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

PPoPP 2011

Kornilios Kourtis, Vasileios Karakasis,
Georgios Goumas, Nectarios Koziris

Computing Systems Laboratory
National Technical University of Athens
Greece

- **Compressed Sparse eXtended (CSX)**:
  *w*hat: storage format for sparse matrices
  *w*hy: optimize sparse matrix-vector multiplication (SpMxV) by (aggressively) compressing structural data

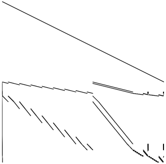
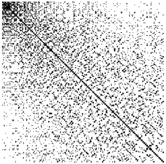
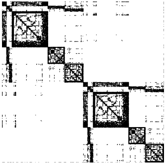
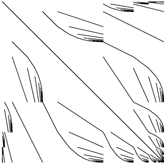
- **background**
  - sparse matrices
  - the SpMxV kernel

- **Compressed Sparse eXtended (CSX)**:
  *w*hat: storage format for sparse matrices
  *w*hy: optimize sparse matrix-vector multiplication (SpMxV)
  by (aggressively) compressing structural data

Sparse matrices and sparse matrix vector multiplication

(application domain)

- Dominated by zeroes
- Applications: PDEs, graphs, linear programming
- Efficient representation: sparse storage formats (space and computation)
  - non-zero values (*value data*)
  - structural information (*index data*)

Sparse matrices and sparse matrix vector multiplication

(application domain)

- Dominated by zeroes
- Applications: PDEs, graphs, linear programming
- Efficient representation: sparse storage formats (space and computation)
  - non-zero values (*value data*)
  - structural information (*index data*)
- sparse matrix vector multiplication (SpMxV)
  - $y = A \cdot x, \quad A$ sparse
  - CG, GMRES, PageRank
  - considerable research attention*

*google scholar:

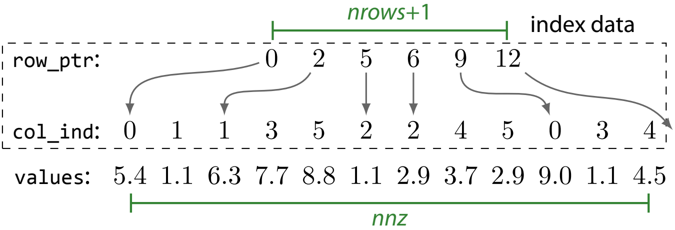
- "sparse matrix vector multiplication" $\rightarrow$ 2280 results
- "multicore" $\rightarrow$ 25100 results

CSR storage format

(Compressed Sparse Row)

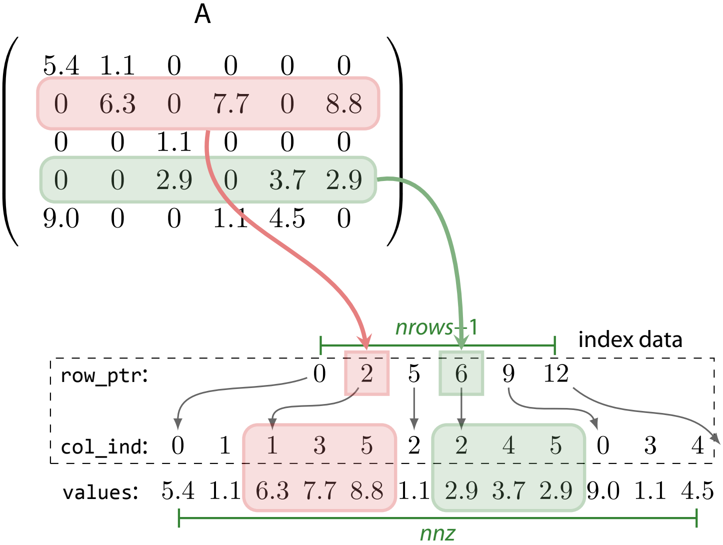
A

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



CSR storage format

(Compressed Sparse Row)



CSR storage format

(Compressed Sparse Row)

$$
\overset{A}{\begin{pmatrix}
5.4 & 1.1 & 0 & 0 & 0 & 0 \\
0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\
0 & 0 & 1.1 & 0 & 0 & 0 \\
0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\
9.0 & 0 & 0 & 1.1 & 4.5 & 0
\end{pmatrix}}
*
\begin{pmatrix}
x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5
\end{pmatrix}
=
\begin{pmatrix}
y_0 = \sum A_{1i} \cdot x_i \\
y_1 = \sum A_{2i} \cdot x_i \\
y_2 = \sum A_{3i} \cdot x_i \\
y_3 = \sum A_{4i} \cdot x_i \\
y_4 = \sum A_{5i} \cdot x_i \\
y_5 = \sum A_{6i} \cdot x_i
\end{pmatrix}
$$

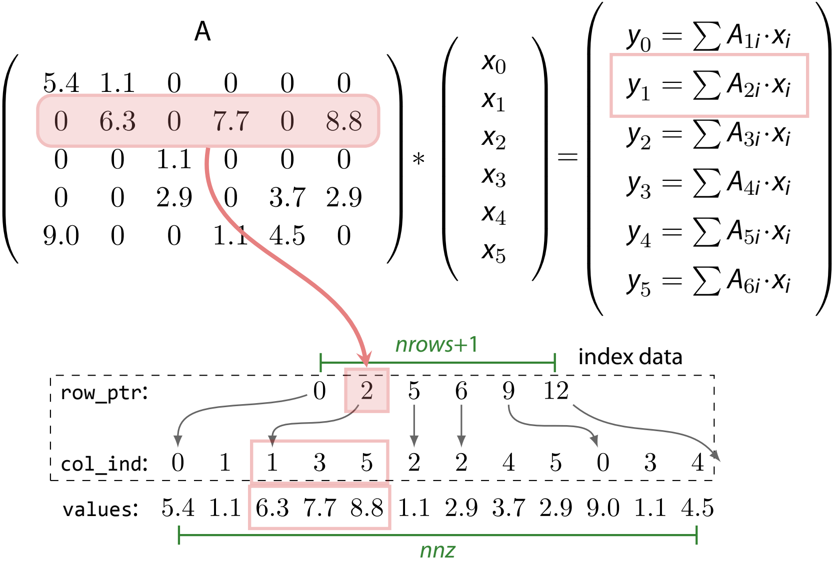
nrows+1 — index data

`row_ptr:`	0	2	5	6	9	12						
`col_ind:`	0	1	1	3	5	2	2	4	5	0	3	4
`values:`	5.4	1.1	6.3	7.7	8.8	1.1	2.9	3.7	2.9	9.0	1.1	4.5

nnz

CSR storage format

(Compressed Sparse Row)



CSR storage format

(Compressed Sparse Row)

$$y_1 = x_1 \cdot 6.3$$

$$\overset{A}{\begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \end{pmatrix}} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} y_0 = \sum A_{1i} \cdot x_i \\ y_1 = \sum A_{2i} \cdot x_i \\ y_2 = \sum A_{3i} \cdot x_i \\ y_3 = \sum A_{4i} \cdot x_i \\ y_4 = \sum A_{5i} \cdot x_i \\ y_5 = \sum A_{6i} \cdot x_i \end{pmatrix}$$

nrows+1

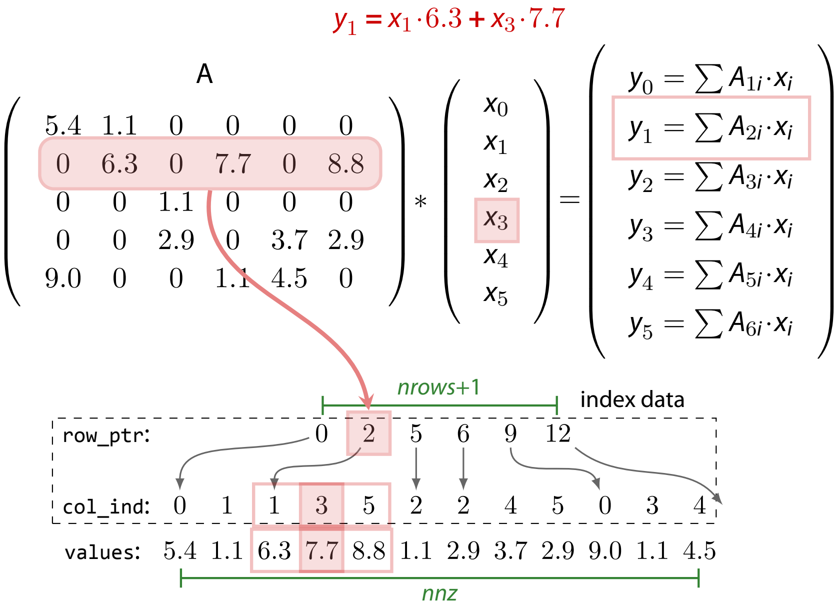
index data

row_ptr:				0	2	5	6	9	12			
col_ind:	0	1	1	3	5	2	2	4	5	0	3	4
values:	5.4	1.1	6.3	7.7	8.8	1.1	2.9	3.7	2.9	9.0	1.1	4.5

nnz

CSR storage format

(Compressed Sparse Row)



CSR storage format

(Compressed Sparse Row)

$$y_1 = x_1 \cdot 6.3 + x_3 \cdot 7.7 + x_5 \cdot 8.8$$

A

$$\begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 0 & 6.3 & 0 & 7.7 & 0 & 8.8 \\ 0 & 0 & 1.1 & 0 & 0 & 0 \\ 0 & 0 & 2.9 & 0 & 3.7 & 2.9 \\ 9.0 & 0 & 0 & 1.1 & 4.5 & 0 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} y_0 = \sum A_{1i} \cdot x_i \\ y_1 = \sum A_{2i} \cdot x_i \\ y_2 = \sum A_{3i} \cdot x_i \\ y_3 = \sum A_{4i} \cdot x_i \\ y_4 = \sum A_{5i} \cdot x_i \\ y_5 = \sum A_{6i} \cdot x_i \end{pmatrix}$$

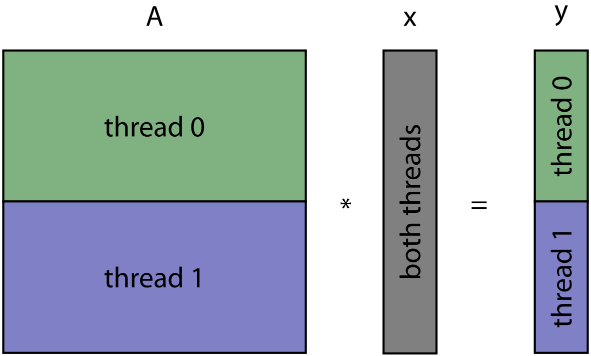
nrows+1 index data

row_ptr:				0	2	5	6	9	12			
col_ind:	0	1	1	3	5	2	2	4	5	0	3	4
values:	5.4	1.1	6.3	7.7	8.8	1.1	2.9	3.7	2.9	9.0	1.1	4.5

nnz

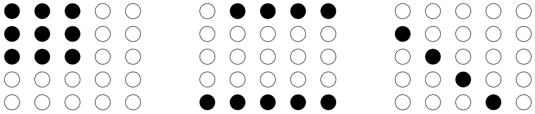
parallel SpMxV for shared memory

- data partitioning
  - per rows
- load balancing
  - based on number of non-zeros



Traditional SpMxV optimization methods

- traditional goal: optimizing computation
- specialized sparse storage formats
  (exploitation of "regularities")
- examples (regularity $\leftrightarrow$ format):
  - 2D blocks of constant size $\leftrightarrow$ BCSR [Im and Yelick '01]
  - 1D blocks of variable size $\leftrightarrow$ [Pinar and Heath '99]
  - Diagonals $\leftrightarrow$ DIAG



Traditional SpMxV optimization: BCSR

[Im and Yelick '01]

- CSR extension: $r \times c$ blocks instead of elements $\Rightarrow$ per-block index information
- optimize computation (register blocking) $\Rightarrow$ specialized SpMxV versions for $r \times c$

$$A = \left(\begin{array}{cc|cc|cc|cc} 4.6 & 9.3 & 0 & 0 & 0 & 0 & 2.4 & 5.6 \\ 8.6 & 8.2 & 0 & 0 & 0 & 0 & 5.3 & 1.6 \\ \hline 0 & 0 & 0 & 0 & 1.9 & 7.9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7.1 & 0 & 0 & 0 \\ \hline 0 & 0 & 8.6 & 1.7 & 2.4 & 7.6 & 0 & 0 \\ 0 & 0 & 3.9 & 2.2 & 3.0 & 3.3 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1.8 & 0 & 7.9 & 1.2 \\ 0 & 0 & 0 & 0 & 0 & 7.8 & 1.0 & 5.3 \end{array}\right)$$

`brow_ptr` : 0 2 3 5 7

`bcol_ind` : (0 6 4 2 4 4 6)

`blocks` : $\begin{matrix} 4.6 & 9.3 \\ 8.6 & 8.2 \end{matrix}$ $\begin{matrix} 2.4 & 5.6 \\ 5.3 & 1.6 \end{matrix}$ $\begin{matrix} 1.9 & 7.9 \\ 7.1 & 0 \end{matrix}$ $\begin{matrix} 8.6 & 1.7 \\ 3.9 & 2.2 \end{matrix}$ $\begin{matrix} 2.4 & 7.6 \\ 3.0 & 3.3 \end{matrix}$ $\begin{matrix} 1.8 & 0 \\ 0 & 7.8 \end{matrix}$ $\begin{matrix} 7.9 & 1.2 \\ 1.0 & 5.3 \end{matrix}$

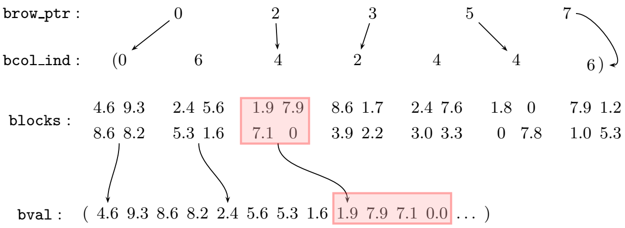
`bval` : (4.6 9.3 8.6 8.2 2.4 5.6 5.3 1.6 1.9 7.9 7.1 0.0 …)

Traditional SpMxV optimization: BCSR

[Im and Yelick '01]

- CSR extension: $r \times c$ blocks instead of elements $\Rightarrow$ per-block index information
- optimize computation (register blocking) $\Rightarrow$ specialized SpMxV versions for $r \times c$
- padding may be required

$$A = \left(\begin{array}{cc|cc|cc|cc}
4.6 & 9.3 & 0 & 0 & 0 & 0 & 2.4 & 5.6 \\
8.6 & 8.2 & 0 & 0 & 0 & 0 & 5.3 & 1.6 \\ \hline
0 & 0 & 0 & 0 & 1.9 & 7.9 & 0 & 0 \\
0 & 0 & 0 & 0 & 7.1 & 0 & 0 & 0 \\ \hline
0 & 0 & 8.6 & 1.7 & 2.4 & 7.6 & 0 & 0 \\
0 & 0 & 3.9 & 2.2 & 3.0 & 3.3 & 0 & 0 \\ \hline
0 & 0 & 0 & 0 & 1.8 & 0 & 7.9 & 1.2 \\
0 & 0 & 0 & 0 & 0 & 7.8 & 1.0 & 5.3
\end{array}\right)$$



SpMxV performance

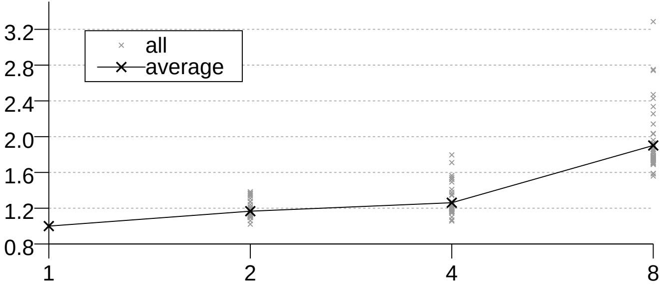
(CSR)

- related work $\rightarrow$ several performance issues
- performance evaluation in 100 matrices [Goumas et. al. '09]
- memory bandwidth is the bottleneck [1]

[1]for matrices larger than cache

SpMxV performance
(CSR)

- related work $\rightarrow$ several performance issues
- performance evaluation in 100 matrices [Goumas et. al. '09]
- memory bandwidth is the bottleneck [1]

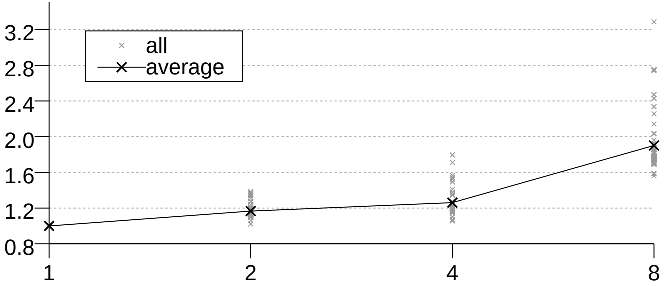


[1] for matrices larger than cache

SpMxV performance

(CSR)

- related work $\rightarrow$ several performance issues
- performance evaluation in 100 matrices [Goumas et. al. '09]
- memory bandwidth is the bottleneck [1]



- **compression** for improving SpMxV performance

  (reduce working set)

---

[1]for matrices larger than cache

CSX: approach

regularities and sparse storage formats

- BCSR, [Pinar and Heath '99], DIAG
- multiple regularities $\leftrightarrow$ *composite formats* [Agarwal et. al '92]
  multiple sub-matrices — each in different format
  $A \cdot x = (A_0 + A_1) \cdot x = A_0 \cdot x + A_1 \cdot x$

CSX: approach

regularities and sparse storage formats

- BCSR, [Pinar and Heath '99], DIAG
- multiple regularities $\leftrightarrow$ *composite formats* [Agarwal et. al '92]
  multiple sub-matrices — each in different format
  $A \cdot x = (A_0 + A_1) \cdot x = A_0 \cdot x + A_1 \cdot x$

(our) requirements

- support multiple regularities on the same matrix
- extendability – arbitrary regularities
- adaptability

CSX: approach

regularities and sparse storage formats

- BCSR, [Pinar and Heath '99], DIAG
- multiple regularities $\leftrightarrow$ *composite formats* [Agarwal et. al '92]
  multiple sub-matrices — each in different format
  $A \cdot x = (A_0 + A_1) \cdot x = A_0 \cdot x + A_1 \cdot x$

(our) requirements

- support multiple regularities on the same matrix
- extendability – arbitrary regularities
- adaptability

approach — CSX (Compressed Sparse eXtended) format

- units: matrix areas that adhere to a regularity
- unified detection of regularities
- code generation of specialized SpMxV routines

CSX outline

- CSX substructures (regularities)
- CSX detection of substructures
  - and how to make it faster
- Experimental evaluation

CSX substructures

(regularities supported by CSX)

- **Horizontal**

x x x x x

(e.g: col. indices: 1,2,3,4,5)

sequential elements

$$(y, x+i) \rightarrow \quad (y, x) \quad (y, x+1) \quad (y, x+2) \quad \ldots$$

CSX substructures

(regularities supported by CSX)

- **Horizontal (delta run-length-encoding — drle)**

x x x x x

(e.g: col. indices: 2,4,6,8,10)

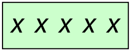
~~sequential~~ elements with a constant difference δ

$(y, x + i \cdot \delta) \rightarrow \quad (y, x) \quad (y, x + \delta) \quad (y, x + 2 \cdot \delta) \ldots$

CSX substructures

(regularities supported by CSX)

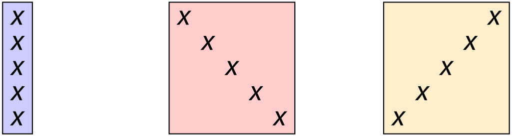
- **Horizontal (delta run-length-encoding — drle)**



~~sequential~~ elements with a constant difference δ

$$(y, x + i \cdot \delta) \rightarrow \quad (y, x) \quad (y, x + \delta) \quad (y, x + 2 \cdot \delta) \ldots$$

- **Other 1D directions (Vertical, Diagonal, Anti-Diagonal)**

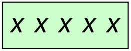


$$(y + i \cdot \delta, x) \qquad (y + i \cdot \delta, x + i \cdot \delta) \quad (y - i \cdot \delta, x + i \cdot \delta)$$

CSX substructures

(regularities supported by CSX)

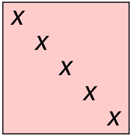
- **Horizontal (delta run-length-encoding — drle)**

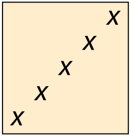


~~sequential~~ elements with a constant difference δ

$$(y, x + i \cdot \delta) \rightarrow \quad (y, x) \quad (y, x + \delta) \quad (y, x + 2 \cdot \delta) \ldots$$

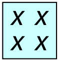
- **Other 1D directions (Vertical, Diagonal, Anti-Diagonal)**





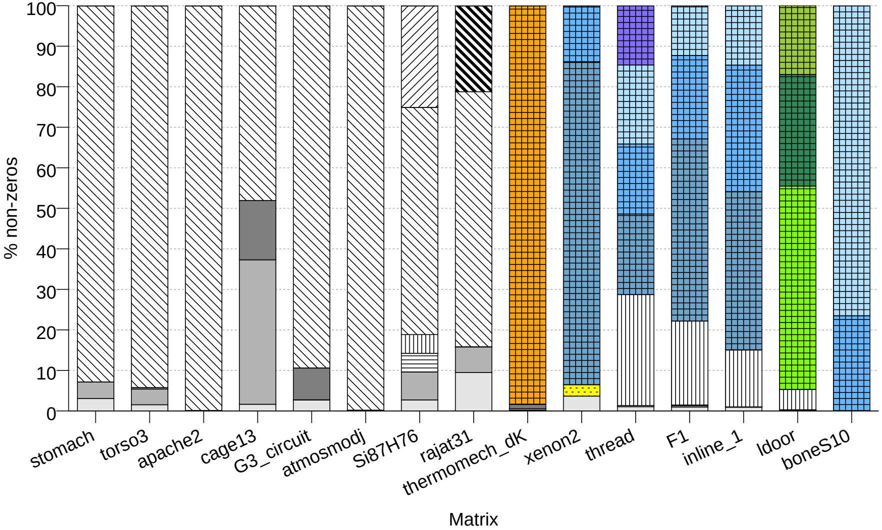
$(y + i \cdot \delta, x)$ $\quad (y + i \cdot \delta, x + i \cdot \delta)$ $\quad (y - i \cdot \delta, x + i \cdot \delta)$

- **2D blocks**

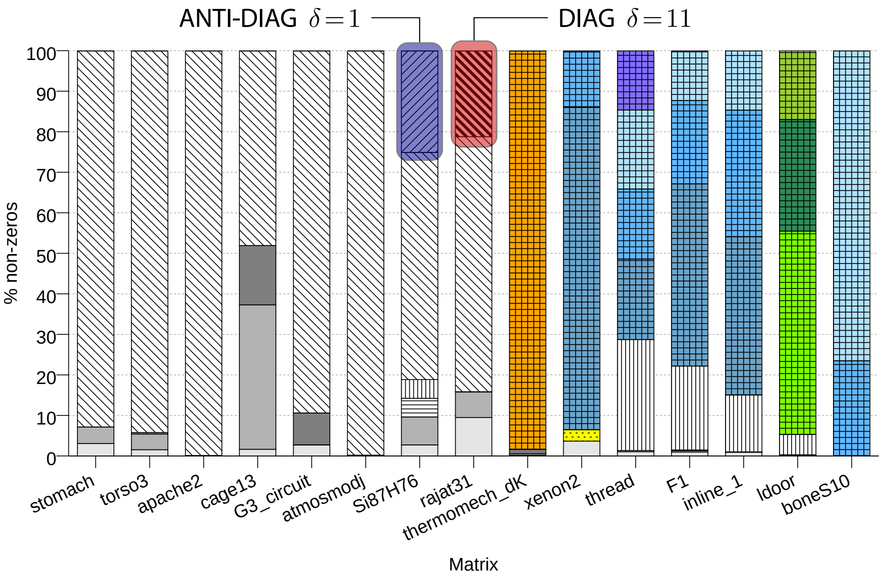


$(x + i) \times (y + j)$ (double nested loop)

CSX substructures on the matrix set



CSX substructures on the matrix set



CSX substructure detection: horizontal

(Delta Run-Length Encoding – DRLE)

$$\begin{pmatrix} & & (1,3) & \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (3,1) & & & \\ & & (4,3) & \end{pmatrix}$$

$$(1,3)\ (2,1)\ (2,2)\ (2,3)\ (2,4)\ (3,1)\ (4,3)$$

CSX substructure detection: horizontal

(Delta Run-Length Encoding – DRLE)

$$\begin{pmatrix} & & (1,3) & \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (3,1) & & & \\ & & (4,3) & \end{pmatrix}$$

detection

column indices:	1	2	3	4
deltas (δ):	1	1	1	1
run-length-encoding:	(δ=1,len=4)			

$$(1,3)\ (2,1)\ (2,2)\ (2,3)\ (2,4)\ (3,1)\ (4,3)$$

- same order with storage $\rightarrow$ detection is simple

CSX substructure detection: horizontal

(Delta Run-Length Encoding – DRLE)

$$\begin{pmatrix} & & (1,3) & \\ (2,1) & (2,2) & (2,3) & (2,4) \\ (3,1) & & & \\ & & (4,3) & \end{pmatrix}$$

detection

column indices:	1	2	3	4
deltas (δ):	1	1	1	1
run-length-encoding:	(δ=1,len=4)			

$$(1,3)\ \ (2,1)\ \ (2,2)\ \ (2,3)\ \ (2,4)\ \ (3,1)\ \ (4,3)$$

unit

start:	$(2,1)$
order:	HORIZ
δ :	1
length:	4

- same order with storage $\rightarrow$ detection is simple

CSX substructure detection: generalization

$$\begin{pmatrix} (1,1) & & (1,3) & \\ & (2,2) & & \\ & & (3,3) & \\ & & & (4,4) \end{pmatrix}$$

$(1,1)\ (1,3)\ (2,2)\ (3,3)\ (4,4)$

CSX substructure detection: generalization

(Transformations)

$$\begin{pmatrix} (1,1) & & (1,3) & \\ & (2,2) & & \\ & & (3,3) & \\ & & & (4,4) \end{pmatrix} \xrightarrow[j' = min(i,j)]{i' = nrows + j - i} \begin{pmatrix} (4,1) & & (2,1) & \\ & (4,2) & & \\ & & (4,3) & \\ & & & (4,4) \end{pmatrix}$$

$(1,1)$ $(1,3)$ $(2,2)$ $(3,3)$ $(4,4)$

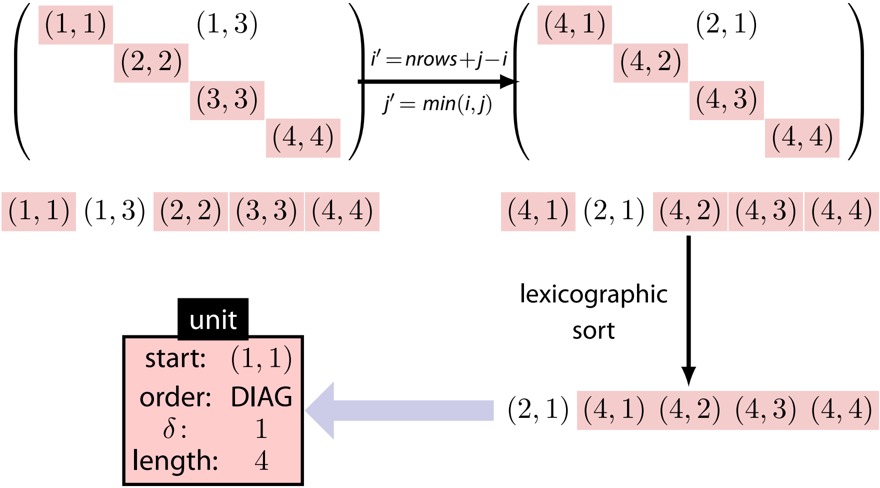
$(4,1)$ $(2,1)$ $(4,2)$ $(4,3)$ $(4,4)$

lexicographic sort

$(2,1)$ $(4,1)$ $(4,2)$ $(4,3)$ $(4,4)$

CSX substructure detection: generalization

(Transformations)



- add a regularity $\rightarrow$ provide transformation

CSX preprocessing phases

❶ Detection: find and select substructures

❷ Encoding:

- index information stored in a byte-array
- each unit: ➨ size (1 byte) ➨ type+markers (1 byte) ➨ payload

❸ Code Generation: matrix-specific SpMxV routines generated programmatically using LLVM
(code iterates substructures and perform the operation)

CSX preprocessing phases

❶ Detection: find and select substructures

❷ Encoding:
- index information stored in a byte-array
- each unit: ➨ size (1 byte) ➨ type+markers (1 byte) ➨ payload

❸ Code Generation: matrix-specific SpMxV routines generated programmatically using LLVM
(code iterates substructures and perform the operation)

➜ what about preprocessing (compression) cost?

- ▸ depends on the application
- ▸ frequently, the matrix is used across numerous SpMxV runs
  - ➻ sufficient repetitions $\rightarrow$ overhead will be amortized
- ▸ methods to reduce preprocessing cost (in the detection phase)
  - ➻ tradeoff: performance vs preprocessing cost

reducing preprocessing cost

(and a more in-depth look at substructure detection)

in: *elems* (matrix elements)
in: *xforms* (set of transformations)

while *True* **do**
    $xf_{best} \leftarrow$ select_best(*xforms*,*elems*)
    if $xf_{best} == \emptyset$ **then** break
    encode *elems* using xf_{best}
    remove xf_{best} from *xforms*

reducing preprocessing cost

(and a more in-depth look at substructure detection)

- **transformations considered:**
  · HORIZ · LINEAR (4) · ALL (18)

in: *elems* (matrix elements)
in: *xforms* (set of transformations)

while *True* **do**
  $xf_{best} \leftarrow$ select_best(*xforms*,*elems*)
  if $xf_{best} == \emptyset$ **then** break
  encode *elems* using xf_{best}
  remove xf_{best} from *xforms*

reducing preprocessing cost

(and a more in-depth look at substructure detection)

- **transformations considered:**
  · HORIZ · LINEAR (4) · ALL (18)

- **preprocesing windows:**
  - sorting is $\mathcal{O}(n \log n)$

```
select_best(xforms,elems):
    xf_best ← ∅ ;
    score_max ← 0 ;
    foreach xf in xforms do
        substr ← detect(xf, elems) ;
        score ← get_score(substr) ;
        if score > score_max then
            xf_best = xf ;
            score_max = score ;
    return xf_best

detect(xf, elems):
    elems ← xf(elems)
    Sort(elems)
    substr ← horiz_detector(elems)
    elems ← xf^{-1}(elems)
    return substr
```

reducing preprocessing cost

(and a more in-depth look at substructure detection)

- **transformations considered:**
  · HORIZ · LINEAR (4) · ALL (18)

- **preprocesing windows:**
  - sorting is $\mathcal{O}(n \log n)$
  - we keep complexity to $\mathcal{O}(nnz)$ by running detection in windows of constant size w

```
select_best(xforms,elems):
    xf_best ← ∅ ;
    score_max ← 0 ;
    foreach xf in xforms do
        substr ← detect(xf, elems) ;
        score ← get_score(substr) ;
        if score > score_max then
            xf_best = xf ;
            score_max = score ;
    return xf_best

detect(xf, elems):
    substr ← ∅
    for i ← 1 to ⌈nnz/w⌉ do
        welems ← window(elems, w)
        welems ← f(welems)
        Sort(welems)
        substr += horiz_detector(elems)
        welems ← f^{-1}(welems)
return substr
```

reducing preprocessing cost

(and a more in-depth look at substructure detection)

- **transformations considered:**
  · HORIZ · LINEAR (4) · ALL (18)

- **preprocesing windows:**
  - sorting is $\mathcal{O}(n \log n)$
  - we keep complexity to $\mathcal{O}(nnz)$ by running detection in windows of constant size w

- **sampling:**

select_best(*xforms*,*elems*):
  $xf_{best} \leftarrow \emptyset$;
  $score_{max} \leftarrow 0$;
  foreach *xf* in *xforms* **do**
    substr $\leftarrow$ detect(*xf*, *elems*) ;
    score $\leftarrow$ get_score(*substr*) ;
    if $score > score_{max}$ **then**
      $xf_{best} = xf$;
      $score_{max} = score$;
  return xf_{best}

detect(*xf*, *elems*):
  substr $\leftarrow \emptyset$
  for $i \leftarrow 1$ **to** $\lceil \frac{nnz}{w} \rceil$ **do**
    welems $\leftarrow$ window(*elems*, *w*)
    welems $\leftarrow$ *f*(*welems*)
    Sort(*welems*)
    substr += horiz_detector(*elems*)
    welems $\leftarrow f^{-1}$(*welems*)
return *substr*

reducing preprocessing cost

(and a more in-depth look at substructure detection)

- **transformations considered:**
  · HORIZ · LINEAR (4) · ALL (18)

- **preprocesing windows:**
  - sorting is $\mathcal{O}(n \log n)$
  - we keep complexity to $\mathcal{O}(nnz)$ by running detection in windows of constant size *w*

- **sampling:**
  detection on a constant number of windows (uniformly distributed)

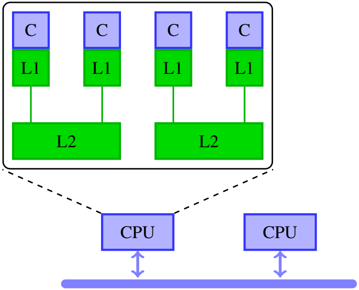
```
select_best(xforms,elems):
    xf_best ← ∅ ;
    score_max ← 0 ;
    foreach xf in xforms do
        substr ← detect(xf, elems) ;
        score ← get_score(substr) ;
        if score > score_max then
            xf_best = xf ;
            score_max = score ;
    return xf_best

detect(xf, elems):
    substr ← ∅
    foreach i in samples do
        welems ← window(elems, w)
        welems ← f(welems)
        Sort(welems)
        substr += horiz_detector(elems)
        welems ← f^-1(welems)
return substr
```

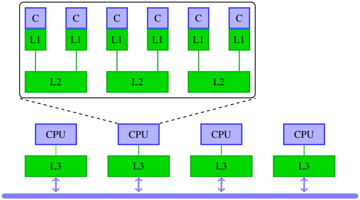
Experimental evaluation

- Machines:

Harpertown



Dunnington



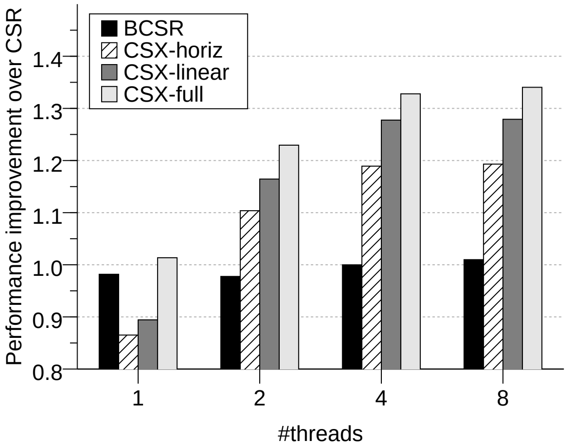
- 15 matrices from real-world applications
- compare against:
  - CSR
  - BCSR (select the best performing block)
- double (64-bit) floating point values

Experimental results: performance improvement

(over multithreaded CSR)

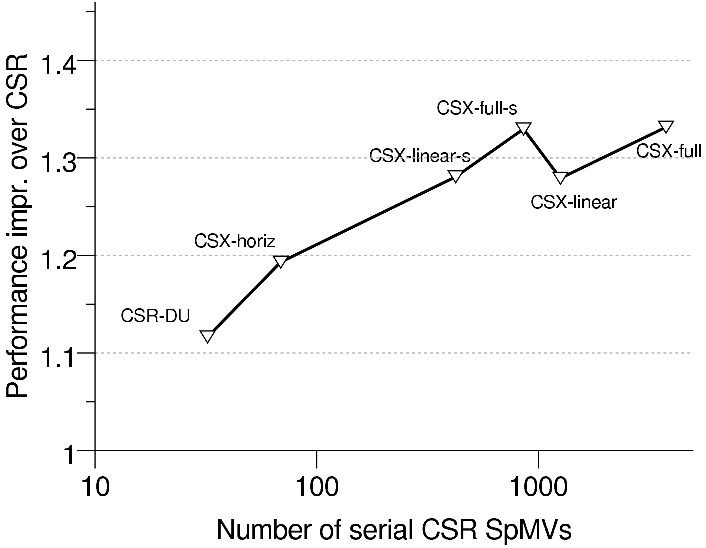
for 8 cores:

- average speedup: 2.21 (33% better than CSR)
- BCSR outperforms CSX only for one matrix
- no matrix with slowdown for CSX



Experimental results: sampling

CSX average performance improvement vs preprocessing cost



Conclusions & future work

CSX:

- aggressive index data compression to optimize SpMxV
- supports arbitrary regularities
- tunable preprocessing cost
- code available at: `http://www.cslab.ece.ntua.gr/csx/`

Conclusions & future work

CSX:

- aggressive index data compression to optimize SpMxV
- supports arbitrary regularities
- tunable preprocessing cost
- code available at: `http://www.cslab.ece.ntua.gr/csx/`

can SpMxV scale?

CSR: | index data (32-bit) | value data (64-bit) |

- index data compression $\rightarrow$ diminishing returns
  (since value data dominate)

Conclusions & future work

CSX:

- aggressive index data compression to optimize SpMxV
- supports arbitrary regularities
- tunable preprocessing cost
- code available at: `http://www.cslab.ece.ntua.gr/csx/`

can SpMxV scale?

CSR: | index data (32-bit) | value data (64-bit) |

- index data compression → diminishing returns
  (since value data dominate)

currently working on:

- improving CSX (e.g., NUMA support, improved heuristics)
- integrating CSX on ELMER (Open Source Finite Element Software)
- power efficiency considerations

EOF

Thank you!
Questions ?

The First Rule of Program Optimization:
Don't do it.

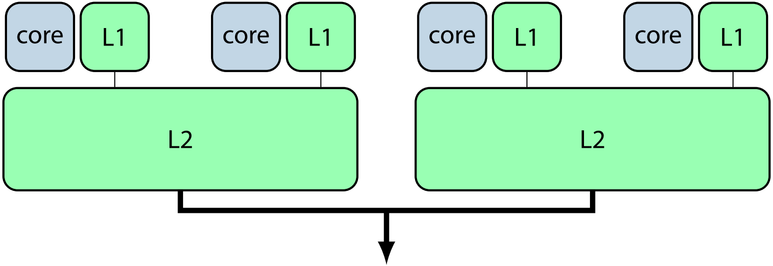
The Second Rule of Program Optimization (for experts only!):
Don't do it yet.

- Michael A. Jackson

Backup slides

Application classes

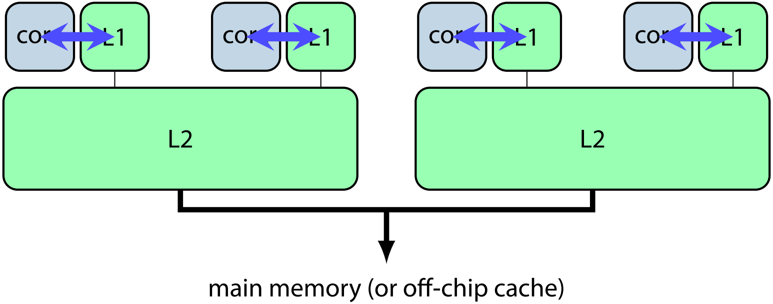
(based on their performance on shared memory systems)



Application classes

(based on their performance on shared memory systems)

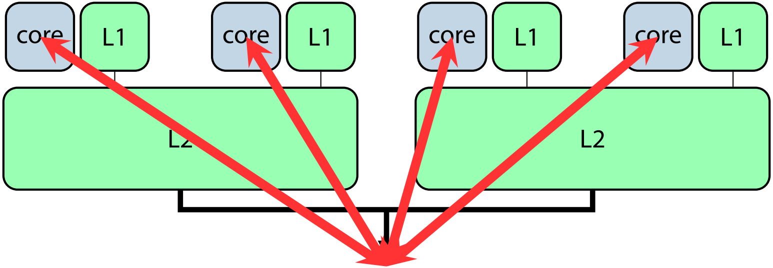
- ✔ Good scalability
  - ✔ temporal locality
  - ✔ no dependecies



Application classes

(based on their performance on shared memory systems)

- ✗ Applications with intensive memory accesses
  - ✗ (very) poor temporal locality
  - ✗ high memory-to-computation ratio
  - ✗ limited scalability due to contention on memory



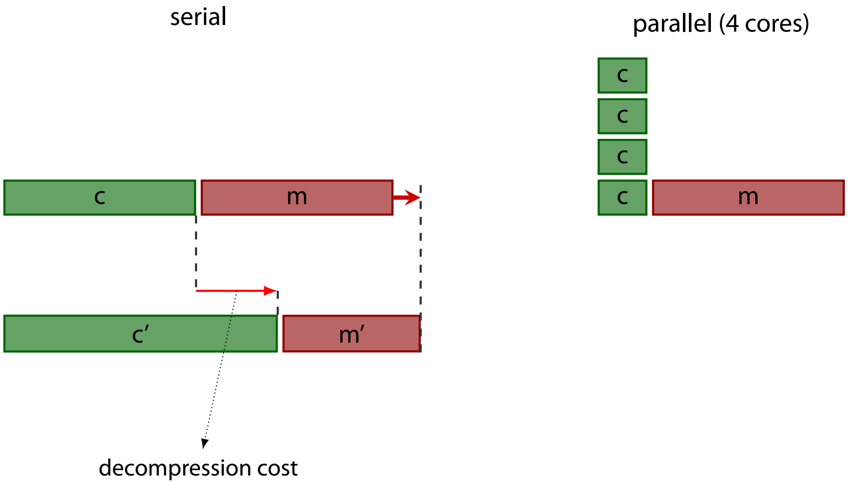
main memory (or off-chip cache)

Improving performance using compression

exchange memory cycles for CPU cycles

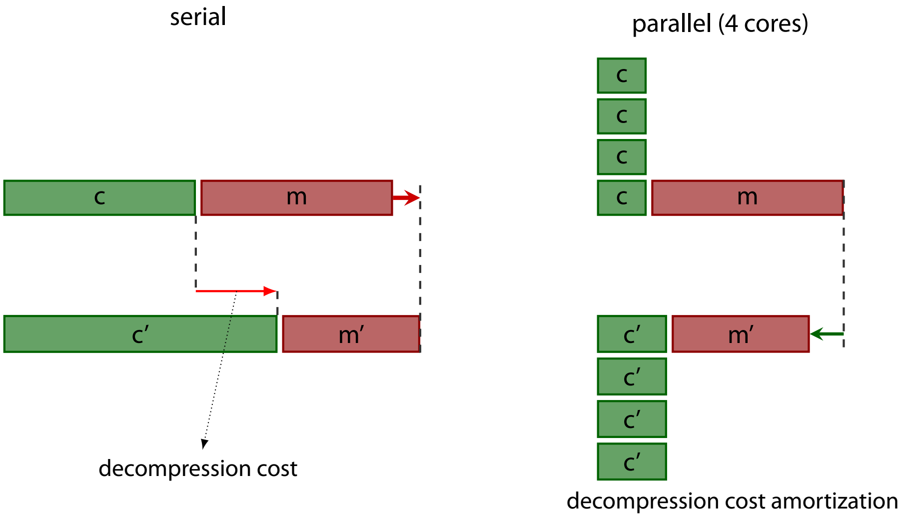
Improving performance using compression

exchange memory cycles for CPU cycles



Improving performance using compression

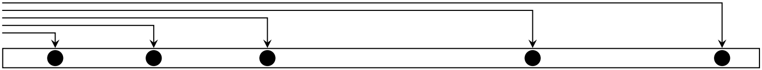
exchange memory cycles for CPU cycles



optimizing SpMxV using index compression

(connection with previous work)

- index data: column indices

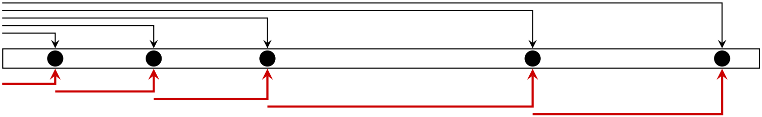


optimizing SpMxV using index compression

(connection with previous work)

- index data: column indices

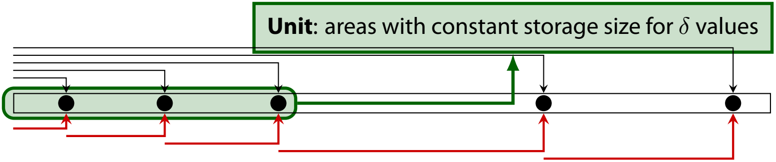
- delta encoding ([Willcock and Lumsdaine '06]):
  instead of ci_i, store $\delta_i = ci_i - ci_{i-1}$
  $\Rightarrow \delta_i \leq ci_i \Rightarrow$ (potentially) less space per index



optimizing SpMxV using index compression

(connection with previous work)

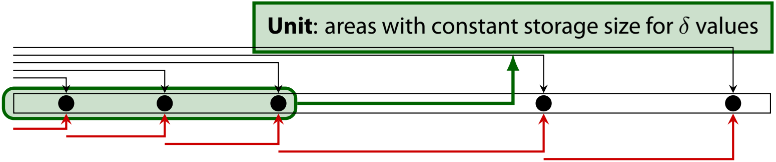
- index data: column indices
- delta encoding ([Willcock and Lumsdaine '06]):
  instead of ci_i, store $\delta_i = ci_i - ci_{i-1}$
  $\Rightarrow \delta_i \leq ci_i \Rightarrow$ (potentially) less space per index
- CSR-DU ([Kourtis et al. '08]): coarse-grained delta encoding



optimizing SpMxV using index compression

(connection with previous work)

- index data: column indices
- delta encoding ([Willcock and Lumsdaine '06]):
  instead of ci_i, store $\delta_i = ci_i - ci_{i-1}$
  $\Rightarrow \delta_i \leq ci_i \Rightarrow$ (potentially) less space per index
- CSR-DU ([Kourtis et al. '08]): coarse-grained delta encoding



- **CSX**: (more) aggressive compression by supporting units with arbitrary *regularities* ($\mathcal{O}(1)$ space)