

# Reordering Sparse Matrices for Cache-Oblivious Computations

Albert-Jan Yzelman & Rob Bisseling

February 2010



# Outline

- 1 Memory and multiplication
- 2 Cache-friendly data structures
- 3 Cache-oblivious sparse matrix structure
- 4 Moving to two dimensions
- 5 Experimental results

Reference: Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SISC, 2009



# Memory and multiplication

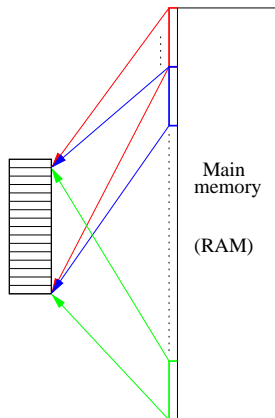
- 1 Memory and multiplication
- 2 Cache-friendly data structures
- 3 Cache-oblivious sparse matrix structure
- 4 Moving to two dimensions
- 5 Experimental results



# Naive cache

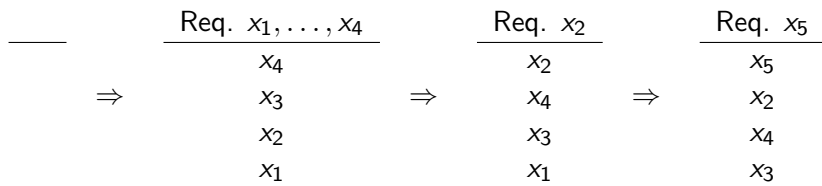
$k = 1$ , modulo mapped cache

Dividing main memory (RAM) in stripes of size  $L_S$ , the  $x$ th line is mapped to the cache line number  $x \bmod L$ :



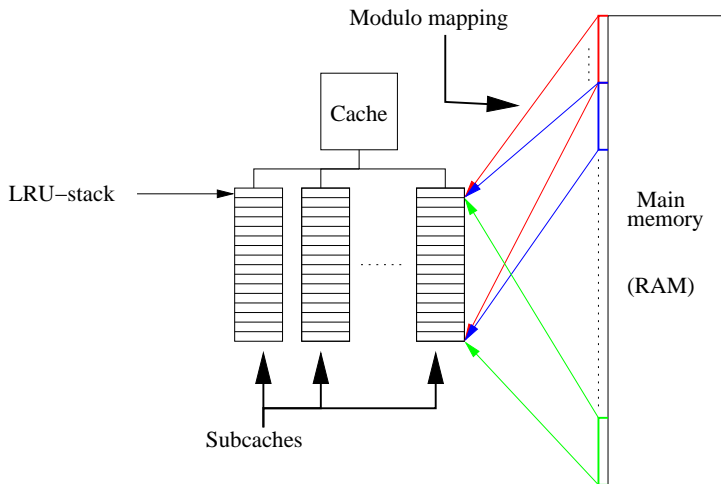
## 'Ideal' cache

Instead of a naive modulo mapping, we use a smarter policy. We take  $k = L = 4$ , using 'Least Recently Used (LRU)' policy:

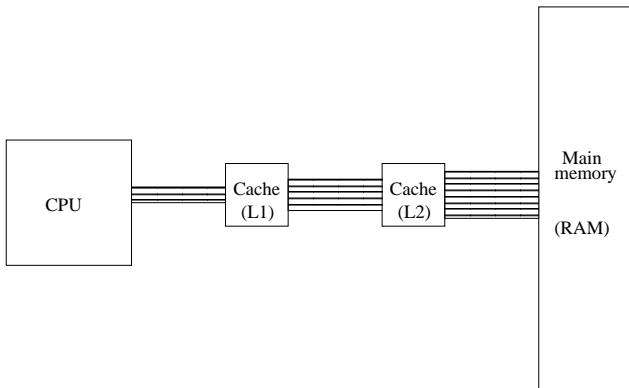


# Realistic cache

$1 < k < L$ , combining modulo-mapping and the LRU policy



# Multilevel caches



Intel Core2 (Q6600)

L1: 32kB  $k = 8$

L2: 4MB  $k = 16$

L3: - -

AMD Phenom II (945e)

$S = 64\text{kB}$   $k = 2$

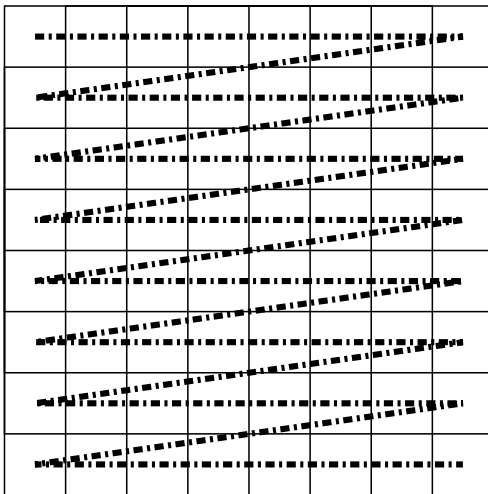
$S = 512\text{kB}$   $k = 8$

$S = 6\text{MB}$   $k = 48$



# The sparse case

Standard datastructure: Compressed Row Storage (CRS)





# The sparse case

Sparse matrix–vector multiplication (SpMV)

$x?$



# The sparse case

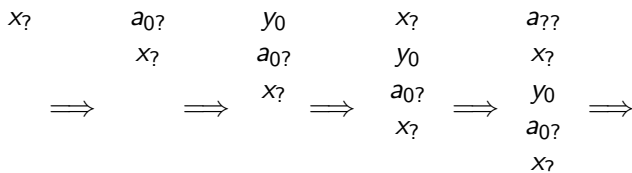
## Sparse matrix–vector multiplication (SpMV)

$$\begin{array}{ccc}
 x_0 & a_0 & y_0 \\
 & x_1 & a_1 \\
 & & x_2 \\
 \Rightarrow & \Rightarrow & \Rightarrow
 \end{array}$$



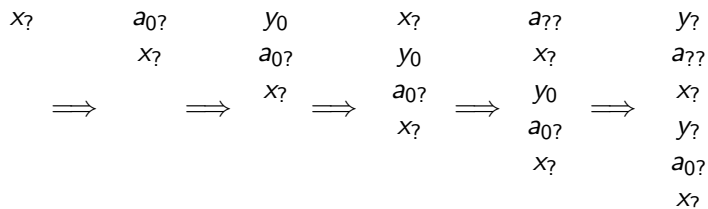
# The sparse case

## Sparse matrix–vector multiplication (SpMV)



# The sparse case

## Sparse matrix–vector multiplication (SpMV)



We cannot predict memory accesses in the sparse case!



# Cache-friendly data structures

- 1 Memory and multiplication
- 2 Cache-friendly data structures
- 3 Cache-oblivious sparse matrix structure
- 4 Moving to two dimensions
- 5 Experimental results



## CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

nzs: [4 1 3 2 3 1 2 7 1 1]  
 col: [0 1 2 2 3 0 3 0 2 3] ,  $2nnz + (m + 1)$  accesses  
 row: [0 3 5 7 10]



# Incremental CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

$$\begin{array}{l} \text{nzs:} \quad [4 \ 1 \ 3 \ 2 \ 3 \ 1 \ 2 \ 7 \ 1 \ 1] \\ \text{col\_increment:} \quad [0 \ 1 \ 1 \ 4 \ 1 \ 1 \ 3 \ 1 \ 2 \ 1] \ , \ 2nnz + m \ \text{accesses} \\ \text{row\_increment:} \quad [0 \ 1 \ 1 \ 1] \end{array}$$

Note: accesses like plain CRS, but requires less instructions for SpMV

Reference: Joris Koster, *Parallel templates for numerical linear algebra, a high-performance computation library* (Masters Thesis), 2002.



## Blocked CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}, \text{ dense blocks: } 4, 1, 3 / 2, 3 / 1 / 2 / 7, 0, 1, 1$$

Stored as:

nzs: [4 1 3 2 3 1 2 7 0 1 1]

blk: [0 3 5 6 7 11]

col: [0 2 0 3 0]

row: [0 1 2 4 5]

,  $nnz + (2nblk + 1) + (m + 1)$  accesses

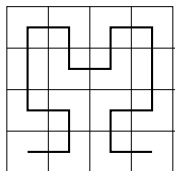
Reference: Pinar and Heath, *Improving Performance of Sparse Matrix-Vector Multiplication*, 1999





## Fractal datastructures (triplets)

$$A = \begin{pmatrix} 4 & 1 & 0 & 2 \\ 0 & 2 & 0 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 0 \end{pmatrix}$$



Stored as:

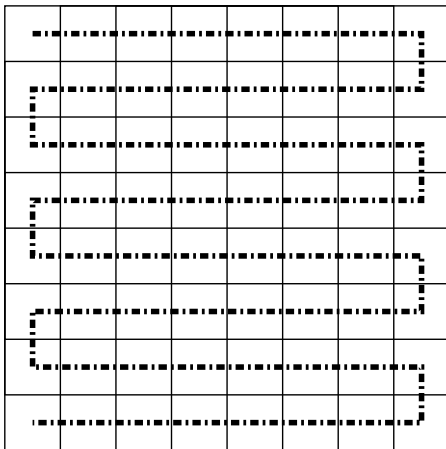
nzs: [7 1 4 1 2 2 3 2 1]  
 i: [3 2 0 0 1 0 1 2 3] , **3nnz** accesses per nonzero  
 j: [0 0 0 1 1 3 3 3 2]

Reference: Haase, Liebmann and Plank, *A Hilbert-Order Multiplication Scheme for Unstructured Sparse Matrices*, 2005



# Zig-zag CRS

Change the order of CRS:



## Zig-zag CRS

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$

Stored as:

nzs: [4 1 3 3 2 1 2 1 1 7]

col: [0 1 2 3 2 0 3 3 2 0] ,  $2nnz + (m + 1)$  accesses

row: [0 3 5 7 10]

Reference: Yzelman and Bisseling, *Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods*, SISC, 2009



# Cache-oblivious sparse matrix structure

- 1 Memory and multiplication
- 2 Cache-friendly data structures
- 3 Cache-oblivious sparse matrix structure
- 4 Moving to two dimensions
- 5 Experimental results

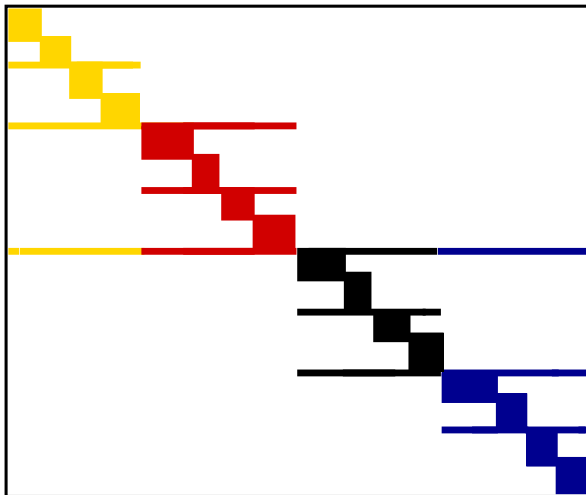


## Why not also change the input matrix structure?

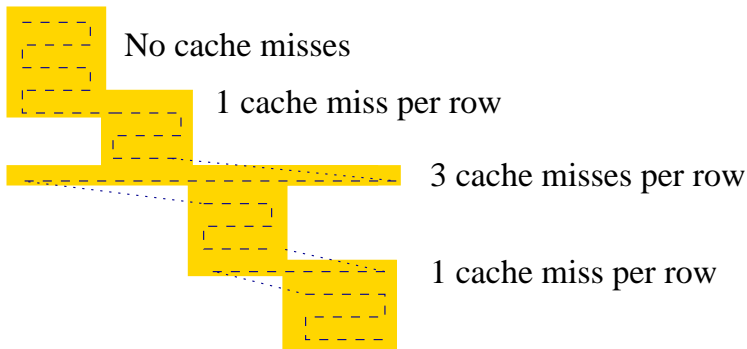
- Assuming zig-zag CRS ordering (theoretically)
- Allowing only row and column permutations



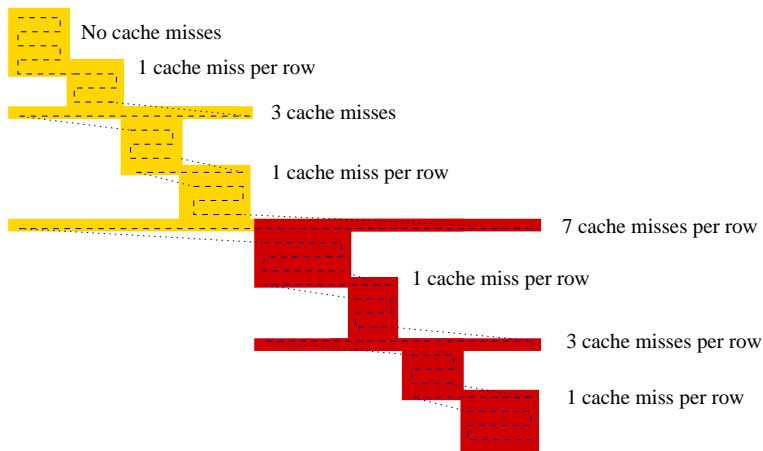
# Separated Block Diagonal form



# Separated Block Diagonal form

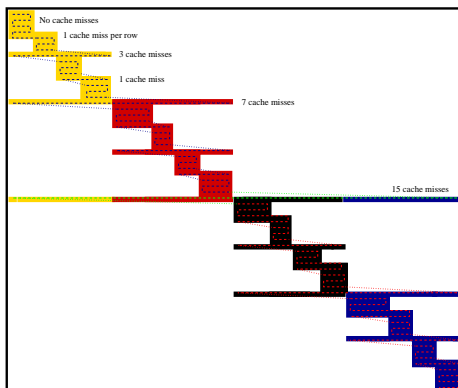


# Separated Block Diagonal form





# Separated Block Diagonal form

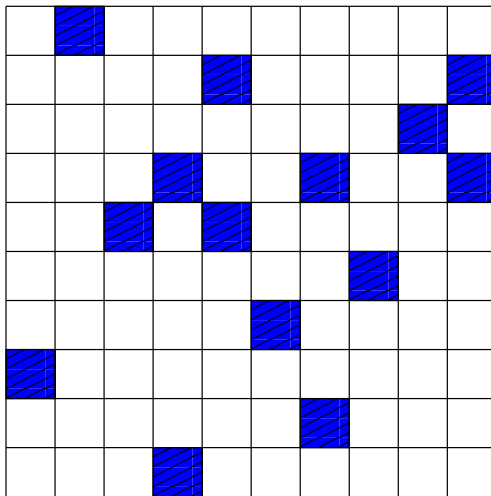


(Upper bound on) the number of cache misses:  $\sum_i (\lambda_i - 1)$



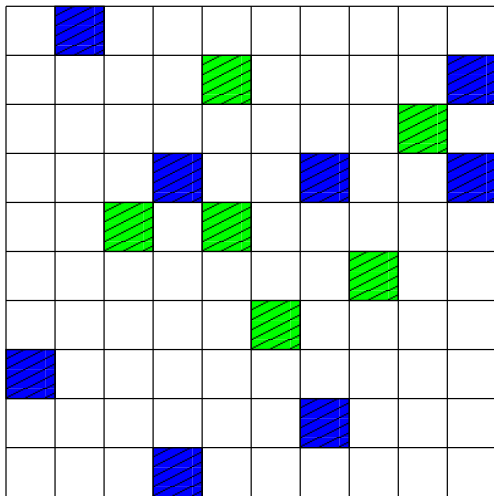
# Permuting to SBD form

Input



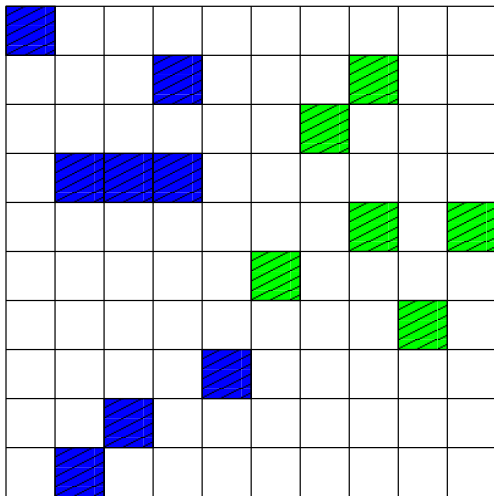
# Permuting to SBD form

## Column partitioning



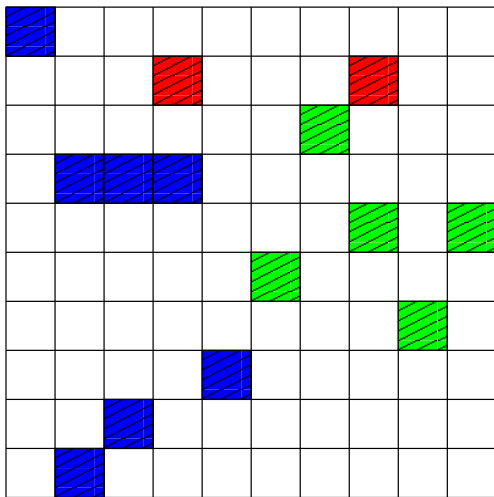
# Permuting to SBD form

## Column permutation



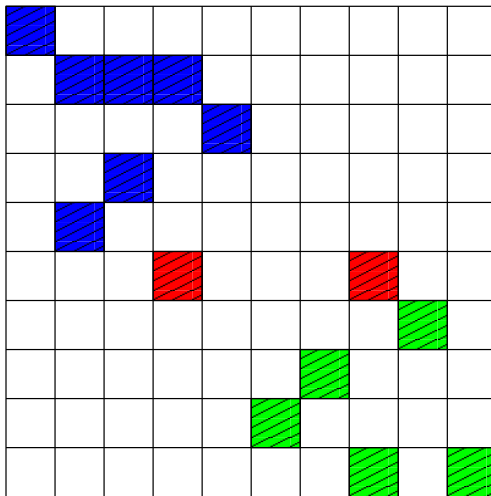
# Permuting to SBD form

## Mixed row detection



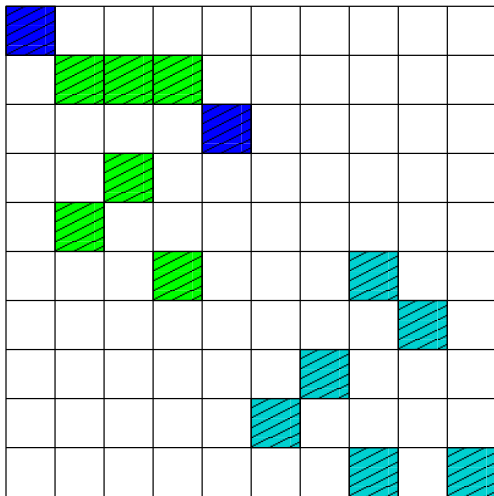
# Permuting to SBD form

Row permutation



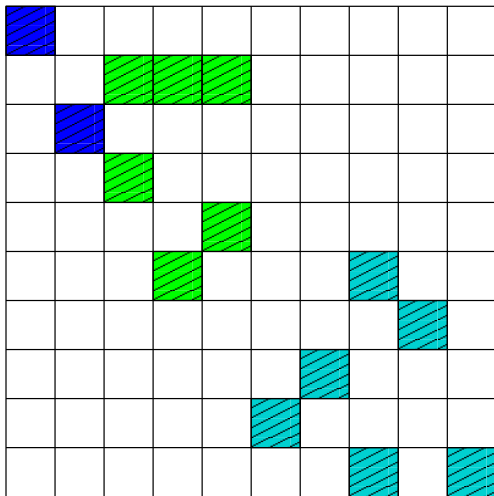
# Permuting to SBD form

## Column subpartitioning



# Permuting to SBD form

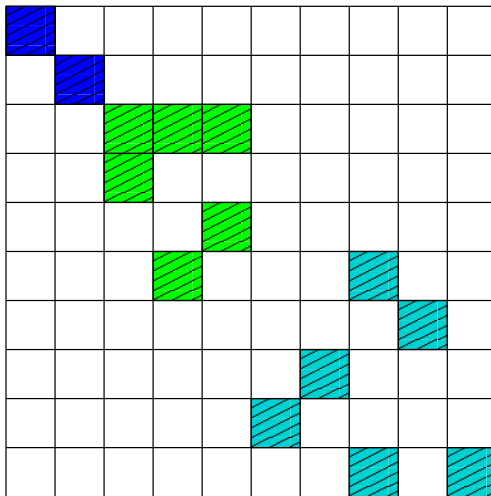
## Column permutation





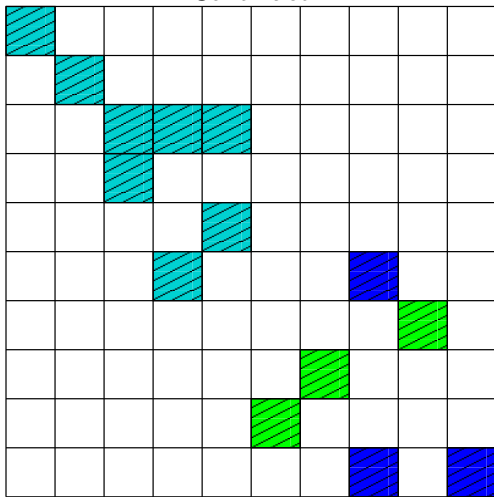
# Permuting to SBD form

No mixed rows - row permutation



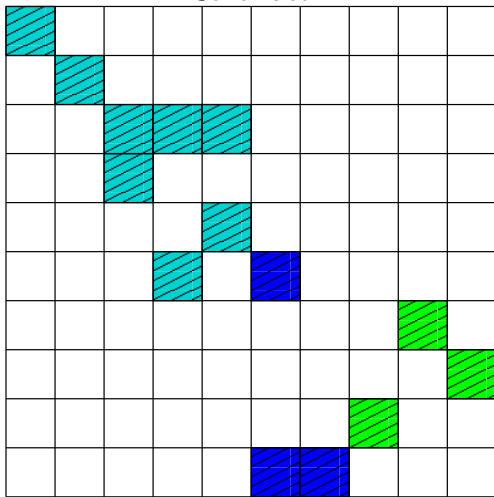
# Permuting to SBD form

Continued



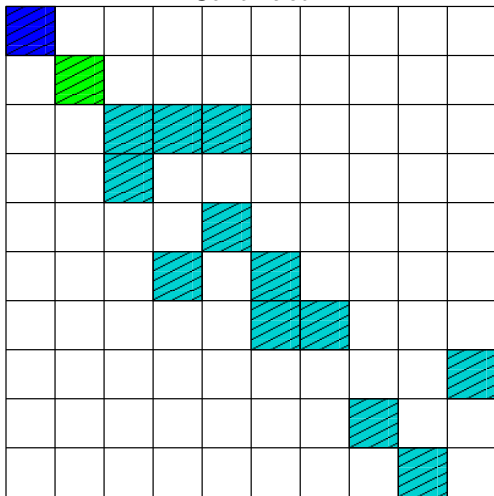
# Permuting to SBD form

Continued



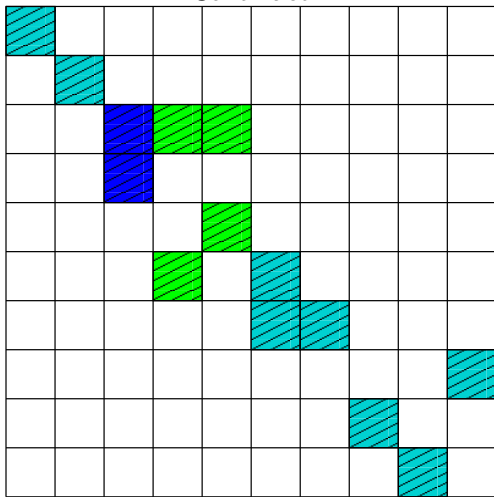
# Permuting to SBD form

Continued



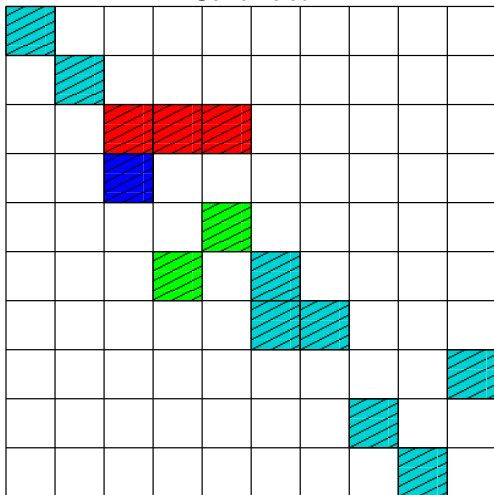
# Permuting to SBD form

Continued



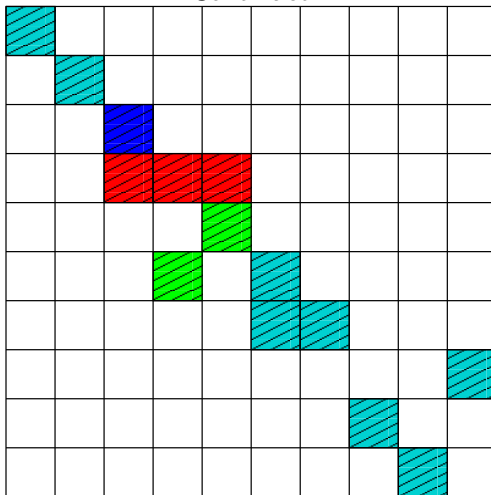
# Permuting to SBD form

Continued



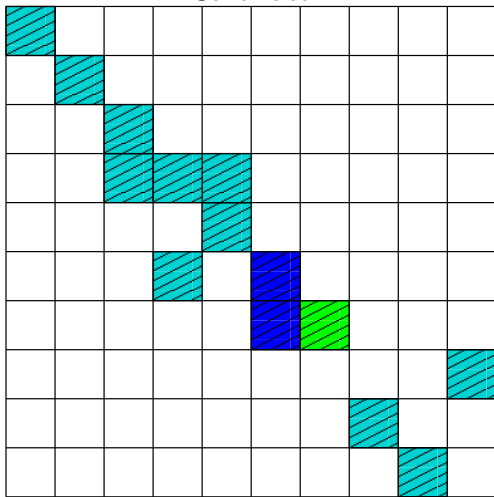
# Permuting to SBD form

Continued



# Permuting to SBD form

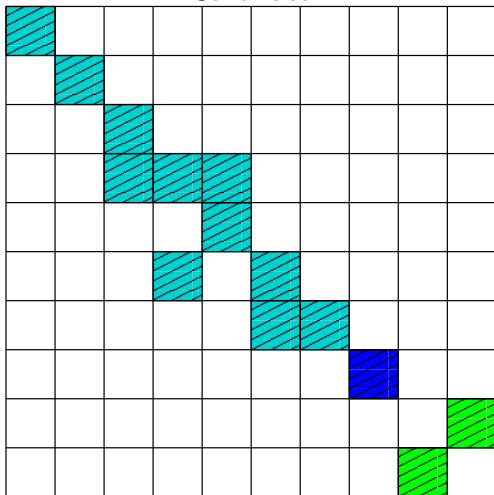
Continued





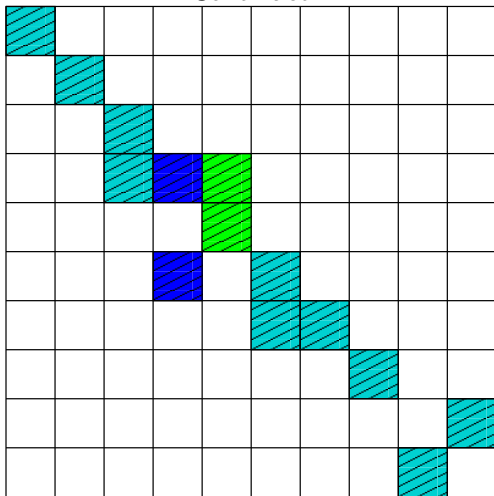
# Permuting to SBD form

Continued



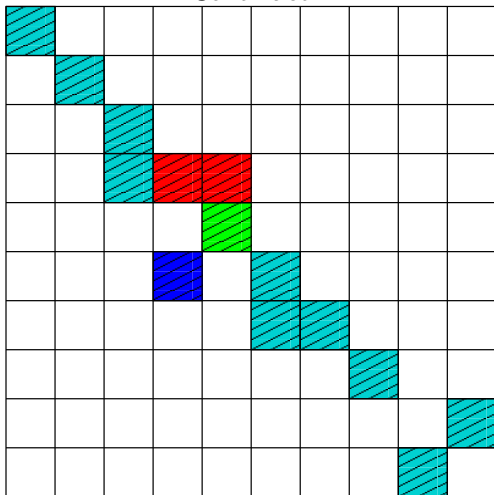
# Permuting to SBD form

Continued



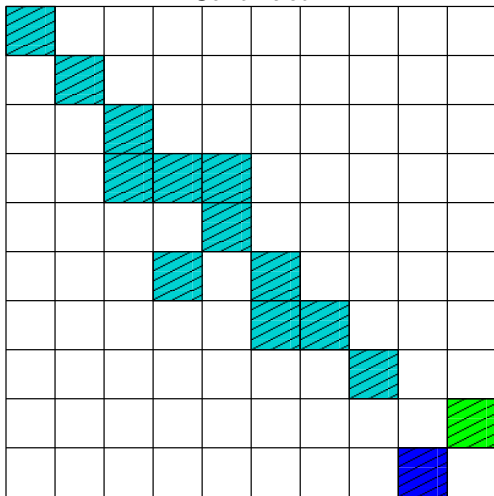
# Permuting to SBD form

Continued



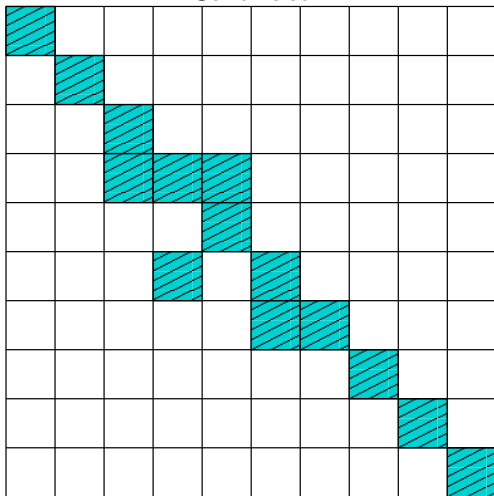
# Permuting to SBD form

Continued

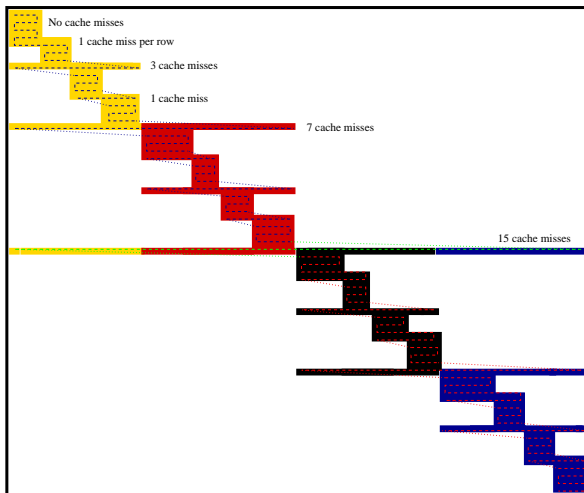


# Permuting to SBD form

Continued



Assuming a row-net hypergraph representation  $(\mathcal{V}, \mathcal{N})$ , we will have partitioned  $\mathcal{V}$  into subsets  $\mathcal{V}_0, \dots, \mathcal{V}_{p-1}$ . Nets spread over multiple parts correspond to mixed-row blocks incurring cache misses.



For  $p = \frac{n}{wL}$ , the number of cache misses is given by

$$\sum_{i: n_i \in \mathcal{N}} (\lambda_i - 1), \quad \text{assuming Zig-zag CRS.}$$

This is also called the  $\lambda - 1$  *metric*, and is already used extensively in parallel computing. Partitioners should also take into account the *load-imbalance*, typically denoted by  $\epsilon$ .

References:

- Çatalyürek and Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, 1999
- Vastenhouw and Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, 2005



For a truly cache-oblivious approach, the number of subsets  $p$  we partition  $\mathcal{V}$  into, must be as large as possible:

$$p \rightarrow \infty \quad (p = n).$$

Taking  $p$  higher than needed for the actual cache size does not harm efficiency, and will even optimise for the smaller caches closer to the CPU; *we optimise access for all cache levels irrespective of the cache parameters.*



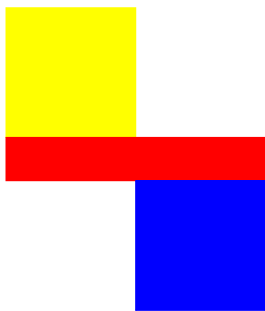


# Moving to two dimensions

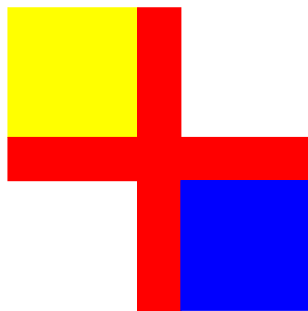
- 1 Memory and multiplication
- 2 Cache-friendly data structures
- 3 Cache-oblivious sparse matrix structure
- 4 Moving to two dimensions**
- 5 Experimental results



# Two-dimensional SBD



1D



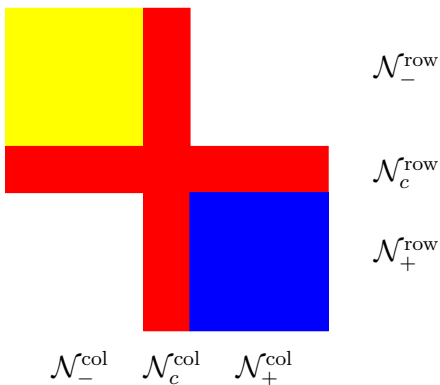
2D



## Two-dimensional SBD

Using a fine-grain model of the input sparse matrix, individual nonzeros each correspond to a vertex; each row and column have a corresponding net.

The quantity to minimise remains  $\sum_i(\lambda_i - 1)$ .



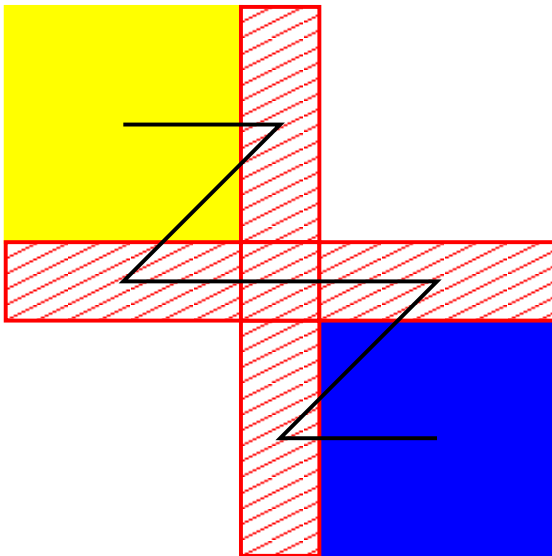
## Two-dimensional SBD



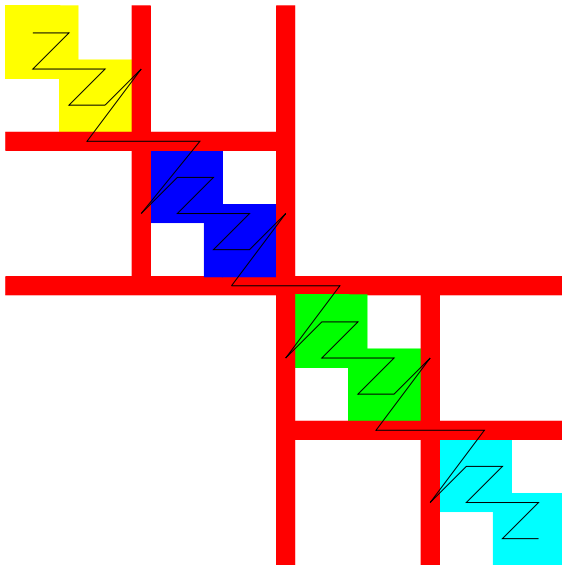
Zig-zag CRS is not suitable for handling 2D SBD



# Two-dimensional SBD

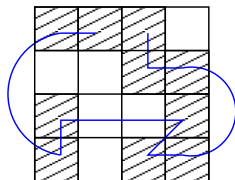


# Two-dimensional SBD



## Bi-directional Incremental CRS (BICRS)

$$A = \begin{pmatrix} 4 & 1 & 3 & 0 \\ 0 & 0 & 2 & 3 \\ 1 & 0 & 0 & 2 \\ 7 & 0 & 1 & 1 \end{pmatrix}$$



Stored as:

$$\begin{aligned} \text{nzs:} & \quad [3 \ 2 \ 3 \ 1 \ 1 \ 2 \ 1 \ 7 \ 4 \ 1] \\ \text{col\_increment:} & \quad [2 \ 4 \ 1 \ 4 \ -1 \ 5 \ -3 \ 4 \ 4 \ 1] \ , \\ \text{row\_increment:} & \quad [0 \ 1 \ 2 \ -1 \ 1 \ -3] \end{aligned}$$

$2n_{nz} + (\text{row\_jumps} + 1)$  accesses



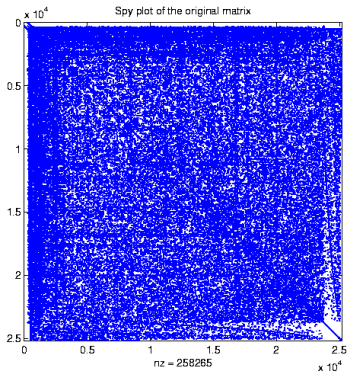
# Experimental results

- 1 Memory and multiplication
- 2 Cache-friendly data structures
- 3 Cache-oblivious sparse matrix structure
- 4 Moving to two dimensions
- 5 Experimental results

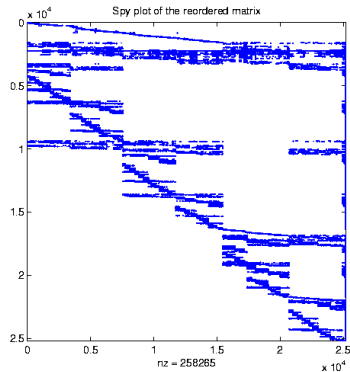




# The rhipientium matrix



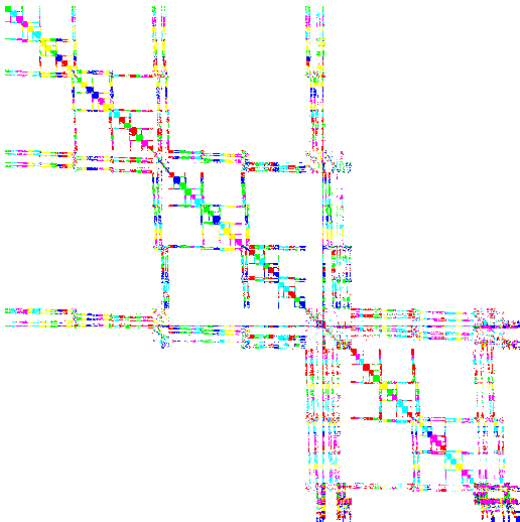
$\rho = 1$  (original)



$\rho = 100, \quad \epsilon = 0.1$



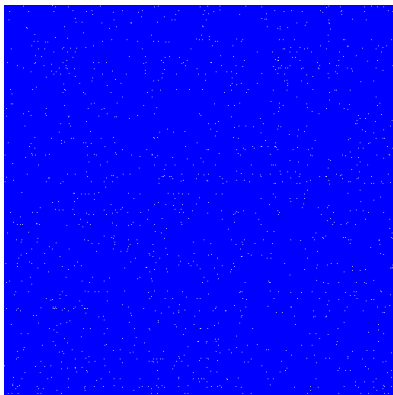
# The rhentium matrix – 2D SBD



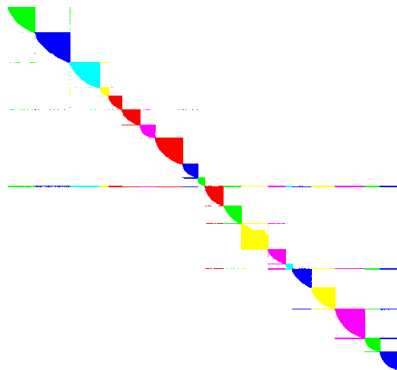
$$p = 100, \quad \epsilon = 0.1$$



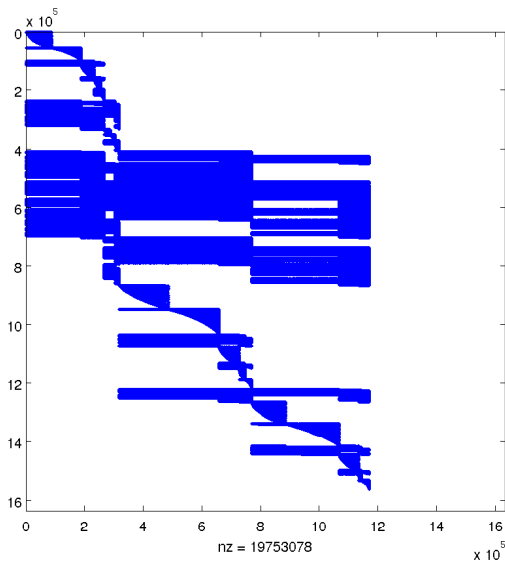
# The Stanford link matrix

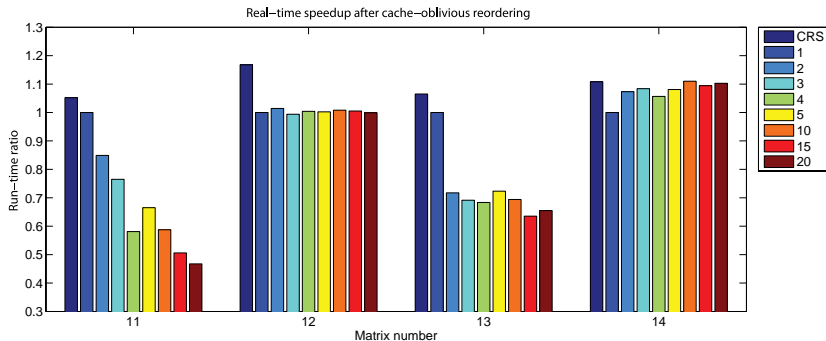


Original

Finegrain (hybrid,  $p = 20$ ,  $\epsilon = 0.1$ )

# The wikipedia matrix (May 11, 2005)





Matrices used are: 11. Stanford, 12. Stanford\_Berkeley, 13. wikipedia20051105, and 14. cage14.



# Pre-processing times

## 1D

Matrix	Reordering time	SpMV time (old/new)
rhpentium, $p = 400$ :	1 minute	(0.9 / 0.7 ms.)
memplus, $p = 100$ :	4 minutes	(0.4 / 0.4 ms.)
stanford, $p = 20$ :	4 minutes	(27 / 14 ms.)
cage14, $p = 20$ :	12 minutes	(109 / 123 ms.)
stanford_berkeley, $p = 20$ :	13 minutes	(31 / 30 ms.)
wikipedia, $p = 10$ :	16 minutes	(353 / 236 ms.)
wikipedia, $p = 20$ :	45 minutes	(353 / 237 ms.)

Results from 2009, run on an Intel Q6600



# Pre-processing times

2D, (*preliminary*)

Matrix	Reordering time	SpMV ratio (new/old)
rhpentium, $p = 2$ :	< 1 minute	1.01 (ICRS/ICRS)
rhpentium, $p = 2$ :	< 1 minute	0.88 (BICRS/BICRS)
stanford, $p = 20$ :	8 minutes	0.44 (BICRS/ICRS)
wikipedia, $p = 20$ :	17 minutes	0.83 (BICRS/BICRS)

*Old*: SpMV on the original matrix  $A$

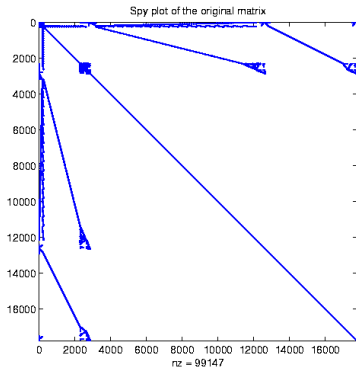
*New*: SpMV on the reordered matrix  $PAQ$

Reordering done on Huygens (IBM Power6), SpMVs on an Intel Atom

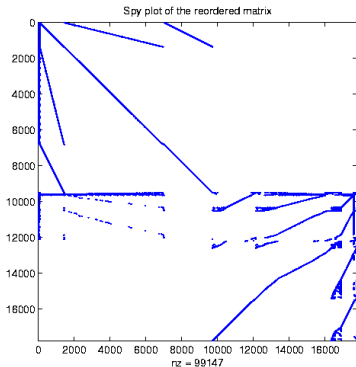
*BICRS nonzero ordering is only partially implemented*



# The memplus matrix



$p = 1$  (original)

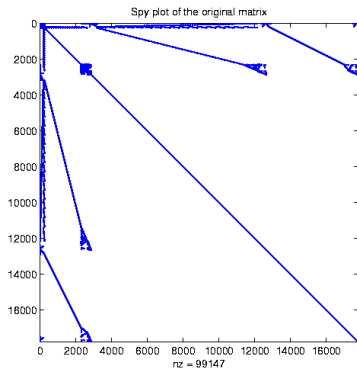


$p = 2, \epsilon = 0.1$

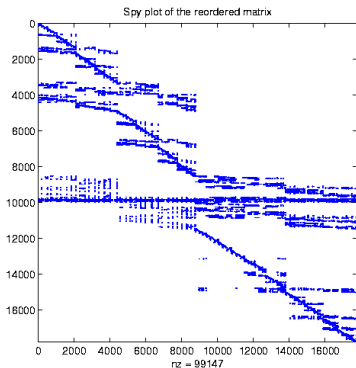




# The memplus matrix



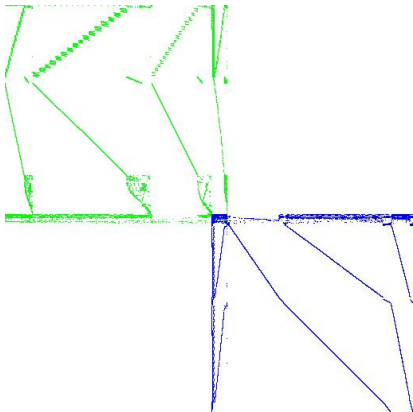
$p = 1$  (original)



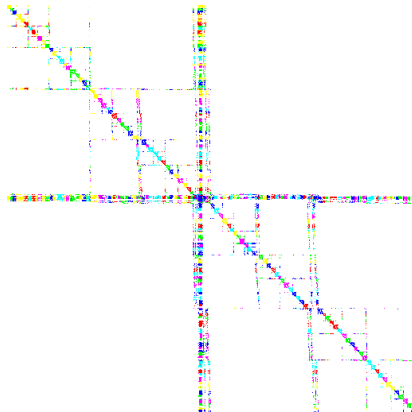
$p = 100, \quad \epsilon = 0.1$



# The memplus matrix – 2D SBD



$p = 2$



$p = 100$



## Conclusions:

we have introduced a scheme capable of increasing SpMV performance up to a factor two, while:

- remaining cache-oblivious, and
- keeping open the option of using specialised sparse BLAS libraries (1D).

For already well-structured matrices our approach does not obtain significant speedups, but does not decrease performance much either.

The two-dimensional method promises even greater speedups, more than doubling SpMV speed on the Stanford link matrix in its current preliminary version.

