SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms

Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas and Nectarios Koziris

Presenter: Rawn Henry

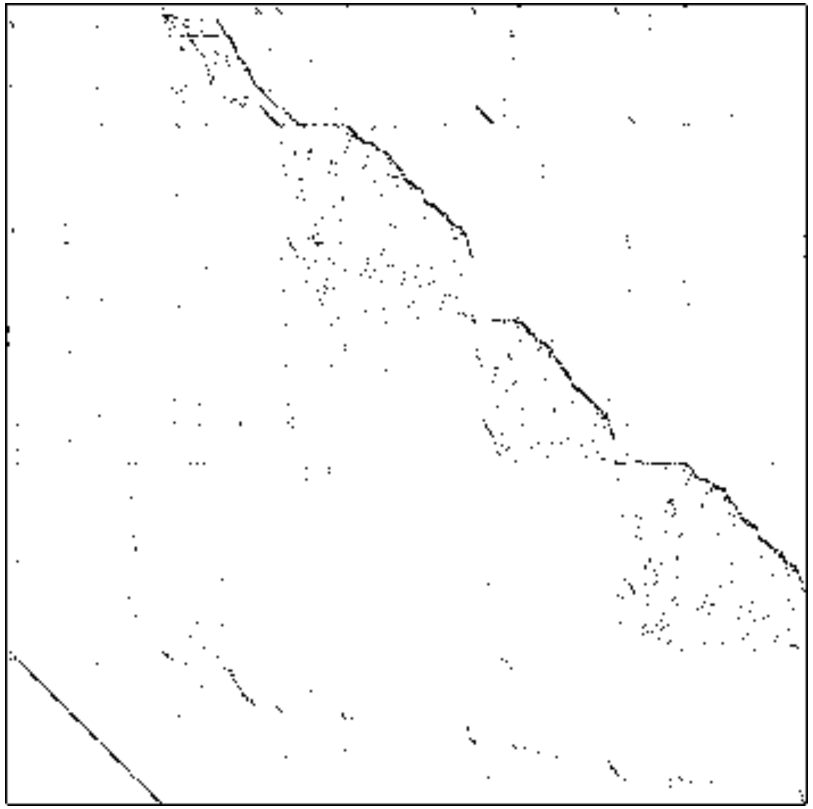
April 25 2019

Overview

- ➢Motivation
- ➢Related Work
- ➢Compression Techniques
- ➢The CSX Format
- ➢Performance of SparseX
- ➢Conclusion

Motivation: Why Compress?

- The dimensions of sparse matrices are usually a lot larger than the number of non-zeros.
  - That is – for an N x M matrix, we usually have that NNZ << N x M
  - A lot of work doing computation can be saved



Motivation: Use Cases

- Building block of iterative methods to solve large, sparse linear systems ($Ax = b$)
- Approximation of Eigen values and Eigen vectors ($Ax = \lambda x$)
- Applications in Economic Modelling, Physics, Medicine, etc…

Motivation: Difficulty with Compression

- SpMV hard to optimize due to:
  - Low Operational Intensity
  - Irregular accesses into input vector
  - Indirect memory accesses due to sparse structure
  - For very short rows loop overhead can be high
  - Large amount of storage formats

Related Work

- Implement SpMV for several formats
  - Drawbacks
    - Complicates library since we need a lot of kernels that do the same thing
    - Requires users to have deep understanding of the problem to pick the right format for their specific domain
- Auto tune kernels based on architecture and application parameters
  - Architecture parameters: cache and registers size, vectorization capabilities
  - Application parameters: symmetry, sparsity pattern
  - Drawbacks
    - Tuning still limited to the formats that the library supports
    - Incurs high overhead making tuning impractical for online use.

SparseX

- Key idea is the use of a custom matrix format, namely the Compressed Sparse eXtended format (CSX)
  - Designed to be auto tuned
  - Can detect a large number of features in a matrix
- Allows SparseX to export a simple BLAS-like interface while maintaining performance of special matrix formats

Compression Techniques: Coordinate Format (COO)

$$A = \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & 0 & 0 & 0 \\ 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & 0 & 0 & 2.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{pmatrix}$$

Redundant

Redundant

Row index - {0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 4, 4, 5, 5}

Col index - {0, 1, 2, 3, 0, 1, 2, 0, 1, 2, 0, 3, 4, 5, 4, 5}

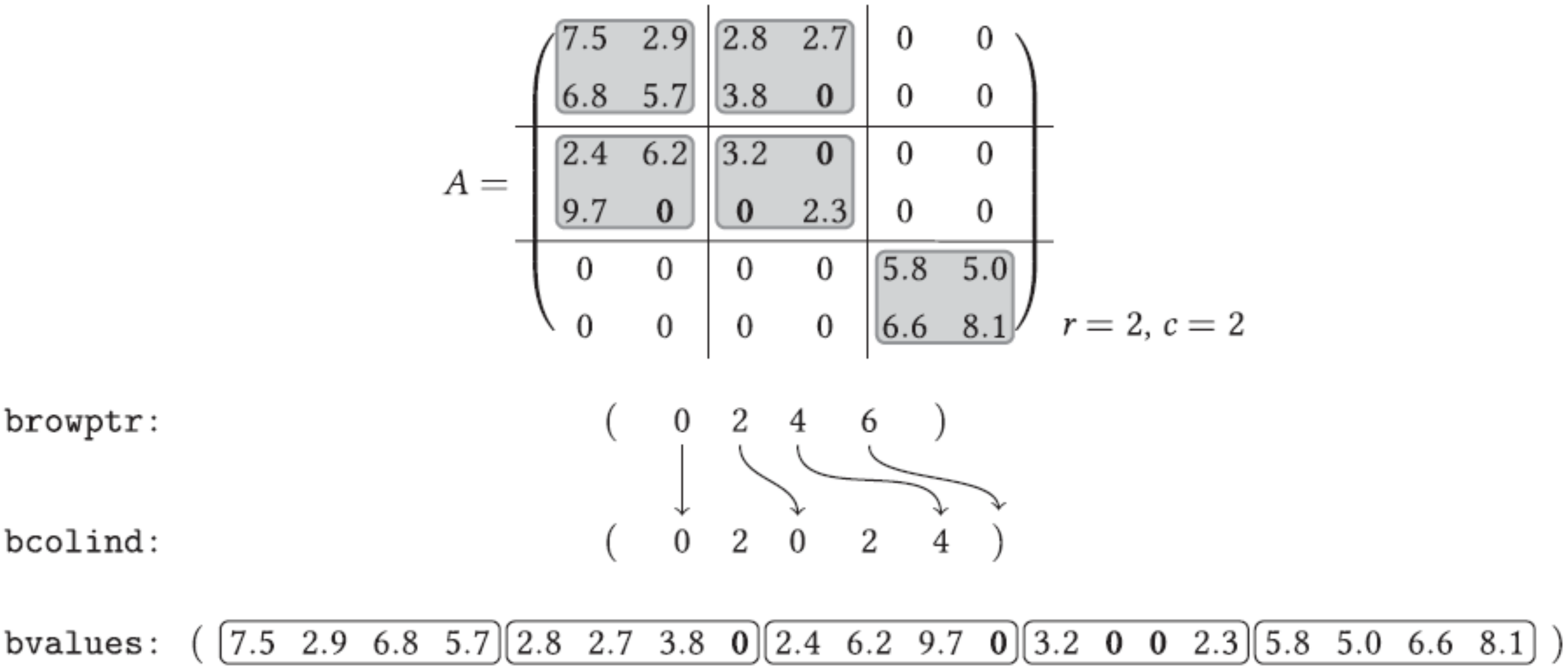
Value - {7.5. 2.9, 2.8, 2.7, 6.8, 5.7, 3.8, 2.4, 6.2, 3.2, 9.7, 2.3, 5.8, 5.0, 6.6, 8.1}

Compression Techniques: Compressed Sparse Row (CSR)

$$A = \begin{pmatrix} 7.5 & 2.9 & 2.8 & 2.7 & 0 & 0 \\ 6.8 & 5.7 & 3.8 & 0 & 0 & 0 \\ 2.4 & 6.2 & 3.2 & 0 & 0 & 0 \\ 9.7 & 0 & 0 & 2.3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5.8 & 5.0 \\ 0 & 0 & 0 & 0 & 6.6 & 8.1 \end{pmatrix}$$

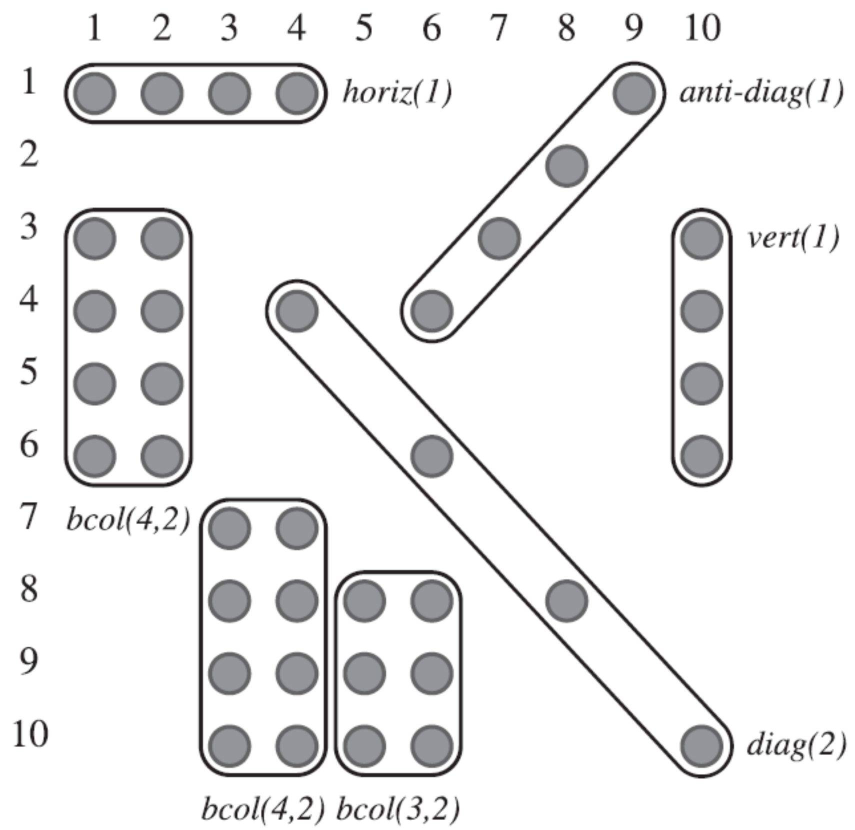
rowptr:	0	4	7	10	12	14	16									
colind:	0	1	2	3	0	1	2	0	1	2	0	3	4	5	4	5
val:	7.5	2.9	2.8	2.7	6.8	5.7	3.8	2.4	6.2	3.2	9.7	2.3	5.8	5.0	6.6	8.1

Compression Techniques: Blocked Compressed Sparse Row (BCSR)



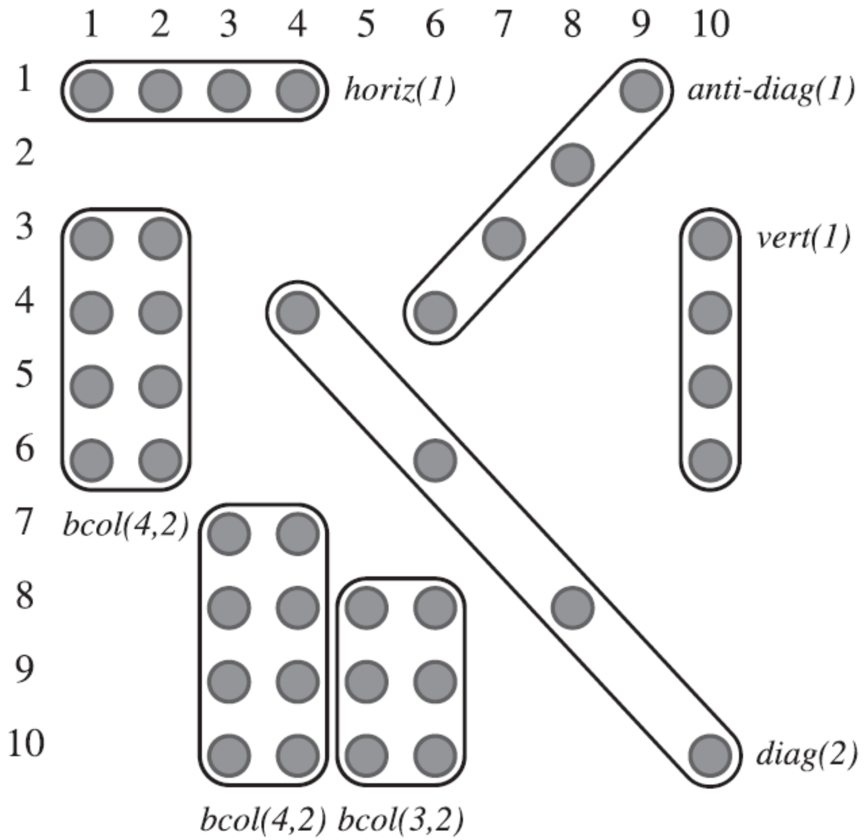
CSX Format: Basics

- ➢Decomposes matrix into units
  - ➢Units can be substructure units encoding blocks, vertical components, diagonals etc
  - ➢Can also be delta units
    - ➢Delta units store the distance from the previous column to the next column. This allows less bytes to be used per index element



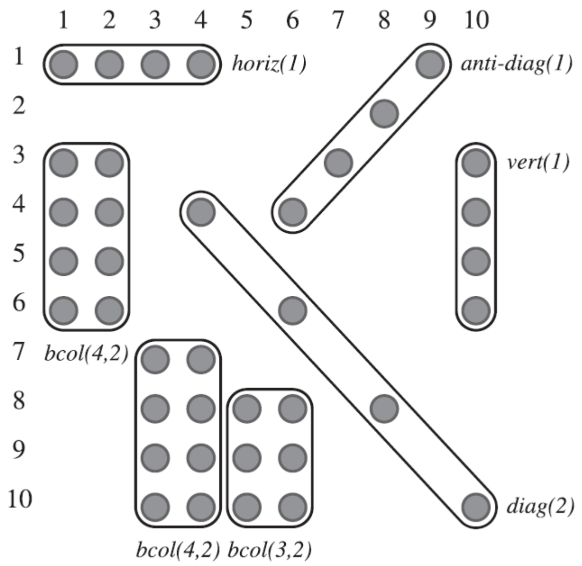
CSX Format: The Layout

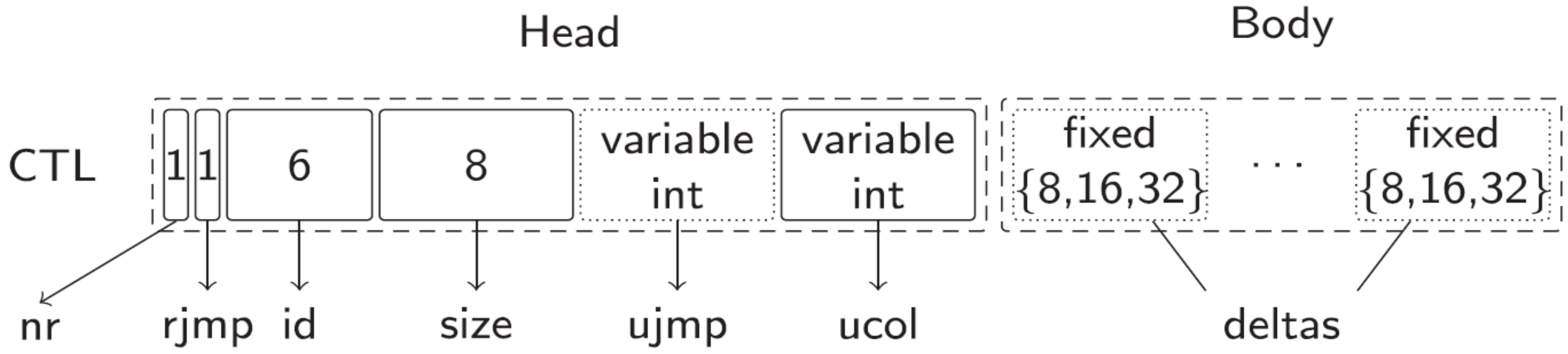
- ➢Stores elements of each substructure in a value array
- ➢Stores substructures in row-wise order
- ➢For matrix on right:
  - ➢Horiz(1), anti-diag(1), bcol(4,2), vert(1), diag(2), bcol(4,2) and bcol(3,2)



CSX Format: The Unit

- A unit comprises of a head and a body
  - nr: Start of new row
  - Rjmp and ujmp: Tells us if we need to skip rows
  - ID: type of substructure
  - Size : number of elements in the body
  - ucol: initial column of the unit
  - Body: Only present in delta unit otherwise substructure values





CSX Format: Detecting Substructures

- To Facilitate detection, CSX uses an internal COO format with (i, j, e) tuples lexicographically sorted on the (i, j) where e is either a substructure or a single element.
  - In the case of a substructure, (i, j) is the coordinate of the first element in that substructure
- CSX also stores row pointers for fast row access

CSX Format: Detecting Substructures

Applies Run-Length encoding

- ➢Computes delta distances of column indices and assembles groups called runs from the same distance values
- ➢Each run is identified by a common delta value and its length

Algorithm 1. Substructure detection in CSX.

```
procedure DETECTSUBSTR(matrix::in, stats::inout)
    matrix: CSX's internal repr., lexicographically sorted
    stats: substructure statistics
    colind ← ∅                                  ▷ Column indices to encode
    for all rows in matrix do
        for allgeneric elements e(i, j, v) in row do
            if e is not a substructure then
                colind ← colind ∪ e.j
                continue
            enc ← RLENCODE(colind)
            UPDATESTATS(stats, enc)             ▷ Update statistics
                                                  for this encoding
            colind ← ∅
        enc ← RLENCODE(colind)
        UPDATESTATS(stats, enc)                 ▷ Update statistics for
                                                  this encoding
        colind ← ∅
```

CSX Format: Detecting Non-horizontal Substructures

- ➢Transform coordinates to desired iteration order sort lexicographically and use algorithm 1
- ➢r, c – block row size and block column size

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	$(i', j') = \begin{cases} (i + j - 1, i), & i + j - 1 \leq N \\ (i + j - 1, N + 1 - j), & i + j - 1 > N \end{cases}$
Block (row aligned)	$(i', j') = (\lfloor \frac{i-1}{r} \rfloor + 1, \mathrm{mod}(i - 1, r) + r(j - 1) + 1)$
Block (column aligned)	$(i', j') = (\lfloor \frac{j-1}{c} \rfloor + 1, c(i - 1) + \mathrm{mod}(j - 1, c))$

CSX Format: Encoding Substructures

- For each substructure type
  - Transform the matrix to the corresponding iteration order
  - Scan the result and collect statistics for the examined substructure type
  - Filter out substructures created that encode less than 5% of total non-zeros
  - Select most appropriate type based on some criterion
  - Repeat until no more substructure types can be selected

CSX Format: Encoding Substructures

Algorithm 2. Detection, selection and encoding of the substructures in CSX.

```
procedure ENCODEMATRIX(matrix::inout)
matrix: CSX's internal matrix in row-wise order
    repeat
        stats ← ∅
        for all available substructure types t do
            TRANSFORM(matrix, t)
            LEXSORT(matrix)
            DETECTSUBSTR(matrix, stats)
            TRANSFORM^-1 (matrix, t)
        FILTERSTATS(stats)  ▷ Filter out instantiations that
                              encode less than 5% of the non-zero
                              elements
        s ← SELECTTYPE(stats)
        if s ≠ NONE then
            TRANSFORM(matrix, s)
            LEXSORT(matrix)
            ENCODESUBSTR(matrix)  ▷ Encode the selected
                                    substructure
    until s = NONE
```

CSX Format: Criterion for Substructure Selection

- Select substructures based on a rough estimate of the reduction over original CSR.
- S_{colind} := Size of colind structure from normal CSR
- S_{ctl} := Size of ctl structure from CSX (depends on number of units)

$$
\begin{aligned}
G &= S_{colind} - S_{ctl} \\
&= NNZ - (\overbrace{N_{units}}^{\text{encoded}} + \overbrace{NNZ - NNZ_{enc}}^{\text{unencoded}}) \\
&= NNZ_{enc} - N_{units},
\end{aligned}
$$

CSX Format: Takeaways

- CSX can automatically detect a variety of substructures in a matrix removing the need for users to carefully choose format types
- CSX format naturally lends itself to autotuning

Performance: Preprocessing of CSX

- Initially 500 serial SpMV operations if entire matrix is processed
- Can get down to ~100 serial SpMV operations through sampling and other techniques

Performance: Operational Intensity

- Intensity for general SpMV: $y = \alpha A x + \beta y$
- Flops: $2N_{Nz} + 3N_r$
- Memory: Index information is 4 bytes while values are 8 bytes
- M_x is the memory of vector x: $8N_r$ bytes since read only
- M_y is the memory of vector y: $16N_r$ bytes since read and write
- M_{CSR} = $12N_{Nz} + 4N_r$
- M_{CSX} = $S_{ctl} + 8N_{NZ}$
- $I_F = \frac{flops}{M_F + M_x + M_y}$ where F is the format type and I is the intensity

Performance: Operational Intensity

Let $M_{x,y}$ = $M_x + M_y$ so we have the following operational intensities:

—*CSR*

$$I_{CSR} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{M_{CSR} + M_{x,y}} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{4 \cdot N_r + 12 \cdot N_{nz} + 4 + 24 \cdot N_r}$$

$$= \frac{1 + 1.5 \cdot \frac{N_r}{N_{nz}}}{6 + 14 \cdot \frac{N_r}{N_{nz}} + \frac{2}{N_{nz}}} \text{ (flops/bytes).}$$

—*CSX*

$$I_{CSX} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{M_{CSX} + M_{x,y}} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{S_{ctl} + 8 \cdot N_{nz} + 24 \cdot N_r}$$

$$= \frac{1 + 1.5 \cdot \frac{N_r}{N_{nz}}}{4 + 0.5 \cdot \frac{S_{ctl}}{N_{nz}} + 12 \cdot \frac{N_r}{N_{nz}}} \text{ (flops/bytes),}$$

where S_{ctl} is $O(N_r)$.

For common case NNZ >> N_r:

- I_{CSR} = 0.167
- I_{CSX} = 0.25

CSX has higher intensity which reduces pressure on memory subsystem.

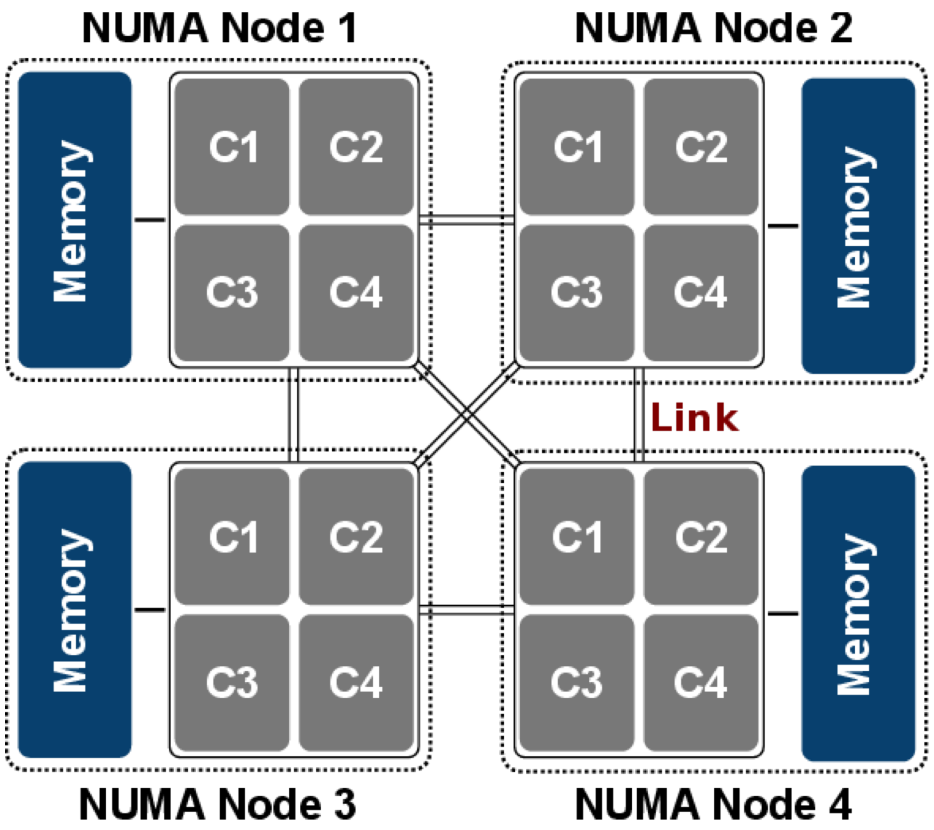
Performance: Compression vs CSR

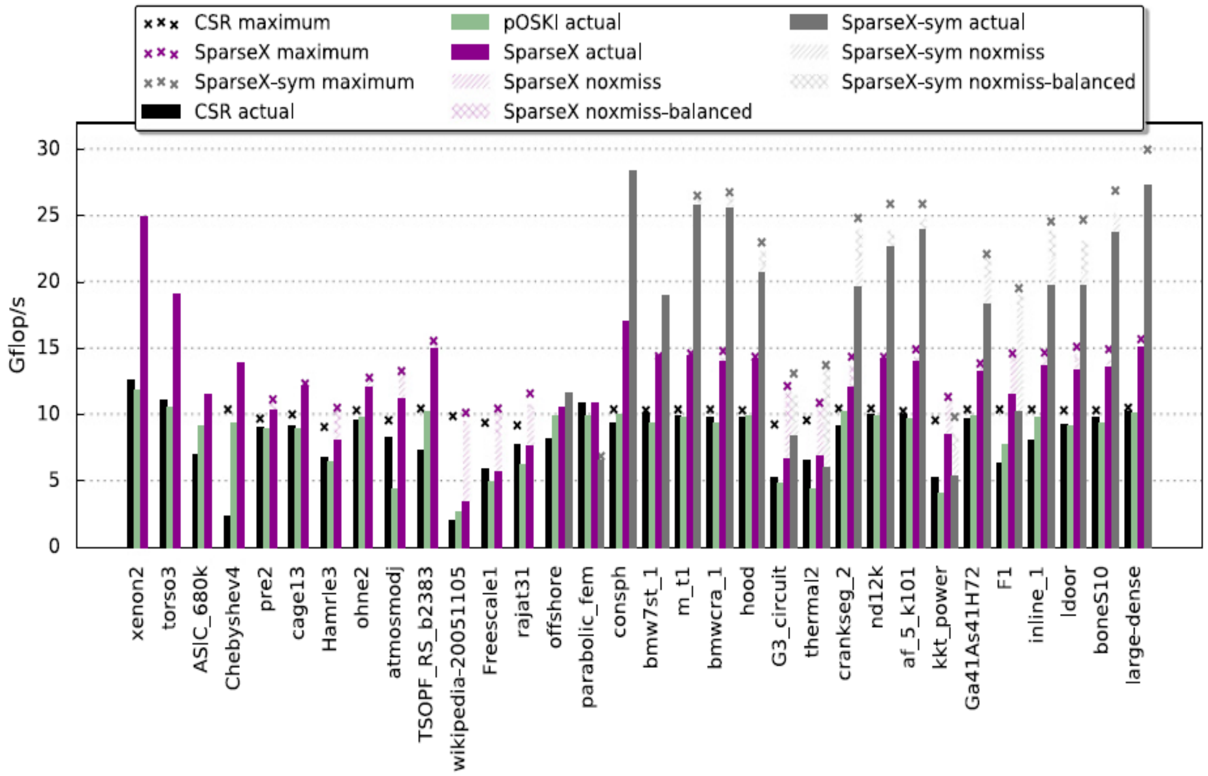
Matrix	N	NNZ	S_{CSR} (MiB)	S_{CSX} (MiB)	$S_{CSX_{sym}}$ (MiB)	I_{CSR}	I_{CSX}	$I_{CSX_{sym}}$	Problem
xenon2	157,464	3,866,688	44.85	32.20	-	0.162	0.219	-	Materials
ASIC_680k	682,862	3,871,773	46.91	37.09	-	0.149	0.177	-	Circuit Sim.
torso3	259,156	4,429,042	51.67	35.35	-	0.160	0.222	-	Other
Chebyshev4	68,121	5,377,761	61.80	46.09	-	0.165	0.219	-	Structural
Hamrle3	1,447,360	5,514,242	68.63	54.51	-	0.144	0.167	-	Circuit Sim.
pre2	659,033	5,959,282	70.71	60.15	-	0.154	0.176	-	Circuit Sim.
cage13	445,315	7,479,343	87.29	69.19	-	0.159	0.196	-	Graph
atmosmodj	1,270,432	8,814,880	105.72	67.61	-	0.152	0.211	-	C.F.D.
ohne2	181,343	11,063,545	127.30	103.08	-	0.164	0.202	-	Semiconductor
TSOPF_RS_b2383	38,120	16,171,169	185.21	124.42	-	0.166	0.247	-	Power
Freescale1	3,428,755	18,920,347	229.61	199.21	-	0.149	0.165	-	Circuit Sim.
wikipedia-20051105	1,634,989	19,753,078	232.29	224.63	-	0.157	0.162	-	Directed Graph
rajat31	4,690,002	20,316,253	250.39	176.66	-	0.146	0.184	-	Circuit Sim.

Performance: Benchmark Terminology

- *noxmiss* - eliminates irregular accesses by setting the column indices of all nonzero elements to 0. Indicative of the performance loss due to excessive cache misses when accessing right-hand side vector.
- *noxmiss-balanced* - performance of the noxmiss benchmark using the average execution time of all threads. Designates the performance loss due to both excessive cache misses and workload imbalance

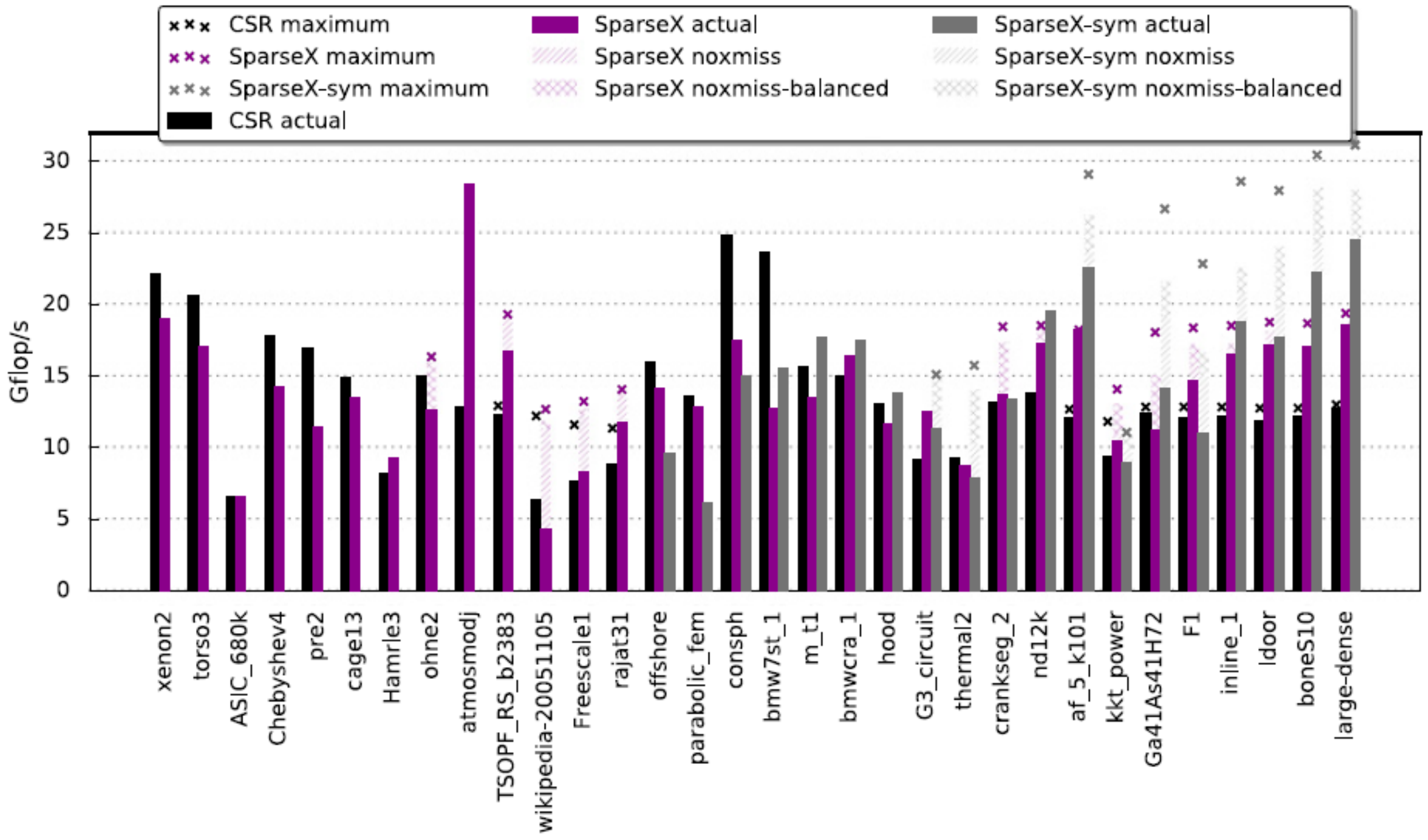
Performance: Single NUMA Node



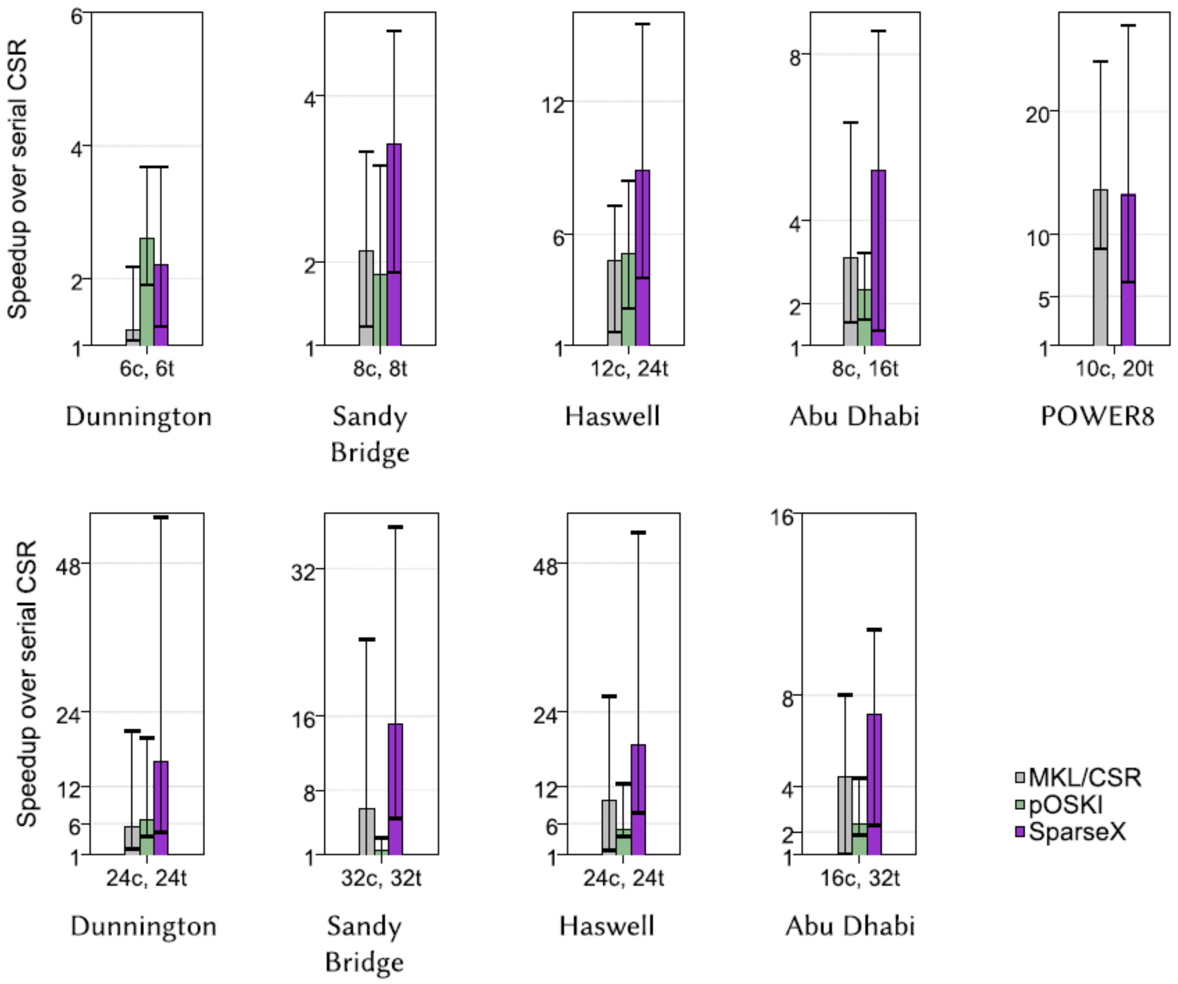


Performance: Single NUMA Node

Matrices fit in last level cache for power 8 machine (80MB)



Performance: Overall



Conclusion

- SparseX provides an easy to use library that automatically autotunes to matrix structure due to the CSX format
- Achieves speed ups of 1.2 to 2x on a variety of matrices