An Extended Compression Format for the Optimization of Sparse Matrix-Vector Multiplication

Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, Nectarios Koziris

***Abstract*—Sparse matrix-vector multiplication (SpM×V) has been characterized as one of the most significant computational scientific kernels. The key algorithmic characteristic of the SpM×V kernel, that inhibits it from achieving high performance, is its very low flop:byte ratio. In this paper, we present an extended and integrated compressed storage format, called Compressed Sparse eXtended (CSX), that is able to detect and encode simultaneously multiple commonly encountered substructures inside a sparse matrix. Relying on aggressive compression techniques of the sparse matrix's indexing structure, CSX is able to considerably reduce the memory footprint of a sparse matrix, therefore alleviating the pressure to the memory subsystem. In a diverse set of sparse matrices, CSX was able to provide a more than 40% average performance improvement over the standard CSR format in symmetric shared memory architectures and surpassed 20% improvement in NUMA architectures, significantly outperforming other CSR alternatives. Additionally, it was able to adapt successfully to the non-zero element structure of the considered matrices, exhibiting very stable performance. Finally, in the context of a 'real-life' multiphysics simulation software, CSX was able to accelerate the SpM×V component nearly 40% and the total solver time approximately 15% after 1000 linear system iterations.**

I. Introduction

SParse matrices arise in a variety of scientific disciplines, ranging from physics simulations to data mining and financial analysis. Most of these applications boil down to the execution of basic sparse matrix kernels. Among these kernels, the Sparse Matrix-Vector Multiplication kernel (SpM×V), which computes the product of a sparse matrix and a dense vector, is probably the most prominent and challenging, being recently classified as one of the kernels particularly important for science and engineering in the next decade [1], [2]. The algorithmic characteristic that renders the optimization of SpM×V so challenging in modern multiprocessor systems is its very low flop:byte ratio [3]–[5]. In effect, this means that the algorithm must retrieve a significant amount of data from the memory hierarchy in order to perform a useful operation. This adds significant pressure to the memory subsystem for large problems that do not fit in the system's cache hierarchy and tends to saturate the memory bandwidth of modern symmetric shared memory multicore architectures. In the case of NUMA architectures, although the higher memory bandwidth offered alleviates the pressure to the memory subsystem, the high memory intensity of the SpM×V kernel renders its performance very sensitive to the placement of the data on the different memory nodes.

V. Karakasis, T. Gkountouvas, G. Goumas and N. Koziris are with the National Technical University of Athens, Greece.

K. Kourtis is with ETH Zürich, Switzerland.

In order to store a sparse matrix efficiently, one needs to store only the non-zero values of the matrix along with some location information for iterating over the non-zero elements. The most straightforward format is the Coordinate format (COO) [6], which stores the row index and column index for each non-zero element. However, the most common format for storing sparse matrices for use with the SpM×V kernel is the Compressed Sparse Row (CSR) format [6]. The idea behind CSR is that it suffices to store only the column indices and 'pointers' to the start of each row for locating each non-zero element of the matrix. In a typical implementation of CSR with 32-bit integers for row and column indices and double precision non-zero values, the gain in matrix size is approximately 25% compared to the COO format.

Nonetheless, even CSR has a lot of redundant information. The non-zero elements of sparse matrices, especially of those arising from physical simulations, are usually arranged in dense substructures, e.g., horizontal and/or diagonal sequences, 2-D blocks etc. One could keep just a single column index for each encountered substructure, since it is known beforehand how to iterate over the elements inside the substructure; the matrix size, therefore, could be significantly reduced, especially for matrices with a lot of dense regions. Several storage formats have been proposed in the past that target the reduction of the matrix size, either explicitly or implicitly. For example, the Variable Block Length (VBL) [7] format exploits horizontal substructures grouping together sequential elements into one-dimensional blocks with variable length. To achieve this, it also adds a new data structure that holds the length of every block. The Blocked Compressed Sparse Row format (BCSR) [6], [8] takes a different approach: it groups together the non-zero elements into fixed-size 2-D blocks adding zero-padding to construct full blocks. Other approaches include formats that exploit diagonal structures [9], [10] using zero-padding or formats that split the input matrix into multiple smaller matrices each one holding a different substructure [9], [11]. The main drawback of these formats, however, is that they take an 'all-or-nothing' approach: they will either offer very high performance improvement over CSR in cases where they find the correct substructures or plummet in cases where they fail to do so. The most typical example is the BCSR, which can offer 2× performance improvement in certain matrices, but half the CSR performance in others.

Our approach to the optimization of the SpM×V kernel is the Compressed Sparse eXtended (CSX). When designing CSX we set a number of goals for the new format:

(a) It should specifically target on the minimization of the memory footprint of the matrix, since SpM×V is a memory

bandwidth bound kernel,
(b) it should cover a wide range of substructures inside the matrix, including horizontal, diagonal and 2-D blocks,
(c) it should expose a stable high performance behavior across different matrices and symmetric shared memory and NUMA architectures,
(d) it should be extensible and adaptive in the sense of supporting new substructures or implementing variations of the existing ones.

In order to meet the two first design goals, we employ an aggressive compression scheme for CSX. We believe that data compression techniques will play a more significant role in future multicore and manycore chips as a means not only for minimizing the communication cost in different levels (processor-to-memory, processor-to-processor etc.), but also for increasing energy-efficiency. CSX builds on top of the CSR-DU format [12], by adding run-length encoding to the delta encoding of the column indices performed by CSR-DU, in order to detect sequences of non-zero elements either continuous or separated by some constant delta distance. For each encoded substructure, CSX keeps a two-byte descriptor (type and size) of the substructure and its initial column index encoded as a delta distance from the previous one. To detect non-horizontal substructures, we employ the notion of coordinate transformations to transform the elements of the matrix according to the desired iteration order (e.g., vertical, diagonal, block-wise etc.) and then reuse the very same detection process that we use for the horizontal substructures. This technique adds also to the design goal of the extendibility, since it suffices to define a '1-1' coordinate transformation in order to detect whatever substructure inside the matrix. Our third design goal is verified mostly by the experimental results: CSX manages to significantly reduce the matrix size and provides a 42% average performance improvement over CSR on a 24-core symmetric shared memory system. For NUMA architectures, where the memory bottleneck is not so intense, we relax the compression scheme by storing the full initial column index (instead of a delta distance) to save some of the time-consuming decompression computations. In these architectures, CSX manages an 18% average performance improvement over CSR, which climbs to 21% when simultaneous multi-threading is enabled. Compared to VBL and BCSR, CSX gains 10% and 33% on average, respectively, in symmetric shared memory architectures, while in NUMA architectures the gap closes to 8.4% over VBL and 11% over BCSR on average. A key advantage of CSX over other CSR alternatives is its performance stability; even for matrices that CSX does not achieve the highest performance, its performance lies within 5% of the best. In cases of very irregular and rather 'ill-based' matrices, CSX manages to successfully adapt to the matrix structure and achieves performance comparable to CSR, while other alternatives encounter a significant performance degradation. Finally, we add to the versatility of CSX by employing runtime code generation for the substructure-specific SpM×V routines. For every matrix we generate specific C code that we compile programmatically using the Clang [13] front-end compiler infrastructure and pass it to the LLVM [14] optimization back-end. Thanks to its advanced mechanism for detecting and encoding substructures and the flexibility provided by the runtime code generation, the CSX format can be adapted to the specificities of almost every sparse matrix and provides consistent and high performance over a variety of modern commodity architectures, outperforming every major CSR alternative.

In this paper, we take a step further from the simple optimization of the SpM×V kernel, by focusing also on the minimization of the preprocessing cost of CSX, in order to make it practical for 'real-life' SpM×V applications. With the use of advanced sampling of the input matrix and a careful implementation, we were able to reduce the CSX preprocessing cost down to approximately 100 serial CSR SpM×V iterations. This low preprocessing cost can be amortized easily in the context of a large sparse linear solver that may need a few thousands of SpM×V iterations to converge. Indeed, despite its preprocessing cost, integrating CSX into the Elmer multiphysics simulation software offered a nearly 40% performance improvement of the SpM×V component and a 15% average improvement of the overall solver time.

The rest of the paper is organized as follows: Section II provides background information about sparse matrices, SpM×V, iterative solvers and modern commodity architectures. Section III presents the data structures of CSX, Section IV describes how CSX detects and encodes the substructures, Section V presents the runtime code generation process employed by CSX, Section VI describes the techniques used for minimizing the preprocessing cost, while Section VII details the porting of CSX in NUMA architectures. Section VIII presents the experimental evaluation of the performance of CSX, as well as its impact on the overall performance of the Elmer multiphysics simulation software. Finally, Section IX presents the related work in the field of SpM×V optimization and Section X concludes the paper.

II. Background & Motivation

A. Significance of the SpM×V and performance bottlenecks

Sparse matrices are finite matrices dominated by zero elements. These matrices arise often with the discretization of parallel differential equations (PDE) in finite element methods (FEM) and are usually involved in the solution of large linear systems. The most widely used class of iterative algorithms for solving linear systems are the Krylov subspace methods [15], [16], which include, among others, the well known GMRES (Generalized Minimum Residual [17]) and the CG (Conjugate Gradient [18]) iterative methods. Krylov methods involve the execution of three time-consuming kernels [16], [19], namely sparse-matrix vector products (SpM×V), vector-vector operations (AXPY) and dot products. If the method is preconditioned, then the preconditioner must also be included in this listing. Fig. 1 shows an execution time breakdown for a non-preconditioned serial CG implementation. The majority of the execution time, surpassing 80%, is spent executing the SpM×V kernel. Similar is the case with other Krylov methods, e.g., the Bi-CG Stabilized [20] method employed by the Elmer multiphysics software [21], where the SpM×V kernel takes up 60–90% of the total execution time of the solver, depending on the input matrix.

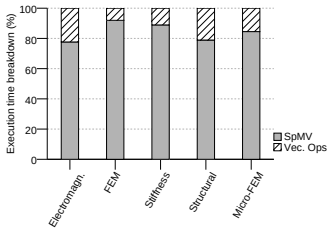


Fig. 1. Execution time breakdown of a non-preconditioned CG implementation.

1: **procedure** CSRSPMV($rowptr$, $colind$, $values$, N, x, y)
2: **for** $i \leftarrow 0$ **to** N **do**
3: $yr \leftarrow 0$
4: **for** $j \leftarrow rowptr[i]$ **to** $rowptr[i+1]$ **do**
5: $yr \leftarrow yr + values[j] \cdot x[colind[j]]$
6: $y[i] \leftarrow yr$

Alg. 1: A typical SpM×V implementation for the CSR storage format for $N \times N$ sparse matrices.

The major performance problem of the SpM×V kernel (Alg. 1) in modern commodity microarchitectures stems primarily from its algorithmic nature. The matrix-vector kernels (dense and sparse) sweep once through the whole matrix and perform a constant number of floating point operations per element. This automatically leads to a $\Theta(1)$ flop:byte ratio compared to the $\Theta(N)$ (N being the rank of the matrix) of the matrix-matrix multiplication kernels. In practice, this means that in order to avoid bottlenecks, the memory hierarchy must be able to provide data to the processor at a comparable speed, which is hardly ever the case for any modern commodity microarchitecture. The situation gets worse with sparse matrices, where the kernel must first retrieve the non-zero element's location information, before accessing it[1]. Fig. 2 shows the speedup of a multithreaded SpM×V CSR implementation and the consumed main memory bandwidth in GB/s in a two-way quad-core symmetric shared memory system. Despite exhibiting ample parallelism [22], the SpM×V kernel fails to scale beyond four threads due to the saturation of the system's memory bandwidth. An 85% of the available memory bandwidth is already consumed by the two threads and it is totally saturated from four threads onward. The minimization of the memory footprint of the sparse matrix, therefore, becomes of vital importance for the optimization of the SpM×V kernel, especially for the symmetric shared memory architectures.

B. Sparse matrix storage formats and their compression potential

Storing a sparse matrix in a dense matrix format is at least impractical. Therefore, special storage formats have been proposed to handle sparse matrices under the general rule that it suffices to store only the non-zero elements (possibly, with a reasonable zero-padding) along with some location metadata for locating the elements inside the matrix. Ideally, in order to store an $N \times N$ sparse matrix with NNZ non-zero elements, at least $8NNZ$ bytes of storage are required, assuming double precision floating-point values for the non-zero elements and no need for location metadata (e.g., suppose a fictional matrix that the location of its non-zero elements could be computed in the runtime).

[1]The best possible flop:byte ratio of SpM×V is 0.25, while stencil computations can approach 0.5 and a 3D FFT kernel reaches 1.64 [5].

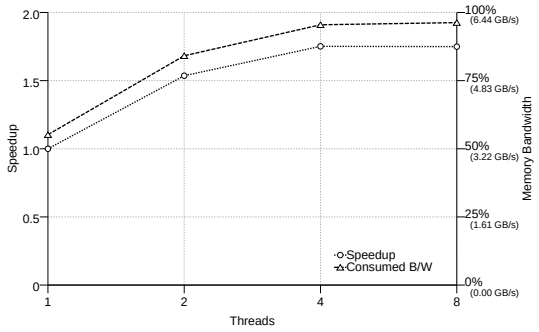


Fig. 2. Demonstration of the SpM×V kernel speedup in relation to the memory bandwidth consumption in a two-way quad-core SMP system.

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \\ 1.1 & 0 & 2.9 & 3.7 & 0 & 1.1 \end{pmatrix}$$

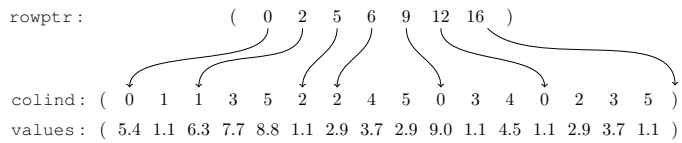


Fig. 3. An example of a sparse matrix stored in the CSR format.

The most straightforward sparse matrix storage format is the Coordinate (COO) format [6], which stores the row index and column index of every non-zero element. Assuming four-byte integers for indices, the matrix size in the COO format is $16NNZ$ bytes, twice as much the size of the theoretical lower bound. The Compressed Sparse Row (CSR) format [6] addresses the problem of the increased matrix size by compressing the row indices and leaving the column indices intact. More specifically, instead of storing all the row indices, the CSR format stores only N 'pointers' to the start of each row. Since for most sparse matrices $N \ll NNZ$, the total matrix size now falls to $12NNZ$, which is a 25% gain in the total matrix size compared to the COO format. Fig. 3 shows the typical implementation of the CSR format, where the `rowptr` array stores the row pointers, the `colind` the column indices and the `values` the non-zero values. The great advantage of CSR is that it provides a good compromise between the reduction of the matrix size and the ease of construction and manipulation (no preprocessing of the matrix needed), making it ideal for SpM×V applications.

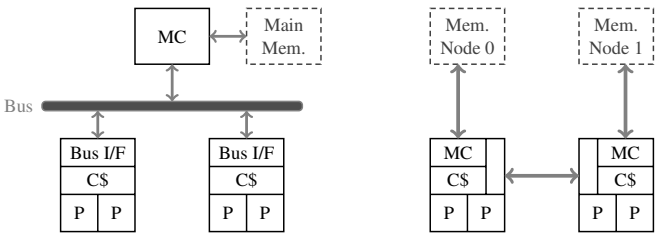
However, the `colind` structure of CSR has a lot of redundant information. Indeed, the matrix size of CSR is still 50% over the theoretical lower bound. The most common approach for compressing the column indices is grouping together

neighboring elements in blocks and keeping one column index per block. We can distinguish two large categories of blocking storage formats: (a) those with fixed size blocks and (b) those with variable size blocks. The most representative storage format in the first category is the Blocked Compressed Sparse Row (BCSR) format [6], [8], which is actually a blocked version of the CSR format. BCSR arranges all the non-zero elements of the matrix into fixed size $r \times c$, strictly aligned blocks, employing zero-padding, if necessary, to construct full blocks, and keeps a single column index per block. Despite being initially utilized for register optimizations [8] and still being very computation-friendly, the BCSR format can significantly reduce the size of matrices with large amounts of dense blocks. Nonetheless, from a theoretical standpoint, the compression potential of BCSR is limited: suppose a matrix that its non-zero elements can be grouped perfectly in $r \times c$ blocks without padding ($r, c \ll N$), then the condensed `colind` structure will be (in bytes) $4\frac{NNZ}{r \times c} = \Omega(NNZ)$, still remaining in the order of NNZ.

In the category of variable size block formats, the most representative is the Variable Block Length (VBL) format [7], which forms one-dimensional variable sized horizontal blocks. Again, the VBL format keeps a single column index per block, but now, since the block size is variable, an additional data structure is required to keep the different block sizes. A single byte for storing the block size is usually more than enough for the majority of sparse matrices, splitting larger blocks in 255-element chunks. The compression potential of VBL is higher than BCSR's and can approach the theoretical lower bound. Suppose a matrix with NNZ_{row} elements per row and that k, $k \ll NNZ_{row}$, VBL blocks are formed on average per row. Since in a typical sparse matrix, $NNZ_{row} \ll N$, k can be considered as constant. Therefore, the size of the compressed `colind` will be $4kN$ and the required size for holding the block sizes will be kN (assuming one-byte size representations). These sum up to a size of $5kN = \Omega(N)$ bytes, which brings the total matrix size very close to the theoretical lower bound. However, in sparse matrices with a non-horizontal non-zero element structure, VBL will form degenerate size-one blocks, and thus, the number of blocks per row cannot be decoupled from the non-zero elements per row. In this case, the resulting `colind` will be in the order of NNZ. Nonetheless, this cannot overshadow the higher compression potential of the variable size block methods over their fixed size counterparts, which are restricted mainly by the large number of resulting blocks. In practice, VBL achieves almost always higher compression ratios than BCSR.

C. Trends in modern mainstream computer architectures

It is not very long ago that the quest for higher processor frequencies was abandoned by the leader chip manufacturers, due to the need for higher energy efficiency and the need to keep Moore's law alive at the same time. The interest of academia and industry has since shifted toward incorporating multiple cores inside the same physical processor, inaugurating the *multicore era* and setting up new challenges. The use of multiple cores or hardware threads inside the same physical processor has broaden the gap between the rate that the processor can now consume data and the rate that the memory subsystem can supply data, making the 'memory-wall' problem [23] even tenser.



(a) Symmetric shared memory. (b) NUMA.

Fig. 4. The two current trends in commodity architectures: symmetric shared memory and NUMA architectures.

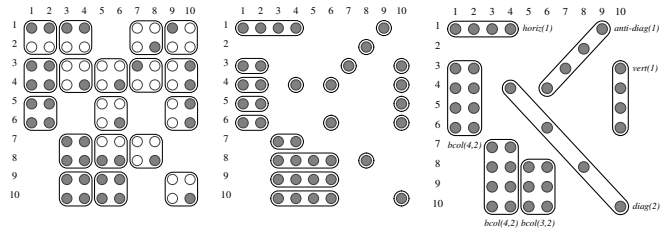
Symmetric shared memory multicore architectures are affected the most from the memory-processor speed gap. In these architectures, all the memory and interprocessor communication requests are routed through the same front-end bus to the central, off-chip memory controller and the peer processors (Fig. 4(a)). Apparently, this centralized logic, in conjunction with the low bandwidth and high latency that the off-chip communication carries with, can easily become the hotspot of a memory-intensive application, like the SpM×V kernel. Large and complex cache hierarchies, unfortunately, can limit this effect in the short term only.

The need to extract more parallelism from the hardware, given the slow DRAM speed evolution pace, demands a more decentralized approach. The Non Uniform Memory Access (NUMA) architectures 'move' the memory controller inside the processor chip and use dedicated hardware for the interprocessor communication (Fig. 4(b)). The main memory, though still shared, is no more uniformly accessible from every processor in the system: it is split into multiple nodes, each one assigned to a single processor. The available bandwidth is now ample for the communication between a processor and its local memory node, but accessing remote nodes requires multiple and costly hops. Two more challenges arise with the NUMA architectures for the memory-intensive kernels: (a) the increase in the available memory bandwidth may reveal weaknesses in the computational part of the kernel, that were otherwise hidden by the very slow access to the main memory, and (b) the kernel's performance can be now very sensitive to the correct placement of its data on the different memory nodes. The latter poses an additional burden for the programmer, who might need to explicitly alter her kernel to fully exploit the NUMA capabilities of a system.

III. The Compressed Sparse eXtended format

A. The need for an integrated storage format

Sparse matrices arising from the discretization of partial differential equations have their non-zero elements arranged in substructures either extending to some one-dimensional direction (e.g., horizontal, vertical, diagonal) or expanding to two-dimensional blocks. The exact nature of these substructures depends chiefly on the underlying application domain



(a) BCSR. (b) VBL. (c) CSX.

Fig. 5. The advantage of detecting multiple substructures inside a sparse matrix (gray dots are non-zero elements, white dots are padding elements). The use of padding in fixed size blocking (BCSR) can be excessive for matrices with irregular non-zero element structure, while detecting only one substructure type (VBL) cannot exploit substructures in different directions.

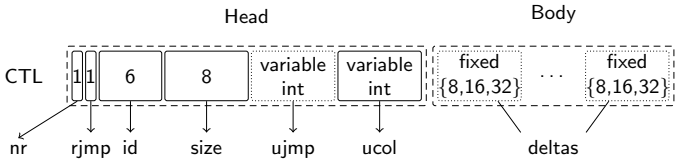


Fig. 6. The `ctl` byte-array used by CSX to encode the location information of the non-zero elements of a sparse matrix. Optional fields are denoted with a dotted bounding box.

and any preprocessing performed in the original matrix (e.g., bandwidth minimization techniques) that alters the distribution of the non-zero elements. As already discussed in Section II, exploiting these substructures can result in a reduction of the matrix size. However, restraining a storage format to detect a single substructure may have diminishing returns in cases where the detected substructure does not exist in adequate quantities inside the sparse matrix (Fig. 5), leading even to an increase in matrix size compared to the standard CSR. Alternatives that split the matrix into multiple submatrices of the same rank, each one storing a different substructure [9], [24], should experience additional scaling problems, since a final reduction of partial output vectors is required; the SpM×V kernel performance can be hindered as well, due to very small rows of the resulting submatrices [3], and load balancing issues might also creep in. The need for an integrated storage format that could incorporate a multitude of substructures is seemingly the most viable approach to a performant CSR alternative.

The proposed Compressed Sparse eXtended (CSX) format integrates five different substructure categories (horizontal, vertical, diagonal, anti-diagonal, 2-D blocks) into a single storage format and can be expanded easily to support more. Since we focus on the minimization of the matrix size, we build CSX on top of the CSR-DU [12] format. CSR-DU is a CSR alternative that applies delta indexing (*delta encoding*) in the CSR's `colind` structure; instead of storing the full, four-byte column index for every non-zero element, CSR-DU stores the delta distance from the previous index based on the premise that one or two bytes will be enough to store the delta distances in most of the cases. Basing CSX on CSR-DU has the additional benefit of using a more compressed storage format than the standard CSR as a baseline, giving rise to performance improvements even in the case that no substructures are detected.

Detecting sequences of continuous column indices, i.e., an horizontal substructure, given the delta distances of the columns is straightforward: a sequence of constant delta distances d forms an horizontal substructure. It then suffices to store only the initial column index (as a delta index) and a descriptor of the substructure, including its type and length. This type of column index encoding employed by CSX is commonly known as *run-length encoding*. For non-horizontal substructures, we transform the coordinates of the matrix's non-zero elements (originally in row-wise order) into the desired iteration order and reuse the very same horizontal substructure detector to detect non-horizontal ones.

B. The data structures

CSX replaces both the `rowptr` and `colind` arrays of the standard CSR with a single control byte-array containing all the required information, called `ctl` (Fig. 6). Similar to CSR-DU, CSX divides the matrix into units. In CSX's terminology, a *unit* represents a substructure inside the sparse matrix, will it be a substructure (*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 a two-byte descriptor of the unit and its initial column index (`ucol` field) encoded as a variable-size integer. The two byte descriptor stores a 6-bit ID of the substructure unit (`id` field) and the size of the unit in non-zero elements. The `nr` bit denotes the start of a new row. Finally, the body part is present only in delta units and stores the column index delta distances as fixed-size integers.

The `rjmp` bit and the `ujmp` field serve a special purpose. Since CSX encodes non-horizontal substructures, it is possible for substructures to group together all the elements of subsequent rows. For example, the anti-diagonal substructure in Fig. 5(c) starting at position $(1, 9)$ contains also the sole element of the second row, leaving it empty. These rows must be skipped when computing the SpM×V. Therefore, the `rjmp` bit denotes the existence of empty rows, while the `ujmp` field stores the number of empty rows to skip in a variable-size integer.

Similar to CSR, CSX stores the non-zero elements of the matrix in a `values` array, but in a substructure row-wise order. For example, the substructures in Fig. 5(c) will be stored in the following order: horiz(1), anti-diag(1), bcol(4,2), vert(1), diag(2), bcol(4,2) and bcol(3,2).

Encoding variable-size integers in CSX: In an attempt to be as compact as possible, the CSX format employs the use of variable-size integers to encode the initial column indices and the row jumps. In this encoding, an arbitrary integer is stored in 7-bit chunks reserving the most significant bit (bit 7) of each byte as a link between the different chunks (Fig. 7). The gain in the total matrix size by the use of variable-size integers is 2–3% and has an almost direct impact on the performance of the SpM×V kernel in symmetric shared memory systems, where the memory bandwidth saturation leaves enough space for the additional computational burden of the decoding process.

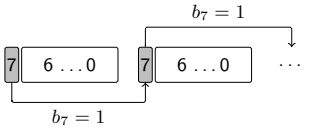


Fig. 7. The variable-size integer encoding employed by CSX.

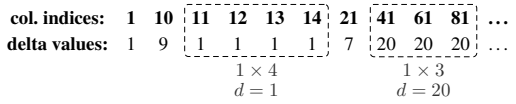


Fig. 8. Example of the run-length encoding of the column indices.

```
1: function RLEncode(colind::in)
   colind: sorted column indices
2:     deltas ← DeltaEncode(colind)
3:     d ← deltas[0]
4:     f ← 1
5:     rle ← (d, f)
6:     for i ← 1 to N do
7:         if deltas[i] = d then
8:             f ← f + 1
9:         else
10:            if f ≥ min_run_length then
11:                rle ← rle ∪ (d, f)
12:            d ← deltas[i]
13:            f ← 1
14:    return rle
```

Alg. 2: Run-length encoding of the column indices

IV. Detection and encoding of substructures

A. Mining the matrix for substructures

CSX can detect a variety of non-horizontal substructures with the use of coordinate transformations on the non-zero elements of the matrix. To facilitate the detection process, CSX uses a special internal representation for the sparse matrix, which is a hybrid of CSR and a generic version of the COO format. Instead of simple non-zero elements, the CSX's internal representation stores *generic elements*; a generic element is either a single non-zero element or a substructure. Similarly to the COO format, each generic element is stored as an (i, j, v) tuple, lexicographically sorted on the (i, j). In the case of a substructure, (i, j) represents the coordinate of the first element in the substructure and v is an array of the substructure's elements. We also keep the CSR's row pointers for fast row accessing. This internal representation is constructed once during the 'loading' of the original matrix, either from CSR or from the disk.

CSX detects substructures by applying run-length encoding on the column indices of the matrix. The run-length encoding first computes the delta distances of the column indices and then assembles groups, called *runs*, from the same distance values (Fig. 8, Alg. 2). Each run is identified by the common delta value and its length. Runs with length greater or equal to two form a substructure; nonetheless, we impose a restriction on the minimum length of a run (currently set to four) to avoid a proliferation of very small runs that could be more of an overhead than a benefit. There is also another subtle issue when mapping the detected runs into CSX units: all runs—except those that start at the beginning of a row—miss the first element of the real substructure. For example, in Fig. 8 the length of the real units are 5 and 4, instead of the detected runs of 4 and 3, which miss the 10 and 21 column indices, respectively. This can be crucial for the detection of 2-D substructures where additional alignment limitations exist. In our actual implementation, we fix this issue and also split large runs into 255-element chunks, so as to fit in the one-byte size field of a CSX unit. The process of scanning the whole matrix for a specific type of substructures is described in Alg. 3; we iterate over the rows of the matrix and for each row we gather all the column indices that are not yet part of a substructure and apply run-length encoding. For all the detected substructure instantiations (different delta values), we keep statistics (number of units and number of non-zero elements covered) to guide us on the final selection of the substructures to encode in CSX.

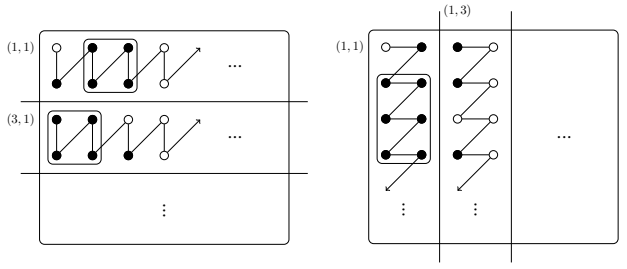
Detecting non-horizontal substructures: In our previous discussion on the detection process, we have not mentioned about horizontal substructures specifically. Indeed, the detection process 'sees' just column indices and the only requirement is that these indices must be sorted. Detecting horizontal substructures is therefore straightforward, since the matrix elements are arranged in row-wise (horizontal) order and are lexicographically sorted by default. To detect non-horizontal substructures then, it suffices to transform the matrix elements into the desired iteration order, sort them lexicographically and use the DetectSubstr() procedure described in Alg. 3. The case of one-dimensional, non-horizontal substructures is straightforward and Tab. I shows the exact coordinate transformations that we use for their detection. The case of 2-D substructures, though, is a bit more complex, since we need to 'linearize' the coordinates of the elements.

The approach we take for the 2-D substructures in CSX is depicted in Fig. 9. We segment the matrix into fixed-width bands, either row- or column-aligned, and apply a space-filling transformation inside every band, passing the resulting sequence to the DetectSubstr() procedure. However, care must be taken during the detection, since now not all detected units are valid blocks; for a unit to be valid, it must begin at a column index (in the transformed space), which is a multiple of the band width. Using this segmentation technique, we have managed to detect loosely aligned, variable-sized blocks, further increasing the compression potential of CSX. We support two categories of block substructure types, namely row-aligned and column-aligned blocks, and each category defines seven different substructure types for different widths of the segmentation bands ($r, c \in [2, 8]$ in Tab. I). In total, the use of coordinate transformations has allowed CSX to support seamlessly 18 different substructure types and enables the straightforward expansion to other substructure families, e.g., diagonal blocks.

Fig. 10 shows the substructures detected by CSX in our matrix suite and is very characteristic of the capability of CSX in detecting a variety of substructures inside a sparse matrix. It is very interesting that CSX was able to detect and encode even 'weird' substructures, such as the diagonal

```
1: procedure DETECTSUBSTR(matrix::in, stats::inout)
   matrix: CSX's internal repr., lexicographically sorted
   stats: substructure statistics
2:     colind ← ∅                                   ▷ Column indices to encode
3:     for all rows in matrix do
4:         for all generic elements e(i, j, v) in row do
5:             if e is not a substructure then
6:                 colind ← colind ∪ e.j
7:                 continue
8:             enc ← RLENCODE(colind)
9:             UPDATESTATS(stats, enc) ▷ Update statistics for this encoding
10:            colind ← ∅
11:        enc ← RLENCODE(colind)
12:        UPDATESTATS(stats, enc)      ▷ Update statistics for this encoding
13:        colind ← ∅
```

Alg. 3: Detecting substructures in CSX's internal representation.



(a) Row-aligned detection. (b) Column-aligned detection.

Fig. 9. Detection process of 2-D substructures in CSX (black dots denote non-zero elements).

ones with delta distances of 857 and 1714 in the pre2 matrix. In matrices dominated by dense blocks, CSX was able to detect large blocks in significant amounts, as is the case of the inline_1 and af_k_101 matrices. The detection of large dense blocks not only has a positive impact on the overall compression ratio, but also provides a performance advantage in the SpM×V computations. Finally, the *substructure map* of a sparse matrix produced by CSX can be very revealing for the specific performance behavior of SpM×V. For example, the matrices Freescale1, parabolic_fem and offshore are dominated by delta units with very large delta distances meaning that they are quite sparse and rather random. Indeed, their performance is poor using any format, suffering from irregular accesses in the input vector and load imbalances.

Substructure types and their instantiations: In CSX we make a distinction between a substructure type and its *instantiation*. In CSX's terminology, a substructure instantiation denotes how exactly a substructure type is encountered inside the sparse matrix. For example, the horizontal substructures with $d = 1$ and $d = 20$ of Fig. 8 belong to the same substructure type (horizontal), but are different instantiations. Similarly, the blocks 2×10 and 2×20 are instantiations of the same block, row-aligned, $r = 2$ type. A substructure type, therefore, may have indefinitely many instantiations in a sparse matrix, which adds great flexibility to the CSX format, allowing it to detect almost every dense pattern inside a sparse matrix. The 6-bit `id` field in the `ctl` structure of CSX (Fig. 6) refers to the exact substructure instantiation being encoded by the related unit; it does not refer to an abstract substructure type. This has the downside of limiting the total number of substructure instantiations in a sparse matrix to 64, but, in practice, only 4–5 instantiations are selected for encoding (see Fig. 10).

TABLE I
THE COORDINATE TRANSFORMATIONS APPLIED BY CSX ON THE MATRIX ELEMENTS FOR ENABLING THE DETECTION OF NON-HORIZONTAL SUBSTRUCTURES (ONE-BASED INDEXING ASSUMED).

Substructure	Transformation
Horizontal	$(i', j') = (i, j)$
Vertical	$(i', j') = (j, i)$
Diagonal	$(i', j') = (N + j - i, \min(i, j))$
Anti-diagonal	$(i', j') = \begin{cases} (i+j-1, i), & i+j-1 \geq N \\ (i+j-1, N+1-j), & i+j-1 > N \end{cases}$
Block (row aligned)	$(i', j') = (\lfloor \frac{i-1}{r} \rfloor + 1, \mathrm{mod}(i-1, r) + r(j-1) + 1)$
Block (column aligned)	$(i', j') = (\lfloor \frac{j-1}{c} \rfloor + 1, c(i-1) + \mathrm{mod}(j-1, c))$

B. Selecting substructures for final encoding

So far we have described how a single substructure is detected by CSX. The full detection process and the selection of the substructures for the final encoding is described in Alg. 4, which is a typical local search optimization algorithm. More specifically, for each available substructure type, we transform the matrix to the corresponding iteration order and scan it, collecting statistics for the examined substructure type. After having collected statistics for all the available substructure types, we filter out all the instantiations that fail to surpass a certain threshold in encoding the non-zero elements of the matrix (in our current implementation, this is set to 5% of the total non-zero elements) and proceed with the selection of the most suitable type for encoding the matrix (SELECTTYPE()). The algorithm then proceeds by encoding the matrix with the selected substructure type (ENCODESUBSTR()) and repeating the same procedure until no type can be selected.

The criterion we use for selecting the substructures to encode is a rough estimate of the achieved reduction in the size of the original CSR's `colind` structure. Assuming that we keep a single, full column index per detected substructure (ignoring delta units), the size of the `ctl` structure will be

$$S_{ctl} = \overbrace{N_{units}}^{\text{encoded}} + \overbrace{NNZ - NNZ_{enc}}^{\text{unencoded}}, \quad (1)$$

where N_{units} is the total number of encoded substructure units and NNZ_{enc} is the number of the non-zero elements encoded by the substructure type. Therefore, the gain in the CSR's `colind` size will be roughly

$$G = NNZ - S_{ctl} = NNZ_{enc} - N_{units}, \quad (2)$$

which is the metric that a substructure type must maximize in order to be selected for encoding.

V. GENERATING THE SPM×V CODE

The indefinitely large and a priori unknown number of substructure instantiations inside a sparse matrix dictates the

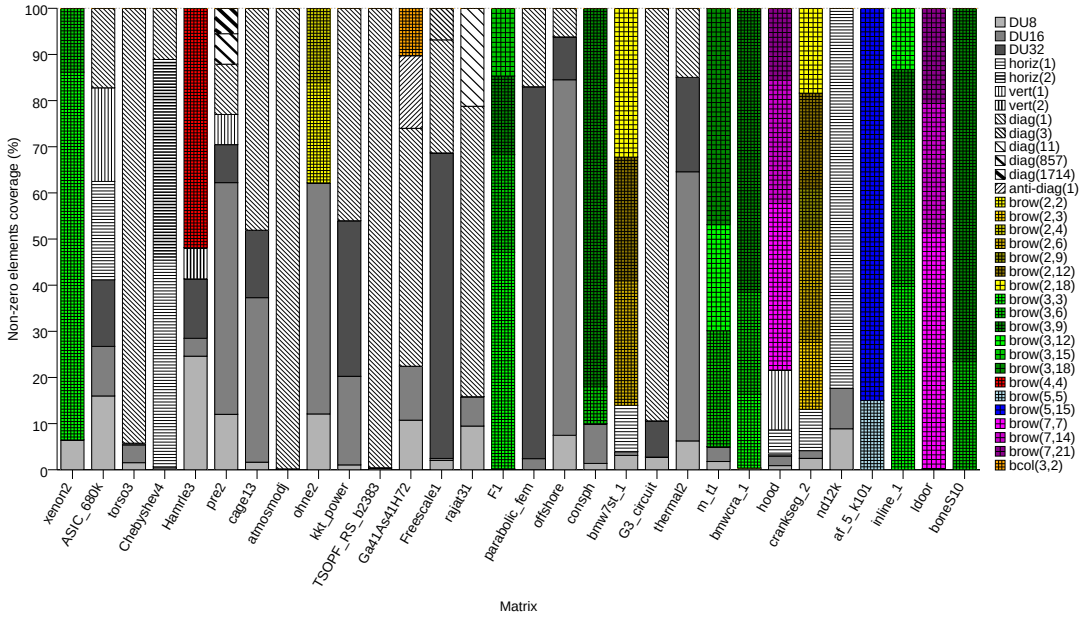


Fig. 10. The substructures detected and encoded by CSX in a diverse set of sparse matrices. The substructures are identified by their type name and the delta distance or the block dimensions for the one- and two-dimensional substructures, respectively. Delta units are denoted with the 'DU' prefix.

```
1:  procedure EncodeMatrix(matrix::inout)
    matrix: CSX's internal matrix in row-wise order
2:      repeat
3:          stats ← ∅
4:          for all available substructure types t do
5:              Transform(matrix, t)
6:              LexSort(matrix)
7:              DetectSubstr(matrix, stats)
8:              Transform^{-1}(matrix, t)
9:          FilterStats(stats)        ▷ Filter out instantiations that encode less
                                        than 5% of the non-zero elements
10:         s ← SelectType(stats)
11:         if s ≠ NONE then
12:             Transform(matrix, s)
13:             LexSort(matrix)
14:             EncodeSubstr(matrix)  ▷ Encode the selected substructure
15:     until s = NONE
```

Alg. 4: Detection, selection and encoding of the substructures in CSX.

use of runtime code generation for the substructure-specific SpM×V routines. We have built a new Just-In-Time (JIT) compilation framework for CSX based on Clang and LLVM [13], [14]. LLVM is a low-level compiler infrastructure that provides a collection of modular and reusable compiler and toolchain technologies. Clang is a C language family front-end for the LLVM infrastructure. Its main purpose is to parse normal C/C++ source code, convert it to the LLVM's Intermediate Representation (IR) and pass it to the LLVM back-end for the generation of the final native code. The great advantage of Clang and LLVM is that they provide a very rich programming interface that allows the development of efficient JIT code and its easy integration into an existing application.

In our initial implementation of CSX [25], we used to generate the LLVM IR directly through the related library interface. However, this is a tedious and error-prone task (also in terms of performance), since not only a good understanding of the intermediate representation is required, but also the actual SpM×V code is 'obfuscated' by the complex calls needed to construct the IR. For this reason, we now generate simple C code for the substructure-specific SpM×V routines and use Clang to generate optimized LLVM IR. The overhead of parsing the resulting C source (one translation unit, less than 150 l.o.c.) and generating the optimized LLVM IR is negligible compared to the matrix preprocessing time to justify the extra pain of maintaining a pure LLVM IR codebase or even an on-disk cache of precompiled SpM×V routines. Fig. 11 shows how the just-in-time compilation is organized in CSX. CSX maintains a directory of C source templates[2], which define a set of text hooks to be filled in the runtime. For each substructure type we maintain a source template for performing the SpM×V computation inside the substructure[3], and we also have a 'top-level' template that is responsible for the execution of the full SpM×V kernel in the CSX format. After we have selected the substructures for encoding and have generated the CSX's data structures, we pick the suitable templates and generate the corresponding SpM×V C code passing it to the Clang front-end. We program the front-end to emit an optimized LLVM IR module, from which we get a function pointer to the generated SpM×V kernel and, eventually, the LLVM back-end takes over to generate the final native code for the host machine.

The SpM×V kernel that we execute for CSX is shown in Alg. 5. First, we should note that in CSX we must explicitly zero out the output vector (lines 4–5), an arti-

[2]Not to be confused with the C++ templates.

[3]For block substructures we maintain just two generic templates: one for row-aligned blocks and one for column-aligned ones.

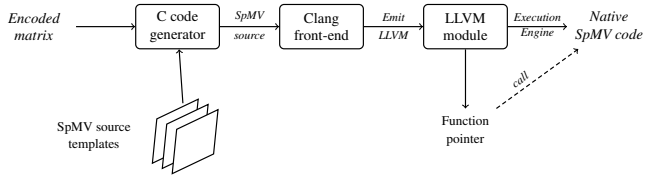


Fig. 11. The just-in-time compilation framework of CSX.

```
1: procedure CsxSpmv(ctl::in, values::in, x::in, y::out)
2:     y_curr ← y                          ▷ Current position in y vector
3:     x_curr ← x                          ▷ Current position in x vector
4:     for i ← 0 to N do                   ▷ We must zero-out the output vector
5:         y_curr[i] ← 0
6:     yr ← 0                              ▷ Local accumulator
7:     repeat
8:         flags ← *ctl
9:         size ← *(ctl + 1)
10:        ctl ← ctl + 2
11:        if TestBit(flags, 7) then       ▷ Check if nr bit is set
12:            *y_curr ← *y_curr + yr
13:            yr ← 0
14:            __NEW_ROW_HOOK()            ▷ Advances y_curr
15:            x_curr ← x
16:        x_curr ← x_curr + DecodeColumn(ctl)
17:        id ← GetID(flags)               ▷ Retrieve the ID of the next unit
18:        __BODY_HOOK()                   ▷ Unit-specific SpM×V code
19:    until ctl ends
```

Alg. 5: The SpM×V kernel template used the CSX storage format. The hooks are filled in during the runtime. The '$*$' is a dereference operator.

fact that is prescribed by the use of multiple non-horizontal substructures. However, the effect of this operation is very small, especially in multithreaded configurations, where the initialization is performed in parallel. The algorithm iterates over the `ctl` structure decoding a field at a time. If a new row is detected, we store the computed dot product (yr) in the output vector and advance the current position (y_{curr}) with the __NEW_ROW_HOOK(). If the matrix does not contain row jumps, the generated code simply advances y_{curr} by one. In the opposite case, it checks the `rjmp` bit and if there is a jump, it advances y_{curr} with the value of the variable size integer containing the jump size. The algorithm proceeds by decoding the variable size integer containing the delta distance from the previous column index (DECODECOLUMN()) and computes the starting column index (x_{curr}) of the current CSX unit. It then retrieves the ID of the unit and executes the unit-specific SpM×V code within the __BODY_HOOK(). The __BODY_HOOK() is actually replaced by a C `switch` statement, switching on the id variable. The unit-specific SpM×V routines are responsible for correctly advancing the `ctl` structure and the current column index, as well as updating the local accumulator yr, if necessary.

Since we encode specific substructure instantiations in CSX, we know during the compilation time the exact delta distances of the one-dimensional substructures and the exact block dimensions of the 2-D ones. This allows us to generate efficient code with hardcoded constant delta distances and constant block dimensions, matching the computational advantage of the fixed-size blocks of BCSR. Additionally, if only one substructure instantiation is encoded, we optimize out the switch statement completely.

Parallelization: CSX follows the parallelization pattern of CSR. During the construction of the CSX's internal representation, we split the input matrix row-wise maintaining roughly the same number of non-zero elements per partition. From this point on, the detection and encoding phases of CSX proceed independently, producing different final CSX submatrices per partition. The SpM×V kernel in Alg. 5 changes only in line 2, where we initialize y_{curr} to the beginning of the partition and in line 4, where we iterate only within the bounds of the partition.

VI. Tackling the preprocessing cost

The preprocessing time of CSX is bounded by the multiple calls to the LEXSORT() routine, which performs a lexicographic sort on the transformed matrix elements, during the scanning for substructures (Alg. 4, lines 4–8). We address this issue with a combination of partial sorting and sampling. More specifically, instead of scanning the matrix as a whole, we split it into constant size preprocessing windows based on the non-zero elements and scan every window separately. This modification reduces automatically the asymptotic complexity of the scanning phase from $\Theta(NNZ \lg NNZ)$ to $\Theta(NNZ)$ at the expense of missing the substructures that cross the boundaries of the windows. To further reduce the preprocessing cost, we only examine a certain number of windows uniformly distributed over the whole matrix covering only a small amount of the total non-zero elements. In our experiments, sampling a mere 1% of the matrix non-zero elements using 48 sampling windows reached the same SpM×V performance levels as the full-fledged preprocessing at nearly an order of magnitude less preprocessing time.

Having minimized the substructure scanning phase, the preprocessing time is now bound from the final encoding of the matrix (Alg 4, lines 12–14), which is still in the order of $\Theta(NNZ \lg NNZ)$. In this phase, unfortunately, partitioning and, especially, sampling cannot be applied, since we need to encode the *whole* matrix, after all. Nonetheless, the impact of this phase on the overall preprocessing cost is not so crucial for two main reasons: in most of the cases a very small number of substructure types will be finally encoded, and the full cost of the sorting will be paid only during the encoding of the first substructure type, since every time we sort only the unencoded elements.

In addition to the use of sampling and preprocessing windows, we further reduce the preprocessing cost by parallelizing the whole preprocessing process. The parallelization is straightforward: after we have loaded the initial CSR matrix, we setup a new thread for every partition that proceeds independently with the scanning and the construction of the final CSX matrix.

Finally, we take particularly care of the memory management during the preprocessing, by avoiding completely memory reallocations that would result in huge memory copies. For this reason, we try to infer the exact allocation requirements for a data structure right from the beginning or, in case this is

not possible, we are generous with the initial allocation and then truncate the extra space. This optimization has led to a considerable acceleration of the preprocessing phase, reaching a 2× to 3× speedup.

The preprocessing cost of CSX for detecting all the supported substructure types ranges now in the order of 100 serial CSR SpM×V operations (cf. the approximately 500 operations of [25]), rendering it viable even for online processing of the input matrix.

VII. Porting to NUMA architectures

The key factor for the high performance of SpM×V in NUMA architectures is the correct placement of the involved data on the system's memory nodes [3], [26]. The data each thread accesses must lie on its local memory node, in order to avoid the saturation of its peers' memory links and also the increased latency incurred by the multiple hops required for a remote access. A thread in the multithreaded SpM×V kernel accesses the matrix data structures of its own matrix partition, the corresponding parts of the output vector and arbitrary parts of the input vector. The correct placement of the matrix partitions in CSX is straightforward: since the construction of the CSX matrix happens independently for every partition, we just need to make sure that the CSX's data structures are allocated on the correct node using calls to a NUMA-aware memory allocator, e.g., the `numa_alloc_onnode()` of the Linux *numactl* library.

For the output vector though, we have to implement a more low-level allocation scheme. The problem with the output vector is that, ideally, different parts of it must lie on different physical memory nodes, while still being continuous in the virtual address space. Our approach is a Linux-based coarse-grained interleaved allocation scheme and is depicted in Fig. 12. The idea is simple: we allocate the whole output vector continuously in the virtual address space with a single call to `mmap()` and then bind every partition on its local node with subsequent calls to `mbind()`, rounding the partition sizes to the nearest multiple of the system's page size. The advantage of this allocation scheme is that the data placement is completely transparent to the user code, which now becomes NUMA-aware with the least effort. We used this technique to implement NUMA-aware versions of the SpM×V kernel with the CSR, BCSR and VBL formats (see Section VIII for the experimental evaluation) by just replacing the standard allocation calls with calls to our coarse-grained interleaved allocator.

Finally, for the sake of our experimentation, we place a copy of the input vector on every memory node. This arrangement provides a better balancing of the overall memory operations and exposes further the computational part of the kernel. In practical applications, however, where the input vector changes during the SpM×V iterations, placing it on a single node is a more reasonable option without losing significantly in SpM×V performance for the great majority of sparse matrices[4].

[4]We measured a 2.5% average performance overhead using a single instance of the input vector, which falls to 1.5% when using the same interleaved allocation scheme as for the output vector.

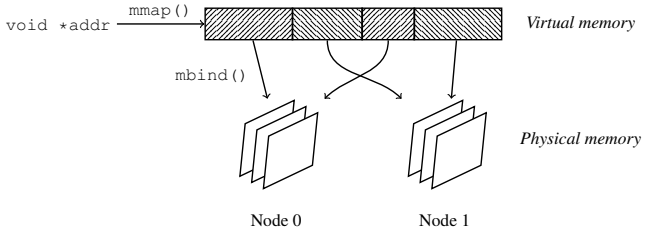


Fig. 12. Coarse-grained interleaved allocation of data structures involved in SpM×V. The data placement is completely transparent to the user code, yet NUMA-aware. The actual physical page allocation happens on memory write.

Optimizing the computations: Modern NUMA architectures have an important side-effect on the execution of the SpM×V kernel: the increased memory bandwidth when accessing a local memory node, exposes more the computational part of the kernel, which is no longer negligible. CSX performs some heavy decompression operations in decoding the variable size integers (Fig. 7) used to store the column delta distances and the row jumps. The case of row jumps is not of considerable concern, since we do not generate the decoding code at all for matrices that do not have them, and in cases they do, the decoding is not in the critical path. What stays in the critical path, though, is the decoding of the column delta distance before computing every substructure (Alg. 5, line 16). The decoding is trivial if the delta distance is less than 128, but it demands hefty bit operations if it is larger. Unfortunately, when encoding substructures (and not only delta units[5]) there is a proliferation of multi-byte column delta distances, whose decoding can considerably hinder the overall SpM×V performance. For this reason, we replace the column delta distance with the full starting column index of the substructure, stored in a standard 32-bit integer. This optimization degrades the overall compression ratio 2–3%, but the gain in performance can exceed 15% in certain matrices.

The proliferation of delta units when encoding multiple substructures can also overwhelm the benefit of compression in NUMA architectures due to the increased performance overhead of the switch statement (branch mispredictions). For this reason, we revise our score function (equation (2)) for NUMA architectures to include an estimate of the total switch statements executed as follows:

$$G = NNZ_{enc} - N_{units} - N_{switch}, \tag{3}$$

where $N_{switch} = N_{units} + N_{deltas}$, N_{deltas} being the total number of delta units generated. This heuristic balances better the tradeoff between the memory and the computational part of the kernel by penalizing the encodings that lead to a large computational cost. We have encountered more than 20% performance improvement for certain matrices in NUMA-aware thread configurations with the revised heuristic.

VIII. Experimental Evaluation

A. Setup and methodology

For the evaluation of the CSX format, we have selected 30 matrices from the University of Florida Sparse Matrix

[5]The case of using only delta units is not critical, because we encode the full matrix row as a single delta unit.

TABLE II
THE MATRIX SUITE USED FOR THE EXPERIMENTAL EVALUATION. ALL MATRICES ARE SQUARE. THE MAXIMUM POSSIBLE COMPRESSION RATIO (C.R.), USING NO INDEXING STRUCTURES, IS DENOTED, ALONG WITH THE COMPRESSION RATIO ACHIEVED BY CSX.

Matrix	N	NNZ	Size (MiB)	C.R. (CSX)	C.R. (Max.)	Problem
xenon2	157,464	3,866,688	44.85	30.6%	34.2%	Materials
ASIC_680k	682,862	3,871,773	46.91	24.0%	37.0%	Circuit Sim.
torso3	259,156	4,429,042	51.67	32.3%	34.6%	2D/3D
Chebyshev4	68,121	5,377,761	61.80	29.7%	33.6%	Structural
Hamrle3	1,447,360	5,514,242	68.63	22.9%	38.7%	Circuit Sim.
pre2	659,033	5,959,282	70.71	20.4%	35.7%	Circuit Sim.
cage13	445,315	7,479,343	87.29	21.1%	34.6%	Graph
atmosmodj	1,270,432	8,814,880	105.72	36.1%	36.4%	C.F.D.
ohne2	181,343	11,063,545	127.30	21.0%	33.7%	Semiconductor
kkt_power	2,063,494	14,612,663	175.10	20.0%	36.3%	Optimization
TSOPF_RS_b2383	38,120	16,171,169	185.21	33.1%	33.4%	Power
Ga41As41H72	268,096	18,488,476	212.61	28.4%	33.7%	Chemistry
Freescale1	3,428,755	18,920,347	229.61	10.3%	37.1%	Circuit Sim.
rajat31	4,690,002	20,316,253	250.39	30.9%	38.1%	Circuit Sim.
F1	3,428,755	26,837,113	308.44	31.1%	33.6%	Structural
parabolic_fem	525,825	3,674,625	44.06	6.7%	36.4%	C.F.D.
offshore	259,789	4,242,673	49.54	16.5%	34.7%	Electromagnetics
consph	83,334	6,010,480	69.10	30.9%	33.6%	F.E.M.
bmw7st_1	141,347	7,339,667	84.54	31.1%	33.8%	Structural
G3_circuit	1,585,478	7,660,826	93.72	32.3%	37.6%	Circuit Sim.
thermal2	1,228,045	8,580,313	102.88	15.7%	36.4%	Thermal
m_t1	97,578	9,753,570	111.99	31.9%	33.4%	Structural
bmwcra_1	148,770	10,644,002	122.38	32.1%	33.6%	Structural
hood	220,542	10,768,436	124.08	31.9%	33.8%	Structural
crankseg_2	63,838	14,148,858	162.16	30.6%	33.4%	Structural
nd12k	36,000	14,220,946	162.88	29.3%	33.4%	2D/3D
af_5_k101	503,625	17,550,675	202.77	33.3%	34.0%	Structural
inline_1	503,712	36,816,342	423.25	31.4%	33.6%	Structural
ldoor	952,203	46,522,475	536.04	33.2%	33.8%	Structural
boneS10	914,898	55,468,422	638.28	32.4%	33.7%	Model Reduction

Collection [27] (Tab. II). We made an effort to include large matrices from various scientific disciplines with different non-zero element structures. In order to evaluate the adaptivity and the performance stability of CSX in unfavorable cases, we have selected a large amount of matrices with a rather irregular structure. These matrices are challenging for memory bandwidth minimization formats, since contention to the memory bandwidth is not the key performance problem in these cases [3].

The alternative storage formats we consider in this paper are the BCSR and the VBL storage formats. We have selected these formats as the most established paradigms of CSR alternatives and as the best representatives of the fixed and variable size blocking storage formats, respectively. Of course, the baseline storage format in this comparison remains the CSR. We have implemented our own optimized versions of all these formats, including NUMA-aware implementations. Specifically for BCSR, we have implemented block-specific, optimized SpM×V routines for all the block sizes (1-D and 2-D) up to size eight plus the 3×3 block. The results reported for BCSR correspond to the best performing block, which was obtained after an exhaustive search of the 20 available blocks. In practice, where a heuristic will be most probably used (e.g., in OSKI [28]), the real performance of BCSR might be less. For the parallelization of the SpM×V routines, we use a static, row-wise partitioning scheme based on the non-zero elements of the matrix. Specifically for BCSR, we partition the input matrix after we have build it, taking into account the zero-padding as well, in order to achieve a better load balance.

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

	Harpertown	Dunnington	Gainestown
Model	Intel Xeon E5405	Intel Xeon X7460	Intel Xeon W5580
Microarchitecture	Intel Core	Intel Core	Intel Nehalem
Clock freq.	2.00 GHz	2.66 GHz	3.20 GHz
L1 cache (D/I)	32 KiB/32 KiB	32 KiB/32 KiB	32 KiB/32 KiB
L2 cache	6 MiB (per 2 cores)	3 MiB (per 2 cores)	256 KiB (per core)
L3 cache	–	16 MiB	8 MiB
Cores/Threads	4/4	6/6	4/8
Peak Front-end B/W	10.7 GB/s	8.5 GB/s	2× 30 GB/s
Sustained B/W	5.8 GB/s	5.4 GB/s	2× 15.5 GB/s
Multiprocessor Configurations			
Processors	2	4	2
Cores/Threads (total)	8/8	24/24	8/16

Finally, we use 32-bit integers for the indexing structures of all the storage formats and 64-bit, double precision floating point values for the non-zero elements. In the case of VBL, we use one-byte block size fields.

Our computational testbed comprises two symmetric shared memory systems and a NUMA platform. Specifically, the *Harpertown* system is a two-way quad-core Intel Xeon E5405 configuration (8 cores, symmetric shared memory), the *Dunnington* system is a four-way six-core Intel Xeon X7460 configuration (24 cores, symmetric shared memory), and the *Gainestown* system is a two-way quad-core Intel Xeon W5580 configuration (8 cores/16 threads, NUMA). Tab. III presents the technical characteristics of our platforms in more detail. All systems were running a 64-bit version of the Linux OS (kernel version 2.6) and we used LLVM 2.9 for the compilation of the SpM×V routines 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[6] 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 `sched_setaffinity()` system call. We followed a 'share-all' policy for the assignment of threads to logical processors by first 'filling' a socket before moving to the next one, with the exception of Gainestown; in this case, we did not use the HyperThreading feature but for the last 16-thread configuration. Compared to a 'share-nothing' policy, the selected core-filling policy gives a better insight on the SpM×V performance as we scale a system to accommodate more sockets and exposes better the impact of the correct data placement in NUMA architectures. Finally, for the NUMA-aware implementations, we used the *numactl* library, version 2.0.7, in conjunction with our low-level interleaved allocator.

Using simply the same compiler is not enough to achieve a completely fair comparison of the different storage formats. For this reason, we have built a common measurements frame-

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

work 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. We made no attempt to artificially pollute the cache after each iteration, in order to better simulate the behavior of iterative scientific applications, where matrix and vector data are present in the cache hierarchy, either because they have just been produced or they have been recently accessed. We should note here that by using multiple iterations, we induce temporal locality to our benchmark, and, thus, the streaming behavior of the SpM×V kernel is maintained only if the working set and, more specifically, the matrix data are larger than the system's aggregate cache.

B. The performance of the CSX format

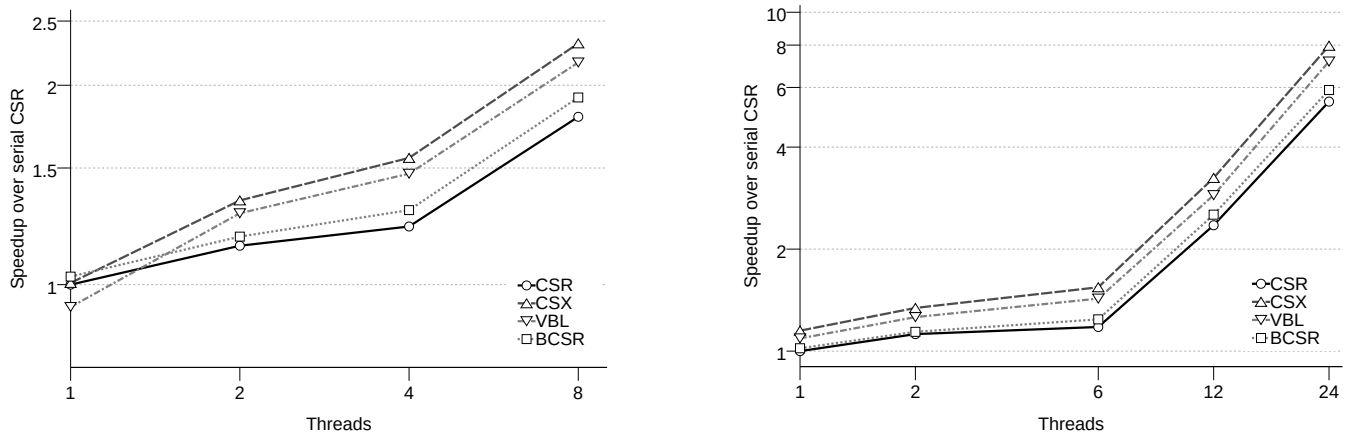
Fig. 13 shows the speedups achieved by all the considered methods in our two symmetric shared memory platforms, Harpertown and Dunnington. The effect of compression is dominant in these architectures, especially in the multithreaded configurations. CSX and VBL take the lead with an average 26.4% and 18.5% performance improvement over CSR in the eight-threaded configuration in Harpertown, respectively, while BCSR gains a mere 4.1%. Similar is the picture in the 24-thread configuration in Dunnigton: CSX offers a 41.8% performance improvement over CSR on average, VBL follows further behind with a 28.8% improvement, while BCSR is able to achieve only a bare 6.3% improvement. BCSR suffers from its inherently low compression capability and the use of padding to construct full blocks. It can be classified as an 'all-or-nothing' storage format, since in almost half of the matrices of our suite, it degraded SpM×V performance more than 30% on average, while for the other half matrices it provided a more than 25% performance improvement. The compression potential of VBL and CSX is definitely higher than BCSR's and this is clearly depicted by the speedup diagrams in our two symmetric shared memory architectures. The ability of CSX to detect and encode multiple types of substructures, and especially blocks, is a significant advantage of CSX over VBL, not only in terms of pure compression ratio, but also in terms of the involved SpM×V computations, since CSX keeps the computational advantage of the BCSR's fixed size blocks, thanks to the runtime code generation, without paying the padding overhead at all. Indeed, like BCSR, CSX matches the average performance of CSR in Harpertown using one thread, despite the additional overhead of zeroing the output vector in every iteration. In Dunnington, CSX is starting off with a significant 13.1% performance advantage over CSR right from the single-threaded configuration.

In the NUMA architectures, where the available memory bandwidth is considerably higher, the performance landscape changes, but CSX remains still the most performant storage format across the full range of multithreaded configurations. Fig. 14 shows the speedups of the considered storage formats in Gainestown using a NUMA-aware data placement. The very first observation is that CSR is now rather competitive, since the memory bottleneck is not so intense as before until the 16-threaded configuration, where HyperThreading is also enabled. CSX achieves a 17.5% performance improvement over CSR when using 8 threads and increases the gap to 20.7% in the 16-threaded configuration, where the contention for shared resources by the hardware threads becomes visible. The respective numbers for VBL are 8.4% and 13.9% for the two aforementioned configurations. The case of BCSR is interesting, since it manages to achieve a performance comparable to VBL, thanks chiefly to the ample memory bandwidth that better exposes its computational advantage. While VBL's performance falls behind CSR's up to the two-threaded configuration, reaching a 23.5% performance degradation for the single-threaded one, CSX experiences a slight 2.8% performance degradation only for the single-threaded configuration. From the two-threaded configuration, however, CSX is already ahead of every other format having achieved an 11.4% performance improvement over CSR and reaching 20.7% when the full system is utilized.

This behavior can be further explained by examining the per-matrix performance results for eight threads in Gainestown presented in Fig. 15. We have selected this configuration instead of the 16-threaded configuration as being less favorable for CSX and most representative of systems where the memory bottleneck is not so intense. Observing the performance results, we can first separate our matrix suite into two matrix categories: (a) low-performing matrices (≈3 Gflop/s in CSR) and (b) high-performing matrices (≈5 Gflop/s in CSR). The first category is formed by matrices with an irregular non-zero element structure and very short rows; SpM×V is rather latency-bound than bandwidth-bound in this case, since it suffers chiefly from cache misses in the input vector [3]. The second category, which is the most typical, consists of matrices with a more regular structure arising mostly from the discretization of PDEs; the key problem here is the contention for memory bandwidth. CSX manages to achieve considerable performance improvements in matrices of the second category, approaching the upper bound of dense-matrix vector multiplication performance in this configuration (8.5 Gflop/s). In the low-performing matrices, CSX exhibits a rather stable behavior, matching or even surpassing the performance of CSR. Although VBL and BCSR exhibit similarly high performance in more regular matrices, with BCSR matching the performance of CSX in block-dominated matrices (xenon2, consph, m_t1, bmwcra_1, inline_1), the situation changes for irregular matrices, where both BCSR and VBL tend to exhibit a significant performance degradation, due to their increased overhead in matrix size and computations. The advantage of CSX is also clear in matrices dominated by diagonal patterns (e.g., torso3, cage13, atmosmodj), which cannot be efficiently exploited by the alternative formats considered.

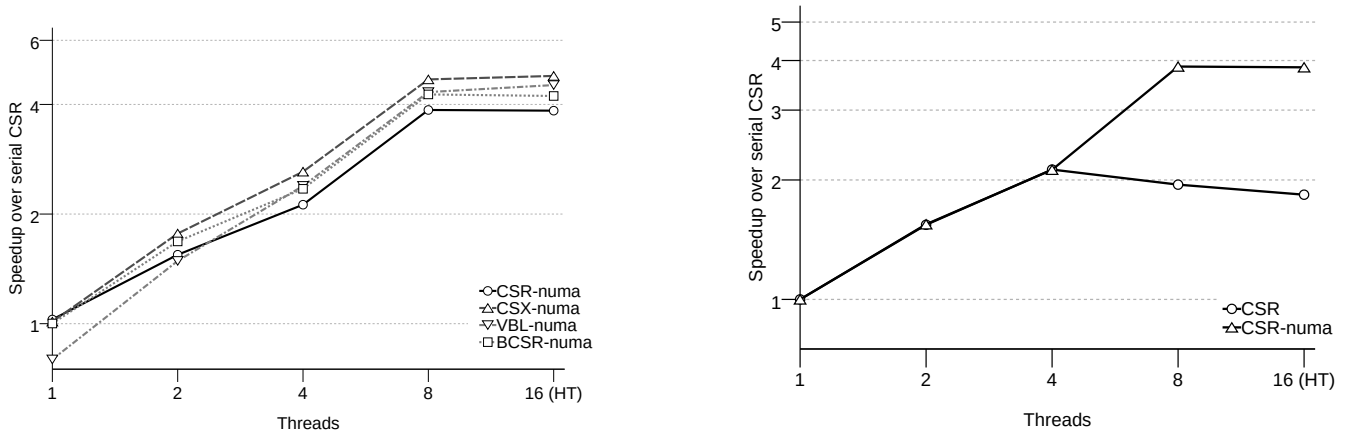
In Tab. IV, we exhibit quantitatively the performance stability of CSX in the eight-threaded, NUMA-aware configuration in Gainestown[7]. The competitiveness of CSR and BCSR is clear from this table, gaining seven and nine matrices, respectively. Nonetheless, even in this not so favorable configuration,

[7]In Harpertown and Dunnington, the predominance of CSX is almost total, providing the best performance in 26 out of the 30 matrices of our suite.



(a) Harpertown. (b) Dunnington.

Fig. 13. Speedup diagrams of CSX in comparison to the rest CSR alternative storage formats. The steep increase of speedup in Dunnington for 12 and 24 threads is due to the exponential increase of the available aggregate L3 cache as we use additional sockets.



(a) Gainestown. (b) The effect of the NUMA-aware data placement in SpM×V.

Fig. 14. Speedup of CSX in comparison to the rest CSR alternative storage formats in the Gainestown NUMA system. The effect of the NUMA-aware data placement is also depicted.

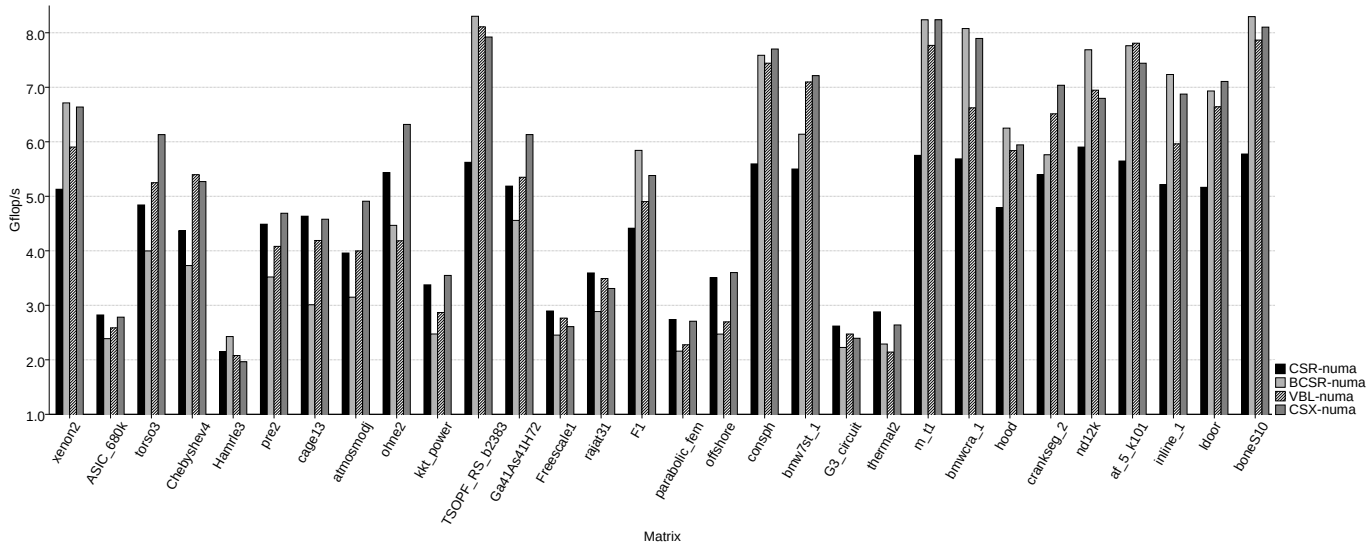


Fig. 15. The SpM×V performance in Gainestown for every sparse matrix in our suite using 8 threads in a NUMA-aware configuration.

TABLE IV
DEMONSTRATION OF THE PERFORMANCE STABILITY OF EACH METHOD IN GAINESTOWN (EIGHT-THREADED, NUMA-AWARE CONFIGURATION).

	CSR	**BCSR**	**VBL**	**CSX**
Best perf.	7	9	2	12
Differences from best				
Average	21.15%	20.13%	12.03%	5.79%
95% C.I.	± 3.53%	± 4.99%	± 2.94%	± 2.16%
Minimum	2.60%	0.01%	1.62%	1.13%
Maximum	32.27%	35.86%	33.79%	19.04%

CSX manages to achieve the absolute best performance in the majority of matrices (12), while VBL contents itself to just two matrices. The most important information of this table, however, are the performance differences of each format from the best overall performance. Specifically, for each considered format, we present its average performance difference from the absolute best for all the matrices that it did not gained. We also present a 95% *confidence interval (C.I.)*[8]. These metrics confirm the predominance of CSX both in terms of overall SpM×V performance and in terms of adaptivity to the different characteristics of each matrix. Despite obtaining the absolutely best performance in less than half of the matrices, CSX manages to be as close as 5.79% to the overall best performance. VBL follows further back with an average difference of 12.03% from the best performance, while BCSR and CSR come last with 20.13% and 21.15% differences, respectively. The stability of CSX is also superior to the other considered formats as it is depicted by the computed confidence intervals, while BCSR, as expected, is the least stable format with a performance variation of almost 5%. CSX can be therefore considered as a high performance storage format for sparse matrices, since not only achieves the highest performance in the majority of matrices in both SMP and NUMA architectures, but also manages to stay very close to the overall best performance in the rest of the matrices, including corner cases, such as very irregular matrices, exhibiting significant performance stability compared to other CSR alternatives.

Before closing the presentation of the performance of CSX, it is important to emphasize the effect of the data placement in the NUMA architectures. Fig. 14(b) compares the speedups of the standard and NUMA-aware versions of the CSX and CSR storage formats. The difference in performance between the two versions is quite large. In the eight-threaded configuration, where we start using the second processor in the system, the CSR standard implementation starts to encounter a performance slowdown, while CSX enters a plateau, before encountering a slowdown with the 16 threads. The correct data placement of the NUMA-aware versions balances the memory traffic between the two available memory controllers and offers an almost 2× performance improvement to both formats. Similar is the behavior for the other two formats, VBL and BCSR. Using our interleaved allocator (Section VII) for the allocation of the matrix's data structures, the data placement remained completely transparent to the user level and this vast performance improvement came with no changes in the SpM×V algorithm whatsoever and at a minimal developer's cost.

[8] An α% confidence interval denotes that α% of the observed samples will lie within the specified interval, assuming that the samples follow the normal distribution.

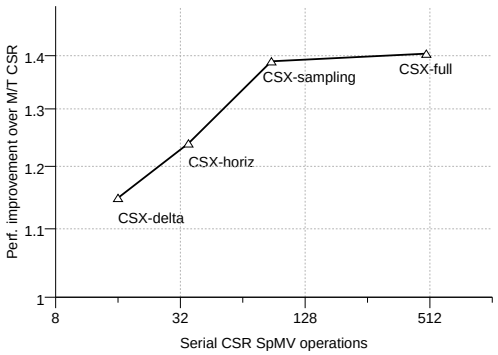


Fig. 16. The preprocessing cost of CSX in Dunnington using 24 threads.

C. The preprocessing cost

CSX can be programmed to detect only delta units or a specific subset of the supported substructures, e.g., horizontal substructures only. This a simple way to reduce the preprocessing cost. Additionally, in this case, there is no need for sampling as well, since the non-zero elements are not transformed and, as a result, there is no need for the expensive sort. Nonetheless, this kind of reduction comes at a cost of the overall CSX performance as it is depicted in Fig. 16. To exploit its full potential, CSX must be programmed to detect as many substructure types as possible. In this case, however, the preprocessing cost climbs to several hundreds of serial CSR SpM×V operations. Though not irrational even for on-line preprocessing, this cost is large and can eradicate the performance benefit of CSX in the midterm. For this reason, we employ uniform statistical sampling on the input matrix as described in detail in Section VI. The use of sampling can drop the preprocessing cost nearly an order of magnitude with a minimal impact in CSX's overall performance. A key aspect for sampling a sparse matrix with CSX is to use a lot of sampling windows scattered all over the matrix, in order to obtain a 'good look' of the whole matrix and avoid any over- or under-estimation of the presence of certain substructures. In our case, we have used 48 sampling windows for sampling only 1% of the total non-zero elements of the matrix.

Fig. 16 shows the preprocessing cost of CSX measured in serial CSR SpM×V operations in relation to the achieved performance improvement over the multithreaded CSR in Dunnington using 24 threads. The cost when detecting only delta units (CSX-delta) or horizontal substructures (CSX-horiz) is very small ranging from 16 to 35 CSR SpM×V operations, but only a 14.8% to 23.9% performance improvement is achieved. The use of sampling in detecting all the available substructure types (CSX-sampling) rises to 88 CSR SpM×V operations—an affordable number for on-line preprocessing of the matrix—providing a 38.9% performance improvement.

The preprocessing cost increases significantly with the activation of full preprocessing (no windows, no sampling) approaching the 500 serial CSR SpM×V operations, only to offer a mere 1.5% additional performance improvement over CSR. Nonetheless, the cost of this case is only slightly higher than the sampling cost of our initial CSX implementation [25] (≈350 operations), which reveals the progress performed in optimizing the preprocessing cost in CSX.

For the completeness of our presentation, we provide here a note on the preprocessing cost of the other CSR alternatives considered in the paper. VBL's preprocessing is very small (3–6 serial CSR SpM×V iterations), since it consists only of the conversion cost from CSR to VBL and its performance is comparable to CSX-horiz. For BCSR, the conversion cost is 17–65 SpM×V operations. However, we do not have an exact number for the detection phase of BCSR, since we performed an exhaustive search; we expect the total preprocessing cost of BCSR using statistical block detection to exceed slightly the statistical preprocessing of CSX.

D. Integrating CSX into multiphysics simulation software

As a further evaluation of the potential of CSX in optimizing SpM×V in the context of a multiphysics application, we have integrated it into the Elmer [21] multiphysics simulation software. Elmer employs iterative Krylov subspace methods for treating large problems using the pre-conditioned Bi-Conjugate Gradient Stabilized (BiCGStab) method for the solution of the resulting linear systems. Elmer supports parallelism across multiple nodes using MPI, but uses only a single thread inside every node. For this reason, we have also implemented a multithreaded CSR version for Elmer, in order to achieve a fair comparison with CSX. Fig. 17 shows the average speedup achieved by Elmer in a 24-node, two-way, quad core Intel Xeon E5405/E5335 (Harpetown/Clovertown) mixed cluster (192 cores) for five large ($>$ 576 MiB) benchmark problems of the Elmer test suite. The Clovertown nodes have less memory bandwidth and we have started using them from the 16-node configuration, thus the knee in the speedup diagrams for the multithreaded CSR/CSX versions. We have also used a diagonal preconditioner, which consumed almost half of the execution time in three of our benchmarks. We show results after 1000 linear system iterations, since in practical applications the solver may need thousands of iterations to converge. CSX, including its preprocessing cost, was able to improve the performance of the Elmer's SpM×V component by 37% over the multithreaded CSR, while the performance of the overall solver was improved by a noticeable 14.8%, despite the large preconditioning cost in three of our benchmarks. We believe this improvement could have been much more significant, have other parts of the solver (e.g., the preconditioner) been also multithreaded.

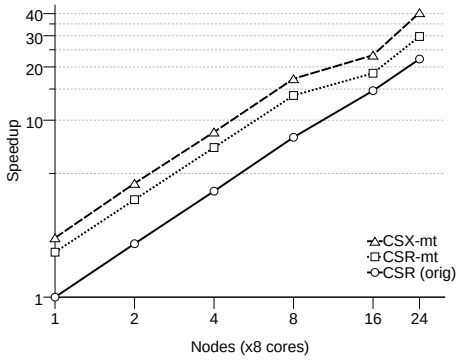
IX. Related work

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 [29]–[31]. 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 [32]. 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. They also describe a *bit map scheme*, where the `colind` structure is replaced by a bit map for identifying the non-zeros pattern inside a row of the matrix. In the same survey, Pooch and Nieder also designate the use of *delta indexing* of the column indices, which is what CSR-DU [12] and DCSR [33] employ for the compression of the `colind` structure.

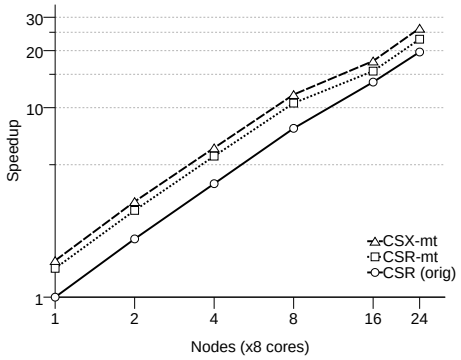
Despite being used as a standard sparse matrix storage format under different descriptions [34], [35], the term *Compressed Sparse Row* format or *CSR* was 'standardized' by Saad [6], [36]. In the same set of works, Saad also coins the terms *Coordinate format (COO)* and *Blocked Sparse Row (BSR)*, which is later standardized as Blocked Compressed Sparse Row (BCSR) by Im and Yelick [8], [37]. Im and Yelick exploit the blocked structure of BCSR for register reuse and also propose a heuristic for automatically selecting the best block size for the matrix. Vuduc et al. [38] investigate the performance bounds of SpM×V using BCSR by modeling the cache and TLB behavior and also extend the auto-tuning capability of SPARSITY [37] by using hybrid offline/online models. In [39], the authors extend BCSR to support unaligned blocks, in order to minimize the BCSR's padding, and also discuss the Variable Block Row (VBR) format, which uses variable size two-dimensional blocks. The work on BCSR and its variations culminated in the OSKI sparse kernel library [28], which provides an auto-tuning sparse matrix optimization framework.

Pinar and Heath [7] employ variable sized one-dimensional blocks and present the Variable Block Length (VBL) format, which we discussed in more detail in previous sections. Agarwal et al. [9] decompose the input matrix into multiple submatrices, each one storing a different substructure. Belgin et al. [24] also employ the same decomposition technique in their Pattern Block Row (PBR) format. Specifically, they store in each submatrix an arbitrary block pattern and generate specific C SpM×V routines for each pattern. Another interesting approach in blocking storage formats for sparse matrices is the Compressed Sparse Block (CSB) format proposed recently by Buluç et al. [22]. The motivation behind this format is the efficient support of both Ax and $A^T x$ sparse matrix operations, the second being less efficient with the row-oriented common storage formats. For this reason, CSB divides the matrix into large sparse square blocks, which are stored using the coordinate storage scheme, but with small integers for the row and column indices. The authors employ task parallelization using Cilk++ and expand their format to support also symmetric matrices in [40].

Willcock and Lumsdaine [33] take a different approach and apply delta compression explicitly in the column indices of CSR, proposing the Delta-Coded Sparse Row (DCSR) format. The also apply run-length encoding for delta distances up to four. Kourtis et al. [12], [41] take a similar approach by delta-encoding the column indices and applying run-length encoding for detecting arbitrary horizontal substructures in the sparse matrix. Kourtis et al. [12] also propose a storage format for compacting the non-zero values by keeping only unique values



(a) Speedup of the SpM×V component.



(b) Speedup of the Elmer solver.

Fig. 17. Speedup of the Elmer multiphysics simulation software by employing CSX (preprocessing time is included) after 1000 linear system iterations.

and an associated indexing structure. Escudero [42] has also presented a similar approach in his *Super-Sparse (SS)* format.

Considerable research has also been conducted recently in characterizing the performance behavior of SpM×V. Williams et al. [4] perform a performance evaluation of the SpM×V kernel in modern commodity and emerging multicore architectures and evaluate the performance impact of a set of different optimizations. Goumas et al. [3] provide an in-depth experimental evaluation of the SpM×V kernel in modern commodity SMP and NUMA arhictectures identifying and classifying the different performance problems of the kernel. Karakasis et al. [26], [43] expand this evaluation in blocked storage formats and investigate the impact of block shapes on the performance of the SpM×V kernel.

X. Conclusions

In this paper, we have presented in detail and evaluated the performance of the Compressed Sparse eXtended (CSX) sparse matrix storage format in a variety of modern multicore architectures. CSX integrates in a single storage format the most commonly encountered substructures inside sparse matrices, including horizontal, vertical, diagonal and two-dimensional substructures. In conjuction with the highly compressed representation of the substructure indexing information, CSX is able to minimize the memory footprint of the sparse matrix, therefore alleviating the pressure exerced to the memory subsystem, induced by the highly streaming nature of the SpM×V kernel. Compared to other CSR alternatives, like BCSR and VBL, CSX is able to provide considerable performance improvements not only in symmetric shared memory architectures, but also in NUMA platforms, where the computational part of the kernel is more prominent. Even in cases that it does not achieve the best overall performance, CSX manages to stay very close to the best, exposing stability that lacks from other, more monolithic, formats. Most notably, this performance advantage does not come at an excessive preprocessing cost, since CSX employs advanced techniques for minimizing this cost. Indeed, CSX was able to considerably accelerate the performance of a multiphysics simulation, despite its initial bootstrap cost.

As a future research direction, we plan to investigate alternative parallelization schemes for CSX, e.g., task-parallelism techniques, and experiment with more advanced heuristics for the selection of the substructures for encoding in CSX, especially in architectures where the computational part of the kernel is more exposed. Finally, we plan to exploit the insight of the matrix structure that CSX offers during its preprocessing phase, in order to guide the SpM×V execution for achieving high performance and increased energy-efficiency.

References

[1] 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.

[2] P. Colella, "Defining software requirements for scientific computing," 2004, DARPA's High Productivity Computer Systems (pres.).

[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] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. Portland, OR, USA: ACM, 1999.

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

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

[10] R. Vuduc, "Automatic performance tuning of sparse matrix kernels," Ph.D. dissertation, University of California, Berkeley, 2003.

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

[12] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proceedings of the 5th conference on Computing frontiers*. Ischia, Italy: ACM, 2008.

[13] C. Lattner, "LLVM and Clang: Advancing Compiler Technology," in *Free and Open Source Developers' European Meeting*, Brussels, Belgium, Feb. 2011. [Online]. Available: http://clang.llvm.org/

[14] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization (CGO'04)*. San Jose, CA, USA: IEEE Computer Society, 2004. [Online]. Available: http://www.llvm.org/

[15] Y. Saad, "Krylov subspace methods for solving large unsymmetric linear systems," *Mathematics of Computation*, vol. 37, no. 155, pp. 105–126, 1981.

[16] ——, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[17] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.

[18] 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.

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

[20] H. A. van der Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, no. 2, pp. 631–644, 1992.

[21] M. Lyly, J. Ruokolainen, and E. Järvinen, "ELMER – a finite element solver for multiphysics," in *CSC Report on Scientific Computing*, 1999–2000. [Online]. Available: http://www.csc.fi/english/pages/elmer

[22] 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.

[23] A. W. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH Computer Architectures News*, vol. 23, no. 1, 1995.

[24] M. Belgin, G. Back, and C. J. Ribbens, "Pattern-based sparse matrix representation for memory-efficient SMVM kernels," in *Proceedings of the 23rd international conference on Supercomputing (ICS'09)*. Yorktown Heights, NY, USA: ACM, 2009, pp. 100–109.

[25] 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.

[26] V. Karakasis, G. Goumas, and N. Koziris, "Exploring the effect of block shapes on the performance of sparse kernels," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. Rome, Italy: IEEE Computer Society, 2009, pp. 1–8.

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

[28] 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.

[29] 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.

[30] A. R. Curtis and J. K. Reid, "The solution of large sparse unsymmetric systems of linear equations," *IMA Journal of Applied Mathematics*, vol. 8, pp. 344–353, 1971.

[31] F. G. Gustavson, "Some basic techniques for solving sparse systems of linear equations," in *Sparse Matrices and Their Applications*. Plenum Press, 1972, pp. 41–52.

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

[33] J. Willcock and A. Lumsdaine, "Accelerating sparse matrix computations via data compression," in *Proceedings of the 20th annual International conference on Supercomputing*. Cairns, QLD, Australia: ACM, 2006, pp. 307–316.

[34] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, "Yale sparse matrix package I: The symmetric codes," *International Journal for Numerical Methods in Engineering*, vol. 18, pp. 1145–1151, 1982.

[35] S. Pissanetzky, *Sparse Matrix Technology*. London, UK: Academic Press, 1984.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] K. Kourtis, G. Goumas, and N. Koziris, "Exploiting compression opportunities to improve SpMxV performance on shared memory systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 3, 2010.

[42] L. F. Escudero, "Solving systems of sparse linear equations," *Advances in Engineering Software*, vol. 6, pp. 141–147, 1984.

[43] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *12th IEEE International Conference on Computational Science and Engineering*. Vancouver, Canada: IEEE Computer Society, 2009.