CSX: An Extended Compression Format for SpMV on Shared Memory Systems

Kornilios Kourtis Vasileios Karakasis Georgios Goumas Nectarios Koziris
National Technical University of Athens
{kkourt,bkk,goumas,nkoziris}@cslab.ece.ntua.gr

Abstract

The Sparse Matrix-Vector multiplication (SpMV) kernel scales poorly on shared memory systems with multiple processing units due to the streaming nature of its data access pattern. Previous research has demonstrated that an effective strategy to improve the kernel's performance is to drastically reduce the data volume involved in the computations. Since the storage formats for sparse matrices include metadata describing the structure of non-zero elements within the matrix, we propose a generalized approach to compress metadata by exploiting substructures within the matrix. We call the proposed storage format *Compressed Sparse eXtended (CSX)*. In our implementation we employ runtime code generation to construct specialized SpMV routines for each matrix. Experimental evaluation on two shared memory systems for 15 sparse matrices demonstrates significant performance gains as the number of participating cores increases. Regarding the cost of CSX construction, we propose several strategies which trade performance for preprocessing cost making CSX applicable both to online and offline preprocessing.

Categories and Subject Descriptors J.2 [*Computer Applications*]: Physical Sciences and Engineering

General Terms Algorithms, Performance

Keywords Sparse Matrix-Vector Multiplication, Shared Memory, compression, SpMV, SMP

1. Introduction

Multicore processors have become the trend in all aspects of computing (commodity products, high performance systems, and future research directions). A factor that limits the ability of applications to scale on a large number of cores is the sharing of the memory hierarchy by the processing units. Applications with no data dependencies and good temporal locality tend to scale well, since each core can work independently using local data residing in its cache without interfering with the operation of other cores. On the other hand, applications with streaming access patterns tend to exhibit poor scaling due to contention on the memory subsystem. A technique to improve the multithreaded performance of these applications is to trade memory cycles for computation cycles via data compression. In other words, as core count increases, performing redundant computations to avoid memory accesses has the potential of increasing the scalability in applications with streaming memory accesses.

An important and ubiquitous computational kernel with streaming memory access pattern is the Sparse Matrix-Vector multiplication (SpMV). It is used in a large variety of applications in scientific computing and engineering. For example, it is the basic operation of iterative solvers, such as Conjugate Gradient (CG) and Generalized Minimum Residual (GMRES), which are extensively used to solve sparse linear systems resulting from the simulation of physical processes described by partial differential equations [21]. Furthermore, SpMV is a member of one of the "seven dwarfs", which are classes of applications that are believed to be important for at least the next decade [3].

The distinguishing characteristic of sparse matrices is that they are dominated by a large number of zeros, making it highly inefficient to perform operations using typical (dense) array structures. Special storage schemes are used instead, which target both the reduction of the storage requirements of the matrix and the efficient execution of various operations by performing only the necessary computations. Thus, the common approach is to store only the non-zero values of the matrix and employ additional indexing information representing the position of these values (*index data*).

Our previous work [2, 9] has identified the memory subsystem as the main performance bottleneck of the SpMV kernel. Obviously, this problem becomes more severe in a multithreaded environment, where multiple processing cores access the main memory. An approach for alleviating this problem is the reduction of the data volume accessed during the execution of the kernel (*working set*). In [15], we proposed the CSR-DU (CSR with Delta Units) storage format as a way to reduce the index data of non-zero elements across the same matrix row by applying compression. The approach is effective as it can significantly benefit the performance of the multithreaded SpMV kernel [14]. CSR-DU employs a coarse grain delta encoding technique; the sparse matrix is divided into areas, called *units*, with a variable number of elements, and for each of these areas the minimum size for representing the encoded delta values is selected. Following the same philosophy, Belgin et al. [5] proposed a storage scheme called PBR that exploits frequently repeated patterns within the matrix to reduce the working set and improve performance. However, both approaches are quite conservative in mining the regular patterns of sparse matrices, since CSR-DU scans patterns along a single row, while PBR scans patterns strictly row and column-aligned within specific block areas.

The work in this paper was motivated by two key observations: first, the utilization of regular patterns within matrices leads to significant performance improvements and, second, thorough inspection of the sparse matrix to mine as many patterns as possible can be a viable approach in a large class of applications, where the

[Copyright notice will appear here once 'preprint' option is removed.]

same matrix is used across numerous runs. For example, the typical linear system $Ax = b$ may be solved repeatedly for the same matrix A and different right-hand statements b. This second observation is also supported by the various preprocessing algorithms that are used to reorder a sparse matrix in order to reduce its bandwidth [21]. To this direction, we propose a generalized and elaborate storage format based on compression that aims at achieving more aggressive reduction in matrix metadata. We call this storage format *Compressed Sparse eXtended (CSX)*. In the proposed implementation, CSX is able to represent frequently occurring substructures along the same row, column, and diagonal, as well as dense two-dimensional blocks. To the best of our knowledge, this is the most general and flexible representation scheme for sparse matrices. We handle the above substructures in a unified way, by expressing each one of them as an appropriate transformation. We transform the initial matrix coordinates according to this transformation, partially sort the transformed coordinates lexicographically, and mine the non-zero elements that belong to the substructure.

Our experimental results indicate that CSX is able to provide significant performance improvements over a number of existing storage formats. Regarding the cost of preprocessing, the complexity is kept linear to the number of the non-zero elements of the input matrix. The total preprocessing time ranges from tens to a few thousands of serial CSR SpMV operations depending on the number of patterns being detected. The higher cost of preprocessing for CSX when trying to detect a large number of substructures makes it practical only for offline matrix preprocessing. However, by limiting the considered substructures, or searching for all substructures in a small part of the matrix using sampling, we manage to drop the preprocessing cost to a few hundreds of SpMV operations without severe loss in the final CSX performance. This makes CSX suitable also for online preprocessing of the input matrix.

The rest of the paper is organized as follows: Section 2 presents some relevant background information. Section 3 presents the CSX storage format together with its implementation details, while Section 4 presents the results of the performance evaluation. Section 5 discusses related work, and Section 6 concludes the paper and discusses directions for future work.

2. Background

2.1 Sparse matrix formats and the SpMV operation

The most commonly used storage format for sparse matrices is the Compressed Sparse Row (CSR) format [4, 21]. In CSR the matrix is stored in three arrays: `values`, `row_ptr`, and `col_ind` . The `values` array stores the non-zero elements of the matrix in row-major order, while the other two arrays store indexing information: `row_ptr` contains the location of the first (non-zero) element of each row within the `values` array and `col_ind` contains the column number for each non-zero element. An example of the CSR format for a 6×6 sparse matrix is presented in Figure 1. The size of the `values` and `col_ind` arrays are equal to the number of non-zero elements (`nnz`), while the `row_ptr` array size is equal to the number of rows (nrows) plus one. Other generic formats for sparse matrices are the Compressed Sparse Column (CSC), which is similar to CSR storing columns instead of rows, and the Coordinate format (COO), where each non-zero is stored as a triplet along with the coordinates of its location in the matrix.

The SpMV operation ($\mathtt{y} = A\mathtt{x}$), is the multiplication of a sparse matrix A and a (dense) vector `x`, with the result stored in the (dense) output vector `y`. The operation is easily implemented for matrices stored in CSR form. The SpMV code for a matrix with N rows in CSR format is shown in Figure 2. The working set (`ws`) of the CSR SpMV operation consists of the matrix and vector data. Its size is

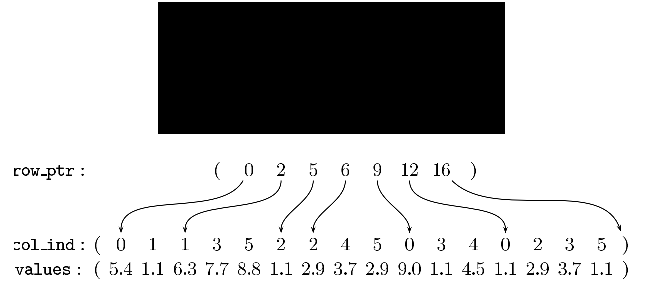


Figure 1. Example of the CSR storage format.

```
for (i=0; i<N; i++)
  for (j=row_ptr[i]; j<row_ptr[i+1]; j++)
     y[i] += values[j]*x[col_ind[j]];
```

Figure 2. SpMV code for the CSR format.

expressed by the following formula:

$$ws = \overbrace{(nnz \times (idx_s + val_s) + (nrows + 1) \times idx_s)}^{\text{sparse matrix}} + \overbrace{(nrows + ncols) \times val_s}^{\text{vectors}}$$

The idx_s and val_s terms represent the storage size required for an index and a value respectively. Since for most real-life sparse matrices it holds $nnz \gg nrows, ncols$, the most dominant terms of the working set is the size of the `col_ind` and `values` arrays, which have `nnz` elements. Commonly, vectors `x` and `y` have less than 2^{32} elements due to memory size restrictions and thus a 4-byte integer is used for index storage. Floating point values, however, normally require double precision, so the typical value for val_s is 8 bytes. Under these conditions, values constitute the larger portion of the working set by a factor of $2/3$, if we consider only the `col_ind` and `values` arrays. Hence, compression of index data after a point leads to diminishing returns because value data dominate the working set.

2.2 The CSR-DU storage format

Sophisticated sparse storage formats traditionally try to exploit contiguous elements, either in one or two dimensions. Examples include the BCSR format [20] and the variable length one-dimensional block format described in [19]. The index data compression approach of the CSR-DU storage format is based on the general premise that sparse matrices have dense areas, which do not necessarily contain contiguous non-zero elements. These areas can contribute significantly to index data size reduction, when delta encoding is used to reveal the highly redundant nature of the `col_ind` array [26]. In a delta encoding scheme the column indices are replaced with *deltas*, each of which is defined as the difference of the current index with the previous one. Since delta values are positive and less or equal than their corresponding column indices, they can be stored in smaller size integers, leading to index data size reduction. The CSR-DU storage format [15] is based on a coarse grain delta encoding scheme. The matrix is divided into areas called units, each of which is characterized by the minimum integer size able to represent the unit's delta-encoded column indices. For example, if for the delta values of the unit stands that $\delta_i \leq 2^8$, then only one byte is required for storing the delta values, while a unit with $2^8 < \delta_i \leq 2^{16}$ requires two-byte integers for storing the column index.

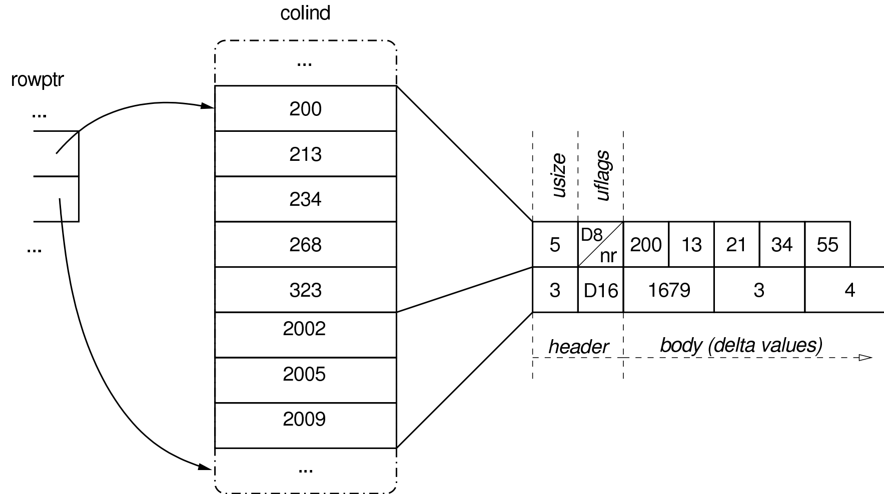


Figure 3. Example of the CSR-DU storage format, where a row is split into two units.

In the implementation of CSR-DU presented in [15], unit information is stored in a single byte-array called `ctl`, which consists of a *header* with the properties of the unit and the *main body* that includes the delta-encoded column indices. In the most simple form of CSR-DU, the header consists of two one-byte fields: `usize`, which is the number of elements the unit contains, and `uflags`, which encodes its characteristics. Since `usize` is stored in a single byte, the maximum possible number of elements per unit is $2^8 = 256$. The size in bytes (1, 2, 4 or 8 bytes) of the delta values stored in the main body can be extracted from the `uflags` field, along with a marker that designates the beginning of a new row. Figure 3 presents an example of the CSR-DU format. In this example a row with 8 elements is split into two units. The first unit has 5 elements, 1-byte delta sizes and a designator for a new row (`nr`), while the second unit has 3 elements and 2-byte delta sizes. The actual implementation of CSR-DU in [15] also includes a column index offset from the previous unit in the header. The offset is called `ujmp` and is stored as a (positive) variable-length integer at the end of the header.

3. Compressed Sparse eXtended (CSX)

3.1 Motivation and approach

Although the SpMV kernel is essentially simple, its actual and maximum performance depends strongly on the nature of the sparse matrix. A generalized storage format, like CSR, does not make any assumptions about the sparse matrix data, and thus it cannot exploit optimization opportunities that arise from special properties of the matrix. In the following paragraphs we describe a general optimization approach for the SpMV kernel based on a new storage format for sparse matrices, that targets the exploitation of regularities in the matrix. We call this storage format Compressed Sparse eXtended (CSX). We argue that this approach can be used for a wide range of optimizations that exploit matrix-specific knowledge. Our method is based on the extension of the concept of *units* introduced by CSR-DU to support different and arbitrary classes of regularities. Focusing on the generality and flexibility of our method, we employ a runtime code generation technique [13], where a specialized, matrix-specific SpMV operation is created, along with the encoded matrix data.

3.2 Exploiting substructures within the sparse matrix

Sparse matrices commonly represent physical structures of computational domains. For this reason, they carry repeated substructures of non-zero elements, expressing, for example, connectivity between elements of a domain. To the best of our experience, the majority of the matrices expose these regularities along the same row, column, diagonal or may span multiple rows (columns or diagonals) in a two-dimensional structure. In the proposed storage format of CSX we consider the above kinds of substructures described in greater detail in the following paragraphs. Nevertheless, if necessary, other classes of regularities can be incorporated in the format to optimize matrices with different characteristics.

Horizontal substructures

In order to mine horizontal substructures within the sparse matrix, we extend the CSR-DU storage format to apply more aggressive index compression by employing run-length encoding in the delta values in multiple directions. Drawing inspiration from the variable length one-dimensional block format described in [19], we generalize the notion of sequential elements to elements with a constant distance. As with sequential elements $(\alpha, \alpha+1, \alpha+2, \ldots)$, elements with a constant distance $(\alpha, \alpha+\delta, \alpha+2\delta, \ldots)$ can be encoded using only the initial value, the constant distance, and their size. Since this technique essentially applies a run-length encoding on the delta values of the column indices, we will refer to it as *delta run-length encoding*.

Vertical and diagonal substructures

To further enhance index compression, we augment our approach to include multiple directions for the sparse matrix elements. The directions we consider are vertical, diagonal and anti-diagonal (see Tab. 1) with the same rationale as in the horizontal case.

Two-dimensional substructures

Two-dimensional substructures are common in sparse matrices, especially in those that arise from problems with underlying 2D/3D geometry [1, 11, 12]. Storage formats that exploit these structures, e.g., BCSR, can provide significant speedups over the standard CSR implementation in many cases, since apart from reducing the SpMV working set, they exhibit good computational characteristics [12, 24]. Therefore, we further extend our approach to support 2D-blocks.

Direction		Elements	
		y	x
Horizontal	$\rightarrow$	y_0	$x_0 + i\delta$
Vertical	$\downarrow$	$y_0 + i\delta$	x_0
Diagonal	$\searrow$	$y_0 + i\delta$	$x_0 + i\delta$
Anti-diagonal	$\swarrow$	$y_0 + i\delta$	$x_0 - i\delta$

Table 1. Directions for delta run-length encoding. The y and x columns contain an expression for generating the matrix elements for the specific direction given its delta value δ. Note that $0 \leq i < size$, where $size$ the number of elements in the unit.

Figure 4 presents the distribution of the most dominant substructures in the 15 matrices of our suite as discovered by CSX. Delta units are designated with DUx, where x is the number of bits used for the delta values. Dense blocks are either brow or bcol, depending on the alignment (see Section 3.3), with the numbers inside parenthesis indicating the block dimensions. Finally, the rest of the substructures are indicated with their literal name and the delta value inside parenthesis. We observe that there is a significant representation of the aforementioned substructures in our matrix suite, indicating that there is space for performance improvement by utilizing them. Interestingly enough, matrix rajat31 contains a rather uncommon substructure, since more than 20% of its non-zero elements lie along the same diagonal with step 11. Our approach is able to capture this substructure. How CSX discovers the different patterns inside a sparse matrix is discussed in the next section, while the matrix suite is detailed in Section 4.

3.3 CSX matrix construction

Our algorithm to construct the CSX matrix handles the supported substructures in a uniform way. The algorithm is based on a delta run-length encoding detector for the horizontal direction, which detects sequences (*runs*) of the same delta value. If the number of elements in a run is larger or equal than a specific configuration parameter, then the items are grouped together in a single unit. The detector can be easily implemented if we assume that the elements can be iterated in lexicographical order. To simplify the detection process, detection of overlapped runs is not supported. An example is presented in Figure 5, where the detector has been configured to detect runs of size larger than or equal to 4. Note that it does not detect the run of the indices $41, 61, 81$, since its size is 3, and it also does not detect the run of the indices $1, 21, 41, 61, 81$ since it overlaps with other elements. The algorithm for the detector of horizontal substructures is given in Alg. 1.

indices:	1	10	11	12	13	14	21	41	61	81
delta values:	1	9	1	1	1	1	7	20	20	20
			1x4							

Figure 5. Horizontal detection example.

In order to detect the rest of the substructures discussed in the previous section, we use the horizontal detector and apply appropriate transformations on the matrix points coordinates. For example, to detect vertical runs we swap the coordinates of the elements. The transformations for the substructures implemented within CSX are shown in Table 2.

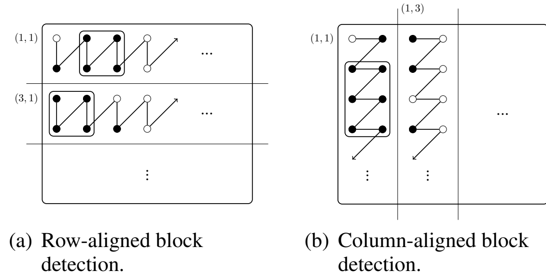
For the 2D substructures, we need a non-linear transformation that will transform the 2D space into an 1D space. Since the input matrix is sparse, we chose not to map the 2D coordinate space of the whole matrix to 1D using a typical space-filling curve, e.g., Morton-order curve, because such an order (a) would require a linear space of $O(nrows^2)$ size, which is impractical for very large and sparse matrices, and (b) would imply a strict requirement on

Algorithm 1: Horizontal detector.

Input: $indices$: array of size n column indices
Input: lim: the minimum size for DRLE units

$deltas$ = deltaEncode($indices$)
$delta_{rle} \leftarrow deltas[0]$
$freq_{rle} \leftarrow 1$

for $i \leftarrow 1$ **to** n **do**
  if $deltas[i] == delta_{rle}$ **then**
    $freq_{rle} \leftarrow freq_{rle} + 1$
  else
    if $freq_{rle} \geq lim$ **then** encode in DRLE units
    else keep individual indices
    $delta_{rle} \leftarrow deltas[i]$
    $freq_{rle} \leftarrow 1$



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

Figure 6. Detection of two-dimensional dense substructures (black dots are non-zero elements).

alignment and size of blocks, since such transformations are based on bit transformations. For that reason, we segment the matrix into horizontal (or vertical) bands of a specific width and apply a simple space-filling transformation inside that band. Figure 6 presents that schematically for bands of width 2, and Table 2 presents the exact formulas of the transformations, when segmenting the matrix horizontally or vertically. In a single detection phase, we are now able to detect all $r \times c$ blocks, with r or c constant, aligned at either r-rows or c-columns boundaries, respectively, depending on the orientation of the segmentation of the matrix. Although it is possible to detect totally unaligned blocks, this would induce additional space and time complexity. In detecting dense sub-blocks of the matrix, we instruct our detector to search only for substructures with $\delta = 1$.

Our generic substructure detector processes the input matrix in windows of w non-zero elements as it is depicted in Alg. 2. For each processing window, we transform the matrix elements and sort them lexicographically before passing them to the horizontal detector. After the detection phase completes, we transform back the window elements. The asymptotic complexity is determined by the complexity of sorting the elements of a window and the total number of processing windows. Assuming non-overlapping windows, the asymptotic complexity of the horizontal detector is $\lceil \frac{nnz}{w} \rceil O(w \log(w)) = O(nnz)$. Note that even when using overlapping windows, the asymptotic complexity does not change. Finally, increasing the processing window so that it approaches nnz lets us mine almost every substructure in the matrix, but increases considerably the preprocessing time.

In order to mine all the considered substructures/patterns from the matrix, we run the generic detector for each supported class of patterns (e.g., horizontal, diagonal, blocks with 2-row alignment,

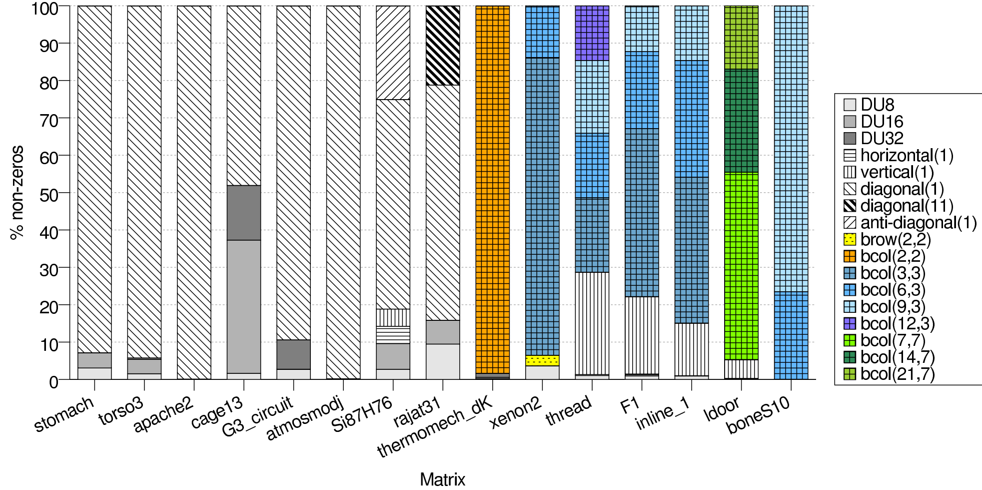


Figure 4. Distribution of substructures in the matrices of our suite.

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

Table 2. The transformations used to convert the different substructures to horizontal, in order to feed the CSX pattern detector. The number of rows of the matrix is denoted by $nrows$, and r and c are the required alignments for row- and column-aligned blocked substructures.

Algorithm 2: Generic detector.

Input: $elems$: array of size nnz of matrix elements
Input: w: size of processing window
Input: f: transformation function
Input: lim: the minimum size for DRLE units

$nw \leftarrow \lceil \frac{nnz}{w} \rceil$
for $i \leftarrow 1$ **to** nw **do**
  $welems \leftarrow$ `extract_window`$(elems, w)$
  $welems \leftarrow f(welems)$
  `sort`$(welems)$
  `horizontal_detector`$(welems, lim)$
  $welems \leftarrow f^{-1}(welems)$

etc.) and record an estimate of the gain in the total size of the matrix (we ignore all patterns that cannot encode more than 10% of the non-zero elements of the matrix). The class of patterns that maximizes this estimate is selected for encoding and the procedure is repeated for the rest, unencoded, elements of the matrix until no other encoding can be applied. The remaining elements are then stored in delta units in row-major order. The estimate used to score the different encodings is based on the idea that the major gain in the size of the matrix would come from the reduction of the `col_ind` structure. In order to keep our estimate simple, we assume that we keep one index for each pattern and one index for each unencoded element. So, if $npatt$ is the number of the encoded patterns and nnz_{enc} the number of the non-zeros encoded by this pattern, the selection criterion is

$$\begin{aligned} G &= \overbrace{nnz}^{\text{initial}} - (\overbrace{npatt}^{\text{encoded}} + \overbrace{nnz - nnz_{enc}}^{\text{unencoded}}) \\ &= nnz_{enc} - npatt \end{aligned} \quad (1)$$

3.4 Multiplication routines

After the matrix construction is complete, the `ctl` byte-array and the SpMV code needs to be generated. We use an encoding similar to CSR-DU where, as depicted in Figure 3, each unit starts with two bytes: `uflags` and `usize`. In CSX, each unit represents an encoded pattern. The `usize` byte contains the number of elements for the specified unit and the `uflags` its type along with some bookkeeping information. From the 8 bits of `uflags`, 6 are reserved for the encoding of the type, and 2 are used for a new row marker and a row offset marker. If the row offset bit is set, the header is followed by a variable-length integer equal to the number of empty rows. This is necessary, because the use of CSX units in directions other than the horizontal may lead to empty rows. For each unit type available, a unique 6-bit identifier is allocated for the type encoding, therefore limiting the maximum number of different unit types to 64. Different unit types may belong to the same class of patterns, e.g., horizontal(1) and horizontal(2) are different unit types, but belong to the horizontal class of patterns.

Our dynamic code generation approach uses the LLVM [16] compiler infrastructure. A core component of LLVM is its intermediate representation (IR), which resembles a RISC-like assembly, and it can be manipulated by optimization passes and used to produce native code for a number of different ISAs. The code for the SpMV operation is generated programmatically in LLVM's IR

and is specific to each unit type. Subsequently, it is optimized and dynamically compiled into native code. An on-disk cache of generated versions can be used to reduce the overhead of compilation and optimization.

3.5 Restrictions and extensions

Overall, CSX is a very flexible storage format supporting several different substructures within the sparse matrix simultaneously. However, to simplify the implementation and reduce the run-time cost of preprocessing, a number of restrictions have been applied. As discussed above, our implementation does not detect overlapping substructures and fully unaligned two-dimensional blocks (row or column alignment is required). Even with these restrictions, however, CSX is able to group the vast majority of the non-zero elements in our matrix suite (see Fig. 4) in utilizable substructures. Quite importantly, our detection approach based on transformations provides a straightforward mechanism to incorporate further meaningful substructures, provided they can be expressed as a coordinate transformation as well.

The preprocessing phase for CSX is broken down into two major steps: substructure detection and matrix construction (encoding). Both steps require sorting of the processed elements. However, as discussed before, processing non-zero elements in windows keeps the overall complexity of the preprocessing phase linear to the number of non-zero elements of the matrix, without wasting a significant number of substructures. However, in the current implementation of CSX, processing in windows is performed only for the first step (detection) while the encoding step sorts all processed elements. This adds some additional cost to the preprocessing, which will be improved in future versions of the preprocessor.

The algorithm used by the detector to select the substructures to be encoded in the final CSX format is greedy in that it always selects the local optimal solution based on the criterion (1). This might lead to a suboptimal set of encoded substructures. Finally, our selection criterion considers only the reduction in the size of the encoded matrix and ignores any differences in the computational characteristics of the different substructures. We are also investigating improvements on these topics, such as faster search methods and more elaborate selection criteria.

4. Experimental evaluation

4.1 Experimental setup and methodology

We performed an experimental evaluation of the proposed storage format on two SMP platforms: (a) a 2-way quad-core system (totaling 8 cores) based on Intel Harpertown processors, and (b) a 4-way six-core system (totaling 24 cores) based on Intel Dunnington processors. Figure 7 and Table 3 present the cache hierarchies and the basic microarchitectural characteristics of our platforms.

	Harpertown	Dunnington
Model	E5405	X7460
Frequency (GHz)	2.0	2.66
L1 (data/instr.)	32K/32K	32K/32K
L2 (unified)	6M (1/2 cores)	3M (1/2 cores)
L3 (unified)	–	16M
Number of cores	$2 \times 4 = 8$	$4 \times 6 = 24$

Table 3. System characteristics of used hardware platforms.

Both systems run a 64-bit version of the Linux OS (kernel version 2.6). We used version 2.5 of the LLVM compiler infrastructure and llvm-gcc 4.2.1 (a modified version of gcc that acts as a front-end for LLVM) as a static compiler. The parallelization of all versions of the SpMV kernel was done explicitly, using the POSIX

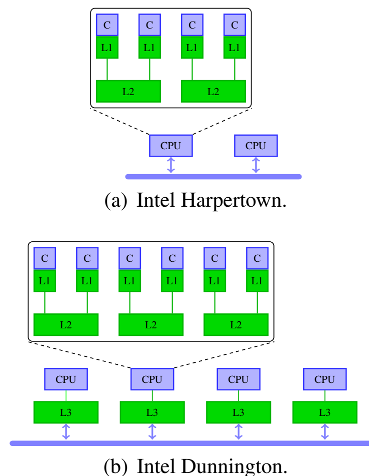


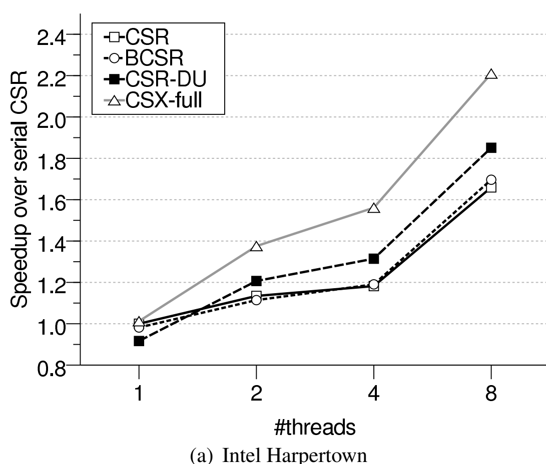
Figure 7. Cache hierarchies of used hardware platforms.

threads interface of the GNU C library (NPTL 2.7). Moreover, the `sched_setaffinity()` system call was used to bind the various threads to predefined cores.

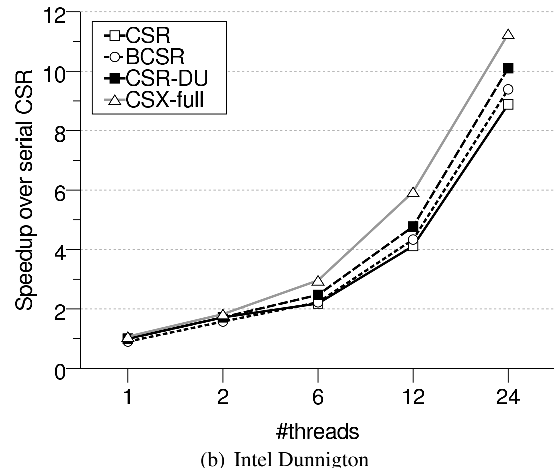
The experiments were conducted by measuring the execution time of 128 consecutive SpMV operations with randomly created input vectors. We made no attempt to artificially pollute the cache after each iteration, in order to better simulate iterative scientific application behavior, where matrix data are present in the cache hierarchy, because either they have just been produced or they were recently accessed. By using multiple iterations, we induce temporal locality to our benchmark, and thus the streaming behavior of the SpMV kernel is maintained only if the working set, and more specifically the matrix data, are larger than the system's cache. For the multithreaded versions of the kernel, we used row partitioning and a static balancing scheme based on the non-zero elements, where each thread is assigned approximately the same number of non-zero elements and, thus, the same number of floating-point operations. The threads are always scheduled to run as "close" to the processors as possible, e.g., on Harpertown 2 threads are scheduled on the cores sharing the L2 cache, while 4 threads are scheduled on the same physical package (similarly for Dunnington). For the CSX matrix construction, we initially partition the data based on the number of non-zero elements and for each partition we perform the analysis and generate different SpMV versions for each thread.

We consider four different formats in our tests: CSR as the baseline standard format, BCSR as a state-of-the-art improvement over CSR, CSR-DU and CSX. The code for the CSR SpMV kernel was optimized to write the `y[i]` value at the end of each innermost loop, by keeping the intermediate result in a register. For BCSR we use block-specific optimized routines and report the best performing block shape out of a large number of candidates. Finally, we use a window of $w = 32K$ non-zero elements for the analysis of the matrix for CSX. We used 64-bit numerical values for all formats and 32-bit indices for CSR and BCSR.

The matrix suite that we used for the performance evaluation of our proposed technique consists of 15 matrices selected from the University of Florida sparse matrix collection [7]. We made an effort to include matrices that arise from different kind of problems. Table 4 presents our matrix suite and the characteristics of every matrix. For the sake of presentation, we have arranged the sparse matrices so that the first matrices are dominated by one-dimensional substructures while the latter from two-dimensional.



(a) Intel Harpertown



(b) Intel Dunnigton

Figure 8. Average speedup of SpMV execution time over the serial CSR version for all considered formats.

Matrix	rows	cols	nnz	size (MB)	Problem
stomach	213360	213360	3021648	35.39	2D/3D
torso3	259156	259156	4429042	51.67	2D/3D
apache2	715176	715176	4817870	57.86	Structural
cage13	445315	445315	7479343	87.29	Graph
G3_circuit	1585478	1585478	7660826	93.72	Circuit Sim.
atmosmodj	1270432	1270432	8814880	105.72	CFD
Si87H76	240369	240369	10661631	122.93	Chemistry
rajat31	4690002	4690002	20316253	250.39	Circuit Sim.
thermomech_dK	204316	204316	2846228	33.35	Thermal
xenon2	157464	157464	3866688	44.85	Materials
thread	29736	29736	4470048	51.27	Structural
F1	343791	343791	26837113	308.44	Structural
inline_1	503712	503712	36816342	423.25	Structural
ldoor	952203	952203	46522475	536.04	Structural
boneS10	914898	914898	55468422	638.28	Model Reduction

Table 4. The matrix suite used for the experimental evaluation. The size column displays the size of the matrix in MB when stored in the CSR format.

4.2 CSX SpMV performance

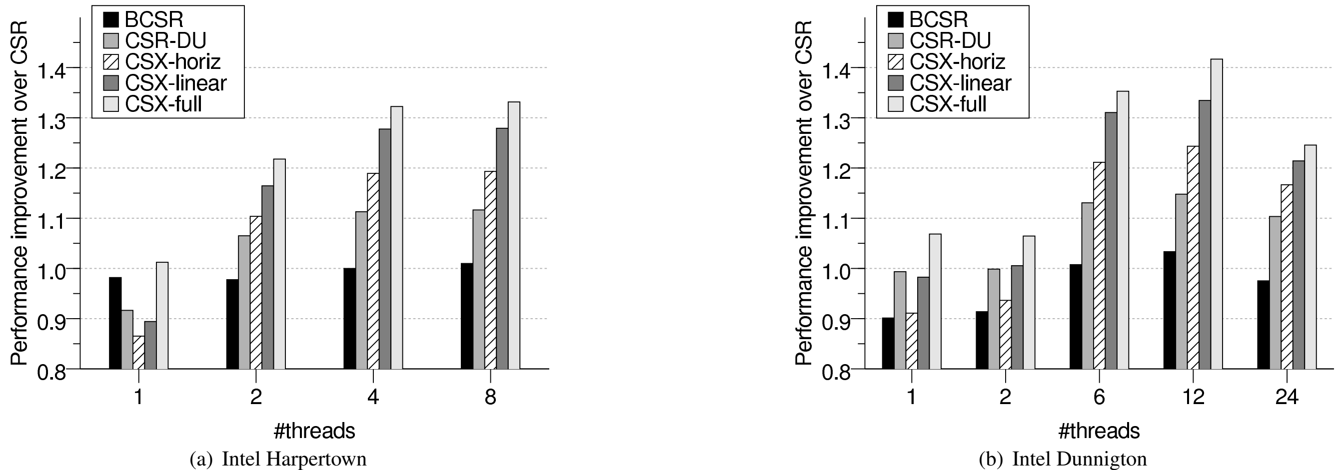
This section discusses the effect of the CSX storage format on the performance of the SpMV kernel. Figure 8 demonstrates the average speedup of the considered formats over serial CSR for all matrices on our platforms, while Fig. 9 presents the performance improvement of different versions of CSX over the multithreaded CSR.

The serial version of CSR-DU and BCSR on Harpertown lead to -8% and -2% performance degradation on average, respectively, while CSX manages to achieve a marginal 1% performance improvement. On Dunnington, CSR-DU almost matches the performance of CSR, BCSR falls further behind at -10%, and CSX gains 7% over CSR. When moving to multiple threads, however, the memory bandwidth problem becomes dominant and the compression formats show their potential. CSX achieves a 2.21 speedup on Harpertown (33% performance improvement over CSR) when all 8 cores are utilized and 11.27 speedup (25% improvement over CSR) on Dunnington when all 24 cores are utilized. The speedups for CSR-DU are 1.85 ($+11.6\%$ over CSR) and 10.1 ($+10.3\%$ over CSR) for the Harpertown and Dunnington platforms, respectively. BCSR cannot provide any speedup over CSR on average, since it suffers from excessive padding. However, BCSR can provide significant performance improvement over CSR for individual sparse matrices with a lot of dense sub-blocks (see Fig. 10). As depicted in Fig. 9, the best performance improvement over CSR on Dunnington is achieved when using 12 threads and not 24, as it might be expected. This happens because 6 matrices of our suite, even in CSR format, fit into the platform's aggregate cache, which totals 64 MB. This also explains the superlinear speedup encountered when moving from 12 to 24 threads on Dunnington for almost all the formats. For this reason, we will focus on the 12-thread configuration on Dunnington in the following, where only two matrices (stomach and thermomech_dK) are close to the aggregate cache (32 MB). In the 12-thread configuration, when the memory bottleneck is the most apparent, the performance improvement of CSX over CSR reaches 42%.

Figure 10 presents also the performance improvement of two lighter versions of CSX, namely CSX-horiz and CSX-linear, which detect only horizontal or one-dimensional patterns. CSX-full is the full version of CSX, which also detects two-dimensional patterns. It is apparent that both these lightweight versions of CSX can provide considerable performance improvement over CSR and CSR-DU. Although Fig. 4 shows that horizontal patterns are not dominant in our suite, the block patterns are mined as horizontal when block detection is disabled. However, the diagonal patterns discovered by CSX-linear and CSX-all cannot be discovered by CSX-horiz and are encoded as CSR-DU delta units in this case. This explains the escalation in performance as we move from the pure CSR-DU to CSX-horiz and then to more elaborate pattern detection schemes.

Figure 10 presents a detailed per matrix view of the performance improvement achieved by BCSR, CSR-DU, and the different CSX variants when 8 cores are used on Harpertown and 12 cores on Dunnington. CSX managed to achieve the best performance in all but one matrix, namely thermomech_dK, where BCSR achieved the best performance. This is not a surprise, since this matrix has almost all of its non-zero elements in 2×2 dense blocks (see Fig. 4), which, apparently, BCSR easily exploits without paying the cost of decompression performed by CSX. However, BCSR's performance is rather poor for the first 8 matrices of our suite, that exhibit mainly one-dimensional substructures, and are actually responsible for the poor average performance of BCSR.

As far as the different CSX variants are concerned, we observe that CSX-horiz falls back to CSR-DU performance for the first matrices of our suite, since these contain mainly diagonal structures. For the same reason, CSX-full cannot provide any additional speedup over CSX-linear. The last 7 matrices of the suite are block-dominated and CSX-all gives considerable improvements



(a) Intel Harpertown

(b) Intel Dunnigton

Figure 9. Average performance improvement over multithreaded CSR for different CSX detection schemes.

over the other two variants. The performance of CSX-horiz and CSX-linear is similar for these matrices, since they detect the same patterns (horizontal). Interestingly, these variants can detect the one-dimensional 'sub-patterns' comprising the blocks and provide considerable improvements over pure CSR-DU.

4.3 CSX preprocessing cost

The preprocessing time for CSX when detecting all available patterns (CSX-full variant) is comparable to a few thousands of serial CSR SpMV operations (the wall-clock time ranges from a few second to less than 10 minutes for the largest of our matrices). Since typical applications using SpMV may involve less than a thousand iterations, CSX preprocessing for all substructures is reasonable only for offline preprocessing. Even in this case, however, we argue that for real-life applications, where the sparse matrix is reused many times across numerous runs (e.g. solution of linear system $Ax = b$ with various values of b), this preprocessing cost is still practical.

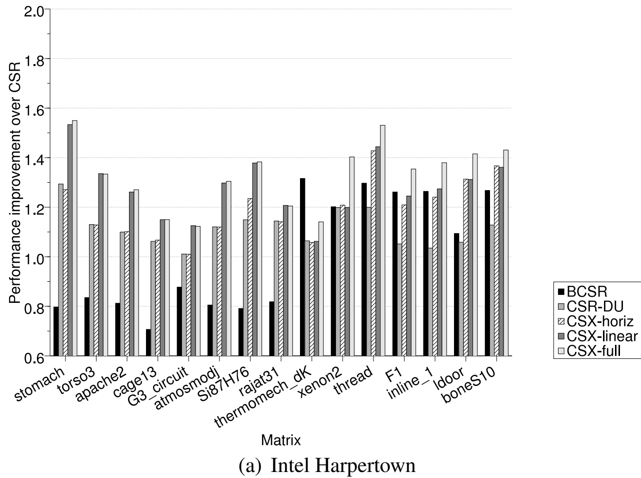
However, for applications that require online CSX preprocessing and need to amortize this cost, we can trade performance for lower preprocessing time. An approach is to restrict the search to fewer substructures, as is the case of the CSX-horiz and CSX-linear variants discussed previously. Both these lightweight variants of CSX outperform significantly existing storage formats, although they do not reach the performance of the 'full' CSX. A second approach is to sample the input matrix, scanning only a representative part of it, in order to decide which substructures to encode in CSX. Figure 11 shows the preprocessing cost (in serial CSR SpMV operations) for CSR-DU and the aforementioned strategies. We applied uniform sampling over the whole matrix for CSX-linear and CSX-full (denoted as CSX-linear-s and CSX-full-s in Fig. 11). Since the substructure detection phase of CSX is already performed using windows, sampling the matrix is rather straightforward: we scan only a certain number of preprocessing windows uniformly distributed all over the input matrix. Specifically, we searched for patterns in 5 windows of 32K non-zero elements, i.e., we sampled 160K of non-zero elements of the matrix, which is less than 6% of the non-zero elements of the smallest matrix in our suite (thermomech_dK). We observe in Fig. 11 that the preprocessing cost is drastically reduced, leading to practical preprocessing times even for online computations. Sampling the matrix is very effective, since it leads to a large reduction in the preprocessing cost with a minor effect on performance. We should also note that the current implementation of the CSX preprocessing can be further reduced with some additional implementation effort, e.g., avoiding the cost of sorting the matrix during the encoding (see Section 3.5) by using processing windows, and providing a multithreaded implementation of the preprocessing. We expect that these optimizations will further reduce the preprocessing cost to less than a hundred serial CSR SpMV operations, making CSX more suitable for online preprocessing of the matrix.

5. Related Work

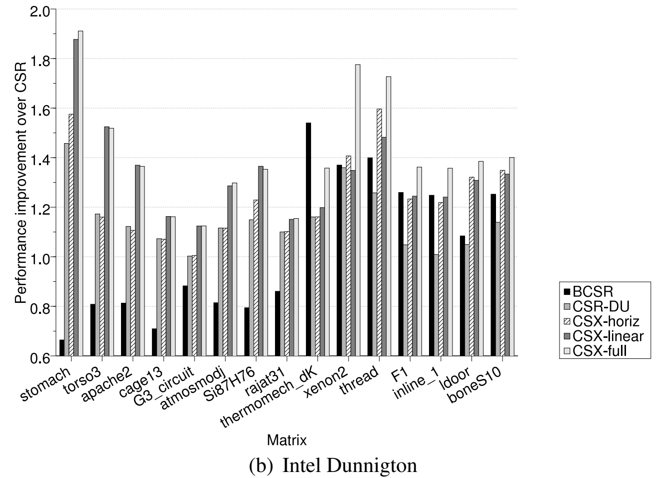
Due to the importance of SpMV, there is an abundance of scientific work targeting the optimization of the serial version of the kernel. Several of these efforts [11,19,22–24] aim at the optimization of the irregular and indirect accesses on the input vector using methods such as matrix reordering, register blocking, and cache blocking. Other works [17,25] are concerned with the performance problems that arise in matrices with a large number of rows with small length.

A significant part of the SpMV optimization techniques reported in the related literature result in index data reduction. Typical examples are blocking methods such as BSCR [20] that store only per-block index information. A problem with these approaches is that, depending on the structure of the matrix, they may require appropriate padding with zero values. One of the first works that explicitly targets the compression of the index data is [26]. In this paper, Willcock and Lumsdaine propose two methods: *DCSR*, which compresses column indices using a byte-oriented delta encoding scheme to exploit the highly redundant nature of the `col_ind` array and *RPCSR*, which generates matrix-specific dynamic code by applying aggressive compression on column indices patterns for the whole matrix. In [15] we propose the CSR-DU format, which employs a delta encoding scheme to group areas of non-zero elements. CSR-DU is restricted to non-zero elements along the same row. Another recent work that targets performance improvement by reducing the index data volume is [5], where Belgin et al. propose a matrix representation that exploits repeated block patterns. The approach here divides the matrix into sub-blocks and searches for frequently encountered block patterns within these sub-blocks. The proposed storage format exhibits good serial performance, but is not so successful with multiple cores.

As far as the multithreaded version of the code is concerned, past work focuses mainly on SMP clusters, where researchers either apply and evaluate known uniprocessor optimization techniques

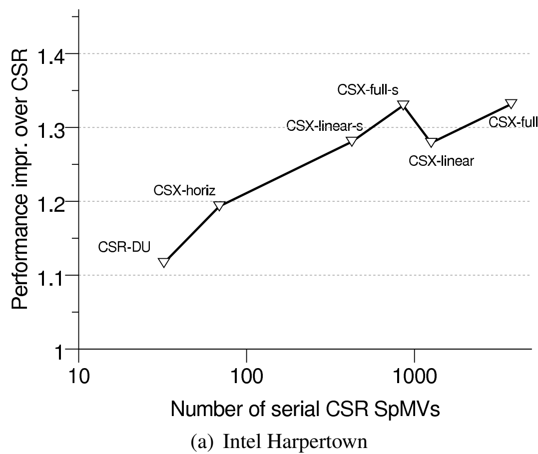


(a) Intel Harpertown

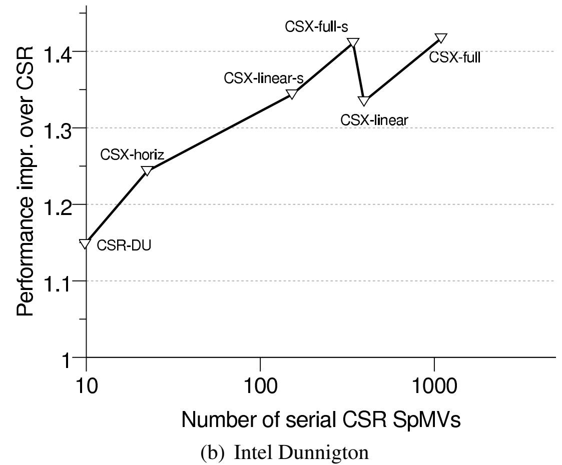


(b) Intel Dunnigton

Figure 10. Per matrix performance improvement over multithreaded CSR for Harpertown (8 threads) and Dunnington (12 threads).



(a) Intel Harpertown



(b) Intel Dunnigton

Figure 11. Tradeoff between preprocessing time and performance improvement over multithreaded CSR. The reported performance improvement is for 8 threads on Harpertown and 12 threads for Dunnington.

(e.g., register and cache blocking) on SMPs or examine reordering techniques to improve locality of references and minimize communication cost [6, 8, 10, 18]. Williams et. al [27] presented an evaluation of SpMV on a set of emerging multicore architectures. Their study covers a wide and diverse range of high-end chip multiprocessors, including recent multicores from AMD (Opteron X2) and Intel (Clovertown), Sun's Niagara2, and platforms comprised of one or two Cell processors. Their work includes a rich collection of optimizations, including some that are targeted specifically at multithreading architectures, on a set of 14 matrices. Although this work focuses on a different class of optimizations, the authors state that they generate different kernel versions for different architectures. Hence this work, along with [5], conforms with our general approach and reveals the necessity for specialized SpMV routines, based on the execution environment (e.g., underlying architecture) or the matrix data for maximizing performance.

6. Conclusions – Future work

In this paper we present a new storage format for sparse matrices called Compressed Sparse eXtended (CSX) that targets the optimization of the SpMV kernel. CSX utilizes a variety of substructures within the sparse matrix, falling in two main categories: one-dimensional substructures across the same row, column or diagonal compressed using a delta run-length encoding scheme, and two-dimensional blocks. CSX is a very flexible storage format that can be further extended to incorporate other families of substructures if necessary. Experimental evaluation on 15 matrices and two multicore platforms has demonstrated that CSX can lead to significant and steady performance improvements over CSR and other, state-of-the-art storage formats. Regarding preprocessing, CSX incorporates a unified approach for the supported substructures based on transformations and exhibits a complexity linear to the number of non-zero elements of the input matrix. Thorough inspection of the matrix to collect the vast majority of substructures leads to a preprocessing cost that is practical for offline preprocessing. To extend the applicability of CSX to online preprocessing, we propose preprocessing strategies that scan a subset of the considered substructures or perform sampling of the matrix. In this way we are able to trade a small part of performance for a large part of preprocessing cost. Our view is that the above two characteristics of

CSX (superior performance and low preprocessing cost) make this storage format the best approach for SpMV. For future work we intend to further reduce the preprocessing cost of CSX and report results from the incorporation of SpMV with CSX in applications like iterative solvers (CG, GMRES, etc.).

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance algorithm using pre-processing for the sparse matrix-vector multiplication. In *Supercomputing'92*, pages 32–41, Minn., MN, November 1992. IEEE.

[2] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, page 69, New York, NY, USA, 1999. ACM.

[3] 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. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

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

[5] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient smvm kernels. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 100–109, New York, NY, USA, 2009. ACM.

[6] U. V. Catalyuerek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. *Lecture Notes In Computer Science*, 1117:75–86, 1996.

[7] T. Davis. University of Florida sparse matrix collection. *NA Digest*, 97(23):7, 1997.

[8] R. Geus and S. Röllin. Towards a fast parallel sparse matrix-vector multiplication. In *Parallel Computing: Fundamentals and Applications, International Conference ParCo*, pages 308–315. Imperial College Press, 1999.

[9] 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*, 2008.

[10] E. Im and K. Yelick. Optimizing sparse matrix-vector multiplication on SMPs. In *9th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.

[11] E. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. *Lecture Notes in Computer Science*, 2073:127–136, 2001.

[12] 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 (CSE-09)*, Vancouver, Canada, 2009. IEEE Computer Society.

[13] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[14] K. Kourtis, G. Goumas, and N. Koziris. Improving the performance of multithreaded sparse matrix-vector multiplication using index and value compression. In *37th International Conference on Parallel Processing (ICPP'08)*, pages 511–519, Sept. 2008.

[15] K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2008. ACM.

[16] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *International Journal of High Performance Computing Applications*, 18(2):225, 2004.

[18] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Improving the locality of the sparse matrix-vector product on shared memory multiprocessors. In *PDP*, pages 66–71. IEEE Computer Society, 2004.

[19] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing'99*, Portland, OR, November 1999. ACM SIGARCH and IEEE.

[20] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[21] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, USA, 2003.

[22] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–725, 1997.

[23] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Supercomputing*, Baltimore, MD, November 2002.

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

[25] J. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *HiPC '97: 4th International Conference on High Performance Computing*, 1997.

[26] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS '06: Proceedings of the 20th annual International Conference on Supercomputing*, pages 307–316, New York, NY, USA, 2006. ACM Press.

[27] S. Williams, L. Oilker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, NV, November 2007.