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 $N \times M$ matrix, we usually have that $\text{NNZ} \ll N \times M$
  - 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)

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

Row index: \{0, 0, 0, 0, 1, 1, 1, 2, 2, 2, 3, 3\}
Col index: \{0, 1, 2, 3, 0, 1, 2, 0, 1, 2, 0, 3\}
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)

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

\( r = 2, c = 2 \)

\[ \text{browptr:} \quad \begin{pmatrix} 0 & 2 & 4 & 6 \end{pmatrix} \]

\[ \text{bcolind:} \quad \begin{pmatrix} 0 & 2 & 0 & 2 & 4 \end{pmatrix} \]

\[ \text{bvalues:} \begin{pmatrix} 7.5 & 2.9 & 6.8 & 5.7 & 2.8 & 2.7 & 3.8 & 0 & 2.4 & 6.2 & 9.7 & 0 & 3.2 & 0 & 0 & 2.3 & 5.8 & 5.0 & 6.6 & 8.1 \end{pmatrix} \]
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

Head

Body

<table>
<thead>
<tr>
<th>CTL</th>
<th>nr</th>
<th>rjmp</th>
<th>id</th>
<th>size</th>
<th>ujmp</th>
<th>ucol</th>
<th>fixed</th>
<th>variable</th>
<th>fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>11</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td>{8,16,32}</td>
<td>int</td>
<td>{8,16,32}</td>
</tr>
</tbody>
</table>
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.

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:     end for
9:     enc ← RLENCODE(colind)
10:    UPDATESTATS(stats, enc)  // Update statistics for this encoding
11:   end for
12:   colind ← ∅
13:   enc ← RLENCODE(colind)
14:   UPDATESTATS(stats, enc)  // Update statistics for this encoding
15: end procedure
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 1

The coordinate transformations applied by CSX on the matrix elements for enabling the detection of non-horizontal substructures (one-based indexing assumed).

<table>
<thead>
<tr>
<th>Substructure</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizontal</td>
<td>((i', j') = (i, j))</td>
</tr>
<tr>
<td>Vertical</td>
<td>((i', j') = (j, i))</td>
</tr>
<tr>
<td>Diagonal</td>
<td>((i', j') = (N + j - i, \min(i, j)))</td>
</tr>
</tbody>
</table>
| Anti-diagonal        | \[
|                     | \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)  | \((i', j') = \left(\left\lfloor \frac{i - 1}{r} \right\rfloor + 1, \text{mod}(i - 1, r) + r(j - 1) + 1\right)\) |
| Block (column aligned)| \((i', j') = \left(\left\lfloor \frac{j - 1}{c} \right\rfloor + 1, \text{c}(i - 1) + \text{mod}(j - 1, c)\right)\) |
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
Algorithm 2. Detection, selection and encoding of the substructures in CSX.

1: procedure ENCODEMATRIX(matrix::inout)
   2:     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
CSX Format: Criterion for Substructure Selection

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

\[
G = S_{\text{colind}} - S_{\text{ctl}}
\]

\[
= \text{NNZ} - \left( \text{NNZ}_{\text{units}} + \text{NNZ} - \text{NNZ}_{\text{enc}} \right)
\]

\[
= \text{NNZ}_{\text{enc}} - \text{NNZ}_{\text{units}},
\]
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: $y = \alpha Ax + \beta y$
- 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
- $M_{CSR} = 12N_{Nz} + 4N_r$
- $M_{CSX} = S_{ctl} + 8N_{NZ}$
- $I_F = \frac{\text{flops}}{M_F + M_x + M_y}$ 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:

For common case $\text{NNZ} \gg N_r$:

- $I_{\text{CSR}} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{M_{\text{CSR}} + M_{x,y}} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{4 \cdot N_r + 12 \cdot N_{nz} + 4 + 24 \cdot N_r}$
  \[= \frac{1 + 1.5 \cdot \frac{N_r}{N_{nz}}}{6 + 14 \cdot \frac{N_r}{N_{nz}} + \frac{2}{N_{nz}}} \text{ (flops/bytes).} \]

- $I_{\text{CSX}} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{M_{\text{CSX}} + M_{x,y}} = \frac{2 \cdot N_{nz} + 3 \cdot N_r}{S_{cxt} + 8 \cdot N_{nz} + 24 \cdot N_r}$
  \[= \frac{1 + 1.5 \cdot \frac{N_r}{N_{nz}}}{4 + 0.5 \cdot \frac{S_{cxt}}{N_{nz}} + 12 \cdot \frac{N_r}{N_{nz}}} \text{ (flops/bytes),} \]

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

CSX has higher intensity which reduces pressure on memory subsystem.
## Performance: Compression vs CSR

<table>
<thead>
<tr>
<th>Matrix</th>
<th>N</th>
<th>NNZ</th>
<th>$S_{CSR}$ (MiB)</th>
<th>$S_{CSX}$ (MiB)</th>
<th>$S_{CSX_{sym}}$ (MiB)</th>
<th>$I_{CSR}$</th>
<th>$I_{CSX}$</th>
<th>$I_{CSX_{sym}}$</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>xenon2</td>
<td>157,464</td>
<td>3,666,688</td>
<td>44.85</td>
<td>32.20</td>
<td>-</td>
<td>0.162</td>
<td>0.219</td>
<td>-</td>
<td>Materials</td>
</tr>
<tr>
<td>ASIC_680k</td>
<td>682,862</td>
<td>3,871,773</td>
<td>46.91</td>
<td>37.09</td>
<td>-</td>
<td>0.149</td>
<td>0.177</td>
<td>-</td>
<td>Circuit Sim.</td>
</tr>
<tr>
<td>torso3</td>
<td>259,156</td>
<td>4,429,042</td>
<td>51.67</td>
<td>35.35</td>
<td>-</td>
<td>0.160</td>
<td>0.222</td>
<td>-</td>
<td>Other</td>
</tr>
<tr>
<td>Chebyshev4</td>
<td>68,121</td>
<td>5,377,761</td>
<td>61.80</td>
<td>46.09</td>
<td>-</td>
<td>0.165</td>
<td>0.219</td>
<td>-</td>
<td>Structural</td>
</tr>
<tr>
<td>Hamrle3</td>
<td>1,447,360</td>
<td>5,514,242</td>
<td>68.63</td>
<td>54.51</td>
<td>-</td>
<td>0.144</td>
<td>0.167</td>
<td>-</td>
<td>Circuit Sim.</td>
</tr>
<tr>
<td>pre2</td>
<td>659,033</td>
<td>5,959,282</td>
<td>70.71</td>
<td>60.15</td>
<td>-</td>
<td>0.154</td>
<td>0.176</td>
<td>-</td>
<td>Circuit Sim.</td>
</tr>
<tr>
<td>cage13</td>
<td>445,315</td>
<td>7,479,343</td>
<td>87.29</td>
<td>69.19</td>
<td>-</td>
<td>0.159</td>
<td>0.196</td>
<td>-</td>
<td>Graph</td>
</tr>
<tr>
<td>atmosmodj</td>
<td>1,270,432</td>
<td>8,814,880</td>
<td>105.72</td>
<td>67.61</td>
<td>-</td>
<td>0.152</td>
<td>0.211</td>
<td>-</td>
<td>C.F.D.</td>
</tr>
<tr>
<td>ohne2</td>
<td>181,343</td>
<td>11,063,545</td>
<td>127.30</td>
<td>103.08</td>
<td>-</td>
<td>0.164</td>
<td>0.202</td>
<td>-</td>
<td>Semiconductor</td>
</tr>
<tr>
<td>TSOPF_RS_b2383</td>
<td>38,120</td>
<td>16,171,169</td>
<td>185.21</td>
<td>124.42</td>
<td>-</td>
<td>0.166</td>
<td>0.247</td>
<td>-</td>
<td>Power</td>
</tr>
<tr>
<td>Freescale1</td>
<td>3,428,755</td>
<td>18,920,347</td>
<td>229.61</td>
<td>199.21</td>
<td>-</td>
<td>0.149</td>
<td>0.165</td>
<td>-</td>
<td>Circuit Sim.</td>
</tr>
<tr>
<td>wikipedia-20051105</td>
<td>1,634,989</td>
<td>19,753,078</td>
<td>232.29</td>
<td>224.63</td>
<td>-</td>
<td>0.157</td>
<td>0.162</td>
<td>-</td>
<td>Directed Graph</td>
</tr>
<tr>
<td>raji31</td>
<td>4,690,002</td>
<td>20,316,253</td>
<td>250.39</td>
<td>176.66</td>
<td>-</td>
<td>0.146</td>
<td>0.184</td>
<td>-</td>
<td>Circuit Sim.</td>
</tr>
</tbody>
</table>
**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

NUMA Node 1
- C1
- C2
- C3
- C4

NUMA Node 2
- C1
- C2
- C3
- C4

NUMA Node 3
- C1
- C2
- C3
- C4

NUMA Node 4
- C1
- C2
- C3
- C4

Graph showing performance metrics for different benchmarks and workloads.
Performance: Single NUMA Node

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

![Graphs showing performance measurements for different hardware configurations.]
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.