Improving the Performance of Multithreaded 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)

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

`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)

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)
`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)

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]];
```

$\mathbf{i = 3}$

`row_ptr`: (0 2 5 6 9 12 16)

(row limits)

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

(indirect access)

`x`: (x_0 x_1 x_2 x_3 x_4 x_5)

$(*)$

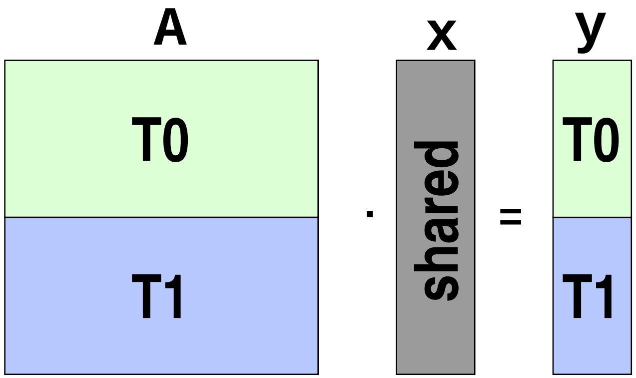
`values`: (5.4 1.1 6.3 7.7 8.8 1.1 2.9 3.7 2.9 9.0 1.1 4.5 1.1 2.9 3.7 1.1)

$(\sum)$

`y`: (y_0 y_1 y_2 y_3 y_4 y_5)

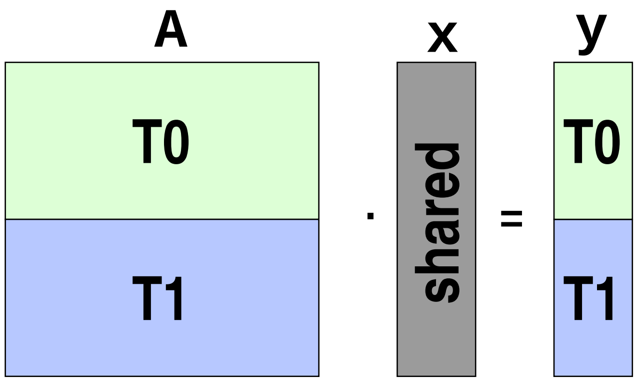
Multithreaded CSR SpMxV

- **Row** (Column, Block) Partitioning
- Only x shared (read-only)
- nnz balancing



Multithreaded CSR SpMxV

- **Row** (Column, Block) Partitioning
- Only x shared (read-only)
- nnz balancing



Example:

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)
y :		(y_0 y_1 y_2 y_3	y_4 y_5)

CSR SpMxV performance

- memory bandwidth is main bottleneck (Goumas et al. PDP08)
- poor scaling for shared memory architectures
- 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
- `values`, `col_ind` dominate working set

CSR SpMxV working set

$$\texttt{ws} \approx \underbrace{\texttt{nnz} \cdot value\_size}_{\texttt{values}} + \underbrace{\texttt{nnz} \cdot index\_size}_{\texttt{col\_ind}}$$

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

64-bit indices, 64-bit values ($\sim 1T$ `ws` size)

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 storage schemes (BCSR, VBR)
  per block indexing $\Rightarrow$ index data reduction
- Delta encoding for column indices
  (Willcock and Lumsdaine : DCSR, RPCSR – ICS 06)

  col_ind : $61311 \; 61336 \; 61390 \; 61400 \; 61428$

  deltas : $\ldots \quad 25 \quad 54 \quad 10 \quad 28$
- DCSR:
  - byte-oriented
  - 6 sub-operations for implementing SpMxV
  - decoding overhead $\rightarrow$ 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:

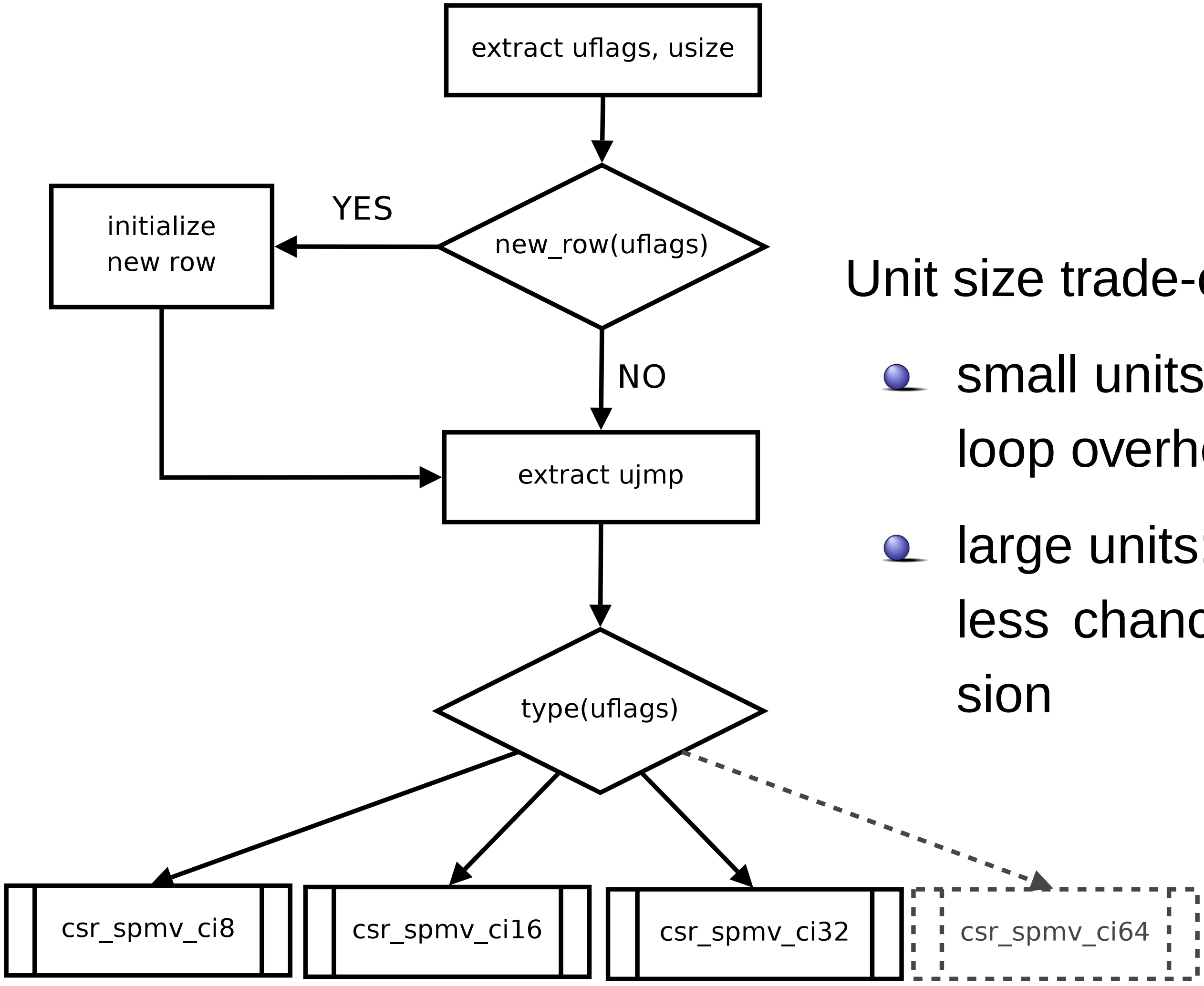
field	description	size
`usize`	size	1 byte
`uflags`	flags (new row, delta_size)	1 byte
`ujmp`	initial delta	variable length
`ucis`	subsequent deltas	`usize` $\cdot\, delta\_size$

- 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

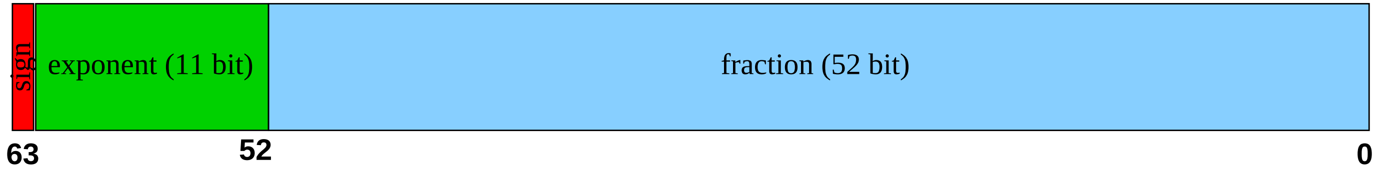


Unit size trade-off:

- 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:
    - not inherently compressibile
    - 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

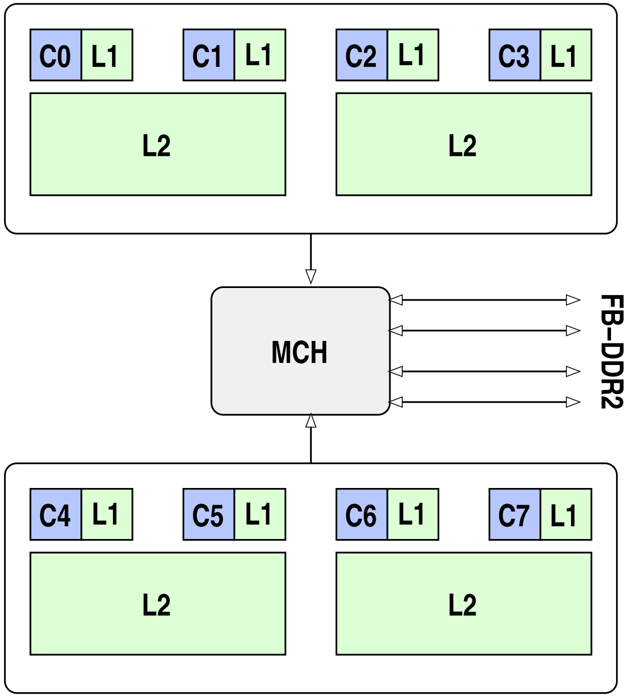
```
for(i=0; i<N; i++)
  for(j=row_ptr[i]; j<row_ptr[i+1]; j++){
    val = vals_unique[val_ind[j]];
    y[i] += val*x[col_ind[j]];
  }
```

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

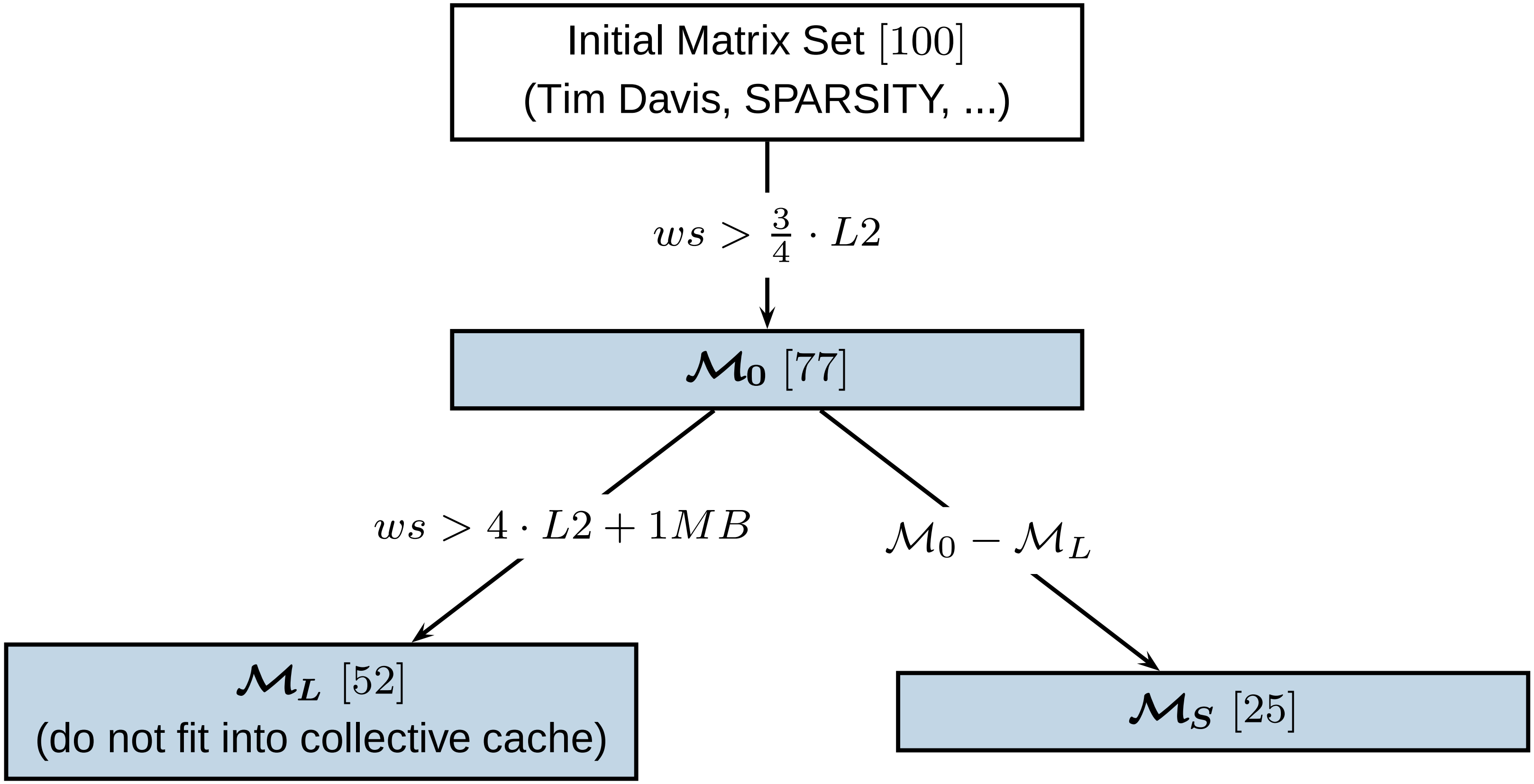
Experimental Evaluation

Experimental Setup

- System
  - 2 quad Clovertown processors
  - shared caches
  - 2GHz, 4MB L2
  - 64-bit linux, gcc-4.2 -O3
- SpMxV Benchmark
  - 32-bit indices, 64-bit values
  - 128 iterations



Matrix Set(s)


Initial Matrix Set $[100]$
(Tim Davis, SPARSITY, ...)
$ws > \frac{3}{4} \cdot L2$
$\boldsymbol{\mathcal{M}_0}$ $[77]$
$ws > 4 \cdot L2 + 1MB$
$\mathcal{M}_0 - \mathcal{M}_L$
$\boldsymbol{\mathcal{M}_L}$ $[52]$
(do not fit into collective cache)
$\boldsymbol{\mathcal{M}_S}$ $[25]$

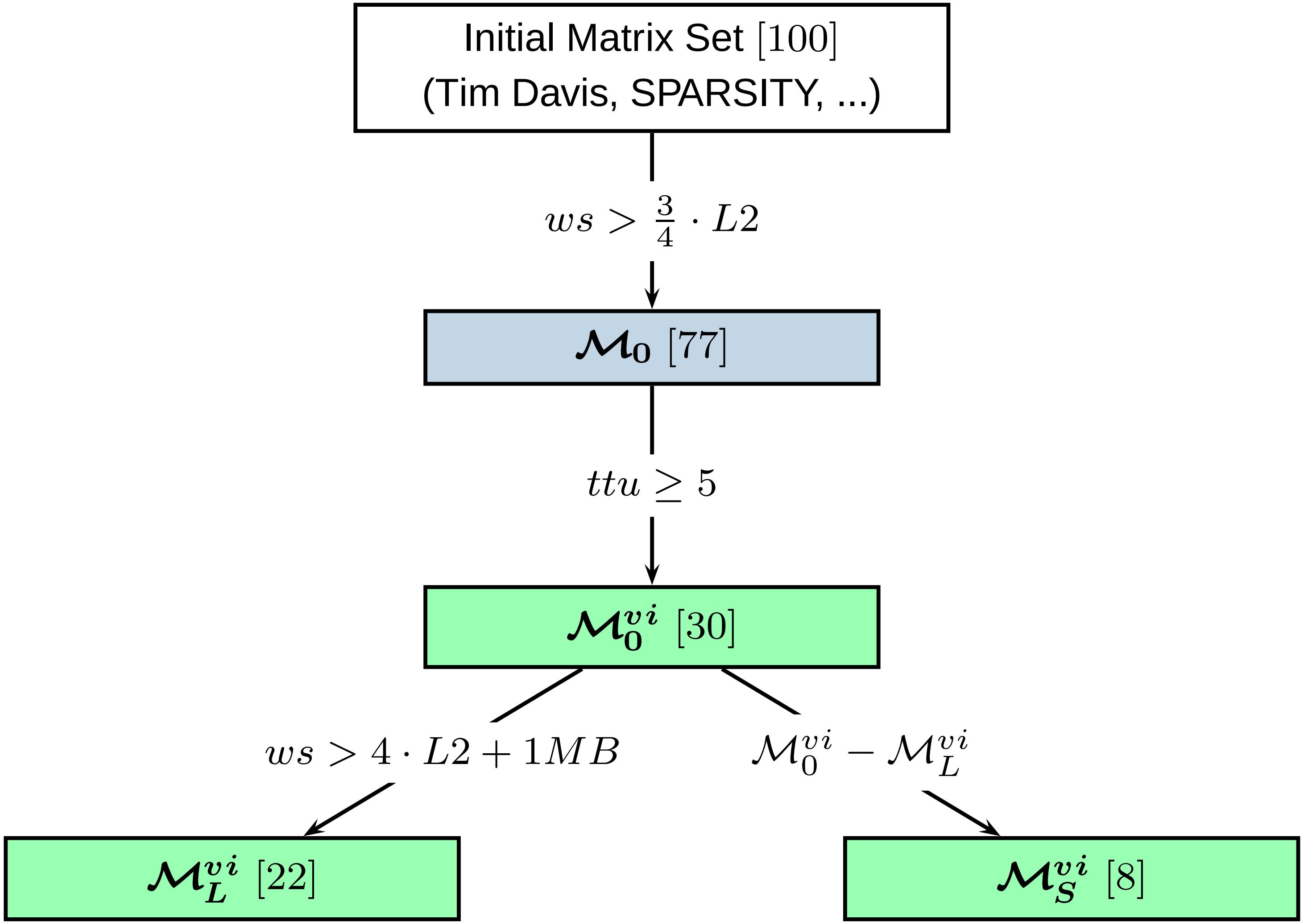
CSR MT Performance

	$\mathcal{M}_0$	$\mathcal{M}_S$ (25 matrices)			$\mathcal{M}_L$ (52 matrices)		
core(s)	avg	avg	max	min	avg	max	min
1	**523.6**	**619.4**	886.6	465.2	**477.8**	594.4	202.4
2 ($1\times L2$)	**1.16**	**1.17**	1.62	0.90	**1.15**	1.40	1.07
2 ($2\times L2$)	**1.46**	**1.93**	2.59	1.24	**1.24**	1.47	1.09
4	**1.72**	**2.63**	4.32	1.54	**1.28**	1.73	1.12
8	**3.44**	**6.19**	8.71	2.12	**2.12**	6.30	1.58

CSR-DU MT Performance

	$\mathcal{M}_0$	$\mathcal{M}_S$ (25 matrices)				$\mathcal{M}_L$ (52 matrices)			
core(s)	avg	avg	max	min	<0.98	avg	max	min	<0.98
1	**1.01**	**1.02**	1.12	0.80	5	**1.01**	1.14	0.69	17
2	**1.15**	**1.24**	1.49	1.06	0	**1.10**	1.19	0.90	2
4	**1.18**	**1.24**	1.89	0.81	4	**1.15**	1.36	0.99	0
8	**1.15**	**1.05**	1.40	0.86	8	**1.20**	1.82	0.99	0

Matrix Set(s) – CSR-VI



CSR-VI MT Performance

	$\mathcal{M}_0^{vi}$	$\mathcal{M}_S^{vi}$ (8 matrices)				$\mathcal{M}_L^{vi}$ (22 matrices)			
core(s)	avg	avg	max	min	<0.98	avg	max	min	<0.98
1	**1.10**	**1.03**	1.17	0.94	2	**1.12**	1.54	0.65	7
2	**1.35**	**1.30**	1.56	0.99	0	**1.36**	2.07	0.80	3
4	**1.47**	**1.25**	2.04	0.96	1	**1.55**	2.16	1.00	0
8	**1.44**	**1.02**	1.15	0.92	3	**1.59**	2.50	0.99	0

Conclusions and Future Directions

- Conclusions:
  - index : inherent, smaller part of `ws`
  - value : contrained regularity, larger part of `ws`
  - compression can lead to improved MT performance for large matrices

- Future Directions:
  - combine index/value compression
  - More aggressive compression (serial slowdown)
  - Other memory bound applications

EOF