Perfomance Models for Blocked Sparse Matrix-Vector Multiplication kernels

Vasileios Karakasis, Georgios Goumas, Nectarios Koziris
National Technical University
Athens, Greece
HiPEAC members
Email: $\{$*bkk,goumas,nkoziris*$\}$*@cslab.ece.ntua.gr*

***Abstract*—Sparse Matrix-Vector multiplication (SpMV) is a very challenging computational kernel, since its performance depends greatly on both the input matrix and the underlying architecture. The main problem of SpMV is its high demands on memory bandwidth, which cannot yet be abudantly offered from modern commodity architectures. One of the most promising optimization techniques for SpMV is blocking, which can reduce the indexing structures for storing a sparse matrix, and therefore alleviate the pressure to the memory subsystem. However, blocking methods can severely degrade performance if not used properly. In this paper, we study and evaluate a number of representative blocking storage formats and present a performance model that can accurately select the most suitable blocking storage format and the corresponding block shape and size for a specific sparse matrix. Our model considers both the memory and computational part of the kernel, which can be non-negligible when applying blocking, and also assumes an overlapping of memory accesses and computations that modern commodity architectures can offer through hardware prefetching mechanisms.**

***Keywords*-sparse matrix-vector multiplication; performance models; blocking**

I. Introduction

Sparse Matrix-Vector Multiplication (SpMV) is one of the most important and widely used scientific kernels arising in a variety of scientific problems. The SpMV kernel poses a variety of performance issues both in single and multicore configurations [5], [14], [19], which are mainly due to the memory-intensive nature of the SpMV algorithm. To this end, a number of optimization techniques have been proposed, such as register and cache blocking [7], [8], compression [10], [18], and column or row reordering [12]. The main purpose of all these techniques is to either reduce the total *working set* of the algorithm, i.e., the total amount of data that needs to be fetched from main memory, or create more regular memory access patterns (reordering techniques). Blocking methods fall mainly into the first category, since their main advantage is that they can considerably reduce the total working set of the algorithm, thus applying less pressure to the memory subsystem. By alleviating the memory pressure, blocking techniques leave more space for optimizations targeting the computational part of the kernel as well, such as loop unrolling and vectorization [9], which can further improve performance.

In general, blocking storage formats for sparse matrices can be divided into three categories: (a) storage formats that apply zero-padding aggresively in order to construct full blocks, (b) storage formats that decompose the original matrix into k submatrices, where the $k-1$ submatrices use blocking without padding and the k-th matrix is stored in standard Compressed Sparse Row (CSR) format [2], and (c) storage formats that use variable size blocks without padding, but at the expense of additional indexing structures. Blocked Compressed Sparse Row (BCSR) [8] and Unaligned BCSR [17] formats are typical examples of the first category. Both formats try to exploit small two-dimensional dense subblocks inside the sparse matrix with their main difference being that BCSR imposes a strict alignment to its blocks at specific row- and column-boundaries. Agarwal et al. [1] decompose the input matrix by extracting regular common patterns, such as dense subblocks and partial diagonals. Similarly, Pinar and Heath [12] decompose the original matrix into two submatrices: a matrix with horizontal one-dimensional dense subblocks without padding and a matrix in CSR format containing the remainder elements. Pinar and Heath [12] present also an one-dimensional variable-sized blocking storage format, while in [13] the Variable Blocking Row (VBR) format is presented, which constructs two-dimensional variable blocks at the cost of two additional indexing structures.

In this paper, we study and evaluate the performance of five different storage formats, which are representative of the blocking strategies for sparse matrices: (a) BCSR, (b) BCSR-DEC, which is the decomposed version of BCSR, (c) BCSD (Block Compressed Sparse Diagonal), which is a variation of BCSR exploiting dense diagonal subblocks, (d) BCSD-DEC, which is the decomposed version of BCSD, and (e) 1D-VBL (One-dimensional Variable Block Length), which is the storage format proposed in [12]. Our evaluation shows that none of them can be considered as a panacea, since in our matrix suite consisting of 30 matrices from different problem categories every storage format managed to achieve the best overall performance for at least two matrices. Additionally, the standard CSR format should always be considered as a competing storage format, since in many problems without an underlying 2D/3D geometry, blocking methods could not provide any speedup.

With such a variety of efficient blocking storage formats and considering the different combination of block shapes for each format, that can also have a significant impact in performance [9], it is obvious that a performance model is needed, in order to select the most appropriate combination of storage format and block shape. Such a performance model should not only be able to predict the execution time of a specific blocking method, but also to guide the selection of the best combination of blocking storage and block shape. Towards this direction, Gropp et al. [6] used a simple performance model to predict the performance of CSR, in which case the execution time was computed as the ratio of the algorithm's working set to the machine's memory bandwidth. However, when moving to blocking storage formats, the memory wall problem is not so intense as in the CSR case and, as pointed out in [9], the computational part of the kernel is non-negligible and the selection of a block that favors computational optimizations can lead to even higher performance. Therefore, a model that would also account for the computational part of the kernel should be considered. Vuduc et al. [16] and Buttari et al. [3] propose a simple heuristic that accounts for the computational part of BCSR by estimating the padding of blocks and by profiling a dense matrix, but it is constrained to the BCSR format only. In this work, we propose two general performance models for the prediction of the execution time of blocked sparse kernels that take into account the computational part of the kernel: a model that considers separately the memory and computational part of the algorithm (MEMCOMP) and a model that assumes an overlapping between the two parts (OVERLAP). We compare both models to the simple streaming model (MEM) presented in [6]. Although MEM can generally provide a good selection of method and block shape, it can fall short on its selection, when the problem becomes more computational intensive. The contrary is true with MEMCOMP model. However, the OVERLAP model adapts better to each case and can yield a fairly good selection of method and block shape, which leads to performance that lies on average 2% off the best SpMV performance achieved for the microarchitecture used in this paper.

The rest of the paper is organized as follows: Section II presents and discusses the different blocking storage formats for sparse matrices. Section III discusses the performance issues involved when applying blocking to SpMV, Section IV presents the performance models proposed in this paper, Section V presents an experimental evaluation of the different blocking storage formats considered and evaluates the proposed performance models. Finally, Section VI concludes the paper.

II. AN OVERVIEW OF BLOCKING STORAGE FORMATS

In this section, we consider blocking storage formats for sparse matrices that can be applied to an arbitrary sparse matrix, as opposed to storage formats which assume a special nonzero elements pattern, e.g., tri-band, diagonal sparse matrices, etc. Before proceeding with the description of any blocking format, we describe briefly the standard sparse matrix storage format, namely the Compressed Sparse Row (CSR) format [2]. CSR uses three arrays to store a $n \times m$ sparse matrix with nnz nonzero elements: an array val of size nnz to store the nonzero elements of the matrix, an array col_ind of size nnz to store the column indices of every nonzero element, and an array row_ptr of size $n+1$ to store pointers to the first element of each row in the val array.

Blocking storage formats for sparse matrices can be divided into two large categories: (a) formats with fixed-size blocks that employ aggressively padding with zeros to construct full blocks and (b) formats that do not pad at all. The second category can be further divided depending on the strategy used to avoid padding. There have been proposed two strategies to avoid padding in the literature: (a) decompose the original matrix into two or more matrices, where each matrix contains dense subblocks of some common pattern (e.g., rectangular, diagonal blocks, etc.), while the last matrix contains the remainder elements in a standard sparse storage format [1], and (b) use variable-sized blocks [12], [13]. In the following, we will present each blocking method in more detail.

A. Blocking with padding

Blocked Compressed Sparse Row: The most prominent blocking storage format for sparse matrices that uses padding is the Blocked Compressed Sparse Row (BCSR) format [8]. BCSR is the blocked version of CSR, which instead of storing and indexing single nonzero elements, it stores and indexes two-dimensional fixed-size blocks with at least one nonzero element. BCSR will use padding in order to construct full blocks. Specifically, BCSR uses three arrays to store a sparse matrix: (a) $bval$, which stores linearly in row-wise or column-wise order the values of all blocks present in the matrix, (b) $bcol\_ind$, which stores the block-column indices, and (c) $brow\_ptr$, which stores pointers to the first element of each block row in $bval$. Another property of BCSR is that it imposes a strict alignment to its blocks: each $r \times c$ block should be aligned at r row- and c column-boundaries, i.e., a $r \times c$ block should always start at the position (i, j), such that $\mathrm{mod}(i, r) = 0$ and $\mathrm{mod}(j, c) = 0$. This restriction leads generally to more padding (see Fig. 1), but it has two main advantages: it facilitates the construction of the BCSR format and it can have a positive impact on performance, when using vectorization [9]. A variation of BCSR is the Unaligned BCSR (UBCSR) [17], which relaxes the above restriction, in order to avoid padding.

Blocked Compressed Sparse Diagonal: The Blocked Compressed Sparse Diagonal (BCSD) format is analogous to BCSR, but exploits small diagonal subblocks inside

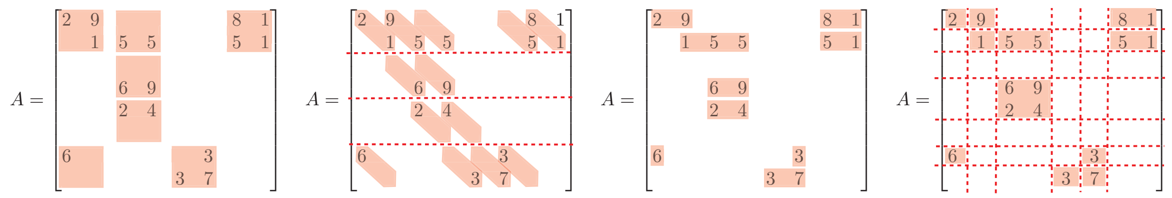

(a) BCSR (b) BCSD (c) 1D-VBL (d) VBR

Figure 1: How the different blocking storage formats split the input matrix into blocks.

the matrix. Like BCSR, it also uses three arrays—$bval$, $bcol\_ind$, and $brow\_ptr$—to store the input matrix, but in this case $bval$ stores the elements of each diagonal subblock, while $bcol\_ind$ continues to store the column index of each subblock. BCSD also imposes an alignment restriction as to where each diagonal block can start. Specifically, each diagonal block of size b should start at the position (i, j), such that $\mathrm{mod}(i, b) = 0$. This restriction effectively separates the matrix into block rows or segments of size b (see Fig. 1). The $brow\_ptr$ array then stores pointers to the first element of each segment in the the $bval$ array. BCSD also uses padding to construct full blocks.

A version of this format has been initially proposed in [1] as part of a decomposed method, which extracted common dense subblocks from the input matrix. A similar format, called RSDIAG, is also presented in [15], but it maintains an additional structure that stores the total number of diagonals in each segment. This format was also part of a decomposed method.

B. Blocking without padding

Decomposed matrices: A common practice to avoid padding is to decompose the original input sparse matrix into k smaller matrices, where the first $k - 1$ matrices consist of elements extracted from the input matrix that follow a common pattern, e.g., rectangular or diagonal dense subblocks, while the k-th matrix contains the remainder elements of the input matrix, stored in a standard sparse matrix storage format. In this paper, we present and evaluate the decomposed versions of BCSR (BCSR-DEC) and BCSD (BCSD-DEC). For these formats $k = 2$, i.e., the input matrix is split into only two submatrices, the first containing full blocks without padding and the second the remainder elements stored in CSR format.

Variable size blocks: An alternative solution to avoid padding when using blocking is to use variable size blocks. Two methods have been proposed in the literature that use variable-size blocks: one-dimensional Variable Block Length (1D-VBL) [12], which exploits one-dimensional horizontal blocks, and Variable Block Row (VBR) [13], which exploits two-dimensional blocks. 1D-VBL uses four arrays to store a sparse matrix: val, row_ptr, $bcol\_ind$, and blk_size. The val and row_ptr arrays serve exactly the same purpose as in CSR, while $bcol\_ind$ stores the starting column of each block and blk_size stores the size of each block. VBR is more complex and it actually partitions the input matrix horizontally and vertically, such that each resulting block contains only nonzero elements. In order to achieve this, it uses two additional arrays compared to CSR to store the start of each block-row and block-column.

Figure 1 summarizes how each block format forms blocks from neighboring elements.

III. Performance issues of blocking

The SpMV kernel in modern commodity microarchitectures is in most cases bound from memory [5] bandwidth. Although there exist other potential performance problems in this kernel, such as indirect references, irregular accesses, and loop overheads, the bottleneck in the memory subsystem is categorized as the most important SpMV performance problem in both single- and multithreaded configurations [5], [9], [19]. The great benefit of blocking methods is that they can substantially reduce the working set of the SpMV algorithm, thus reducing the demands on memory bandwidth and alleviating the pressure to the memory subsystem. The reduction of the working set is mainly due to the fact that blocking methods maintain a single index for each block column instead of an index for each element. Therefore, the col_ind structure of CSR, which comprises almost half of the working set of the algorithm, can be significantly reduced. A consequence of reducing the memory bandwidth demands of the SpMV kernel is that the computational part is then more exposed, since it comprises a larger portion of the overall execution time. Therefore, optimizations targeting the computational part can have significant impact on performance [9]. However, each blocking method has its own advantages and pitfalls, which are summarized in the following.

Fixed size blocking with padding: The main advantage of these methods is that they allow for efficient implementations of block-specific kernels since the size—and in most cases the alignment—of blocks is known a priori. However, if the nonzero elements pattern of the input matrix is rather irregular, these methods lead to excessive padding, overwhelming any benefit from the reduction of the size of col_ind structure. Additionally, the selection of the most appropriate block is not straightforward, especially if vectorization is used, since instruction dependencies and hardware limitations of the vector units of modern commodity architectures can significantly affect the overall performance [9].

Decomposed methods: Although decomposed methods avoid padding, they suffer from three problems: (a) there is no temporal or spatial locality (except in the input vector) between the different k SpMV operations, (b) additional operations are needed to accumulate the partial results to the final output vector, and (c) the remainder CSR matrix will have very short rows, which can further degrade the overall performance due to loop overheads and cache misses on the input vector [5].

Variable size blocking: The variable size blocking has also the advantage of not employing padding to construct blocks, but at the expense of additional data structures. Therefore, any gain in the final working set of the algorithm by eliminating padding and reducing the *col_ind* structure can be overwhelmed by the size of the additional data structures. Finally, the extra level of indirection that variable size blocking methods introduce can further degrade performance.

IV. Performance models

From the above discussion of blocking storage formats for sparse matrices, it is obvious that the correct selection of both a blocking method and an appropriate block (for fixed sized blocking methods) can be a tedious task. Therefore, a performance model is needed that could guide the selection of the appropriate blocking method and exact block and, possibly, the use of a specially tuned multiplication kernel, e.g., with or without vectorization. In this paper, we examine two new performance models that account also for the computational part of the kernel (MEMCOMP and OVERLAP) and compare them to the classic memory bandwidth-based model of Gropp et al. [6] (MEM). The performance models that we propose here are suitable for fixed size blocks with or without padding (decomposed blocking methods). We do not consider variable size blocking methods, since our experiments showed that they have very little potential to provide competitive performance, due to the space and computational overhead that their additional data structures incur. In addition, we have chosen not to directly compare with the heuristic presented in [15], since this targets specifically the BCSR storage format, and thus, it is not as generic as the models presented herein.

According to [5], the total execution time of SpMV is determined primarily by the memory transfer times, and secondarily by the memory latency that should be paid for cache misses due to irregular accesses on the input vector, the loop overheads when the matrix comprises of very short rows, and the useful arithmetic operations that perform the actual multiplication. Each model makes different assumptions for the significance of these times.

The MEM *model:* This model regards the SpMV as a pure streaming kernel and ignores the memory latencies incurred by cache misses and the computational part of the kernel. Therefore, if ws is the working set of the SpMV algorithm for a specific storage format and BW is the effective memory bandwidth of the system, the predicted execution time according to the MEM model is

$$t_{\text{MEM}} = \frac{ws}{BW}. \qquad (1)$$

This model is fairly general and can be applied to any blocking method for sparse matrices.

The MEMCOMP *model:* The MEMCOMP model builds upon the MEM model by considering also the computational part of the kernel. It can be applied on fixed size blocking methods, either decomposed or not. According to this model, the predicted execution time for a storage format decomposed into k matrices is

$$t_{\text{MEMCOMP}} = \sum_{i=1}^{k} \left(\frac{ws_i}{BW} + nb_i \cdot t_{b_i} \right), \qquad (2)$$

where nb_i is the total number of blocks for the i-th matrix in the decomposition and t_{b_i} is the estimated execution time for a single block of the i-th matrix. In general, each matrix of the decomposition can be stored with a different block, even with a different fixed size blocking storage format. If $k = 1$, this model computes the execution time of a fixed size blocking method with padding. The time t_{b_i} differs between block methods, specific blocks, and implementations (e.g., SIMD). These block times can be obtained by profiling the execution of a very small dense matrix, which is stored using every blocking method and block under consideration and fits in the L1 cache of the target machine. The MEMCOMP model also treats CSR as a degenerate blocking method with 1×1 blocks and $nb = nnz$.

The OVERLAP *model:* The idea behind the OVERLAP model is that modern commodity architectures employ smart hardware prefetching mechanisms, which apart from hiding the memory latency by reducing the cache misses, they can effectively overlap computations and memory transfers, since they fetch new data from main memory to cache, while the processor is processing the current data. As discussed in [5], the hardware prefetching mechanisms of modern commodity architectures can accurately predict the memory access pattern of SpMV and can provide large speedups. The OVERLAP model is similar to the MEMCOMP model, but the part of equation (2) representing the computations is adapted with a multiplication factor indicating the percentage of computations that are not overlapped with memory transfers. We name this factor *non-overlapping factor* (nof). Therefore, the execution time of a blocked kernel according to the OVERLAP model is

$$t_{\text{OVERLAP}} = \sum_{i=1}^{k} \left(\frac{ws_i}{BW} + nof_{b_i} \cdot nb_i \cdot t_{b_i} \right). \qquad (3)$$

Similar to t_{b_i}, the nof_{b_i} factor differs between block methods and blocks, and it is obtained through the following formula by profiling a large dense matrix that exceeds the highest level of cache of the target machine:

$$nof_{b_i} = \frac{t_{real_{bi}} - t_{\text{MEM}}}{nb_i \cdot t_{b_i}}, \tag{4}$$

where $t_{real_{bi}}$ is the real execution time for the specific block method and block applied to the profiled dense matrix, and t_{b_i} is as computed for the MEMCOMP model. The nominator of this fraction represents the time of computations that were not actually overlapped with memory transfers, while the denominator represents the estimated time for computations.

Finally, it should be noted that all models presented ignore memory latencies, which means that they actually ignore the cache misses due to the irregular accesses on the input vector. As pointed out in [5], these irregular accesses do not pose a significant performance impedence in most cases, since they can be tackled from hardware prefetching.

V. Experimental evaluation

Matrix suite: The matrix suite used for our experiments is a set of sparse matrices obtained from Tim Davis' sparse matrix collection [4]. We chose to include matrices from different application domains, which could reveal the capabilities and possible shortcomings of the different blocking storage formats under evaluation. The matrix suite consists of 30 matrices (Tab. I). Matrices #1 (dense) and #2 (random) are special purpose matrices, while the remaining 28 are divided into two large categories: matrices #3–#16 come from problems without an underlying 2D/3D geometry, while matrices #17–#30 have a 2D/3D geometry. In general, sparse matrices with an underlying geometry exhibit more regular structure, so we expect blocking methods to perform better on these matrices. Finally, all selected matrices have large enough working sets (>25 MB in CSR format), so that none of them fits in the processor's cache.

System platform and experimental process: For our experimental evaluation we used a dual Intel Core 2 Duo Xeon processor clocked at 2.66 GHz. Each core has separate L1 I-cache and D-cache, 32 KiB each. The two cores share a 4 MiB, 16-way set associative L2 cache. This microarchitecture supports hardware prefetching, which was enabled. The memory subsystem can deliver up to 3.36 GiB/s according to the STREAM benchmark [11]. The system ran GNU/Linux, kernel version 2.6, for the x84_64 ISA. All programs were compiled using `gcc`, version 4.2, with the highest level of optimization (`-O3`). For the evaluation of each blocking storage format, we ran 100 consecutive SpMV operations using randomly generated input vectors.

A. Evaluation of blocking storage formats

We have implemented five different blocking storage formats: two with fixed size blocks using padding (BCSR and BCSD), two decomposed with fixed size blocks (BCSR-DEC and BCSD-DEC), and one with variable size blocks (1D-VBL). We also implemented the standard CSR format, in order to have a common baseline. We used four-byte integers for the indexing structures of every format, and one-byte entries for the additional data structure of 1D-VBL, which contains the block sizes. This restricts the number of maximum elements per block to 255, but in the rare case a greater block is encountered, it is split into 255-element chunks. For the fixed size blocking methods, we used blocks with up to eight elements, and we have implemented a block-specific multiplication routine for each particular block. We did not use larger blocks, since preliminary experiments showed that such blocks cannot offer any speedup over standard CSR. We also implemented vectorized versions of the kernels for the fixed size blocking methods. Table II presents how many matrices each method "won", i.e., in how many matrices managed to provide the overall best performance, for every different configuration. Table III shows in detail the mininum, average, and maximum speedups over standard CSR per matrix, that each method achieved for the double precision configuration without vectorization. The results are similar for the remaining configurations, too.

We can make a number of observations from the results presented. First, the variable size 1D-VBL format cannot be considered very competitive on the microarchitecture used, since it managed to achieve the best performance (with a marginal, less than 3%, difference from the second, BCSR-DEC method) on only one matrix in a single configuration. In addition, it could not outperform CSR in most cases. This is mainly due to the overhead incurred by the additional data structure to store the block sizes. However, 1D-VBL achieved the best speedup for the dense matrix, but this is expected, since it can construct the largest blocks. On the other hand, CSR is still a competitive format, especially for matrices without an underlying 2D/3D geometry, where the blocking methods, even in their decomposed form, cannot perform adequately. Second, the performance of blocking methods with padding can significantly vary, especially in the case of BCSR. For example, the variation between minimum and maximum performance can be greater than 50% on those matrices where an average 10% speedup can be achieved over CSR. On the other hand, although decomposed methods are not so competitive as blocking methods with padding, especially for single precision, they exhibit a more stable behavior across different blocks, since the minimum and maximum performance differ only about 10%–15%.

We have also implemented multithreaded versions of the blocked methods under consideration. We have chosen not to implement a multithreaded version of 1D-VBL, since it proved to be not competitive in the single-threaded configuration. In order to assign work to threads, we have split the input matrix row-wise in as many portions as threads. We

Matrix	Domain	# rows	# nonzeros	ws (MiB)	Matrix	Domain	# rows	# nonzeros	ws (MiB)
01.dense	special	2,000	4,000,000	30.54	16.bone010	Other	986,703	36,326,514	288.44
02.random	special	100,000	14,977,726	115.42	17.kkt_power	Power	2,063,494	8,130,343	121.05
03.cfd2	CFD	123,440	1,605,669	24.95	18.largebasis	Opt.	440,020	5,560,100	45.01
04.parabolic_fem	CFD	525,825	2,100,225	34.05	19.TSOPF_RS	Opt.	38,120	16,171,169	123.81
05.Ga41As41H72	Chemistry	268,096	9,378,286	74.62	20.af_shell10	Struct.	1,508,065	27,090,195	223.94
06.ASIC_680k	Circuit	682,862	3,871,773	37.35	21.audikw_1	Struct.	943,695	39,297,771	310.62
07.G3_circuit	Circuit	1,585,478	4,623,152	76.59	22.F1	Struct.	343,791	13,590,452	107.62
08.Hamrle3	Circuit	1,447,360	5,514,242	58.63	23.fdiff	Struct.	4,000,000	27,840,000	258.18
09.rajat31	Circuit	4,690,002	20,316,253	208.67	24.gearbox	Struct.	153,746	4,617,075	71.04
10.cage15	Graph	5,154,859	99,199,551	815.82	25.inline_1	Struct.	503,712	18,660,027	148.13
11.wb-edu	Graph	9,845,725	57,156,537	548.75	26.ldoor	Struct.	952,203	23,737,339	192.00
12.wikipedia	Graph	3,148,440	39,383,235	336.50	27.pwtk	Struct.	217,918	5,926,171	47.71
13.degme	Lin. Prog.	659,415	8,127,528	65.94	28.thermal2	Other	1,228,045	4,904,179	51.47
14.rail4284	Lin. Prog.	1,096,894	1,000,000	90.31	29.nd24k	Other	72,000	14,393,817	110.64
15.spal_004	Lin. Prog.	321,696	46,168,124	353.54	30.stomach	Other	213,360	3,021,648	25.50

Table I: Matrix suite. The working set (ws) column represents the working set of the matrix stored in CSR format.

Method/Configuration	dp	dp-simd	sp	sp-simd
CSR	9	9	7	7
BCSR	8	9	13	14
BCSR-DEC	6	6	3	2
BCSD	2	2	2	2
BCSD-DEC	2	2	3	3
1D-VBL	1	–	0	–

Table II: Total number of matrices that each format provided the best overall performance for the different configurations tried ('dp' stands for double precision and 'sp' for single-precision). The special category matrices (dense, random) are ignored.

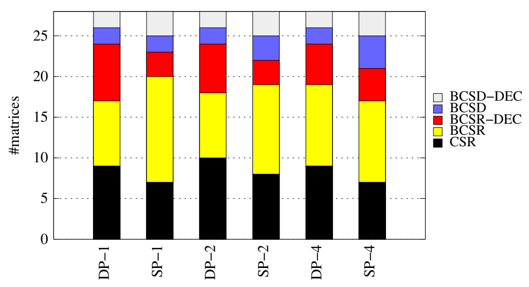


Figure 2: Distribution of wins (overall best performance) across blocking methods for one, two, and four cores, single and double precision.

have also applied a static load balancing scheme, according to which we split the input matrix, such that each thread is assigned the same number of nonzeros. Specifically, for the case of methods with padding, we also accounted for the extra zero elements used for the padding. Figure 2 presents number of "wins" for each method for one, two, and four cores. The picture here is similar to the single-threaded configuration: BCSR continues to gain the majority of matrices with CSR and BCSD following.

From the above evaluation, it is obvious that there is no method that fits all the matrices. All blocking methods presented, except 1D-VBL, managed to achieve the best performance in at least 10% of the matrices of our matrix suite.

B. Evaluation of the performance models

A performance model for the selection of a blocking method can be evaluated with two metrics: (a) the *prediction accuracy*, i.e., how accurately it can predict the actual execution time of a blocked kernel, and (b) the *selection accuracy*, i.e., how close to the optimal combination of block method and block shape is its selection. Although the first metric implies the second, i.e., if a model can accurately predict the actual execution time of the kernel, then its selection would also be accurate, the inverse is not necessarily true. What is important for a performance model to accurately select the proper blocking method and block is to properly rank the different combinations of blocking methods and blocks with the help of its own prediction, even if the predicted execution time is not very accurate. We evaluate the performance models presented in Section IV with both metrics.

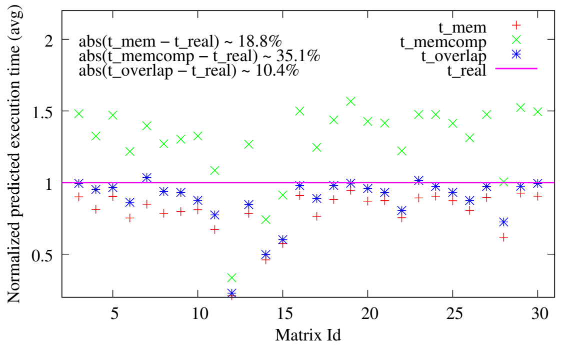
Figure 3 depicts the average predicted execution time of each model normalized over the actual execution time of the kernel for single and double precision. Each point in these graphs represents the average predicted execution time of a model over all possible combinations of blocks and block methods for the matrix whose id lies on the x-axis (we have omitted the first two special matrices). The legend on the left of each subfigure describes the average distance of the predicted execution time from the actual execution time for all matrices. This can be viewed as a metric of the prediction accuracy of each model. These figures reveal several important characteristics of each performance model. First, we can argue that the MEM model provides a lower bound of execution time (performance upper bound), while the MEMCOMP, in general, can provide an upper bound of execution time (performance lower bound). The OVERLAP model approximates better the real execution time of a

Matrix	BCSR			BCSR-DEC			BCSD			BCSD-DEC			1D-VBL
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	
01.dense	1.16	1.28	1.29	1.16	1.28	1.29	1.17	1.27	1.27	1.16	1.27	1.27	**1.32**
02.random	0.21	0.21	0.71	0.98	0.99	**1.00**	0.69	0.69	0.80	0.99	0.99	**1.00**	0.80
03.cfd2	0.41	0.42	0.90	0.91	0.95	1.01	0.92	0.92	0.96	1.00	1.03	**1.03**	0.66
04.parabolic_fem	0.30	0.36	0.80	0.84	0.91	0.91	0.69	0.70	0.73	0.83	0.88	0.88	0.82
05.Ga41As41H72	0.35	0.35	0.85	0.97	0.98	**1.05**	0.77	0.77	0.93	1.01	1.02	1.03	0.86
06.ASIC_680k	0.34	0.35	0.84	0.88	0.94	0.95	0.66	0.66	0.77	0.87	0.91	0.92	0.81
07.G3_circuit	0.34	0.35	0.85	0.76	0.81	0.81	0.83	0.85	0.86	0.84	0.88	0.88	0.81
08.Hamrle3	0.45	0.45	0.96	0.80	0.86	0.87	0.62	0.67	0.86	0.82	0.82	0.86	0.74
09.rajat31	0.30	0.31	0.78	0.79	0.86	0.86	0.80	0.81	0.84	0.87	0.87	0.90	0.74
10.cage15	0.24	0.24	0.71	0.94	0.96	0.96	0.79	0.81	0.90	0.97	1.01	**1.02**	0.75
11.wb-edu	0.40	0.54	0.95	0.82	0.93	0.93	0.73	0.73	0.74	0.83	0.87	0.87	0.74
12.wikipedia	0.34	0.37	0.89	0.98	0.98	0.99	0.83	0.89	0.89	0.98	0.98	0.98	0.66
13.degme	0.28	0.28	0.71	0.88	1.00	**1.00**	0.65	0.75	0.79	0.94	1.00	**1.00**	0.73
14.rail4284	0.35	0.35	0.98	0.97	1.00	**1.04**	0.73	0.75	0.80	0.97	1.00	1.00	0.77
15.spal_004	0.36	0.36	**1.13**	0.99	1.00	1.12	0.76	0.76	0.77	0.99	1.00	1.00	1.03
16.bone010	0.69	0.70	**1.21**	0.98	0.99	1.17	1.03	1.04	1.07	1.06	1.12	1.13	1.07
17.kkt_power	0.26	0.28	0.79	0.83	0.86	0.86	0.75	0.75	0.78	0.84	0.87	0.87	0.83
18.largebasis	0.51	0.52	**1.16**	0.92	0.96	1.12	0.82	0.84	0.98	0.97	0.97	1.01	0.92
19.TSOPF_RS	1.14	1.21	1.27	1.15	1.27	**1.31**	1.15	1.23	1.23	1.15	1.24	1.24	1.27
20.af_shell10	0.75	0.77	**1.18**	0.97	0.99	1.14	0.93	0.93	1.02	1.02	1.02	1.07	1.02
21.audikw_1	0.56	0.60	**1.21**	0.96	0.99	1.17	0.70	0.70	0.92	0.97	0.97	1.02	0.84
22.F1	0.55	0.61	**1.21**	0.98	1.01	1.18	0.69	0.69	0.88	0.95	1.00	1.01	0.86
23.fdiff	0.27	0.28	0.77	0.85	0.89	0.91	1.00	1.09	**1.10**	1.00	1.09	1.09	0.82
24.gearbox	0.63	0.64	**1.23**	0.99	1.00	1.18	0.83	0.83	1.00	1.02	1.02	1.08	0.92
25.inline_1	0.54	0.60	**1.21**	0.96	0.99	1.18	0.71	0.71	0.92	0.98	0.98	1.02	0.82
26.ldoor	0.70	0.78	1.17	0.96	0.99	**1.17**	0.77	0.77	0.96	0.98	0.98	1.00	1.05
27.pwtk	0.82	0.84	1.08	1.02	1.03	1.11	1.01	1.05	1.05	1.07	1.09	1.10	**1.14**
28.thermal2	0.36	0.42	0.89	0.75	0.88	0.88	0.69	0.69	0.74	0.80	0.85	0.85	0.69
29.nd24k	0.92	0.94	1.11	1.10	1.11	**1.17**	1.00	1.00	1.09	1.11	1.12	1.14	0.96
30.stomach	0.29	0.30	0.80	0.92	0.96	0.98	1.02	1.06	**1.09**	1.03	1.05	1.08	0.58
Average	*0.49*	*0.52*	*0.99*	*0.93*	*0.98*	*1.04*	*0.82*	*0.85*	*0.92*	*0.97*	*1.00*	*1.01*	*0.87*

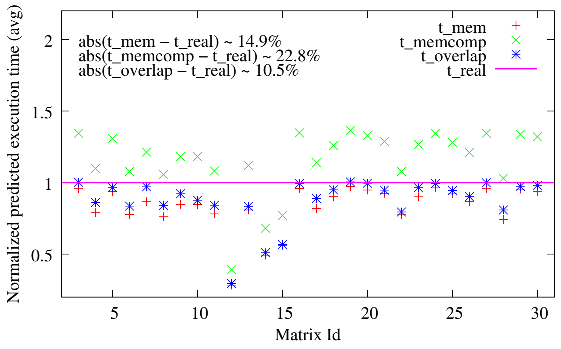
Table III: Comparison of storage formats: speedups over CSR per matrix for all the blocks tested (double precision).

block kernel, and its prediction lies within 10% off the actual execution time. The MEM model can also provide an accurate prediction (≈15%) when the problem is very memory-bound (double precision). On the other hand, the MEMCOMP model cannot provide a precise prediction of the execution time for the majority of matrices, since it assumes no overlapping between memory transfers and computations. The prediction accuracy of the MEM and OVERLAP models verifies our initial assumption that memory latencies do not represent a significant portion of the total SpMV execution time. However, there exist matrices (#12, #14, #15, and #28) where all models, and especially the MEM and OVERLAP, fall short with their prediction. These are matrices that suffer from irregular accesses on the input vector, and therefore they are rather latency-bound than bandwidth-bound. To verify this, we ran a special custom benchmark on these matrices [5]. This benchmark zeros out the col_ind structure of CSR, so that no misses are incurred due to irregular accesses. When we applied the benchmark, the performance of these matrices doubled, and even quadrapled in the case of matrix #12, whose structure is very irregular, meaning that these matrices suffer indeed from cache misses on the input vector. Finally, we should note that the MEMCOMP model provided much better accuracy in these cases, especially for matrix #28, because its assumption of non-overlapping memory transfers and computations holds.

Figure 4 presents the real execution time of the block method and block that each model selected for the matrix on x-axis normalized over the overall best performance for this matrix. In these figures, the MEMCOMP and OVERLAP models selected also the implementation of the kernel, i.e., whether to use vectorization or not. For the MEM model, where the computational part of kernel is ignored, we selected the non-simd version by default. Table IV presents the total number of correct predictions (block method and block) for each model, as well as their average distance from the optimal selection. The OVERLAP model can provide a fairly accurate selection, whose performance lies within 2% off the optimal performance, whereas the selection of the other two models lies between 4% and 9%. The OVERLAP model, as depicted on Fig. 4 managed to provide nearly optimal selections for all matrices. We should also note here the very good selection accuracy of the MEMCOMP model, especially for double precision, despite its inability to accurately predict the actual execution time. Finally, for matrices #16–#30, where the different block methods where beneficial, both the MEMCOMP and OVERLAP models provided nearly optimal block selections, while the MEM model, which considers



(a) Single precision.



(b) Double precision.

Figure 3: Predicted execution time normalized over the real execution time for each matrix (average over all possible combinations of blocks and methods). Average differences of predicted from real execution time are also depicted.

Model	single precision		double precision	
	#correct	off. from best	#correct	off. from best
MEM	12	6.0%	11	6.9%
MEMCOMP	11	8.8%	17	4.2%
OVERLAP	17	1.5%	19	1.9%

Table IV: Total number of optimal predictions for each model and the difference in performance between the real performance of the model's selection and the optimal performance (average over all matrices is presented).

solely the working set of the algorithm, did not produce very accurate selections.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we studied and evaluated a wide set of blocked methods on a matrix suite consisting of a variety of matrices from different application domains. We showed that a single blocking method cannot fit all matrices, and that there still exist cases where the standard CSR provides the best performance. For that reason, we proposed two new performance models that aid in the selection of the correct storage format and its proper configuration. Our models account also for the computational part of the kernel and for the memory latency hiding mechanisms, such as hardware prefetching, that modern commodity architectures offer. We compared these models with the classic memory bandwidth-aware model, showing that they can provide more accurate selections and predictions. As a future work, we intend to extend these models to also account for memory latencies, which in some cases consist the main performance bottleneck of SpMV. Another important future direction is to consider the adaptation of these models on multicore platforms. Finally, we plan to develop more intelligent and adaptive performance models for the execution of sparse kernels based on machine learning.

AKNOWLEDGEMENTS

This work was supported by the Greek Secretariat of Research and Technology (GSRT) and the European Commission under the program 05AKMWN95.

REFERENCES

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 32–41, Minneapolis, MN, United States, 1992. IEEE Computer Society.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. 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*. SIAM, 1994.

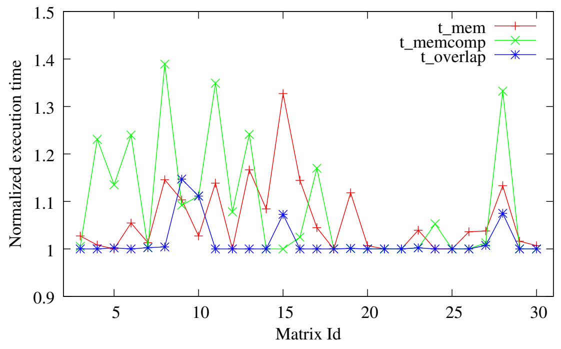
[3] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance optimization and modeling of blocked sparse kernels. *International Journal of High Performance Computing Applications*, 21(4):467–484, 2007.

[4] T. Davis. The University of Florida sparse matrix collection. NA Digest, vol. 97, no. 23, June 1997. http://www.cise.ufl.edu/research/sparse/matrices.

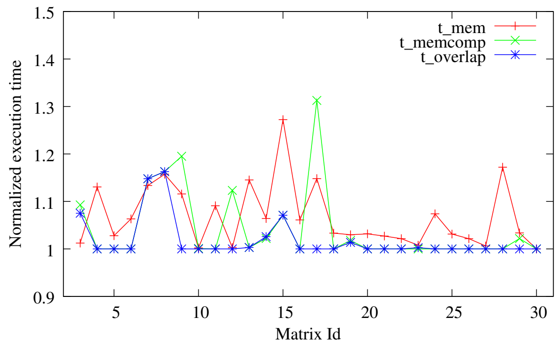
[5] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris. Understanding the performance of sparse matrix-vector multiplication. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, Toulouse, France, 2008. IEEE Computer Society.

[6] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In *International Parallel CFD 1999 Conference*, Williamsburg, VA, United States, 1999.

[7] E. Im and K. Yelick. Optimizing sparse matrix-vector multiplication on SMPs. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.



(a) Single precision.



(b) Double precision.

Figure 4: Actual performance of each method's selection normalized over the best overall performance for each matrix.

[8] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In *Proceedings of the International Conference on Computational Sciences – Part I*, pages 127–136. Springer-Verlag, 2001.

[9] V. Karakasis, G. Goumas, and N. Koziris. Exploring the effect of block shapes on the performance of sparse kernels. In *IEEE International Symposium on Parallel and Distributed Processing (Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications)*, Rome, Italy, 2009. IEEE.

[10] K. Kourtis, G. Goumas, and N. Koziris. Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, Portland, Oregon, United States, 2008. IEEE Computer Society.

[11] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computing, 1995. http://www.cs.virginia.edu/stream/.

[12] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, OR, United States, 1999. ACM.

[13] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations, 1994.

[14] O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 578–587, Minneapolis, MN, United States, 1992. IEEE Computer Society.

[15] R. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, Berkeley, 2003.

[16] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–35, Baltimore, MD, United States, 2002. IEEE Computer Society.

[17] R. W. Vuduc and H.-J. Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications*, volume 3726 of *Lecture Notes in Computer Science*, pages 807–816. Springer Berlin/Heidelberg, 2005.

[18] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual International conference on Supercomputing*, pages 307–316, Cairns, Queensland, Australia, 2006. ACM.

[19] S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, NV, United States, 2007. ACM.