Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 327–335
ISBN 978-83-60810-27-9
ISSN 1896-7094

Use of Hybrid Recursive CSR/COO Data Structures in Sparse Matrix-Vector Multiplication

Michele Martone, Salvatore Filippone, Salvatore Tucci
University of Rome
"Tor Vergata", Italy

Paweł Gepner
Intel Corporation

Marcin Paprzycki
University of Rome
"Tor Vergata", Italy
Polish Academy of Sciences, Poland

***Abstract*—Recently, we have introduced an approach to basic sparse matrix computations on multicore cache based machines using recursive partitioning. Here, the memory representation of a sparse matrix consists of a set of *submatrices*, which are used as leaves of a *quad-tree* structure. In this paper, we evaluate the performance impact, on the Sparse Matrix-Vector Multiplication (*SpMV*), of a modification to our *Recursive CSR* implementation, allowing the use of multiple data structures in leaf matrices (CSR/COO, with either 16/32 bit indices).**

I. Introduction

IT IS known that computations with sparse matrices incur very poor memory performance: indirect addressing causes unpredictable run-time dependencies in memory read/write access; memory access has poor data locality (just to name a two key aspects; see also [1], [2], [3]). To address these issues, recently, we have proposed a *recursive* approach to sparse matrix representation. In [4] we have outlined the proposed method and reported initial experiments with the *SpMV* operation. In the follow-up [5], we have evaluated its performance for the triangular solve and *SpMV* for symmetric matrices. Experimental results lead us to modify the storage scheme in order to reduce the indexing overhead. Encouraging results of an approach employing 16-bit indices have been reported in [6]. Here, we continue investigations leading to the development of methods that can reduce impact of indirect addressing, by reducing the memory traffic incurred in accessing index data. Specifically, we employ *index compression*, and *diversify* the representation of *leaf submatrices* with the intent of raising the floating point performance of the *SpMV* by saving memory bandwidth.

Proceeding, we outline the RCSR storage format with index compression in Section II. Next, we describe modifications to the sparse matrix representations in Section III. Setup for performed experiments can be found in Section IV, while in Section V we analyze the obtained results.

II. The recursive storage format and index compression

We (logically) organize a sparse matrix as a *quad-tree* structure, with nodes consisting of submatrices arising from a recursive partitioning into quadrants. While intermediate nodes are used only as a pointer structure, leaf nodes hold actual subarrays with index and numerical values. The *SpMV* algorithm described in [5] is independent from the actual format of leaf matrices. It only assumes a *coarse* recursive partitioning in leaf submatrices. Similarly to blocking used in dense matrix computations, submatrices at leaf level should be *sized* (in terms of their *memory footprint* during the *SpMV*) in relation to the *cache sizes of the machine*.

In this context, we have investigated a variation to the leaf matrices format, obtained by converting some of the *Compressed Sparse Rows* (CSR) leaves of a matrix to use 16 bit column indices (and thus, reducing the memory traffic). As motivated in Section I (and in the literature; e.g.: [1]), index compression techniques are particularly effective with many active cores. Here, techniques which may not be optimal on a single core (because of a slight memory-bandwidth-to-computation trade-off, in the form of pointer arithmetics), may show their potential when working with multiple cores (where the memory traffic is heavier). As a motivation of our "16-bit" approach, we observe that after partitioning a large sparse matrix (in the RCSR format), it is likely to have many of the leaf submatrices *dimensioned* less than 2^{16}. Thus, using a 16 bit (*halfword*) index type in their CSR *column indices* arrays is possible, and could lead to savings in memory traffic. We name this variant RCSRH. Obviously, for matrices dimensioned less than 2^{16}, the conversion to RCSRH is possible for all submatrices. The outcome of our experiments (documented in [6]) was encouraging: using halfword indices by itself yielded up to a 25% floating point speedup (with a saving in memory usage up to a 16%) on unsymmetric matrices, and 30% on symmetric ones. However, in a number of cases, the RCSRH variant was not helpful. One of the perceived reasons was that CSR itself does not always fit into leaf submatrices, and thus we have decided to convert some leaf matrices to the *COOrdinate* (COO) format. Let us discuss this change with more detail.

III. Recursive storage format with CSR and COO leaves

In this section, we motivate quantitatively why and when storing some submatrices as COO instead of CSR could reduce index overhead, and the way we have chosen to use COO to enhance RCSR.

A matrix is stored in the RCSR format as a quad-tree structure with CSR ([7, Section. 4.3.1]) submatrices at the

leaf level of a *recursive bipartitioning* (see [4]). To store an $r \times c$ matrix with n nonzeroes in CSR, we use an array JA (of size n) with *column indices*, and a *row pointers* array PA (of size $r+1$), referencing *rows* in the JA array. Array JA stores column indices for nonzeroes in a *row-major* order. The array of coefficients (VA) is laid in the same order as JA. To store a matrix in a plain COO format, two n-sized arrays for (row,column) indices (IA,JA) are required. By denoting as $I(r,n)$ the index space requirements for an $r \times c$ matrix (with n nonzeroes) instance we have $I_{CSR}(r,n) \doteq 4(r+1)+4n$ and $I_{COO}(r,n) \doteq 4n+4n$ bytes. Let us call CSRH the CSR format implementation with 16 bit JA indices, and COOH, a COO format implementation with 16 bit IA and JA indices. For these variants, we have $I_{CSRH}(r,n) \doteq 4(r+1)+2n$ and $I_{COOH}(r,n) \doteq 2n+2n$ bytes. This means that for some values of (r,n), COO/COOH would use less indexing space than CSR/CSRH; specifically, $I_{COO}(r,n) < I_{CSR}(r,n)$ when $n < r+1$, and $I_{COOH}(r,n) < I_{CSRH}(r,n)$ when $n < 2r+2$. For this paper, we modified the matrix constructor code to use CSRH whenever a CSR submatrix is dimensioned less than 2^{16}. Similarly, we use COOH whenever a COO submatrix is dimensioned less than 2^{16}; we choose to use COO when $n < r+1$. We adopt COO/COOH as row-major sorted (so we have the same memory access pattern of CSR for JA and VA arrays). In [4] and [6], we have described the *cutoff* function δ as our heuristic regulating subdivision into submatrices; in this paper, we use slightly differing matrix assembly criteria. While we still use the δ_h function from [6], we limit subdivisions by forcing each submatrix not to use more indexing space than a *fullword* COO storage of it would require. The other rules for subdivision are still the same as imposed by δ_h. Please refer to [8] for a full discussion on the new constructor layout. We call the hybrid format resulting from these modifications *Recursive Sparse Blocks* (*RSB*).

IV. Experimental Setup and Methodology

In order to compare the new approach with previously documented experiments using RCSR format (see [6]), we measured performance on the same test set of 36 matrices: 12 of them are symmetric (See Table I), 12 are square unsymmetric (See Table II), and 12 are non square (See Table III). For readability reasons, in Sec. V we left matrices with less significant results (marked with an asterisk (*), in the tables) out of the plots; so the commentary of them is indirect. Furthermore, we have used the same two machines (summarized in table IV). Recall, that **M2** is a lightly loaded network server, while **M1** is a dedicated machine.

For each *matrix/cores* sample, we ran our RSB code, performing 100 times the *SpMV* operation and report the best result. However, timing variation was below 5%, so our results were consistent. We measured timings using the POSIX ([9]) `gettimeofday()` function. Figures in section V depict results, expressed in MFlops (millions of floating point operations per second). Conventionally, we counted 2 Flops per nonzero element for non-symmetric matrices, and 4 for symmetric. We use double precision arithmetic (C's `double` type). Our measurements were performed with *hot caches*; that is, we did not flush deliberately cache contents between subsequent *SpMV*'s; therefore, to avoid artificially high results, all measurements were performed on matrices not fitting entirely in the caches.

TABLE I
SYMMETRIC MATRICES

matrix	r	c	nnz	nnz/r
af_shell10	1508065	1508065	27090195	17.96
BenElechi1	245874	245874	6698185	27.24
bone010	986703	986703	36326514	36.82
crankseg_1	52804	52804	5333507	101.01
ct20stif	52329	52329	1375396	26.28
F1	343791	343791	13590452	39.53
fcondp2	201822	201822	5748069	28.48
kkt_power	2063494	2063494	8130343	3.94
ldoor	952203	952203	23737339	24.93
mip1*	66463	66463	5209641	78.38
nd24k	72000	72000	14393817	199.91
s3dkq4m2	90449	90449	2455670	27.15

TABLE II
GENERAL SQUARE MATRICES

matrix	r	c	nnz	nnz/r
atmosmodl	1489752	1489752	10319760	6.93
av41092	41092	41092	1683902	40.98
cage15	5154859	5154859	99199551	19.24
lhr71	70304	70304	1528092	21.74
patents	3774768	3774768	14970767	3.97
raefsky3	21200	21200	1488768	70.22
rajat31	4690002	4690002	20316253	4.33
rma10*	46835	46835	2374001	50.69
sme3Dc	42930	42930	3148656	73.34
torso1	116158	116158	8516500	73.32
venkat01	62424	62424	1717792	27.52
wb-edu	9845725	9845725	57156537	5.81

TABLE III
GENERAL NON SQUARE MATRICES

matrix	r	c	nnz	nnz/r
12month1	12471	872622	22624727	1814.19
c8_mat11_I	4562	5761	2462970	539.89
cont11_l	1468599	1961394	5382999	3.67
diego-MM-573x230k	573286	230401	41694697	72.73
GL7d19	1911130	1955309	37322725	19.53
neos*	479119	515905	1526794	3.19
rail2586	2586	923269	8011362	3097.97
rel9	9888048	274669	23667183	2.39
relat9	12360060	549336	38955420	3.15
Rucci1	1977885	109900	7791168	3.94
spal_004	10203	321696	46168124	4524.96
tp-6	142752	1014301	11537419	80.82

Our codes were compiled with the Intel `icc` version 11 on **M1**, and `gcc`, version 4.3 on **M2**. In Section V-C we compare our results to that obtained with a publicly available CSB prototype ([2]). On both machines we compiled it using the Cilk++ compiler; version ("`Cilk Arts build 8503`"), based on the `gcc` (*GNU C Compiler*), v.4.2.4. To unify the

TABLE IV
TEST MACHINES.

machine	model	cpus× cores	data caches
M1	Intel Xeon 5670 6-Core 2.93GHz	2×6	2xL3,2x6xL2,2x6xL1: L3:12MB/16-w/64B L2:256KB/8-w/64B L1:32KB/8-w/64B
M2	AMD Opteron 2354 Quad-Core 2.2GHz	2× 4	2xL3,2x4xL2,2x4xL1: L3:2MB/32-w/64B L2:512KB/16-w/64B L1:64KB/2-w/64B

test environment, all codes were compiled using the `-O3` flag only (besides the OpenMP enabling flags).

V. RESULTS

We structure the analysis of results as in [6]. Note that, for brevity, we sometimes reference as RSB-k the k-threaded RSB. In most cases we start by commenting the 8 threaded performance, and proceed from discussing the particularly problematic cases to the best performing ones.

A. Results, Unsymmetric Matrices

For the unsymmetric matrices on **M1**, we observe an improvement when switching from RCSR to RSB in nearly all of the test set matrices; up to 67% on square ones, and up to 33% on non square ones (Fig. 1,2).

The only matrices "suffering" from the switch are: square *av41092* and *raefsky3* (Fig. 1), non square *c8_mat11_I* and *diego-smtxMM-573x230k*, and two borderline cases: *rail2586* and *sme3Dc*.

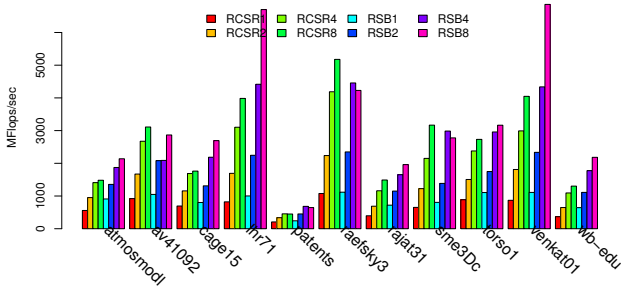

Fig. 1. Unsymmetric *SpMV* on **M1**, square matrices.

On machine **M2** (Fig. 3,4), we see improvements up to 128% for square matrices, and 65% for non square ones, and a single case of a performance drop: a 3% fall for the non square matrix *cont11_l*.

In Fig. 6,7,8,9 we observe index usage saving almost always. Out of 24 non-symmetric matrices, we experience three cases where index usage raises: square matrix *patents* (Fig. 6,8) and non square matrices, *rel9*, *relat9* (Fig. 7,9). We

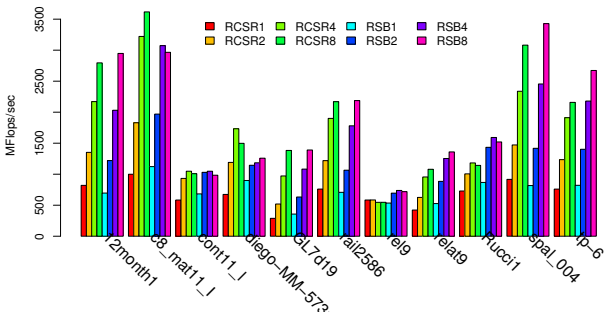

Fig. 2. Unsymmetric *SpMV* on **M1**, non square matrices.

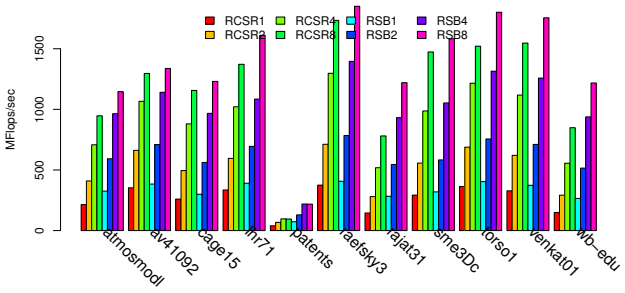

Fig. 3. Unsymmetric *SpMV* on **M2**, square matrices.

note, however, that the effect of RSB is actually an improvement of the performance on these matrices, notwithstanding the increased index usage. Among these matrices, problematic cases remain: *patents* performs better, but continues scaling poorly, (remaining the "slowest" of our entire test set); *relat9* suffers from poor scaling, too (especially on 8 cores **M2**); *rel9* continue not scaling at all.

These matrices have a feature in common: a very low nonzeroes/row elements ratio: 2.39 for *rel9*, 3.15 for *relat9* (see Table III) 3.97 for *patents* (see Table II). Although for such matrices one cannot expect high efficiency for either CSR or COO formats, we have realized why this is also the case for our recursive format (see [6], [5]), so now we present only the particular case for RSB.

Although very poorly performing, *patents*, actually scales up to 4 threads. In facts, *patents* is assembled in 37 COO leaves, regardless the thread count. When working with 8 threads, we observe that scaling is inhibited: this means that particular partitioning leaves a number of threads starving, while most of row intervals are *locked* by other threads. This is a situation occurring when the thread count approaches the number of submatrices in disjoint row intervals (see Fig. 5); and thus threads contend for available row intervals to operate on. In the current formulation of RSB, further partitioning of this matrix is not allowed, for it does not have enough

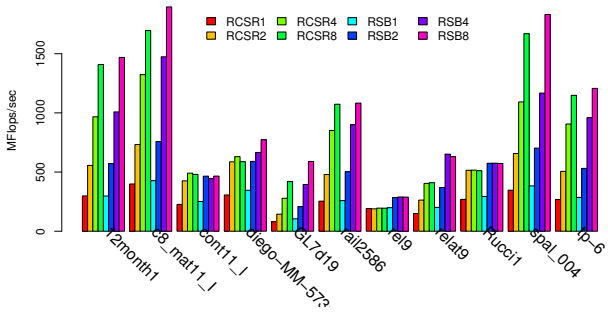

Fig. 4. Unsymmetric *SpMV* on **M2**, non square matrices.

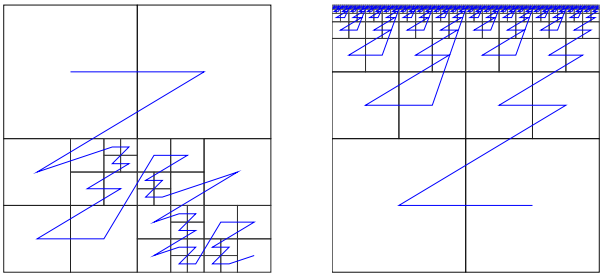

Fig. 5. On the left, matrix *patents* as partitioned on **M1**. On the right (widened, for viewing convenience) *diego-smtxMM-573x230k* on **M1**. Both in RSB format.

nonzeroes per row. On **M2**, the case for *patents* is similar: while on 1,2,4,8 threads, the matrix is partitioned respectively into 13,25,37,37 COO leaves.

The cases of *rel9* and *relat9* (Fig. 2,4) are similar. Since *relat9* has a little higher nonzeroes/row count than *rel9*, it succeeds in scaling in a limited way (up to 30 COO leaves, on both machines), but *rel9* gets partitioned in 7 leaves only, in all cases. Therefore, for *rel9*, more than 2 threads contend for row locking on 7 submatrices, with no possible scaling. Notice, however, that RSB is capable of allowing dual threaded parallelism in these *very sparse* cases, whereas RCSR was not.

The cases we have just discussed are worst/limit cases, and as such are not the primary target of our modifications, so we tolerate them here, and use them for comparison means.

Although quite different, two matrices (*sme3Dc*, *raefsky3*) suffer similar problems, when instantiated as RSB on **M1**. That is, while they are well-performing on RCSR and loosing index overhead from the RSB switch, they also get partitioned into less leaves, giving rise to the same *SpMV* scalability problem. In facts, while RCSR-8 partitions these matrices respectively into 115 (113 CSRH, 2 COOH) and 94 (CSRH) leaves, RSB-8 produces 16 (all CSRH) and 13 (11 CSRH, 2 COOH) leaves. Given the lock-based nature of our *SpMV* algorithm, and the distribution of submatrices, RSB-8 suffers from contention problems on both matrices. It is interesting to note that on **M2**, these matrices get subdivided respectively in 115 and 94 leaves, and we observe in Fig. 3 that this suffices to scale and experience, respectively, a 7% and a 6.7% improvement. Index overhead shifts from 4.44. to 2.55 bytes/nonzero for *sme3Dc*, and from 4.28 to 2.34 bytes/nonzero for *raefsky3*.

Matrix *av41092* on **M1** experiences the same problem *sme3Dc* and *raefsky3* did: insufficient partitioning. While **M1** partitions this matrix in 10 (9 CSRH, 1 COOH) submatrices only, **M2**, due to its smaller caches, partitions it in 72 leaves (64 CSRH, 8 COOH). So, the halving in index overhead experienced on **M1** (from 4.65 to 2.27 bytes/nonzero) could not bring advantage to RSB-8, while on **M2**, the 42% index saving (from 4.5 to 2.61 bytes/nnz) allows for scaling and a modest 3% performance increase.

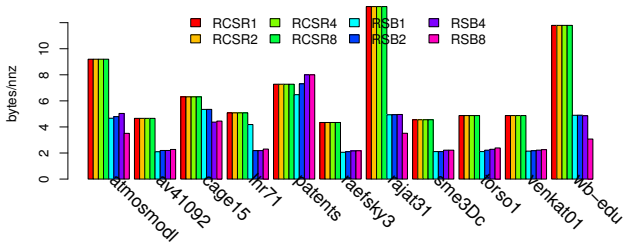

Fig. 6. Index storage requirement (in bytes) per nonzero on **M1** (square matrices).

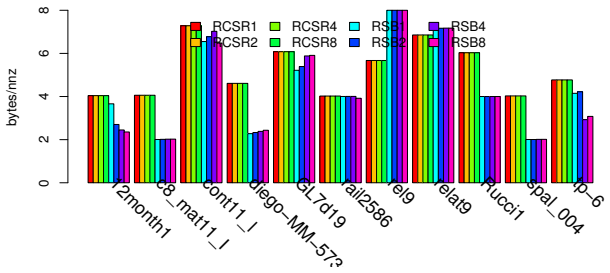

Fig. 7. Index storage requirement (in bytes) per nonzero on **M1** (non square matrices).

The remaining three cases with a missing improvement are non square matrices *c8_mat11_I*, *diego-smtxMM-573x230k*, and *rail2586* (Fig. 2). Matrix *c8_mat11_I*, alike to the matrices we have seen before on **M1**, suffers from poor partitioning, here: RSB partitions it in respectively 1,4,10,13 leaves for 1,2,4,8 threads. On 8 threads, the 13 leaves are not enough to ensure the parallel operation of all the threads, thus leaving some of them *starving*. Similarly to the previous cases, **M2**

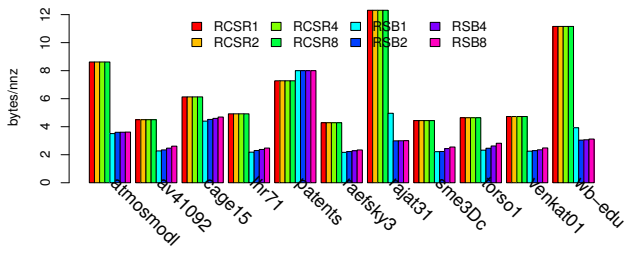

Fig. 8. Index storage requirement (in bytes) per nonzero on **M2** (square matrices).

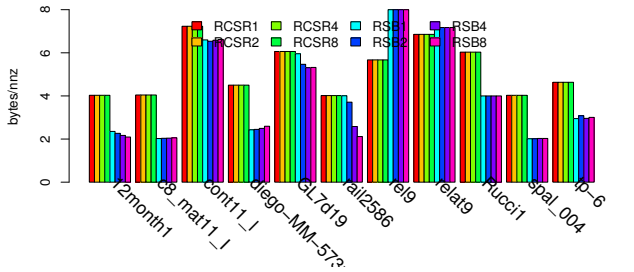

Fig. 9. Index storage requirement (in bytes) per nonzero on **M2** (non square matrices).

divides the matrix in much more leaves, thus avoiding the scaling problem.

The case for matrix *diego-smtxMM-573x230k* is different (and interesting). On **M1**, this matrix performs best as RSCR, while on **M2**, best as RSB. On both machines, though, while not scaling up to 8 threaded RCSR, it scales (although very *slightly*) for RSB, up to 8, but poorly. Poor scaling is evident: RSB-8 on **M1** is only 88% faster than RSB-1; on **M2**, only 123%. By looking at the number of submatrices, we could not say their number is too low. It is only after inspecting the distribution of submatrices (see Fig. 5), that we notice a big unbalance: actually, most of the submatrices are located on the top of the matrix, and it seems that RSB arranged submatrices in "block rows". Given the row-lock-based nature of our *SpMV* algorithm, such a distribution is enough to destroy the parallelism of the computation on this matrix. Here, after completing the bigger-dimensioned submatrices across various row intervals of the matrix, threads will try to acquire a lock on the intervals located on the upper border, with no success for most of them: only a few of them will be able to work at a time, on the upper submatrices. Contention will last during the whole computation for most of the threads, then, because our current *SpMV* algorithm has no mechanism for concurrent update of a single subvector.

Matrix *rail2586* constitutes another special case. For being *wide*, it fits particularly well when stored in a row-oriented storage as CSR. However, for having its nonzeroes scattered quite uniformly around the matrix, it would end up having very sparse submatrices, if it had not as much as 3097 nonzeroes per row, globally. But it happens that for being so wide, the proper introduction of CSRH leaves is only possible after a certain number of subdivisions. On **M1** (Fig. 12), it happens that there are not enough subdivisions for switching much of the submatrices to CSRH. So, the use of RSB for *rail2586* on **M1** does not lighten the index overhead significantly (it remains at about 4 bytes per nonzero), and the performance remains the same (notwithstanding the submatrices reduction: from RCSR's 352, to RSB-8's 55). For architectural reasons, RSB on **M2** ends up partitioning the matrix more finely, and thus falling to switch to CSRH in 335, out of the 352 leaves of RSB-8. The matrix is thus partitioned in number of matrices which is the double of RCSR's. However, in this case, the performance gain expected from RSB is negligible: less than 1%. We conjecture that the *flat* distribution of submatrices in the matrix, and its considerable width, cause a considerable overhead to the memory subsystem, which in turn is forced to continuously load elements from the right hand side vector, which would barely fit in the cache.

We notice that some matrices gain a considerable speedup from the RSB representation: *rajat31* (56%), *lhr71* (17%), *torso1* (18%) on **M2** (Fig. 4), *venkat01* (67%), *cage15* (50%) on **M1** (Fig. 2), *wb-edu* on both (68% on **M1**, 43% on **M2**). The assembled instances of these matrices as RSB differs from RCSR, for the relevant number of COO/COOH submatrices. On **M2**, *rajat31* gets partitioned in 1534 leaves, of which 896 COOH, and 126 COO; *wb-edu* in 4336 leaves, of which 2511 COOH, 254 COO; *torso1* in 357 leaves, of which 39 COOH; *lhr71* in 87 leaves, of which 34 COOH. In all these cases, index overhead is cut down approximately in a half. On matrices *rajat31* and *wb-edu*, index overhead falls down respectively from 12.3 to 3 bytes/nnz and from 11.15 to 3.12 bytes/nnz. This means that RSB *cures* cases where RCSR alone produced subdivisions abusing from CSR leaves; that is, producing CSR leaves with less nonzeroes than rows. The case for matrix *cage15* on **M1** is alike, in that it gets partitioned in 751 leaves, 132 of which are COO, 316 COOH, 6 CSR, 297 CSRH. With RSB, this configuration of *cage15* saves approximately 30% index overhead (from 6.3 bytes/nonzero), which is not much compared to other cases. So probably, the gain is due to the *fuller* submatrices (RSB-8 assembles 751 of them; RCSR as much as 4457). Performance gain on *torso1* is probably due only to index overhead saving: in RSB-8 on **M1**, it gets partitioned in 59 CSRH leaves only, (from 176 CSR), saving 64% of indexing overhead (from 4.6 bytes/nonzero, Fig. 6), which is quite good.

B. Results, Symmetric Matrices

Bar plots in Fig. 10 and 11 present the comparative performance results of RCSR and RSB for symmetric matrices. We observe performance enhancements nearly in all cases. There are three exceptions, though: *crankseg_1*, *ct20stif*, *F1* on **M1**.

We comment these exceptions first, and the remaining cases next.

On **M1**, matrix *F1* in RSB (Fig. 10) does not scale from 4 to 8 threads. On less than 8 threads, *F1* is processed faster with RCSR; e.g.: with 1 thread, *F1* gets partitioned by RSB in 10 submatrices only, all fullword CSR. But with 8 threads, RSB partitions *F1* in 72 leaves, of which 70 are CSRH and 2 COOH. With RCSR, a number of 573 leaves were obtained, which is much more. Given the higher number of subdivisions, load balancing in RCSR ran for sure smoother, while RSB did fall in a lock contention problem here, it seems. Please recall (See [5]) that our symmetric *SpMV* implementation variant incurs in a higher locking overhead than unsymmetric. On **M2**, the situation is almost reversed: for 8 cores, it is RSB that partitions *F1* in more leaves (573: 504 CSRH and 69 COOH), while RCSR divides the matrix in 278 leaves *only*. The index overhead of RCSR is quite high on *F1*: 5.08 bytes/nnz on **M2**, 5.4 on **M1**; on RSB it is always less than this, on both machines. However, the RSB index overhead depends on the threads count: on **M2** (Fig. 13) with more threads, the overhead tends to grow too, from 2.6 to 3.3 bytes/nnz, suggesting that further subdivisions could degrade performance. On the other hand, on **M1**, when going from 1 to 8 threads, this overhead decreases from 4.25 to 2.52 bytes/nnz (Fig. 12). These observations suggest us that the performance improvement over 1-core RCSR (on both **M1** and **M2**) is due to less index overhead, which itself is a consequence of less submatrices fragmentation. We believe that some *optimum partitioning* for 8 cores *F1* is between all of these four instances of RCSR/RSB on **M2/M1**; that is, the algorithm should have partitioned *F1* less coarsely on RSB/**M1**, more coarsely on RCSR/**M1**, and so on.

The cases for matrices *ct20stif* and *crankseg_1* (still on **M1**) are different. With *ct20stif* we observe that 2-threaded RSB fails from partitioning, thus cutting off two-cores parallelism completely (Fig. 10). On more cores the heuristic succeeds partitioning the matrix, but too coarsely to gain a sufficient workload balance. Please note that this matrix is among the smallest in our test set ($1.3 \cdot 10^6$ nonzeroes), stressing the limit of our rule of thumb (*sizing* matrices around the cache sizes). On both **M1** and **M2** machines, index usage for *ct20stif* keeps very low: for RSB it ranges from 2.27 to 2.52 bytes per nonzero, coming from RCSR's approximate 4.5. With an analogy to the previous case, on machine **M2**, partitioning is finer than on **M1**, from the single thread case on (1-threaded RSB partitions *ct20stif* to 7 submatrices), and an adequate workload balancing follows. Thus with *ct20stif* on **M2**, we do not loose the 8 threaded case, and RSB's performance is higher than RCSR's. Here, the sparser leaf submatrices are assembled as COOH (2 out of 7 on 8 cores **M1**, 2 out of 60 on **M2**), the remaining ones in CSRH. Notice that both *F1* and *ct20stif* matrices had more than 25 nonzeroes/row, which is quite sufficient to achieve good results with RCSR/RSB. Matrix *crankseg_1* is a little bit sparser (10 nnz/row). It suffers from the same *poor partitioning* problem on **M1**, having respectively 3,10,16,39 leaves for 1,2,4,8 threads, and loosing 30% of performance on 8 threads. On the other hand, on **M2**, matrix *crankseg_1* performs quite well, achieving an improvement to RCSR. The improvement itself is about 21% on 8 cores, when the matrix is partitioned in 37 COOH and 202 CSRH submatrices.

After having discussed the problematic cases, let's look at the remaining ones.

In one case there is almost no change: *nd24k* on **M1** (Fig. 10). Here, RCSR partitions the matrix in 503 CSR leaves, RSB in 87 CSRH leaves. The index overhead (Fig. 12) gets almost halved (from 4 bytes bytes/nonzero). We are not aware of the reason for the missing performance increase, here, but note that this is our symmetric matrix with the higher nnz/row count (199, see table I). On **M2** (Fig. 11), the same matrix witnesses a slight (5%) speedup, while being partitioned by RSB in 503 (all CSRH, except 5 COOH ones) pieces, and 278 ones by RCSR. The index overhead (Fig. 13) similarly to that of **M1**, halves from RCSR (4.2 bytes/nnz) to RSB (2.1 bytes/nnz). We conjecture that the 87 leaves on **M1** somehow limited parallelism, but we would need to investigate further to confirm this.

In one case, on **M1**, RSB performance boosts up as high as 66%, when compared to RCSR: it is for matrix *s3dkq4m2* (Fig. 10). Here, RCSR partitions in 127 leaves, while RSB in 15 only (8 CSRH, 7 COOH). We observe the index overhead (Fig. 12) is almost halved, switching from RCSR to RSB (for > 1 threads). We deem that this speedup is due to a case in which the matrix offers caching potential (the whole result vector and a matrix portion): on **M2**, where the L3 cache is considerably smaller than on **M1**, the performance of *s3dkq4m2* improves by only 2%, passing from 63 leaves of RCSR to 120 CSRH and 7 COOH leaves of RSB. Performing a run with *cold caches* (that is, making sure that any location caching the matrix or the involved vectors gets overwritten between each *SpMV*), on **M1** the performance of RSB is approximately 7% lower, while on **M2** it made no difference (and the boost becomes 55%, rather than 66%). Please note that the smallest symmetric matrix in the test set is not *s3dkq4m2* but *ct20stif*, which we have commented before.

When switching from RCSR to RSB on **M2** (Fig. 11), we observe speedups in all cases. Probably, L3 cache on **M2**, smaller than on **M1**, induced too coarse partitionings, thus limiting the scalability of our symmetric *SpMV*.

We can now comment the cases where the biggest improvement was observed: *af_shell10* (30%), *BenElechi1* (29%), *bone010* (24%), *fcondp2* (20%), *ldoor* (19%) on **M1** (Fig. 12), and *fcondp2* (28%), *crankseg_1* (21%), *ldoor* (16%), *F1* (12%) on **M2** (Fig. 13). For *af_shell10* on **M1**, we observe that RSB instantiates 255 submatrices (192 CSRH, 48 COOH, 15 COO), while RCSR used to instantiate 1534 CSR leaves. This matrix is also the one to experience the higher saving in index overhead: from 5.22 to 2.5 bytes per nonzero (more than 50%, Fig. 12). Matrix *BenElechi1* gets partitioned by RSB in 63 leaves: 32 CSRH, 30 COOH, 1 COO; by RCSR in 382 CSR matrices. Index usage (Fig. 12) halves: from 4.66 to 2.25 bytes per non zero. Similarly to the *af_shell10* case,

we experience a smaller number of leaf matrices, a more appropriate leaf matrix selection, and a consequent reduction in indexing overhead. On **M2** (Fig. 11), the same matrix improves only by 1.6%. By looking at its partitioning, we notice that it is partitioned in 127 leaves by RCSR, which is much less than RSB's 255 leaves (238 CSRH, 16 COOH, 1 COO). For *bone010*, RCSR assembles 1316 CSR matrices; RSB assembles 170 CSRH, 2 CSR, and 5 COO. Index usage is reduced down from 4.6 to 2.5 bytes/nnz (Fig. 13). On **M2**, RSB assembles 1054 CSRH, 279 COOH, and 5 COO submatrices, while RCSR allocates 630 CSR leaves (index overhead shifting from 4.53 to 2.55 bytes/nonzero). Again, it seems the partitioning proceeded too deeply. Matrix *fcondp2* is partitioned in 31 leaves (19 CSRH, 1 CSR, 11 COOH) with RSB, and with RCSR in 255 leaves. Index overheads falls from 4.63 to 2.42 bytes/nnz. On the same matrix, on **M2** the improvement is even higher, this time. Here, RSB partitions in 257 leaves (182 CSRH, 75 COOH), while RCSR in 127 leaves only. Index overhead falls from 4.56 to 2.5 bytes/nonzero. So, in contrast to the preceding cases, matrix *fcondp2* benefits from increased subdivision, on **M2**. Matrix *ldoor* is partitioned in 157 leaves (122 CSRH, 5 CSR, 26 COOH, 4 COO, 3.14 bytes/nnz) by RSB, and 789 leaves by RCSR (5.47 bytes/nnz, Fig. 12). On **M2**, the performance gain is smaller than on **M1** (16%, rather than 19%). Partitioning of *ldoor*, here, produces 804 (471 CSRH, 329 COOH, 4 COO) submatrices, while RCSR produces 431 leaves. Also index overhead falls more gently: from 5.30 to 3.36 bytes/nnz.

We conclude by observing that there is a strong correlation between the index saving and performance gain: milder index savings on **M2** showed milder performance improvements, while bigger index savings on **M1** were accompanied by higher improvements.

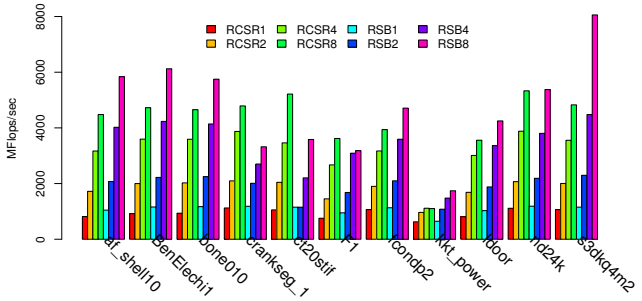

Fig. 10. Symmetric *SpMV* on **M1**.

C. Comparative analysis

Let us now look at the performance of all matrices as RCSR, RCSRH, and RSB, using 8 threads. For unsymmetric matrices, we also give performance results for the CSB prototype. Unfortunately, we had to skip matrix *cage15* (the one with the highest nonzeroes count), because CSB was unable to

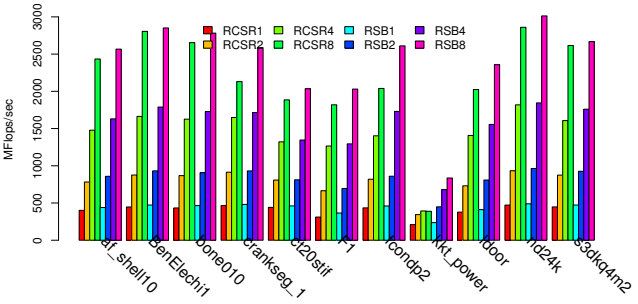

Fig. 11. Symmetric *SpMV* on **M2**.

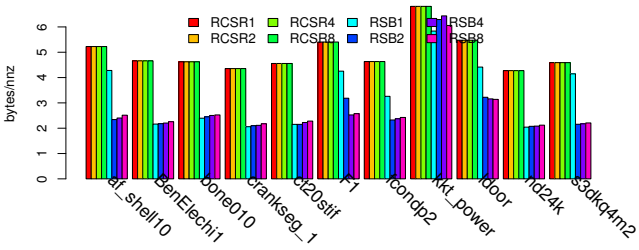

Fig. 12. Index storage requirement (in bytes) per nonzero on **M1** (symmetric matrices).

instantiate it (the CSB implementation needed more memory than the 24 GB available on **M1**).

We observe that for **M2** (Fig. 15): matrices which favor RSB most (over CSB) are *c8_mat11_I*,*spal_004*,*wb-edu*; one matrix looses against RCSR (*cont11_l*); the majority of RSB cases is faster than RCSR (19 matrices out of 20). Summarizing, RSB performs faster than CSB (and is also the fastest among the four cases) in 7 cases out of 20. CSB is the fastest in 12 cases; in one case it is faster than RSB, but not the fastest one.

On **M1** (Fig. 14) we observe that: RSB is much faster than CSB on *wb-edu* and *venkat01*; 6 matrices seem to perform very similarly in both CSB or RSB; the remaining ones perform better in one of the two formats. Some matrices loose performance in RSB, over RCSR: matrices *av41092*,*c8_mat11_I*,*cont11_l*; (slightly) *diego-smtxMM-573x230k*,*sme3Dc*; other matrices favor RSB over RCSR: about 15, out of 20.

For space reasons, we omit figures showing comparative performance for symmetric matrices on RCSR, RCSRH, RSB formats, but include some general comments.

On **M2**, we notice RSB as the fastest format 5 times out of 12; on **M1**, 4 times. Here, RCSRH is the fastest in 7 cases; in all cases, very near to RSB. On **M1**, we see a similar situation, but notice a performance degradation in some additional cases: they are due to the poor partitioning problem discussed in Section V-B. In no case RCSR was the fastest format for

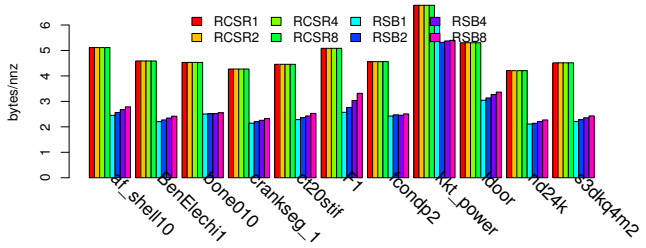

Fig. 13. Index storage requirement (in bytes) per nonzero on **M2** (symmetric matrices).

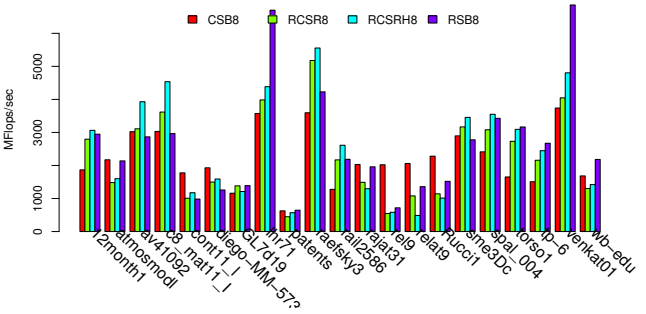

Fig. 14. Results for 8 cores on **M1**, comparing CSB, RCSR, RCSRH, and RSB (unsymmetric matrices).

symmetric matrices (exception made for the poorly scaling three matrices) on **M1**: (*crankseg_1*, *ct20stif*, *F1*).

VI. Conclusions and Future Work

In this paper, we have shown a possible improvement of our BLAS-oriented recursive storage for sparse matrices. We have found that, by using index compression and format diversification techniques, we can improve the floating point performance of *SpMV*. We have also found that, for unsymmetric matrices,

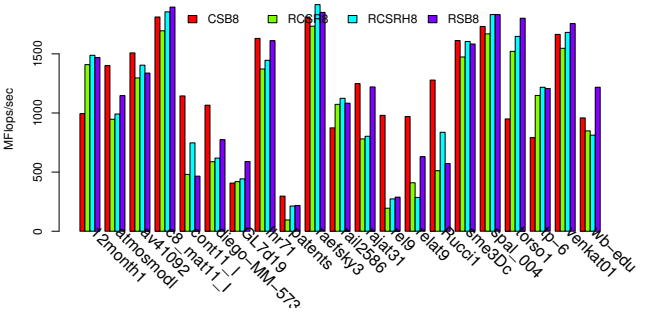

Fig. 15. Results for 8 cores on **M2**, comparing CSB, RCSR, RCSRH, and RSB (unsymmetric matrices).

the performance of our modified format (RSB) is comparable to that of a scalable sparse matrix format (CSB: currently for unsymmetric only). During comparison with RCSR and CSB, we noticed some particular cases that expose *weak points* of both RSB and RCSR; consequently allowing us to identify room for further improvement: (i) To redefine our format in order to obtain some estimate on the parallelism expected from a given partitioning (in Section V-B, we noticed that, despite the apparently adequate partitioning, some instances of matrices (e.g.: smaller symmetric) did not scale on 8-threaded *SpMV*). (ii) To modify the *SpMV* algorithm to be more parallel, by working around the need for row locking (e.g.: by using temporary vectors, as CSB does [2, Sec.4], although this may be challenging in our case). (iii) While our primary interest is focused on bigger matrices, tuning the partitioning algorithm for small matrices could prove useful to ensure parallelism in these cases, too. (iv) Properly subdividing matrices which are big, but with an extremely low nonzeroes/row ratio would be challenging (and fruitful), as well.

Some ideas we have introduced should be developed further. For instance, a more aggressive form of tuning could diversify index types at the *leaf level* and continue using traditional CSR or COO layouts, if profitable. Probably future architectures (with much higher number of cores, and even higher risks for stall due to higher memory latencies and longer instruction pipelines) would render such approaches advantageous.

In summary, we can state that our work illustrates that combinations of hierarchical indexing and index compression techniques can be useful to achieve high efficiency of computing on sparse matrices (on general purpose hardware). In this light, we see the RSB format as a candidate format for a complete multicore sparse BLAS implementation (that is, support for symmetric storage, solve operations, parallel transposed *SpMV*, etc.).

Finally, we would like to thank Jamie Wilcox and Victor Gamayunov from Intel EMEA Technical Marketing HPC Lab for their technical support during experiments.

References

[1] K. Kourtis, G. Goumas, and N. Koziris, “Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression,” *Computing Frontiers*, pp. 87–96, 2008.

[2] A. Buluc, 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 *SPAA*, F. M. auf der Heide and M. A. Bender, Eds. ACM, 2009, pp. 233–244.

[3] S. Williams, L. Oliker, 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*. ACM New York, NY, USA, 2007.

[4] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, “Utilizing recursive storage in sparse matrix-vector multiplication - preliminary considerations,” in *CATA*, T. Philips, Ed. ISCA, 2010, pp. 300–305.

[5] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, “On blas operations with recursively stored sparse matrices,” in *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, September 2010.

[6] ——, “On the usage of 16 bit indices in recursively stored sparse matrices,” in *Proceedings of the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, September 2010.

[7] 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 Edition*. Philadelphia, PA: SIAM, 1994.

[8] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, "About the assembly of recursive sparse matrices," in *Proceedings of the International Multiconference on Computer Science and Information Technology*, October 2010.

[9] "Standard for information technology— portable operating system interface (posix) (ieee std 1003.1)," 2008.