense scheme was optimal for 5 matrices according to Table II).
- The lowest memory footprints were provided by the min-fixed and adaptive schemes; their numbers are considerably lower in comparison with the fixed-format schemes.

B. Block Sizes

Similarly as blocking storage schemes, we assessed block sizes. Fig. 1 shows for how many tested matrices were individual block sizes optimal in case of double precision measurements; for single precision, the results differed only for 2 matrices. We may observe that some block sizes were especially favourable. The 8×8 block size was optimal for 257 matrices, which corresponds to $45.6\,\%$ of their total count. Together with 4×4 and 16×16, these 3 block sizes were optimal for $65.2\,\%$ of tested matrices. However, again, the numbers from Fig. 1 reflect only best cases. To find out how much were particular block sizes better than the others in average and for their worst-cases matrices, we present the average and maximum values of $\mathcal{U}^b_{\mathcal{S}_6,h\times w}$ in Table IV and Table V for single and double precision, respectively. According to these results, some blocks sizes—especially 8×8—provided alone average matrix memory footprints close to their optimal values. However, there was not a single block size that would yield the same outcome for all the tested matrices; the maxima were for all the block sizes relatively high.

Let us remind that one of our goals is a possible reduction of the number of block sizes in the optimization test space. The question thus is whether there is some subset $\mathcal{B} \subset \mathcal{B}_{64}$ that would, at the same time:

1) significantly reduce the number of block sizes ($|\mathcal{B}|$),
2) provide matrix memory footprints close to their optimal values for most of the tested matrices (average of $\mathcal{U}^b_{\mathcal{S}_6,B}$ close to zero),
3) provide low matrix memory footprints for all the tested matrices (low maximum of $\mathcal{U}^b_{\mathcal{S}_6,B}$).

Natural candidates for such a subset would be the first n block sizes from Table IV and Table V; let us denote them by $\mathcal{C}^{64}_n$ and $\mathcal{C}^{32}_n$, respectively. Fig. 2 evaluates these subsets as a function of n. We may notice that

$$\mathcal{C}^{64}_9 = \mathcal{C}^{32}_9 = \big\{\, h \times w : h, w \in \{4, 8, 16\}\big\},$$
$$\mathcal{C}^{64}_{16} = \mathcal{C}^{32}_{16} = \big\{\, h \times w : h, w \in \{4, 8, 16, 32\}\big\};$$

seemingly, block sizes from these subsets are especially suitable for sparse matrices in general.

Despite that, neither these first 9 nor 16 block sizes reduced the maximal matrix memory footprints too much according to

TABLE III: Minimum, average and maximum values of $\mathcal{U}^b_{s,\mathcal{B}_{64}}$ (in percents).

	Single precision ($b = 32$)			Double precision ($b = 64$)		
Scheme (s)	Minimum	Average	Maximum	Minimum	Average	Maximum
COO	0.00	4.78	15.27	0.00	2.52	7.67
CSR	0.73	6.84	19.13	0.41	3.74	11.05
bitmap	0.00	3.13	22.01	0.00	1.75	12.38
dense	0.00	84.61	217.04	0.00	92.40	249.02
min-fixed	0.00	1.19	5.41	0.00	0.64	2.94
adaptive	0.00	0.10	2.24	0.00	0.05	1.30

TABLE IV: Average and maximum values of $\mathcal{U}^{32}_{\mathcal{S}_6,h\times w}$ (in percents), sorted by average.

Rank	$h \times w$	Avg.	Max.	Rank	$h \times w$	Avg.	Max.	Rank	$h \times w$	Avg.	Max.
1	8×8	1.23	18.36	11	16×32	4.03	23.75	21	16×64	5.89	26.15
2	8×16	2.14	19.35	12	32×8	4.13	23.97	22	4×2	6.06	28.77
3	16×8	2.26	21.41	13	4×32	4.36	18.71	23	2×4	6.15	23.07
4	4×8	2.32	17.31	14	32×16	4.53	24.45	24	16×2	6.25	29.98
5	8×4	2.38	19.52	15	32×4	4.87	23.60	25	4×64	6.26	21.53
6	16×16	2.56	21.82	16	32×32	5.20	26.50	26	64×8	6.56	25.83
7	4×4	2.92	21.94	17	2×8	5.59	21.15	...	...	...	...
8	4×16	2.99	16.51	18	8×64	5.61	23.57	62	256×2	14.44	37.33
9	16×4	3.23	20.44	19	8×2	5.66	26.39	63	256×128	14.61	38.32
10	8×32	3.65	21.26	20	2×16	5.84	22.84	64	256×256	14.65	35.42

TABLE V: Average and maximum values of $\mathcal{U}^{64}_{\mathcal{S}_6,h\times w}$ (in percents), sorted by average.

Rank	$h \times w$	Avg.	Max.	Rank	$h \times w$	Avg.	Max.	Rank	$h \times w$	Avg.	Max.
1	8×8	0.69	11.07	11	16×32	2.19	12.84	21	2×16	3.25	13.04
2	8×16	1.18	11.67	12	32×8	2.26	14.45	22	4×2	3.34	15.74
3	16×8	1.25	12.91	13	4×32	2.40	10.56	23	2×4	3.40	12.84
4	4×8	1.30	9.74	14	32×16	2.47	14.04	24	4×64	3.42	11.38
5	8×4	1.33	10.98	15	32×4	2.68	14.23	25	16×2	3.47	15.93
6	16×16	1.40	13.16	16	32×32	2.82	14.18	26	64×8	3.57	15.30
7	4×4	1.63	12.34	17	8×64	3.05	12.62	...	...	...	...
8	4×16	1.66	9.96	18	2×8	3.11	12.08	62	256×2	7.88	21.59
9	16×4	1.79	12.32	19	8×2	3.14	14.02	63	256×128	7.92	19.56
10	8×32	1.99	11.97	20	16×64	3.19	14.00	64	256×256	7.93	18.96

Fig. 2. However, we may observe that there are some block sizes where these maxima significantly dropped. Based on the analysis of the statistics of $\mathcal{U}^b_{\mathcal{S}_6,\mathcal{C}^b_n}$, we propose the following *reduced sets of block sizes*:

$$\mathcal{B}_8 = \big\{2^k \times 2^k : 1 \le k \le 8\big\},$$
$$\mathcal{B}_{14} = \mathcal{B}_8 \cup \big\{2^k \times 2^\ell : 2 \le k, \ell \le 4\big\},$$
$$\mathcal{B}_{20} = \mathcal{B}_8 \cup \big\{2^k \times 2^\ell : 2 \le k, \ell \le 5\big\}.$$

$\mathcal{B}_8$ thus consists of all square block sizes from $\mathcal{B}_{64}$. $\mathcal{B}_{14}$ and $\mathcal{B}_{20}$ equal $\mathcal{B}_8$ plus rectangular block sizes from $\mathcal{C}^{32}_9$ ($\mathcal{C}^{64}_9$) and $\mathcal{C}^{32}_{16}$ ($\mathcal{C}^{64}_{16}$), respectively.

C. Optimization Subspace

Table III revealed that to minimize memory footprints of (all) the tested matrices, we had to use either the min-fixed or the adaptive blocking storage scheme. To reduce the block preprocessing overhead, we now proposed several reduced sets of block sizes. Let us now assess these options together. We measured the statistics of $\mathcal{U}^b_{s,\mathcal{B}_j}$ for all the combinations of $s \in \{\text{min-fixed}, \text{adaptive}\}$ and $j \in \{64, 20, 14, 8\}$; the results are presented in Table VI. The average matrix memory footprints were in all cases close to their optimal values. Moreover, the reduced sets $\mathcal{B}_j$ required much less block sizes than $\mathcal{C}^b_n$ to achieve the same maxima. For instance:

1) $\mathcal{B}_{14}$ in combination with the min-fixed scheme required only 14 block sizes to achieve the same maxima as $\mathcal{C}^b_{43}$ in combination with all the schemes. This would effectively reduce the number of block sizes in the optimization space by a factor of about 3, which would proportionally reduce the preprocessing overhead in practice.
2) $\mathcal{B}_{20}$ in combination with the adaptive scheme required only 20 block sizes to achieve the same maxima $\mathcal{C}^b_{50}$ in combination with all the schemes. This would effectively reduce the number of block sizes by a factor of 2.5.

D. Memory Savings Against CSR32

Likely the most widely-used storage format for sparse matrices in practice is CSR, which is supported by vast majority of software tools and libraries that work with sparse matrices. To distinguish between CSR used for blocks of partitioned matrices and CSR used for whole (not-partitioned) matrices, we call the latter CSR32, since it is typically implemented with 32-bit indices. Researchers frequently demonstrate the

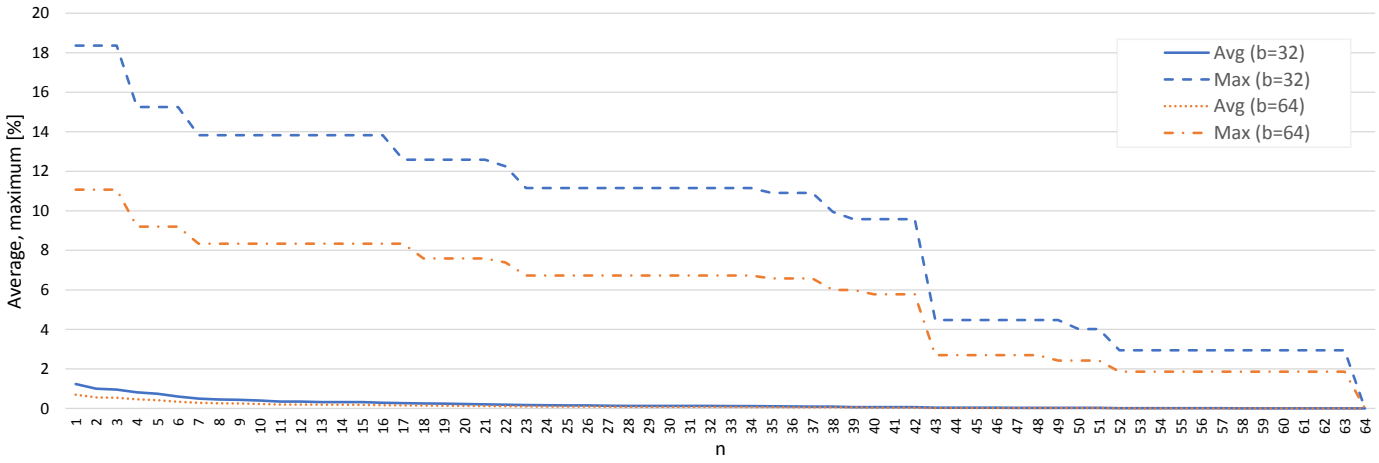

Fig. 2: Average and maximum values $\mathcal{U}^b_{\mathcal{S}_6,\mathcal{C}^b_n}$ (in percents) as a funciton of n.

TABLE VI: Average and maximum values of $\mathcal{U}^b_{s,\mathcal{B}_j}$ (in percents) for $j \in \{64, 20, 14, 8\}$.

(a) Single precision ($b = 32$)

	s = min-fixed		s = adaptive	
Block sizes	Average	Maximum	Average	Maximum
$\mathcal{B}_{64}$	1.19	5.41	0.10	2.24
$\mathcal{B}_{20}$	1.32	6.23	0.22	4.21
$\mathcal{B}_{14}$	1.35	6.89	0.28	6.81
$\mathcal{B}_{8}$	1.51	10.06	0.51	11.07

(b) Double precision ($b = 64$)

	s = min-fixed		s = adaptive	
Block sizes	Average	Maximum	Average	Maximum
$\mathcal{B}_{64}$	0.64	2.94	0.05	1.30
$\mathcal{B}_{20}$	0.71	3.52	0.12	2.37
$\mathcal{B}_{14}$	0.73	3.77	0.16	3.83
$\mathcal{B}_{8}$	0.81	5.34	0.28	5.88

superiority of their algorithms and data structures (formats) by comparison with CSR32, which have become de facto an etalon in sparse-matrix research.

Comparison of memory footprints of sparse matrices partitioned into blocks and the same matrices stored in CSR32 allows us to assess our blocking approach. Let $\mathrm{MMF}_{\mathrm{CSR32}}(A, b)$ denote a memory footprint of a matrix A stored in memory in CSR32 with respect to a precision given by b. The function

$$\Lambda^b(k) = \big(1 - \min\big\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\big\} / \mathrm{MMF}_{\mathrm{CSR32}}(A_k, b)\big) \times 100$$

then expresses how much memory in percents we would save if we stored the tested matrix A_k in its optimal blocking configuration instead of in CSR32. We measured these memory savings for all the tested matrices and processed them statistically; the results are presented by Table VII. The obtained numbers arguments strongly in favour of partitioning of sparse matrices in general. Even in worst cases, our blocking approach reduced the memory footprints of matrices of 25.46 % and 17.08 % for single and double precision, respectively. In average, the savings were 42.29 % and 28.67 %, which significantly reduces the amount of data that needs to be transferred between memory and processors during computations.

TABLE VII: Statistics of $\Lambda^b(k)$, i.e., memory savings of optimal blocking configurations against CSR32 in percents, across the tested matrices.

Statistics	Single precision	Double precision
Minimum	25.46	17.08
Average	42.29	28.67
Maximum	50.21	35.86

E. Memory Footprints Compared with Lower Bounds

Section III-D showed how much memory space we would save if we stored sparse matrices in optimal blocking configurations instead of in CSR32. The last object of our concern within this study was of how much are the memory footprints

of the tested matrices higher than their potential minima, i.e., their lower bounds.

We further do not consider compression of the values of matrix nonzero elements, since it is generally worth applying only for special kinds of matrices where nonzero elements contain few unique numbers. To store nnz nonzero elements of a matrix A in memory with respect to a precision given by b, we thus need $nnz \times b$ bits to store their values and some additional space to store the information about their structure. The lower bound for the latter for any particular structure of nonzero elements is 1 bit, since it is sufficient for distinguishing whether or not a matrix has that particular structure. For instance, we can use this bit to indicate whether a matrix is tridiagonal. If it is, the bit would be set and we can store the values of nonzero elements in a dense array; their row and column indices can then be derived from the positions of values in this array. Such an approach can be generally applied for any particular structure of matrix nonzero elements.

In practice, we would likely store in memory also some additional information about a matrix, such as its dimensions or its number of nonzero elements. However, for large matrices such as those from our tested suite, this additional data require a negligible amount of memory, therefore we define a lower bound for a matrix memory footprint simply as $\mathrm{MMF}_{\mathrm{lb}}(A, b) = nnz \times b$.

Let

$$\Gamma^b_{\boxplus}(k) = \Big(\min\big\{\mathrm{MMF}_{\boxplus}(A_k, s, h \times w, b) : s \in \mathcal{S}_6, h \times w \in \mathcal{B}_{64}\big\} / \mathrm{MMF}_{\mathrm{lb}}(A_k, b) - 1\Big) \times 100$$

and

$$\Gamma^b_{\mathrm{CSR32}}(k) = \left(\frac{\mathrm{MMF}_{\mathrm{CSR32}}(A_k, b)}{\mathrm{MMF}_{\mathrm{lb}}(A_k, b)} - 1\right) \times 100.$$

$\Gamma^b_{\boxplus}(k)$ thus expresses of how much percents is the memory footprint of A_k stored in an optimal blocking way higher than its lower bound. For comparison purposes, we define also a corresponding metric for the CSR32 format denoted by $\Gamma^b_{\mathrm{CSR32}}(k)$.

The measured statistics of $\Gamma^b_{\boxplus}(k)$ and $\Gamma^b_{\mathrm{CSR32}}(k)$ for the tested matrices are shown in Table VIII. Memory footprints of partitioned sparse matrices were obviously much closer to the lower bounds than memory footprints of matrices stored in CSR32; namely, 5 times closer in average and 2 times in worst cases. Moreover, in best cases, partitioned matrices almost reached their lower-bound memory footprints. For instance, in double precision, 7, 26, and 120 matrices out of 563 provided memory footprints up to 1, 2, and 5 percents above their lower bounds, respectively.

TABLE VIII: Statistics of $\Gamma^b_{\boxplus}(k)$ and $\Gamma^b_{\mathrm{CSR32}}(k)$ (in percents) for the tested matrices.

	Single precision		Double precision	
Statistics	Blk.-opt.	CSR32	Blk.-opt.	CSR32
Minimum	0.63	100.02	0.31	50.01
Average	21.85	111.03	10.93	55.51
Maximum	71.31	152.39	35.66	76.19

IV. Conclusions

Within this study, we analyzed memory footprints of 563 representative sparse matrices with respect to their partitioning into uniformly sized blocks. We considered different block sizes and different ways of storing blocks in a computer memory. The obtained results led us to the following conclusions:

1) Partitioning of sparse matrices substantially reduces memory footprints of sparse matrices when compared to the most-commonly used storage format CSR32. The average observed memory savings in case of single and double precision were 42.3 and 28.7 percents of memory space, respectively. The corresponding worst-case savings were 25.5 and 17.1 percents.
2) Partitioning of sparse matrices provides memory footprints much closer to their lower bounds than CSR32. In average, the measured memory footprints for optimal blocking configurations were of only 21.9 and 10.9 percents higher than the lower bounds, while the corresponding memory footprints for CSR32 were higher of 111.0 and 55.5 percents. Moreover, the memory footprints of matrices most suitable for block processing approach the lower bounds; the amount of memory required for storing information about the structure of nonzero elements of such matrices is relatively negligible.
3) For minimization of memory footprints of partitioned sparse matrices in general, we cannot consider only a single format for storing blocks. Instead, we need to choose a format according to the structure of matrix nonzero elements either for all its blocks collectively (min-fixed scheme) or for each block separately (adaptive scheme). The latter approach mostly yields lower memory footprints.
4) For minimization of memory footprints of partitioned sparse matrices in general, we cannot consider only a single block size. However, we can substantially reduce the set of block sizes in the optimization space and still obtain memory footprints close to their optima. In average, the measured memory footprints for the proposed reduced sets of block sizes $\mathcal{B}_{20}$, $\mathcal{B}_{14}$, and $\mathcal{B}_8$ and the min-fixed/adaptive schemes were at most of only 1.51 percents higher than the optimal values. Even considering square blocks only is thus generally sufficient for minimization of memory footprints of sparse matrices. However, there exist matrices for which the corresponding metrics are significantly higher and are inversely proportional to the number of tested block sizes. One should thus be aware of whether or not his/her matrices fall into this category and if yes, he/she might consider using larger sets of block sizes.

Our findings are encouraging since they show that memory footprints of partitioned sparse matrices can be substantially

reduced even when a relatively small block preprocessing optimization space is considered. Whether or not will such a reduction pay off in practice depends first of all on the objective one wants to achieve. A big challenge is to improve the performance of memory-bounded sparse matrix operations due to the reduction of memory footprints of matrices. Within our future work, we plan to face this problem at least partially—we will focus on the development of scalable efficient block preprocessing and SpMV algorithms for the min-fixed and adaptive blocking storage schemes, and we will evaluate them experimentally on mainstream HPC architectures.

ACKNOWLEDGEMENTS

The authors acknowledge support from P. Tvrdík from the Czech Technical University in Prague, P. Vrchota from Výzkumný a zkušební letecký ústav, a.s., and M. Pajr from IHPCI.

REFERENCES

[1] D. Langr, I. Šimeček, and P. Tvrdík, "Storing sparse matrices in the adaptive-blocking hierarchical storage format," in *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2013)*. IEEE Xplore Digital Library, 2013, pp. 479–486.

[2] D. Langr, "Algorithms and data structures for very large sparse matrices," Ph.D. dissertation, Czech Technical University in Prague, 2014.

[3] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, vol. 50, no. 1, pp. 36–77, 2009. doi: 10.1007/s11227-008-0251-8

[4] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient SMVM kernels," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009. doi: 10.1145/1542275.1542294. ISBN 978-1-60558-498-0 pp. 100–109.

[5] ——, "A library for pattern-based sparse matrix vector multiply," *International Journal of Parallel Programming*, vol. 39, no. 1, pp. 62–87, 2011. doi: 10.1007/s10766-010-0145-2

[6] G. E. Blelloch, M. A. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessors," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-93-173, 1993.

[7] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009. doi: 10.1145/1583991.1584053. ISBN 978-1-60558-606-9 pp. 233–244.

[8] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. IEEE Computer Society, 2011. doi: 10.1109/IPDPS.2011.73 pp. 721–733.

[9] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016. doi: 10.1145/2925426.2926278 pp. 37:1–37:12.

[10] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix-vector multiplication for multicore," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-215, 2012.

[11] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010. doi: 10.1145/1693453.1693471 pp. 115–126.

[12] R. Eberhardt and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016. doi: 10.1109/IPDPSW.2016.42 pp. 663–672.

[13] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in SPARSITY," in *Proceedings of the International Conference on Computational Science (ICCS 2001), Part I*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, vol. 2073, pp. 127–136.

[14] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004. doi: 10.1177/1094342004041296

[15] R. Kannan, "Efficient sparse matrix multiple-vector multiplication using a bitmapped format," in *20th Annual International Conference on High Performance Computing*, 2013. doi: 10.1109/HiPC.2013.6799135 pp. 286–294.

[16] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *Proceedings of the 2009 International Conference on Computational Science and Engineering (CSE '09).*, vol. 1, Aug 2009. doi: 10.1109/CSE.2009.223 pp. 247–256.

[17] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, "Adaptive-blocking hierarchical storage format for sparse matrices," in *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2012)*. IEEE Xplore Digital Library, 2012, pp. 545–551.

[18] D. Langr and P. Tvrdík, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 428–440, 2016. doi: 10.1109/TPDS.2015.2401575

[19] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance modeling and analysis of cache blocking in sparse matrix vector multiply," Computer Science Division (EECS), University of California, Tech. Rep. UCB/CSD-04-1335, 2004.

[20] ——, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007. doi: 10.1007/s00200-007-0038-9

[21] I. Šimeček, D. Langr, and P. Tvrdík, "Space-efficient sparse matrix storage formats for massively parallel systems," in *Proceedings of the 14th IEEE International Conference of High Performance Computing and Communications (HPCC 2012)*. IEEE Computer Society, 2012. doi: 10.1109/HPCC.2012.18 pp. 54–60.

[22] I. Šimeček and D. Langr, "Space and execution efficient formats for modern processor architectures," in *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2015)*. IEEE Computer Society, 2015. doi: 10.1109/SYNASC.2015.24 pp. 98–105.

[23] F. S. Smailbegovic, G. N. Gaydadjiev, and S. Vassiliadis, "Sparse Matrix Storage Format," in *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2005*, 2005, pp. 445–448.

[24] P. Stathis, S. Vassiliadis, and S. Cotofana, "A hierarchical sparse matrix storage format for vector processors," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, p. 61.

[25] P. Tvrdík and I. Šimeček, "A new diagonal blocking format and model of cache behavior for sparse matrices," in *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM 2005)*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 3911, pp. 164–171.

[26] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009. doi: 10.1145/1498765.1498785

[27] T. A. Davis and Y. F. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, 2011. doi: 10.1145/2049662.2049663

[28] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. Philadelphia, PA: SIAM, 1994.

[29] D. Langr, I. Šimeček, and T. Dytrych, "Block iterators for sparse matrices," in *Proceedings of the Federated Conference on Computer Science and Information Systems (FedCSIS 2016)*. IEEE Xplore Digital Library, 2016. doi: 10.15439/2016F35 pp. 695–704.

[30] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs for graph applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014. doi: 10.1109/SC.2014.69 pp. 781–792.

[31] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs: Implications for graph mining," *Proc. VLDB Endow.*, vol. 4, no. 4, pp. 231–242, 2011. doi: 10.14778/1938545.1938548

[32] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–58, 2008. doi: 10.1109/IEEESTD.2008.4610935