# Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression

Kornilios Kourtis

kkourt@cslab.ece.ntua.gr

National Technical University of Athens

Computing Systems Laboratory

# Outline

- Introduction and Motivation
- Index Compression (CSR-DU)
- Value Compression (CSR-VI)
- Performance Evaluation
- Conclusions

# SpMxV

- **Sparse Matrices**:
  - Larger portion of elements are $0$'s
  - Efficient representation (storage and computation)
    - non-zero values (`nnz`)
    - indexing information – structure
- **Formats**:
  - CSR, CSC, COO
  - BCSR
  - JD, CDS, Elpack-Itpack
- **Sparse Matrix-Vector Multiplication (SpMxV)**:
  - $y = A \cdot x$, $A$ is sparse
  - important, used in a variety of applications (eg, PDE solvers – CG, GMRES)

# Compressed Sparse Row (CSR)

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

$\texttt{row_ptr}:$ ( 0 2 5 6 9 12 16 )

$\texttt{col_ind}:$ ( 0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5 )

$\texttt{values}:$ ( 5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1 )

# Compressed Sparse Row (CSR)

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

row_ptr : ( 0 2 5 6 9 12 16 )

col_ind : ( 0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5 )

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 1.1 2.9 3.7 1.1 )

# CSR SpMxV

```
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]];
```

row_ptr: ( 0 2 5 6 9 12 16 )

col_ind: ( 0 1 1 3 5 2 2 4 5 0 3 4 0 2 3 5 )

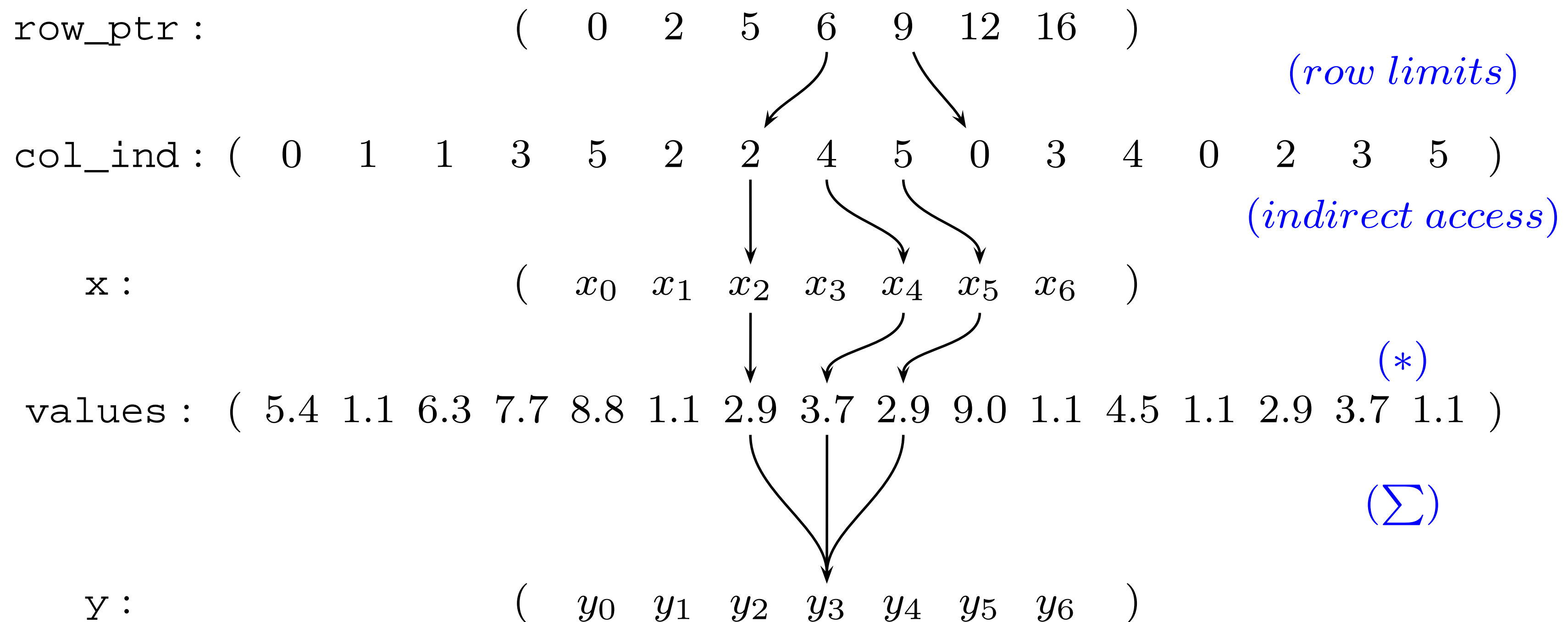
x: ( $x_0$ $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ )

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 1.1 2.9 3.7 1.1 )

y: ( $y_0$ $y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$ )

# CSR SpMxV

```
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]];
```



# CSR SpMxV performance

- memory bandwidth is the main bottleneck (Goumas et al. PDP '08)
- spmv accesses: ($N \times N$ sparse matrix, `nnz` $\gg N$)

| Array | size | accesses | pattern | type |
|---|---|---|---|---|
| `row_ptr` | $N$ | $N$ | sequential | read |
| `values` | `nnz` | `nnz` | sequential | read |
| `col_ind` | `nnz` | `nnz` | sequential | read |
| `x` | $N$ | `nnz` | random, $\uparrow$ | read |
| `y` | $N$ | $N$ | sequential | write |

- Thus, we target working set (`ws`) reduction
- allows better scaling for shared memory architectures
- `values`, `col_ind` dominate working set

# CSR SpMxV working set

$$\mathtt{ws} \approx \underbrace{\mathtt{nnz} \cdot value_size}_{\mathtt{values}} + \underbrace{\mathtt{nnz} \cdot index_size}_{\mathtt{col_ind}}$$

$32$-bit indices, $64$-bit values (common case)

$64$-bit indices, $64$-bit values ($\sim 1T$ $\mathtt{ws}$ size)

# Objective

Explore the design space for accelerating SpMxV using working set reduction techniques

- Propose two methods (index / value compression)
- Evaluate on a rich matrix set
- Investigate issues, identify trade-offs
- Explore future directions

# Compression Methods

# Methods overview

- Compression $\Rightarrow$ trade computation for data size
- data size reduction is not enough (SpMxV run-time)

- Index Compression: **CSR-DU**
  - general
  - coarse-grain delta encoding for column indices

- Value Compression: **CSR-VI**
  - specialized
  - exploits large number of common values

# Index Compression

- Blocking methods (BCSR, VBR)
  per block indexing ⇒ index data reduction
- Delta encoding for column indices
  (Willcock and Lumsdaine : DCSR, RPCSR – ICS 06)

  `col_ind` : $61311$ $61336$ $61390$ $61400$ $61428$

  `deltas` : ... $25$ $54$ $10$ $28$
- DCSR:
  - byte-oriented
  - $6$ sub-operations for implementing SpMxV
  - decoding overhead → performance degradation (branches)
  - patterns of frequent used groups of sub-ops
  - complex, non-portable, matrix-specific

# CSR-DU (CSR Delta Units)

- Exploit dense areas using delta encoding
- Coarse-grain approach:
  - matrix is partitioned into variable-length units
  - each unit has a delta size
  - less compression ratio
  - innermost loops without branches
- Compared to DCSR:
  - comparable performance
  - portable, easier to implement
  - suitable for matrices with large variation

# CSR-DU storage format

- `ctl` byte array replaces `row_ptr`, `col_ind`
- unit contents:

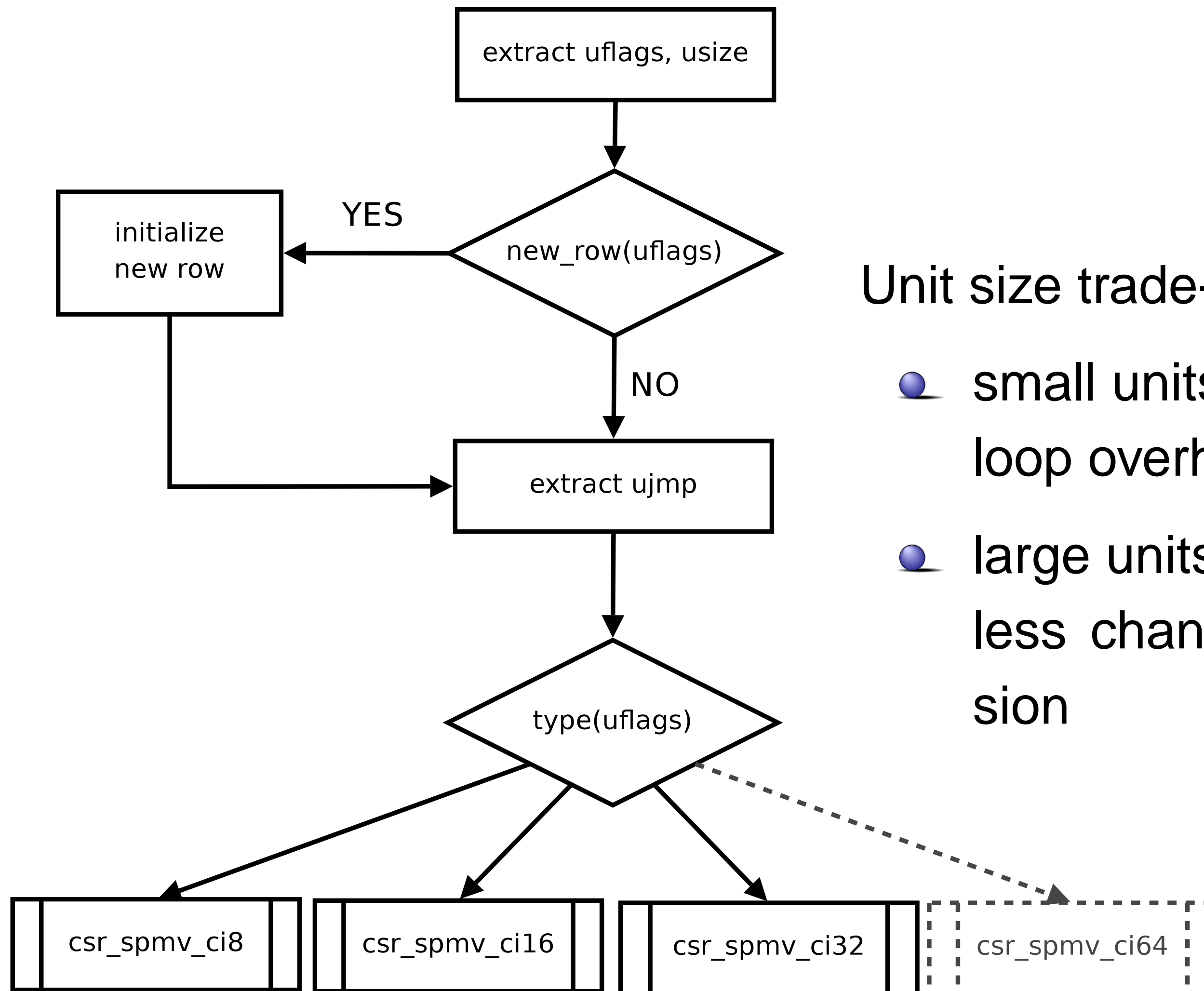
| field | description | size |
|---|---|---|
| `usize` | size | $1$ byte |
| `uflags` | flags (new row, delta_size) | $1$ byte |
| `ujmp` | initial delta | variable length |
| `ucis` | subsequent deltas | `usize` $\cdot\ delta_size$ |

- Example:

$$(7,1)(7,127)(7,250)(7,255)(8,10)(8,1021)$$

$$\underbrace{[4, \overbrace{NR|U8}^{\texttt{uflags}}, 1, \overbrace{(126,123,5)}^{\texttt{ucis}}]}_{\texttt{unit}}\ [2, NR|U16, 10, (1011)]$$

# CSR-DU SpMxV

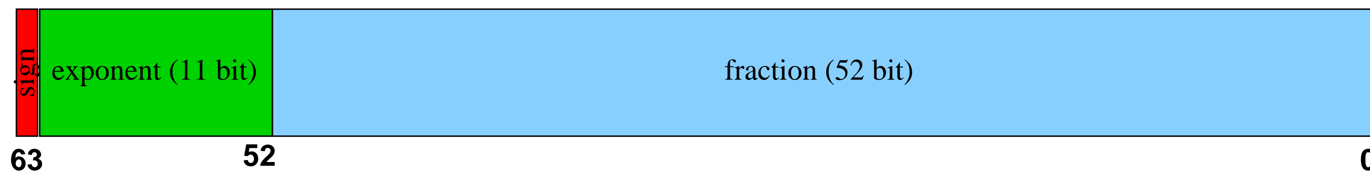



Unit size trade-off:

- small units:
  loop overhead (small rows)
- large units:
  less chances for compression

# Value Compression

- Values:
  - Typically the largest part of the `ws` (32i-64v)
  - (more) difficult to compress:
    - FP arithmetic produces rounded results
    - FP format




- significant number of matrices in our set with a small number of *unique* values.
- feasibility metric: total-to-unique ratio $(ttu = \frac{nnz}{unique\ values})$

# CSR-VI

Indirect access for values:

`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 1.1 2.9 3.7 1.1 )

`val_ind` + `vals_unique`:

( 0 1 2 3 4 1 5 6 5 7 1 8 1 5 6 1 )

( 5.4 1.1 6.3 7.7 8.8 2.9 3.7 9.0 4.5 )

# CSR-VI

Indirect access for values:

`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 1.1 2.9 3.7 1.1 )

`val_ind` + `vals_unique`:

( 0 1 2 3 4 1 5 6 5 7 1 8 1 5 6 1 )

( 5.4 1.1 6.3 7.7 8.8 2.9 3.7 9.0 4.5 )

# CSR-VI

Indirect access for values:

`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 1.1 2.9 3.7 1.1 )

`val_ind` + `vals_unique`:
( 0 1 2 3 4 1 5 6 5 7 1 8 1 5 6 1 )

( 5.4 1.1 6.3 7.7 8.8 2.9 3.7 9.0 4.5 )

| format | values size |
|---|---|
| CSR | $nnz \cdot size_v$ |
| CSR-VI | $nnz \cdot size_vi + uvals \cdot size_v$ |

$size_vi \rightarrow$ smallest integer that can address $uvals$ elements
(e.g. $uvals \leq 256 \Rightarrow size_vi = 1\ byte$)

# CSR-VI SpMxV

```
for(i=0; i<N; i++)
  for(j=row_ptr[i]; j<row_ptr[i+1]; j++){
    val = vals_unique[val_ind[j]];
    y[i] += val*x[col_ind[j]];
  }
```

- one memory access added (indirect)
- access to `vals_unique` is random

# Experimental Evaluation

# Experimental Setup

- System
  - Intel Core 2 Xeon (Woodcrest) @2.6 GHz, 4MB L2
  - 64-bit linux, gcc-4.2 -O3
- SpMxV Benchmark
  - 32-bit indices, 64-bit values
  - 128 iterations
- Matrix set
  - start: 100 matrices (Tim Davis, SPARSITY, ...)
  - memory bound set $\mathcal{M}_0$: $ws > \frac{3}{4}L2$ (77 matrices)

# CSR-DU Performance

- Reject small row matrices: 59 remaining matrices ( $85\%$ $\texttt{nnz}$ in rows with $\leq 6$ elements)
- Summary:

| matrices | | speedup (%) | | | |
|---|---|---|---|---|---|
| total | sp > 1 | avg. | min | max | dense |
| $64$ | $59$ | $8.1$ | $-8.1$ | $18.9$ | $35$ |

- 64-bit indices $+36\%$

detailed results

# CSR-VI Performance

- Reject matrices with low $ttu$: 30 remaining matrices: ($ttu < 5$)
- Summary:

| matrices | | speedup (%) | | |
|---|---|---|---|---|
| total | sp > 1 | avg. | min | max |
| $30$ | $26$ | $21.5$ | $-31.1$ | $74.1$ |

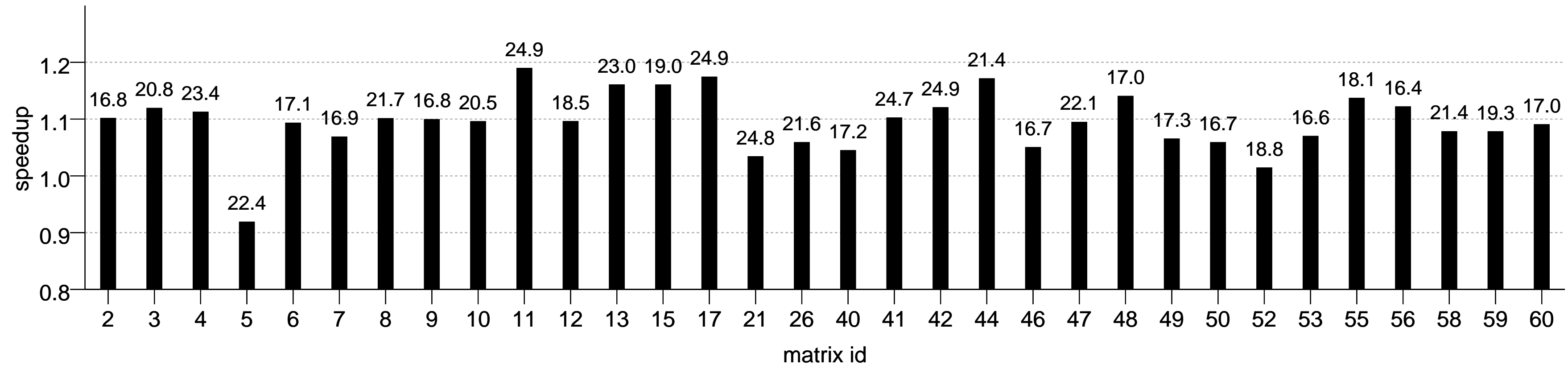
detailed results

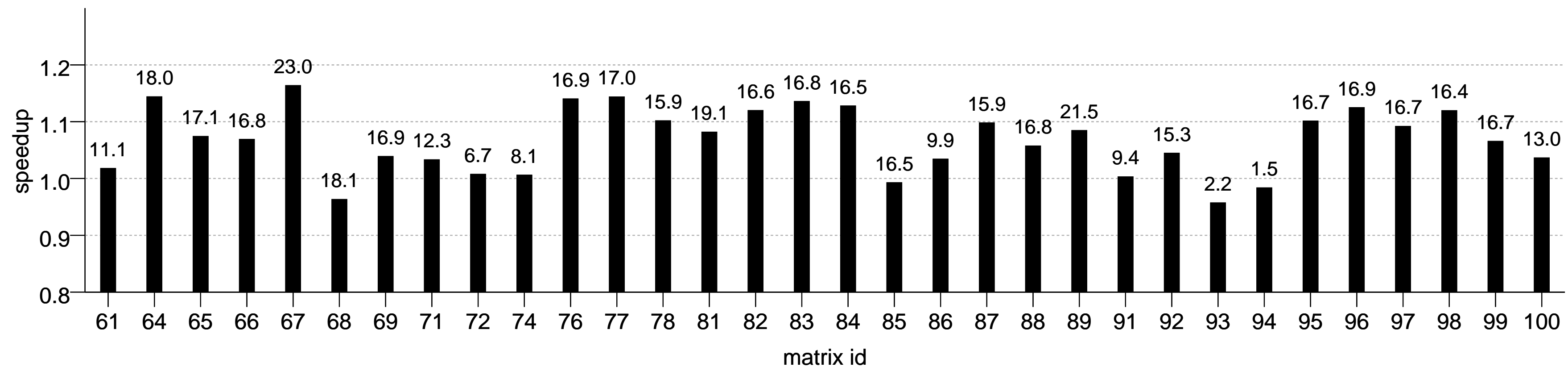
# Conclusions and Future Directions

- Index compression:
  - limited perfomance gain for the 32i-64v case
  - “pure” computation (not hard-to-predict branches)
  - more aggressive compression (global)
  - expand the “unit” concept to support more types of regularities
  - matrix-specific code generation
- Value compression
  - common case: values largest part of `ws`
  - difficult (constrained regularity, nature of FP)
  - specialized schemes
- shared memory architectures
- working set reduction for other applications

# EOF

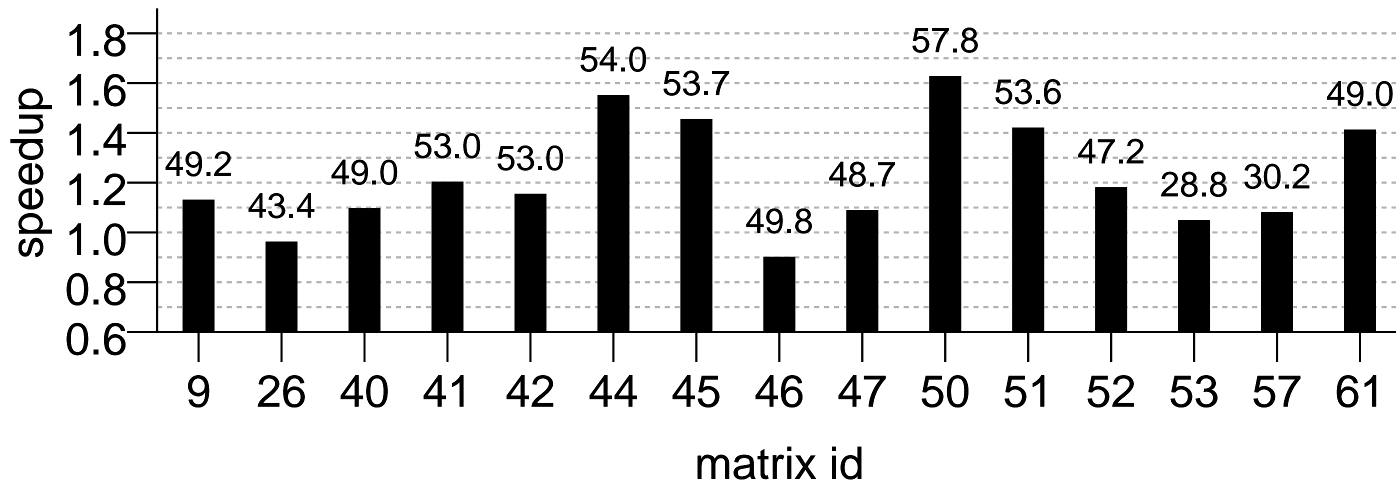
# CSR-DU Performance (2)

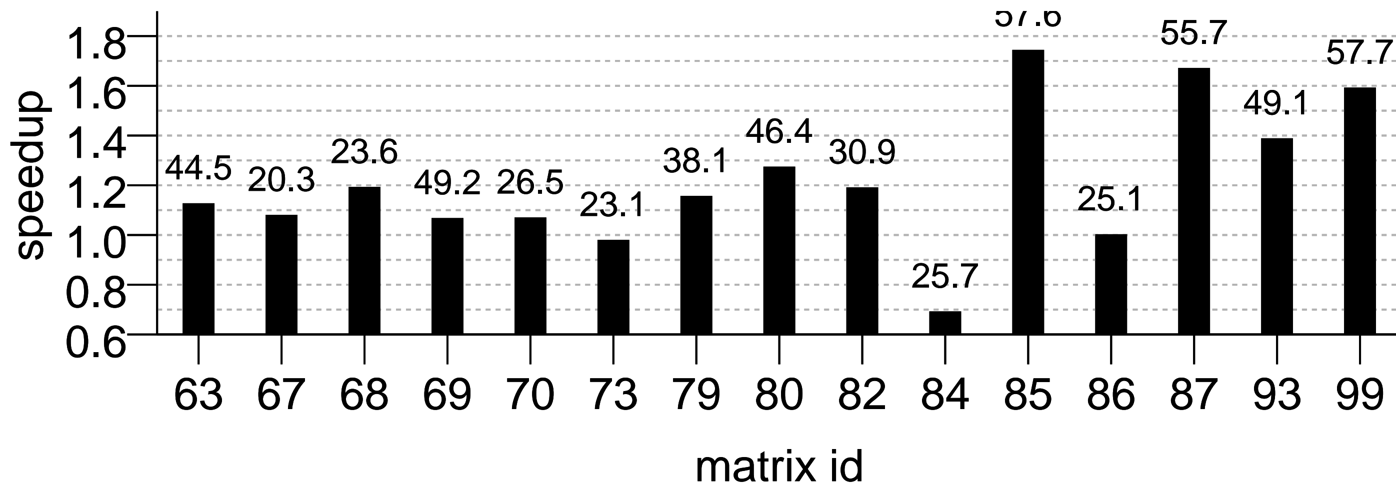



summarized results

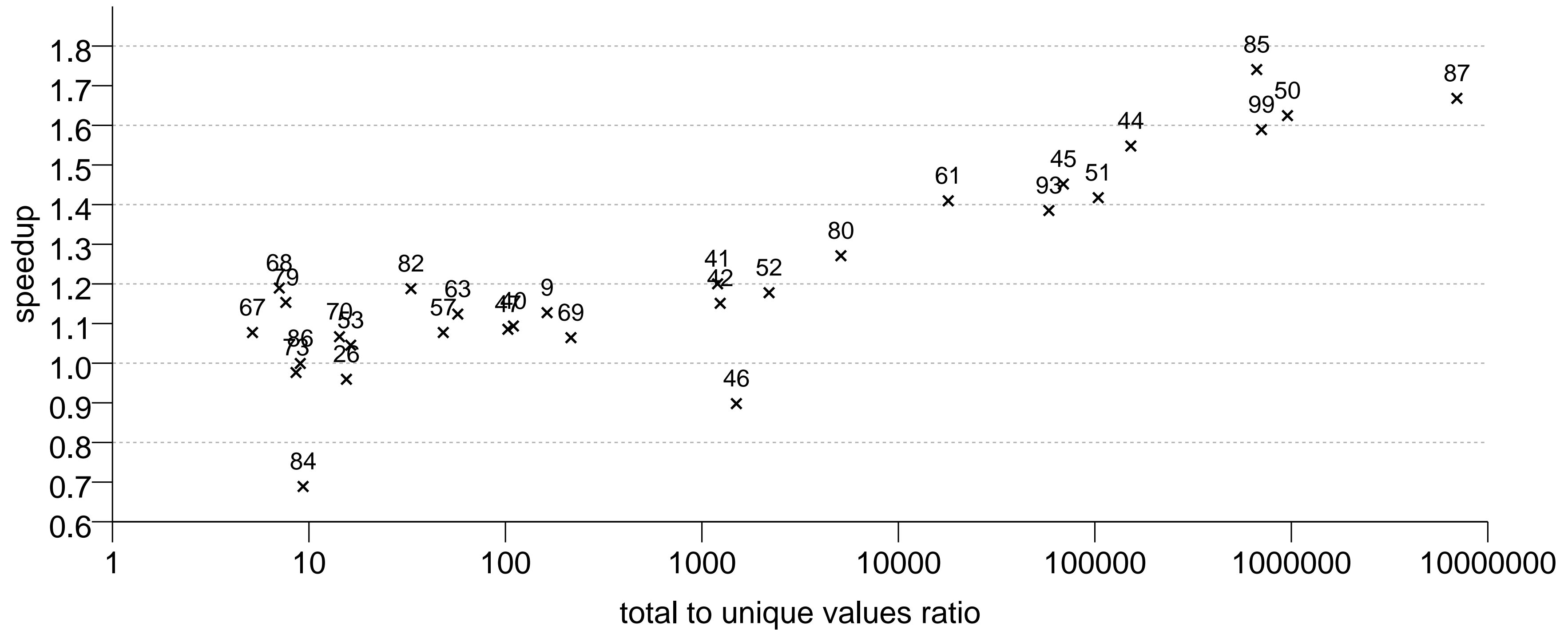
# CSR-VI Performance (2)




summarized results

# CSR-VI Performance (3 – ttu)




summarized results