Improving the Performance of the Symmetric Sparse Matrix-Vector Multiplication in Multicore

Theodoros Gkountouvas*, Vasileios Karakasis†, Kornilios Kourtis‡, Georgios Goumas† and Nectarios Koziris†

†School of Electrical and Computer Engineering
National Technical University of Athens, Greece
E-mail: {bkk,goumas,nkoziris}@cslab.ece.ntua.gr

*Department of Computer Science
Cornell University, Ithaca, NY, USA
E-mail: tg294@cornell.edu

‡Department of Computer Science
ETH, Zurich, Switzerland
E-mail: kkourt@inf.ethz.ch

***Abstract*—Symmetric sparse matrices arise often in the solution of sparse linear systems. Exploiting the non-zero element symmetry in order to reduce the overall matrix size is very tempting for optimizing the symmetric Sparse Matrix-Vector Multiplication kernel (SpM×V) for multicore architectures. Despite being very beneficial for the single-threaded execution, not storing the upper or lower triangular part of a symmetric sparse matrix complicates the multithreaded SpM×V version, since it introduces an undesirable dependency on the output vector elements. The most common approach for overcoming this problem is to use local, per-thread vectors, which are reduced to the output vector at the end of the computation. However, this reduction leads to considerable memory traffic, limiting the scalability of the symmetric SpM×V. In this paper, we take a two-step approach in optimizing the symmetric SpM×V kernel. First, we introduce the CSX-Sym variant of the highly compressed CSX format, which exploits the non-zero element symmetry for compressing further the input matrix. Second, we minimize the memory traffic produced by the local vectors reduction phase by implementing a non-zero indexing compression scheme that minimizes the local data to be reduced. Our indexing scheme allowed the scaling of symmetric SpM×V and provided a more than 2× performance improvement over the baseline CSR implementation and 83.9% over the typical symmetric SpM×V kernel. The CSX-Sym variant has further increased the symmetric SpM×V performance by 43.4%. Finally, we evaluate the effect of our optimizations in the context of the CG iterative method, where we achieve an 77.8% acceleration of the overall solver.**

***Index Terms*—Sparse Matrix Vector Multiplication, symmetric sparse matrices, multicore optimization, compression, SpMV**

I. Introduction

Sparse matrices are two-dimensional matrices primarily populated with zeros. They are routinely used in the solution of large linear systems as part of complex physical simulations based on finite element methods. One of the most time-consuming kernels in iterative solution methods for linear systems is the Sparse Matrix-Vector Multiplication or SpM×V, which computes the product of a sparse matrix with a dense vector [1]. The SpM×V kernel has been characterized as one of the most important computational kernels in science and engineering at least for the current decade [2]. The algorithmic characteristic that renders the study of SpM×V challenging is its very low flop:byte ratio [3]–[5], which very often translates to bottlenecks in the memory hierarchy. This adds a significant overhead in the execution of SpM×V, especially in multithreaded contexts, where the memory bandwidth is shared among the cores, leading to poor scaling and mediocre performance. Storage formats that compress the sparse matrix representation are very common and provide an efficient way of achieving performance gains in SpM×V. The most widely used storage format is the *Compressed Sparse Row* or CSR format [6]. The idea behind CSR is that it suffices to store only the column indices of each non-zero element and 'pointers' to the start of each row for locating each non-zero element of the matrix.

In this paper, we focus on SpM×V for symmetric sparse matrices, which comprise a large subcategory of sparse matrices in real-life applications. Since SpM×V is bound from memory bandwidth in most of the cases, it is tempting to store only the lower triangular submatrix and its main diagonal, reducing therefore the matrix size approximately to the half. This format, known as *Symmetric Sparse Skyline* or SSS format [6], can yield a significant performance improvement over the typical CSR in a single-threaded execution. However, efficient multithreaded implementations of CSR-derived symmetric formats can be a hard task [7], [8]. The key problem in the typical symmetric SpM×V implementation is a race condition during the writing to the output vector, which can be resolved either with locking (prohibitive cost) or with the use of local, per-thread, output vectors [7], [9]. The use of local vectors requires a final reduction step after the SpM×V operation is complete, in order to reduce the local vectors to the final output vector. This step can limit parallelism significantly, chiefly due to the considerable increase in memory traffic[1]. The cost of reduction, therefore, may easily outweigh the benefits of the increased compression achieved by the exploitation of the non-zero elements symmetry and poses severe restrictions to the scalability of the kernel for a larger numbers of threads.

Our approach in optimizing the symmetric SpM×V kernel consists of two steps. First, we introduce a variant of the *Compressed Sparse eXtended* (CSX) [10] format that supports symmetric sparse matrices; we name this variant CSX-Sym. The CSX storage format is able to achieve high compression levels by detecting and encoding a variety of different non-

[1]Theoretically, this step limits parallelism from $\Theta(NNZ)$ to $\Theta(N)$, but since for sparse matrices N is adequately large, it is not a major problem in practice.

zero element substructures (e.g., horizontal, diagonal, 2-D block etc.), leading to considerable performance improvements, while the CSX-Sym variant exploits also the non-zero element symmetry, in order to achieve even higher compression levels. Second, we introduce a non-zero indexing compression scheme for the local vectors for minimizing the memory traffic of the reduction phase of the symmetric SpM×V. Our indexing scheme accumulates only the necessary elements into the corresponding output vector elements; given the high sparsity of the local vectors, the proposed indexing scheme is able to eliminate the overhead of the reduction phase, allowing the symmetric SpM×V to scale. More specifically, the non-zero indexing scheme, when applied to the typical SSS SpM×V implementation, is able to provide a more than 2× performance improvement in SMP systems and surpasses 40% in NUMA architectures. It also allows SpM×V to scale, while the performance of the baseline SSS falls even below CSR in highly multithreaded contexts. Additionally, the aggressive compression of the input matrix achieved by CSX-Sym format provides a further 43.4% and 10% improvement in SMP and NUMA architectures, respectively. In summary, the proposed symmetric SpM×V optimizations (CSX-Sym + local vectors indexing) were able to provide a more than 2× performance acceleration of the typical CSR implementation. Finally, the integration of the CSX-Sym storage format into a baseline non-preconditioned CG implementation yielded a 77.8% improvement of the total execution time of the solver in an SMP system and 28.5% in a NUMA architecture.

The rest of the paper is organized as follows: Section II provides background information about CSR, SSS, symmetric SpM×V and CG. Section III presents the proposed technique for optimizing the local vectors method in symmetric SpM×V, Section IV presents the CSX-Sym variant of CSX and the integration with the optimized local buffer implementation. Section V presents the experimental performance evaluation, as well as the impact of our optimizations on the performance of the CG iterative method. Finally, Section VI discusses the related work in the field of symmetric SpM×V optimization and Section VII concludes the paper.

II. Background

A. The CSR storage format

The most widely used storage format for sparse matrices is the Compressed Sparse Row (CSR) format [6]. CSR uses three arrays for storing a sparse matrix (Fig. 1): the `values` array stores the values of the non-zero elements of the matrix in row-wise order, the `colind` array stores the corresponding column indices and `rowptr` contains 'pointers' to the start of each row. Assuming four-byte indices for the `rowptr` and `colind` arrays and double-precision floating point non-zero values, the size of an $N \times N$ sparse matrix with NNZ non-zero elements in the CSR format will be

$$S_{CSR} = 12NNZ + 4(N+1) \quad (1)$$

The compression potential of CSR is rather limited, since it only compresses the row representation. Moreover, in symmetric matrices almost half of CSR data is redundant, requiring therefore more specialized formats that exploit the benefits of non-zero elements symmetry.

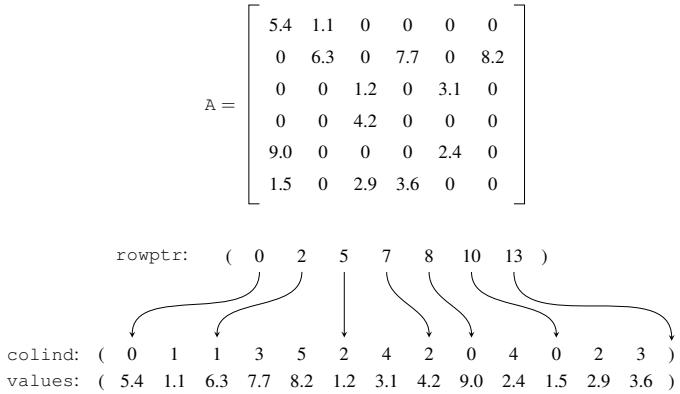


Fig. 1. The CSR sparse matrix storage format.

B. The SSS storage format

Finite element methods usually involve the solution of large linear systems with sparse, structured, symmetric matrices. The most common approach for storing symmetric sparse matrices are variations of the CSR format [6], [7], [11]–[13], where only the non-zero elements of the upper or lower triangular submatrix are stored, almost reducing to the half the overall matrix size. The most established variation for storing symmetric sparse matrices is the CSR-based Sparse Symmetric Skyline (SSS) format [13] (Fig. 2). SSS stores the elements of the main diagonal of the matrix separately in an N-size array, called `dvalues`, and the remaining non-zero elements of the lower triangular matrix using the standard CSR format. As a result, the size of the `values` and `colind` arrays is reduced to $(NNZ - N)/2$. The size of a symmetric matrix stored in the SSS format is therefore given by the following equation:

$$S_{SSS} = 6(NNZ + N) + 4 \quad (2)$$

Assuming $NNZ \gg N$, which is a typical case for SpM×V applications, the SSS format is able to half the matrix representation size. This generous size reduction can yield significant performance benefits in the serial execution of the SpM×V kernel, but the nature of the symmetric SpM×V kernel complicates the multithreaded execution, an issue we discuss in more detail in Section III.

C. The Conjugate Gradient Method

The Conjugate Gradient method (CG) is an iterative algorithm for the solution of linear systems of the form $y = Ax$, where A is a symmetric positive definite coefficient matrix. Originally proposed by Hestenes and Stiefel [14], its variations have become the standard in the solution of large sparse linear systems. Algorithm 1 shows a typical non-preconditioned CG implementation. The key idea of the algorithm is to start from an initial guess x_0 of the system's solution and proceed by computing a new guess x_i at each iteration, until the *residual vector* $r_i = b - Ax_i$ becomes adequately small.

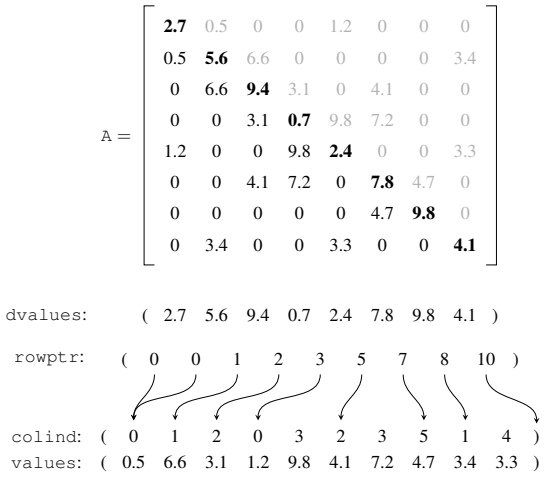


Fig. 2. The SSS symmetric sparse matrix storage format.

Theoretically, CG is guaranteed to converge after N iterations, N being the size of the system [1]. In practice, however, the calculation of residual vector starts to lose in accuracy, due to roundoff errors, and convergence is not guaranteed eventually. Therefore, CG can perform poorly in some matrices, while in others it is able to provide a solution within i iterations, $i \ll N$.

1: **procedure** CG (A, b, x_0)
   x_0: vector that is an initial approximate solution of $A \cdot x = b$
2: $r_0 \leftarrow b - A \cdot x_0$ ▷ *Residual vector*
3: $p_0 \leftarrow r_0$
4: $i \leftarrow 0$
5: **loop**
6: $a_i \leftarrow \frac{r_i^T \cdot r_i}{p_i^T \cdot A \cdot p_i}$ ▷ *SpM×V operation*
7: $x_{i+1} \leftarrow x_i + a_i \cdot p_i$ ▷ *New approximate solution*
8: $r_{i+1} \leftarrow r_i - a_i \cdot p_i$ ▷ *Update residual*
9: **if** r_{i+1} is adequately small **then** exit **loop**
10: $b_i \leftarrow \frac{r_{i+1}^T \cdot r_{i+1}}{r_i^T \cdot r_i}$
11: $p_{i+1} \leftarrow r_{i+1} - b_i \cdot p_i$
12: $i \leftarrow i + 1$
13: **return** x_{i+1}

Alg. 1: The Conjugate Gradient method algorithm.

During an iteration, the CG algorithm performs a series of vector operations (dot products, additions etc.) and an SpM×V product (Alg. 1, line 6). In the case of non-preconditioned CG implementation for multicore architectures, the SpM×V operation tends to be the most expensive, taking up the majority of the CG execution time [1], [15]. In the case of a preconditioned CG implementation, the preconditioning process can be quite expensive, depending on the exact preconditioner used, while in highly parallel distributed implementation, the dot products become also quite significant. In this paper, we focus on a simple, non-preconditioned CG implementation as a baseline iterative method for solving linear systems in multicores, since improving the performance of a preconditioner is orthogonal to the SpM×V optimization examined in this paper.

III. The reduction phase of the symmetric SpM×V

The serial version of the symmetric SpM×V is straightforward and is described in Alg. 2. Iterating over the rows of the matrix, we first compute the product of the corresponding diagonal elements and then proceed with the computation of the rest of the products in a CSR-like way. The important difference from the simple CSR SpM×V is that we must also compute the product of every symmetric element (Alg. 2, line 7).

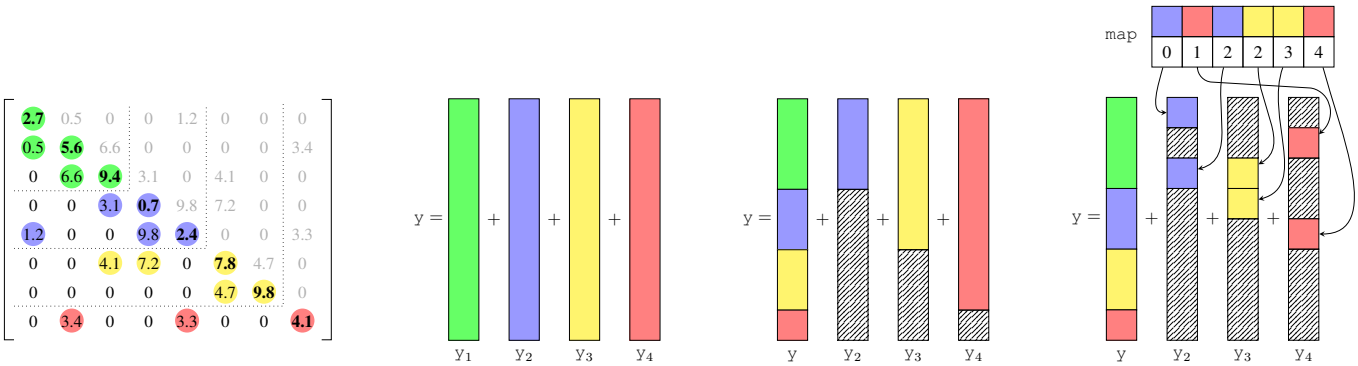
1: **procedure** SSSSPMVSERIAL(A, x, y)
   A: $N \times N$ symmetric sparse matrix in SSS format
2: **for** $r \leftarrow 0$ **to** N **do**
3: $y[r] \leftarrow dvalues[r] \cdot x[r]$
4: **for** $j \leftarrow rowptr[r]$ **to** $rowptr[r+1]$ **do**
5: $c \leftarrow colind[j]$
6: $y[r] \leftarrow y[r] + values[j] \cdot x[c]$
7: $y[c] \leftarrow y[c] + values[j] \cdot x[r]$
8: **end for**
9: **end for**

Alg. 2: Serial implementation of SpM×V using the SSS format.

A. Local Vectors Method

Since SpM×V is bound from memory bandwidth, storing a symmetric matrix using a condensed format, like SSS, is expected to lead to considerable performance improvement over the conventional CSR storage for large matrices. Although this is true for the serial version, the case of the multithreaded version is more complicated. Similarly to CSR, the input matrix in SSS is assigned to threads row-wise ensuring an approximately equal number of non-zero elements per partition (Fig. 3a). Unfortunately, the accesses to the output vector are no more independent, since the calculations for the upper triangular elements (Alg. 2, line 7) may write on output vector elements assigned to other threads. These conflicts can be avoided either with the use of locks or with the use of local output vectors per thread. The cost of locks in such a large extent would be prohibitive and serialize the accesses on the output vector. The second option, however, is more viable, since the accesses in the local output buffers can be performed independently. Nonetheless, an additional reduction step must be performed to compute the final output vector from the local vectors partial results (Fig. 3b). This reduction step can be easily performed in parallel, thus reducing the additional overhead.

Algorithm 3 shows the multithreaded SpM×V implementation using the SSS format. Each thread performs the SpM×V operation for its own partition writing the result in its local output vector. At a second phase (lines 12–15), the local buffers are reduced to the output vector in parallel. During this phase, the output vector and the local buffers are split equally row-wise and each thread is responsible for the reduction of its corresponding part of the output vector.



(a) Sample matrix. (b) Naive. (c) Effective ranges. (d) Indexed.

Fig. 3. Local vector methods for the reduction phase of the symmetric SpM×V kernel. An example implementation of symmetric SpM×V with $p = 4$ threads. The naive method (b) uses simply four local buffers that reduces later to the final output vector. The effective ranges method (c) uses $p - 1$ local vectors writing only the possibly conflicting regions. The indexing scheme proposed 3d uses $p - 1$ local vectors and an indexing structure that points only to the really conflicting elements.

1: **procedure** SSSSPMV(A, x, y, n, $start[n]$, $end[n]$)
   A: $N \times N$ symmetric sparse matrix in SSS format
   n: number of threads utilized
   $start[n]$, $end[n]$: region for each thread
2: **for** $i \leftarrow 0$ **to** n **do in parallel**
3: **for** $r \leftarrow start[i]$ **to** $end[i]$ **do**
4: $y_i[r] \leftarrow dvalues[r] \cdot x[r]$
5: **for** $j \leftarrow rowptr[r]$ **to** $rowptr[r+1]$ **do**
6: $c \leftarrow colind[j]$
7: $y_i[r] \leftarrow y_i[r] + values[j] \cdot x[c]$
8: $y_i[c] \leftarrow y_i[c] + values[j] \cdot x[r]$
9: **end for**
10: **end for**
11: **end for**
12: **for** $r \leftarrow 0$ **to** N **do in parallel**
13: **for** $i \leftarrow 0$ **to** n **do**
14: $y[r] \leftarrow y[r] + y_i[r]$
15: **end for**
16: **end for**

Alg. 3: Multithreaded implementation of the symmetric SpM×V kernel using the SSS storage format.

The reduction phase is seemingly memory bandwidth bound. Assuming p participating processors and a symmetric matrix of rank N, the reduction phase performs $\Theta(pN)$ floating point operations for $\Theta(pN)$ memory accesses, leading to $\Theta(1)$ flop:byte ratio. An important side-effect of the local vectors method is that the working set of the symmetric SpM×V increases linearly with the number of threads. Specifically, assuming double precision values for the vectors, the working set overhead will be

$$ws = 8pN \tag{3}$$

For a small number of threads, the increase in the working set introduced by the reduction phase will be insignificant, since $N \ll NNZ$ in most of the cases. However, as the number of threads increases, the total size of the local vectors becomes comparable to the matrix size, incurring a significant overhead in the execution of the SpM×V kernel. It is crucial, therefore, to reduce the working set overhead of the reduction phase, in order to allow SpM×V to scale beyond a certain number of threads.

B. *Effective ranges of local vectors*

The naive local vectors reduction method is suboptimal, since the full range of the output vector needs to be accessed by every thread. The method of effective ranges, proposed by Batista et al. [7], updates only a specific region of each local vector and redirects the rest of updates directly to the output vector. According to the symmetric SpM×V algorithm (Alg. 3), the i-th thread is assigned the calculations for the SSS submatrix between the $start[i]$ and $end[i]$ rows. Since we store the lower triangular matrix, there is no way for the thread i to access elements below the $end[i]$ row boundary. Therefore, it is safe to exclude this region from the reduction phase. Furthermore, thread i can obtain access to the output vector between the $start[i]$ and $end[i]$ elements directly without ruining SpM×V consistency, exactly as in the regular unsymmetric SpM×V implementation. The SpM×V operations at the remaining region, from the first row up to $start[i]$, may conflict with the output vector updates and, therefore, the algorithm must write to the local vector; this region is the *effective region* of the local vector (Fig. 3c) and should be updated during the reduction phase. Assuming, without loss of generality, that each thread obtains almost the same number of rows, the working set overhead of the reduction phase for this method using p threads can be calculated as follows:

$$ws_{\text{eff}} \approx 8\frac{p(p-1)}{2} \cdot \frac{N}{p} = 4(p-1)N \tag{4}$$

The reduction overhead is now halved compared to the naive version, but the key problem of the symmetric SpM×V execution remains: the overhead of the reduction phase still grows linearly with the number of participating threads.

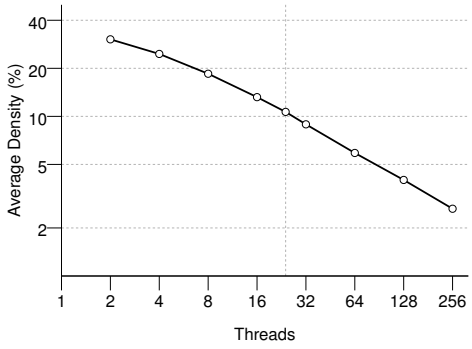


Fig. 4. The density of the effective regions of local vectors. Local vectors become more sparse as the thread count increases, reaching a 2.7% density at 256 threads. The vertical line marks the density at the 24 threads of the Dunnington system.

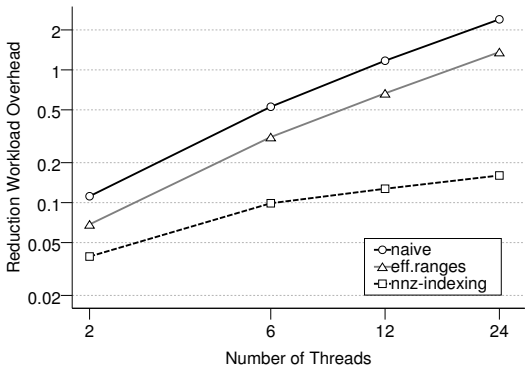


Fig. 5. The workload overhead of the reduction phase (over the serial SSS implementation) for the three local vectors methods considered. The overhead of the indexing scheme proposed tends to stabilize as the thread count increases.

C. Local vectors indexing

Our approach on minimizing the reduction phase overhead is based on the observation that the effective regions of the local vectors are quite sparse, i.e., very few elements of the effective regions are actually updated during the SpM×V computation. Another interesting trait of the effective regions, as depicted in Fig. 4, is that their density is continuously decreasing as more threads are added to the SpM×V computation, reaching an average of 10.7% at the 24 threads of our Dunnington system and 2.6% at 256 threads.

Motivated by this behavior, we introduce an indexing scheme for the non-zero elements of the local vectors, in order to update only the conflicting elements during the reduction phase, minimizing the workload overhead. For each non-zero element in the effective regions of the local vectors (Fig. 3d), we keep a pair (vid, idx), where vid is a unique local vector ID and idx is the index of the non-zero element inside the local vector. The size of idx equals the matrix index size, i.e., four-bytes, while vid can vary depending on the maximum expected thread count. In our implementation, we use generously four bytes for the vid field, but two or even a single byte is enough for current multicore architectures.

The workload overhead of the reduction phase using our indexing scheme is now dependent on the density d of the effective regions of the local vectors. More specifically, assuming, without loss of generality, that the N matrix rows are equally partitioned among p threads, the working set overhead is

$$ws_{\text{idx}} = ws_{\text{eff}} d + \underbrace{8\frac{(p-1)Nd}{2}}_{\text{index size}} \tag{5}$$

$$\approx 8(p-1)Nd \tag{6}$$

Although, theoretically, the local vectors indexing does not decouple the reduction phase overhead from the thread count, in practice, the effect of the thread count increase is considerably attenuated. This is due to the inverse relation between the thread count and the density of the effective regions, which eventually tends to stabilize the workload overhead as the thread count increases and is visually depicted in Fig. 5. The reduction overhead of both the naive and the effective ranges methods increases linearly with the thread count, exceeding significantly the multiplication phase at the 24 threads. The overhead of the indexing scheme proposed, on the other hand, increases at a much slower pace and tends to stabilize around 15% at the 24 threads. Indeed, with an average density of almost 11% at this thread count, the indexing scheme's working set overhead is more than four times lower than the case of the effective ranges.

Parallelization: The parallelization of the reduction phase in our index-based scheme is based on the local vectors index, since this specifies the actual reduction operations. More specifically, we first sort the index in an ascending order of the idx field and then split it equally among the participating threads. The only restriction in the splitting process is that an idx value must not be shared among any pair of threads, in order to guarantee the independence of the updates to the final output vector.

IV. Extending CSX to symmetric matrices

A. Overview of the CSX format

The Compressed Sparse eXtended (CSX) [10] storage format is a highly compressed storage format for sparse matrices that is able to detect and encode multiple non-zero element substructures simultaneously (Fig. 6). This allows CSX to adapt to the specificities of each matrix and reach significant compression levels. In conjunction with an efficient runtime code generation, specifically tuned for SMP and NUMA architectures, CSX is able to provide significant performance improvements to the SpM×V kernel, exhibiting also a considerable performance stability.

CSX uses only two arrays for representing a sparse matrix, namely `ctl` and `values`, and discards both the `rowptr` and `colind` data structures of the typical CSR representation. The `ctl` data structure (Fig. 7) stores all the necessary metadata, while the `values` array contains the non-zero elements arranged in a row-major substructure-wise order. A sparse matrix in CSX is arranged in units. A CSX unit represents either a

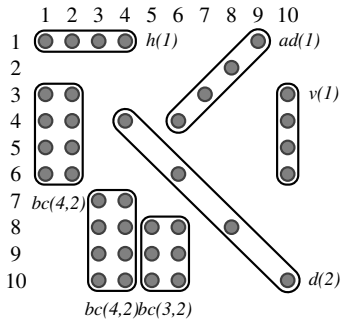


Fig. 6. Detection and encoding of non-zero elements substructures by CSX.

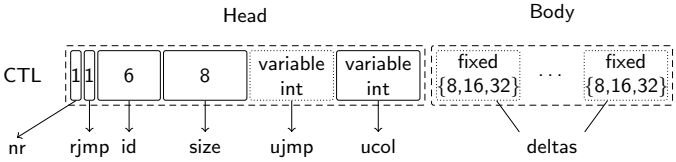


Fig. 7. The `ctl` byte-array used by CSX to encode the location information of the non-zero elements of a sparse matrix. Optional fields are denoted with a dotted bounding box.

substructure inside the sparse matrix (substructure unit) or a sequence of column index delta distances represented by the same number of bytes (delta unit). A CSX unit is comprised of two parts: the head and the body. The head contains the necessary information for denoting a new row (`nr`, `rjmp` and `ujmp` fields) and the column index of the first element of the unit, stored as a delta distance from the previous column in a variable size integer. When encoding a substructure unit, the body is empty. When encoding a delta unit, however, the body contains a sequence of the delta distances of the unit. A delta unit in CSX is a sequence of column indices that can be represented by a specified number of bits (namely, 8, 16 or 32 bits). A detailed description and performance evaluation of CSX can be found in [16].

B. CSX-Sym – Exploiting symmetry

CSX-Sym is the symmetric variant of the CSX format. It detects and encodes non-zero element substructures just as CSX, but only for the lower triangular part of the input matrix; every substructure in the lower triangular submatrix has a symmetric counterpart in the upper triangular one. For example, a *horizontal* substructure at the lower half implies also a *vertical* substructure with the same size and element values starting at the symmetric position in the upper half. Therefore, the description of the lower half substructures suffices for describing precisely the whole matrix.

CSX-Sym, apart from the `ctl` and `values` arrays of the typical CSX introduces a new array, `dvalues`, for storing the elements of the main diagonal, similarly to the SSS format. The only restriction we impose to CSX-Sym is that the writes incurred by a symmetric substructure must not be directed to both the local and the output vector. Such substructures are ignored from CSX-Sym and not encoded at all. The purpose of this restriction is to avoid the per-element check

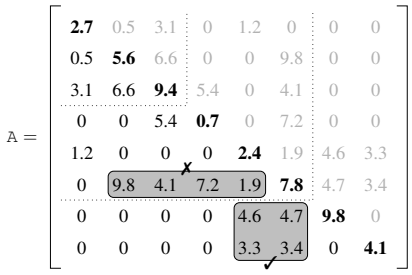


Fig. 8. Substructures encoded by CSX-Sym in an 8×8 matrix. The elements of the upper triangular submatrix are not stored; thread partitions are denoted with dotted lines. Legal substructures for CSX-Sym are marked with tick, while illegal ones are marked with a cross.

TABLE I
THE MATRIX SUITE USED FOR THE EXPERIMENTAL EVALUATION. ALL MATRICES ARE SQUARE AND POSITIVE DEFINITE. THE MAXIMUM POSSIBLE COMPRESSION RATIO (C.R.) FOR SYMMETRIC FORMATS IS DENOTED (NO INDEXING INFORMATION), ALONG WITH THE COMPRESSION RATIO ACHIEVED BY CSX-SYM. AS A MATTER OF REFERENCE THE SSS COMPRESSION RATIO DOES NOT EXCEED 50%.

Matrix	Rows	Nonzeros	Size (MiB)	C.R. (CSX-Sym)	C.R. (Max.)	Problem
parabolic_fem	525,825	3,674,625	44.06	49.6%	63.6%	C.F.D.
offshore	259,789	4,242,673	49.54	56.1%	65.3%	E/M
consph	83,334	6,010,480	69.10	63.9%	66.4%	F.E.M.
bmw7st_1	141,347	7,339,667	84.54	64.4%	66.2%	Structural
G3_circuit	1,585,478	7,660,826	93.72	60.2%	62.4%	Circuit
thermal2	1,228,045	8,580,313	102.88	53.4%	63.6%	Thermal
bmwcra_1	148,770	10,644,002	122.38	65.1%	66.4%	Structural
hood	220,542	10,768,436	124.08	64.4%	66.2%	Structural
crankseg_2	63,838	14,148,858	162.16	64.9%	66.6%	Structural
nd12k	36,000	14,220,946	162.88	64.9%	66.6%	2D/3D
inline_1	503,712	36,816,342	423.25	64.7%	66.4%	Structural
ldoor	952,203	46,522,475	536.04	64.5%	66.2%	Structural

of whether a multiplication must write to the output or to the local vector. Figure 8 shows an example of the CSX-Sym format. The highlighted *horizontal* substructure is not selected for encoding, since its symmetric substructure crosses the boundary of local and 'remote' writes. We should note here that the local vector indexing optimization of the symmetric SpM×V kernel is orthogonal to the CSX-Sym format.

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

For the performance evaluation of the CSX-Sym format and the local vectors indexing method, we have selected 12 matrices from the University of Florida Sparse Matrix Collection [17]. We include large symmetric and positive definite sparse matrices, in order for the CG method to be applicable. Table I summarizes the main characteristics of the matrices and presents the compression ratio achieved by CSX-Sym (the working set overhead of the reduction phase is not included). We use eight-byte double-precision floating point numbers for the non-zero element values and four-byte integers for the indexing information of the SSS format.

Our computational testbed comprises a quad-way six-core SMP system (24 cores, Dunnington) and a two-way quad-core

TABLE II
EXPERIMENTAL PLATFORMS. THE NUMBERS FOR THE SUSTAINED BANDWIDTH ARE OBTAINED WITH THE STREAM BENCHMARK.

	Dunnington	**Gainestown**
Model	Intel Xeon X7460	Intel Xeon W5580
Microarchitecture	Intel Core	Intel Nehalem
Clock freq.	2.66 GHz	3.20 GHz
L1 cache (D/I)	32 KiB/32 KiB	32 KiB/32 KiB
L2 cache	3 MiB (per 2 cores)	256 KiB (per core)
L3 cache	16 MiB	8 MiB
Cores/Threads	6/6	4/8
Peak Front-end B/W	8.5 GB/s	2× 30 GB/s
Sustained B/W	5.4 GB/s	2× 15.5 GB/s
Multiprocessor Configurations		
Processors	4	2
Cores/Threads (total)	24/24	8/16

NUMA system (8 cores, 16 threads, Gainestown). Table II presents the technical characteristics of our platforms in more detail. All systems were running a 64-bit version of the Linux OS. We used LLVM 2.9 for all the considered formats, in order to achieve a fair comparison. We should note here that beyond our initial expectations, LLVM 2.9 offered an average 5% performance improvement to the non-CSX[2] formats compared to GCC 4.5. For the parallelization of the SpM×V routines and the preprocessing phase of CSX, we used explicit, native threading with the Pthreads library (NPTL 2.7) and bound the threads to specific logical processors using the Linux kernel's system call interface. Finally, for the NUMA-aware implementations, we used the *numactl* library, version 2.0.7, in conjunction with our low-level interleaved allocator [16].

In order to ensure a fair comparison, we have built a common measurements framework that interfaces with the storage format implementations through a well-defined sparse matrix-vector multiplication interface. We performed 128 consecutive SpM×V operations with randomly created input vectors, swapping the input and output vectors at every iteration.

Our performance evaluation is organized in four parts. First, we evaluate the impact of local vectors indexing technique and compare it with the naive and the effective ranges methods. Second, we evaluate the performance of the CSX-Sym format and compare to the optimized SSS version. Third, we examine the behavior of the storage formats in reduced bandwidth matrices, and, finally, we evaluate the impact of the proposed optimizations in the CG iterative solver.

B. Performance of local vectors indexing method

The performance improvement achieved by the different local vector methods considered is depicted in the speedup diagrams of Fig. 9. All methods start with a significant improvement in the single-threaded configuration, especially in the Dunnington platform, but the naive and the effective ranges methods scale at a lower rate compared to the baseline CSR. The performance improvement offered by these methods is continuously shrinking as the thread count increases and is completely eliminated when the memory bandwidth is saturated. Due to its reduced working set overhead, the effective ranges method exhibits a slightly better behavior in such cases, but it still does not allow symmetric SpM×V to scale. In Gainestown, the performance of the naive and the effective ranges methods is very close to the original CSR at the eight-threaded configuration and deteriorates significantly at the 16-threaded.

The performance benefit of the proposed local vectors indexing method is prominent in both considered architectures. The considerably reduced working set overhead, which remains almost stable with the thread count, allows symmetric SpM×V to scale at the same rate as the original CSR implementation, without compromising the performance improvement in the cases of memory bandwidth saturation. More specifically, our indexing scheme achieves an 83.9% performance gain over the best SSS configuration in Dunnington (12 threads) and a 44% improvement in Gainestown. Overall, the symmetric SSS kernel using the local vectors indexing scheme was able to provide a more than 2× improvement over the standard CSR implementation in Dunnington and 1.5× in Gainestown in the multithreaded configurations. This is a rather significant achievement, since the proposed technique allows the efficient exploitation of the symmetric structure of certain matrices, which otherwise would remain unexploited.

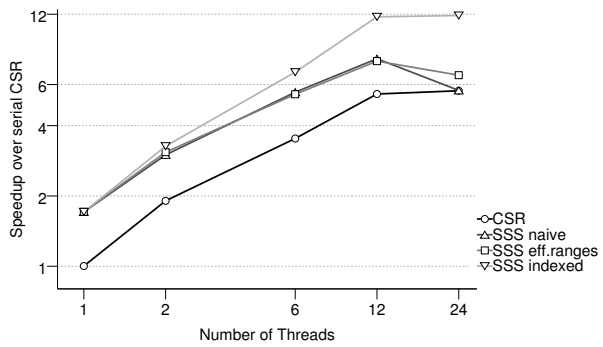
In order to provide more insight on the importance of the reduction phase in the symmetric SpM×V implementation, we present in Fig. 10 the execution time breakdown of the symmetric SpM×V kernel for all the considered reduction methods at the 24-threaded configuration in Dunnington. It is clear that the proposed local vectors indexing scheme reduces considerably the reduction phase overhead, keeping it at a minimal level. This reduction method has a beneficial side-effect: the multiplication phase has also decreased with the proposed indexing scheme. This is mainly due to the lower cache interference introduced by the modest working set overhead of our method. The high working set overhead of the alternative methods in many-threaded configurations is likely to spill out useful data from the cache, incurring an increased overhead to the multiplication phase of the next iteration.

Finally, it is essential to point the four cases (*parabolic_fem*, *offshore*, *G3_circuit*, *thermal2*) that CSR's performance surpasses or reaches our indexing method. These matrices are high-bandwidth matrices with a lot of their non-zero elements lying at very long distances from the main diagonal, leading to considerable memory traffic during the reduction phase. However, the local vectors indexing method seems to handle efficiently even such cases, exhibiting a rather low overhead, while the naive and effective ranges methods are overwhelmed by the reduction cost.

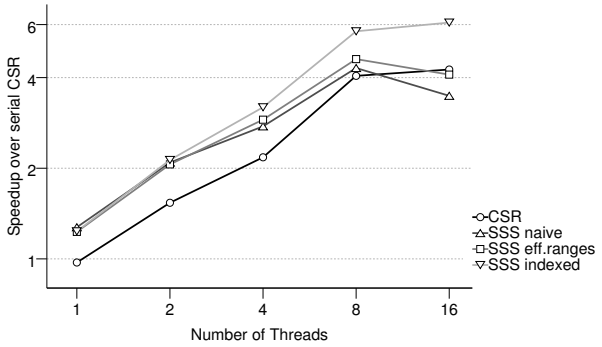
C. Evaluation of the CSX-Sym variant

The performance of the CSX-Sym variant and the optimized SSS format (local vectors indexing) compared to the

[2]CSX routines are generated using LLVM by default.



(a) Dunnington



(b) Gainestown

Fig. 9. Symmetric SpM×V speedup with different local vectors reduction methods. The significant memory traffic incurred by the naive and the effective ranges technique as the number of threads increases eliminates any benefit from the gain of the symmetric storage.

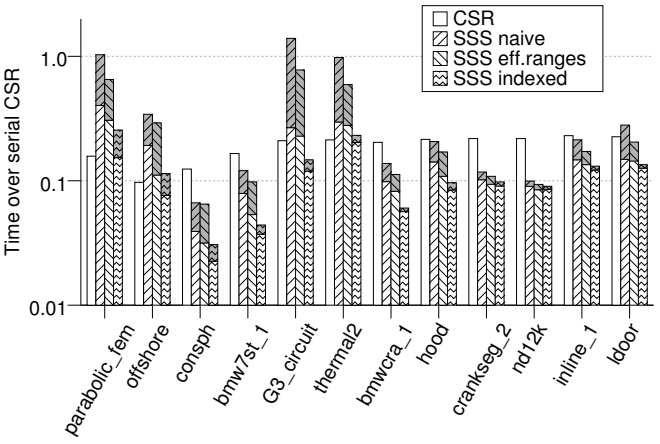


Fig. 10. Symmetric SpM×V execution time breakdown at 24 threads in Dunnington. The reduction overhead (shaded regions) is considerably reduced with the use of local vector indexing.

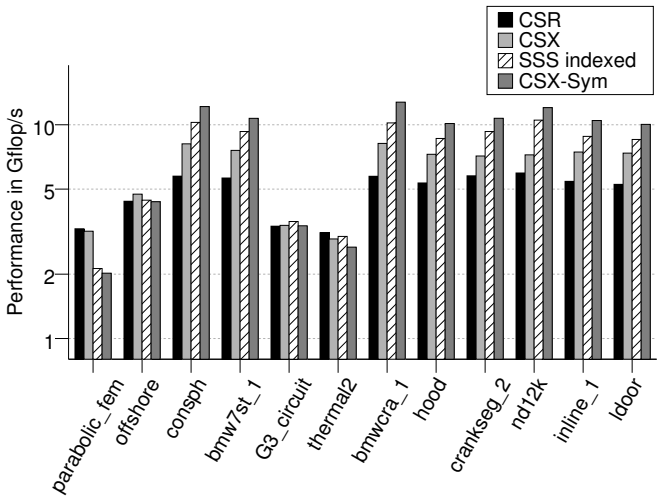


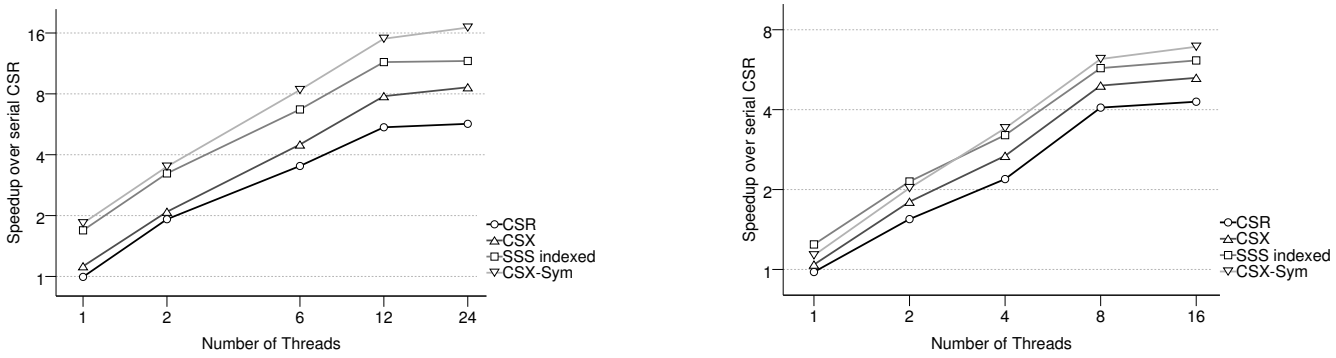
Fig. 12. Per-matrix performance for the CSX-Sym format at 16 threads in Gainestown. CSX-Sym's performance in more regular matrices surpasses 10 Gflop/s, while staying close to baseline CSR performance in less regular ones.

unsymmetric CSR and CSX implementations are depicted in Fig. 11 (see next page). Thanks to its highly compressed representation, CSX-Sym provides a 43.4% further performance improvement over the optimized SSS format in Dunnington, an architecture that is mostly affected by the matrix size representation. In Gainestown, where the available memory bandwidth is ample, the performance gap closes to 10% on average, but CSX-Sym is still able to provide a performance gain. The unsymmetric CSX and CSR implementations are well below in performance, especially in Dunnington, where the memory bottleneck is more prominent.

In order to gain a further insight in the CSX-Sym performance, Fig. 12 shows the per-matrix performance of the considered formats at the 16-threaded configuration in Gainestown. CSX-Sym manages to achieve the best performance, surpassing 10 Gflop/s, in 8 matrices from our suite. The remaining four matrices are the high-bandwidth corner cases, where no symmetric format did achieve performance improvement over CSR. The reason behind this behavior is that non-zero elements in matrices with a high bandwidth are often scattered across the whole matrix, leading to a rather low substructure frequency. However, with the exception of *parabolic_fem*, which has a rather irregular structure and very high bandwidth, CSX-Sym was able to achieve near-best performance for almost all these corner-case matrices.

D. Evaluation in reduced bandwidth matrices

As depicted in Fig. 12, all symmetric SpM×V implementations exhibited poor performance in high bandwidth matrices. The minimization of the matrix bandwidth has been extensively studied in the past, in order to reduce the communication overhead in distributed SpM×V implementations [18]–[20]. These techniques try to 'bring' the non-zero elements as close as possible to the main diagonal of the matrix by applying row and column permutations. This non-zero elements rearrangement can considerably benefit the multithreaded SpM×V performance, and especially the symmetric kernel, for a number of reasons:



(a) Dunnington (b) Gainestown

Fig. 11. Symmetric SpM×V speedup with the CSX-Sym format. All symmetric formats use the optimized local vector indexing method..

TABLE III
SPM×V PERFORMANCE IMPROVEMENT DUE TO MATRIX REORDERING (RCM ALGORITHM).

	Dunnington (24 threads)	Gainestown (16 threads)
CSR	22.0%	11.1%
CSX	63.0%	14.0%
SSS	92.2%	43.6%
CSX-Sym	106.8%	48.5%

1) The access pattern in the input vector becomes more regular minimizing possible cache misses.
2) The interference between the participating threads is reduced, leading to a smaller index size for the local vectors indexing method and, as a result, to lower working set overhead.
3) The probability of CSX and CSX-Sym formats for detecting and encoding more substructures is increased, due to the larger concentration of the non-zero elements in a specific matrix region (main diagonal).

This benefit of matrix reordering is depicted in Tab. III depicts, where the average performance improvement of the different SpM×V implementation due to matrix reordering is reported. In Dunnington, the standard CSR gains a 22% improvement, while baseline CSX is benefited by 63%. As expected, the effect of matrix reordering in the symmetric kernel is much more important, with the improvement of SSS surpassing 90%, while CSX-Sym gets a more than 2× acceleration. Similar is the picture in Gainestown, but both the encountered improvements and the differences among the different formats are attenuated. Such behavior is quite typical in NUMA architectures, where the memory bandwidth contention is not so intense as in SMP systems.

Figure 13, finally, shows the absolute SpM×V performance in the reordered matrices of our suite. The performance of the four matrices highlighted as corner cases in the previous section is now considerably improved, though still not at the level of the regular ones. This is mainly due to their increased sparsity, which leads to considerable loop overheads due to very short rows. Nonetheless, symmetric SpM×V is able to

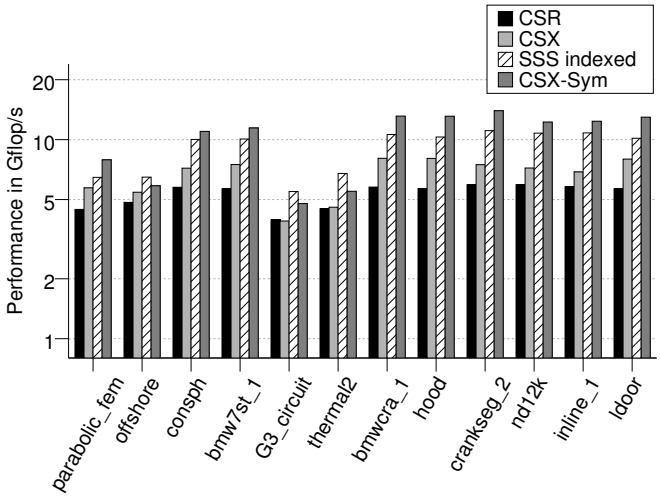


Fig. 13. Per-matrix performance on reordered symmetric matrices (Gainestown, 16 threads).

provide performance improvements, while CSX-Sym stays at the top for the majority of the matrices, with a performance surpassing 12 Gflop/s in six matrices (from *bmwcra_1* to *ldoor*).

E. Preprocessing cost of CSX-Sym

The detection and encoding of non-zero element substructures by CSX (and CSX-Sym as a result) entails a preprocessing cost that must be paid during the construction of the CSX matrix. However, thanks to a careful implementation and the use of advanced matrix sampling techniques, the cost of CSX preprocessing is rather contained [16]. Indeed, the preprocessing cost in Dunnington and Gainestown, using 24 and 16 threads, respectively, amounts to 49 and 94 serial SpM×V operations in CSR format on average, while these numbers are slightly increased to 59 and 115 operations for the set of reordered matrices, a rather expected increase, since the serial SpM×V execution is considerably reduced in this case. The higher numbers for Gainestown are due to the more elaborate preprocessing in NUMA machines needed

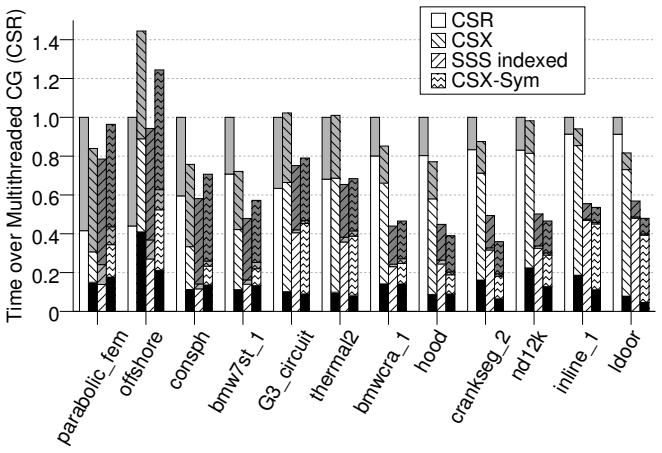


Fig. 14. CG execution time breakdown using 24 threads in Dunnington at the RCM reordered matrices after 2048 iterations. *[Execution time breakdown legend – white: SpM×V, light gray: SpM×V reduction, dark gray: vector operations, black: CSX/CSX-Sym preprocessing.]*

for balancing the compression benefit and the decompression overhead [16].

F. Impact on the CG iterative method

Figure 14 shows the execution time breakdown of the CG method using the unsymmetric CSR and CSX formats and their symmetric counterparts, SSS and CSX-Sym (both with local vectors indexing). Results are shown at the 24 threads in Dunnington for the RCM reordered matrices after 2048 iterations. The first generic observation is that the vector operations can be quite significant in smaller and sparse matrices, such as *parabolic_fem*, *offshore* etc., and may exceed 50% of the overall execution time of the multithreaded CG kernel. CG performs several vector operations, including dot products, during an iteration (see Alg. 1), but only a single SpM×V operation. For small matrices, therefore, that fit in the aggregate cache, the overhead of vector operations can easily dominate the total execution time of the solver. With the exception of the very sparse *parabolic_fem* and *offshore* matrices, where the computation is dominated by the vector operations, CG is greatly benefited by the symmetric storage formats, encountering a more than 50% overall performance improvement in large matrices. CSX-Sym is hindered by its preprocessing cost in smaller matrices and offers similar or lower performance the SSS format with the local vectors indexing optimization technique. In larger matrices, however, CSX-Sym compensates its preprocessing cost and manages to offer a further performance improvement to the CG kernel.

VI. Related work

Research in sparse matrices has been active since the times of the first computer systems. Early descriptions of the CSR format go back in late 1960's, where the CSR format was described as a possible way of storing a sparse matrix [21]. One of the first and very revealing surveys on the indexing structures of sparse matrices, dating back in 1973, is that of Pooch and Nieder [12]. In this paper, the authors describe a series of different indexing structures for sparse matrices, referred to as *row-column schemes*, including the Coordinate format, the CSR and BCSR, and also designate the use of *delta indexing* of the column indices. The term *Compressed Sparse Row* format or *CSR* was 'standardized' by Saad [6], [13], who also coined the terms *Coordinate format (COO)* and *Blocked Sparse Row (BSR)*, later standardized as Blocked Compressed Sparse Row (BCSR) by Im and Yelick [22], [23]. Much reasearch has been conducted on BCSR and its variations [24], including register and cache optimizations [22] and auto-tuning [25], which culminated in the OSKI sparse kernel library [26]. An interesting approach is the Compressed Sparse Block (CSB) format proposed recently by Buluç et al. [8], targeted at supporting efficiently both the Ax and A^Tx operations. CSB divides the matrix into large sparse square blocks, which are stored with the coordinate storage scheme, but using small integers for the row and column indices. The authors employ also task parallelism across the different blocks. The authors extend their approach in symmetric matrices in [27]. The key idea is to use three different local buffers to store results from the three innermost block diagonals which cover the majority of non-zero elements in most cases and store the rest of the results with the use of atomic operations to guarantee safety, in the output buffer. The key achievement is that the reduction phase will always include only three vector additions, based on the assumption that there is a small amount of non-zero elements outside the three block diagonals that will trigger atomic operations. However, in matrices with a relatively high bandwidth, this method is expected to be bound by the atomic operations.

Another interesting technique for symmetric SpM×V is the colorful method [7]. The main approach is to avoid completely the reduction phase without using local vectors. More particularly, they represent a sparse matrix as a graph where its nodes are the rows and the edges are the conflicts between them. By finding subgraphs without any connection, the authors divide SpM×V into n tasks, where n is the number of subgraphs, and each of these tasks can run in parallel. However, the geometry of the graphs limits the potential of this approach and could not achieve a performance gain over the typical local vectors method.

VII. Conclusions

In this paper, we have presented a two-step approach for the optimization of the symmetric SpM×V kernel. First, we extend the highly compressed CSX format to support also symmetric matrices and, second, we implement a local vectors indexing scheme for reducing the memory traffic of the symmetric SpM×V reduction phase. We have shown that our indexing technique has significantly improved the symmetric SpM×V performance compared to alternative local vector techniques. In addition, by integrating CSX-Sym format we managed a significant additional performance gain not only in SMP architectures, but also in NUMA platforms, where the computational part of the kernel is more prominent. Finally, the

performance advantage of the proposed techniques accelerated significantly the performance of a typical CG implementation.

ACKNOWLEDGMENTS

This work was partly funded by the European Community's Seventh Framework Programme (FP7/2007–2013) under grant agreement n° RI-261557 (PRACE1IP).

REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

[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.

[4] 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, USA: ACM, 2007.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM – A Direct Path to Dependable Software*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[6] Y. Saad, *Numerical methods for large eigenvalue problems*. Manchester University Press ND, 1992.

[7] V. H. F. Batista, G. O. Ainsworth Jr., and F. L. B. Ribeiro, "Parallel structurally-symmetric sparse matrix-vector products on multi-core processors," *Computing Research Repository (CoRR)*, vol. abs/1003.0952, 2010.

[8] 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 twenty-first annual Symposium on Parallelism in Algorithms and Architectures (SPAA'09)*. Calgary, Canada: ACM, 2009, pp. 233–244.

[9] R. Geus and S. Röllin, "Towards a fast parallel sparse matrix-vector multiplication," *Parallel Computing*, vol. 27, pp. 883–896, 2001.

[10] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An extended compression format for SpMV on shared memory systems," in *Proceedings of the 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. San Antonio, Texas, USA: ACM, 2011, pp. 247–256.

[11] A. Jennings, "A compact storage scheme for the solution of symmetric linear simultaneous equations," *The Computer Journal*, vol. 9, pp. 281–285, 1966.

[12] U. W. Pooch and A. Nieder, "A survey of indexing techniques for sparse matrices," *ACM Computing Surveys*, vol. 5, pp. 109–133, 1973.

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

[14] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[15] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, University of California, Berkeley, 2010.

[16] V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, "An extended compression format for the optimization of sparse matrix-vector multiplication," *IEEE Transaction on Parallel and Distributed Systems*, 2012, to appear.

[17] T. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, pp. 1–25, 2011.

[18] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*. ACM, 1969.

[19] G. Karypis and V. Kumar, "METIS – Unstructured graph partitioning and sparse matrix ordering system, version 2.0," University of Minessota, Department of Computer Science, Tech. Rep., 1995.

[20] U. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[21] W. Tinney and J. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *IEEE Proceedings*, vol. 55, no. 11, pp. 1801–1809, 1967.

[22] 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*. Springer-Verlag, 2001, pp. 127–136.

[23] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications*, vol. 18, pp. 135–158, 2004.

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

[25] 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*. Baltimore, MD, USA: IEEE Computer Society, 2002, pp. 1–35.

[26] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 521, 2005.

[27] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *IEEE International Parallel & Distributed Processing Symposium*. Anchorage, AK, USA: IEEE Computer Society, 2011, pp. 721–733.