Introduction to Parallel and Distributed Computing

Selected Parallel Algorithms
Sparse Matrix-Vector Multiplication

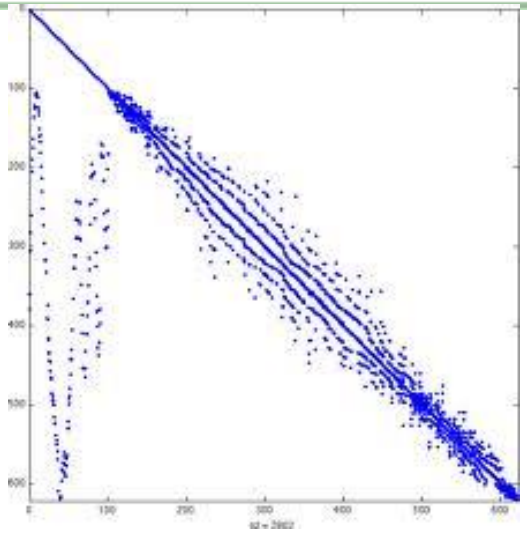
Lecture 15, Spring 2024

Instructor: 罗国杰

gluo@pku.edu.cn

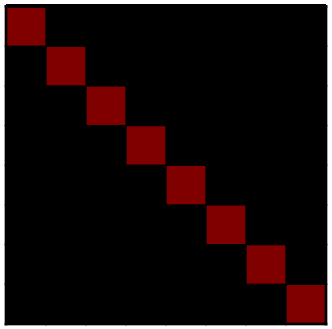
adapted from J. Demmel et al., “Applications of Parallel Computers,” UC Berkeley CS267 S18/S22.

What is a sparse matrix?

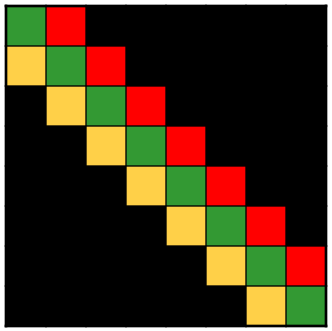


- **A matrix with primarily zeros (>> 90%)**
- **Representing them using dense data structures wastes**
  - Memory
  - Computation
- **Sparse matrices arise in many applications**
  - Simulating climate
  - Analyzing images (photos, MRIs,…)
  - Web page ranking for search
  - Graphs, including Graph Neural Nets

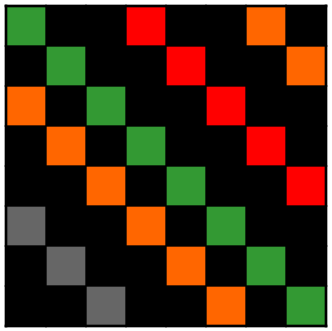
Examples of sparse matrices



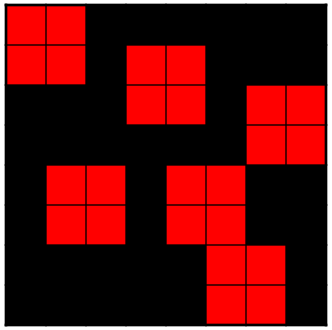
Diagonal



Tridiagonal

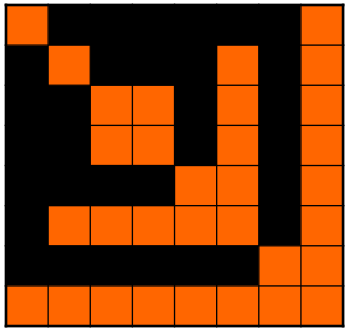


“Generalized diagonal”

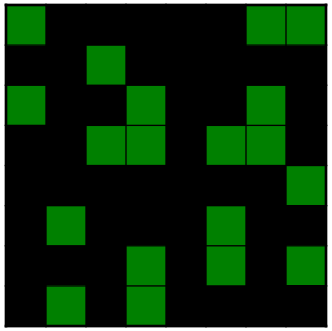


Block matrix

(2x2 dense blocks)



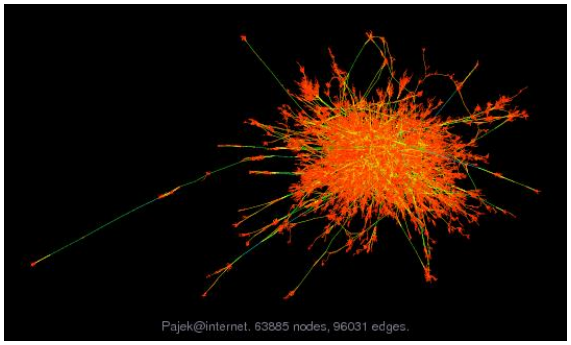
Symmetric

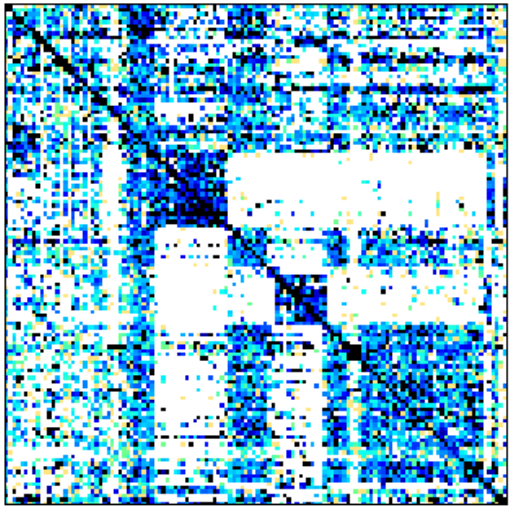


Irregular

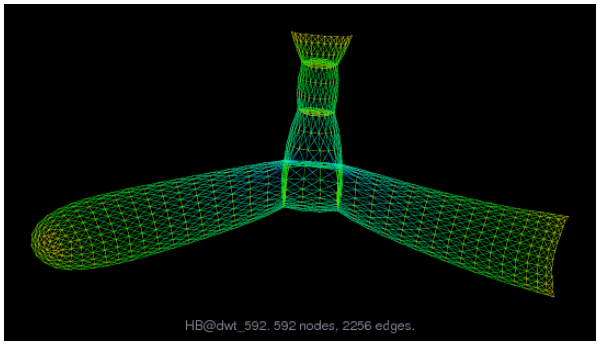
Sparse Matrices are Everywhere

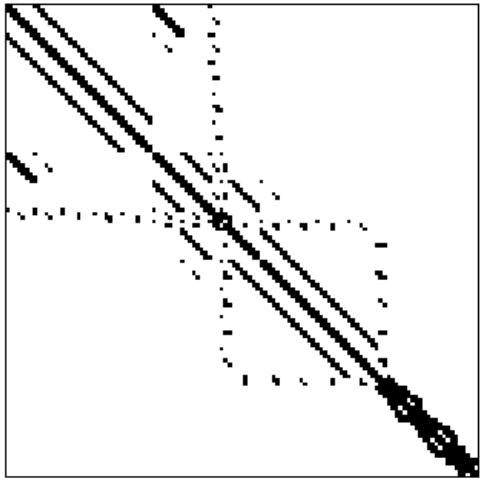
Internet connectivity



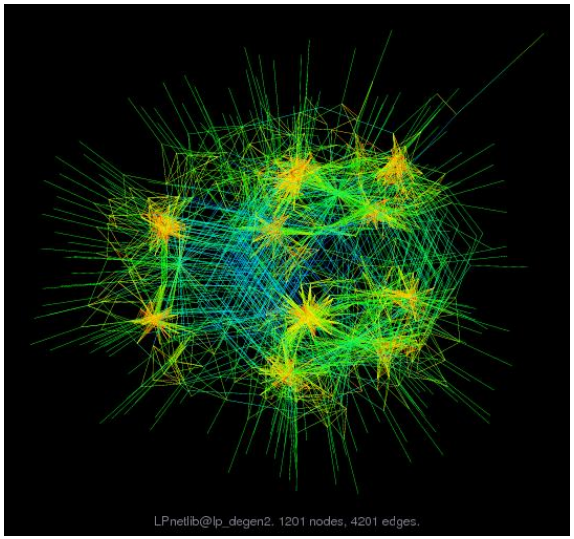


Structural design





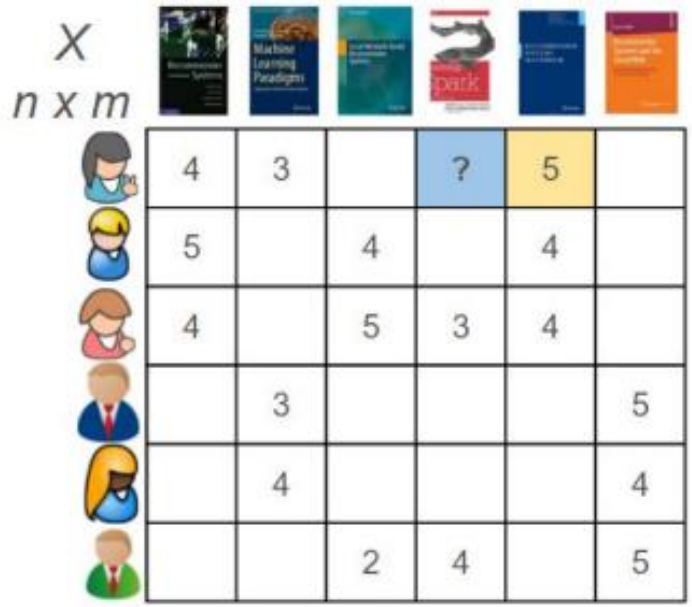
Linear Programming



Recommendation Matrix

Products

Population



Ratings

Image: Oliver Gindele, Data Scientist at Datatonic

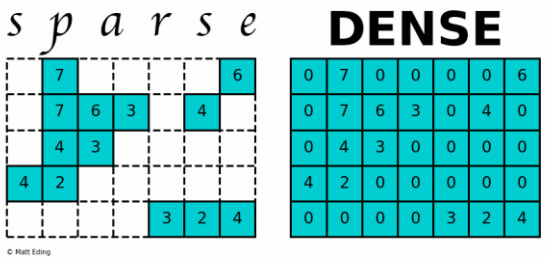
More applications

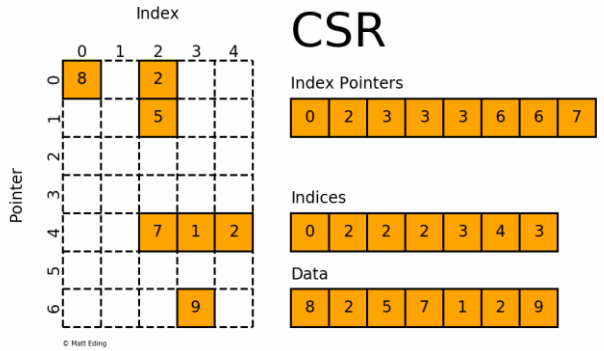
- **Graphs**
  - Google's PageRank (originally an eigen-problem on the web adjacency matrix)
  - Transportation network analysis
- **Text analysis**
  - Latent Semantic Indexing finds topics in a document corpus by doing a singular value decomposition of a bag-of-words matrix
- **Scientific and engineering**
  - Solving differential equations (climate modeling, etc.)
  - Optimization problems
- **And many more…**

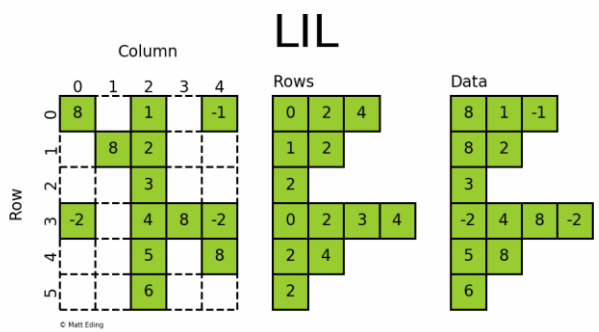
Outline

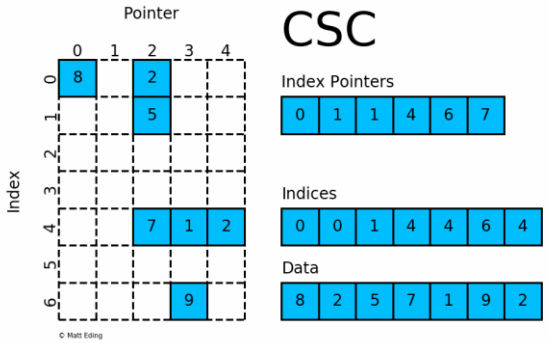
- Sparse matrices in the world
- Sparse matrix formats and serial SpMV
- Parallel and distributed SpMV
- Register / cache blocking and autotuning SpMV
- CA iterative solvers
- Sparse matmul (SpGEMM, SPMM, ...)

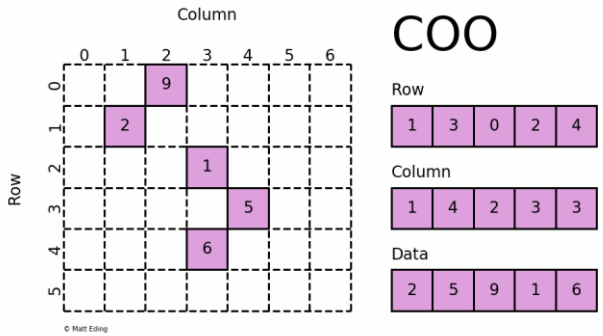
Sparse Matrix Formats: Examples

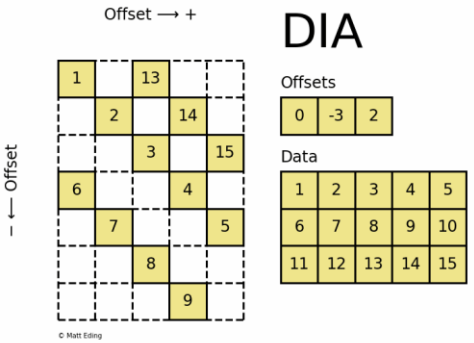








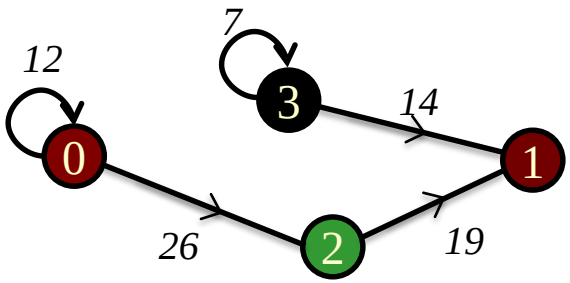




https://matteding.github.io/2019/04/25/sparse-matrices/

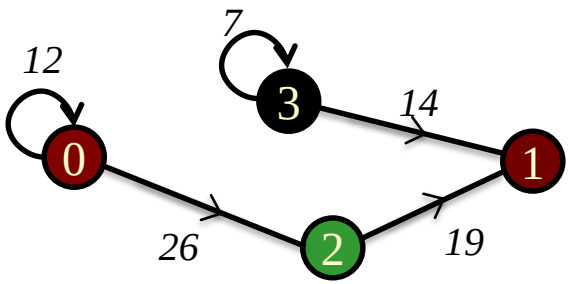
Graph representations

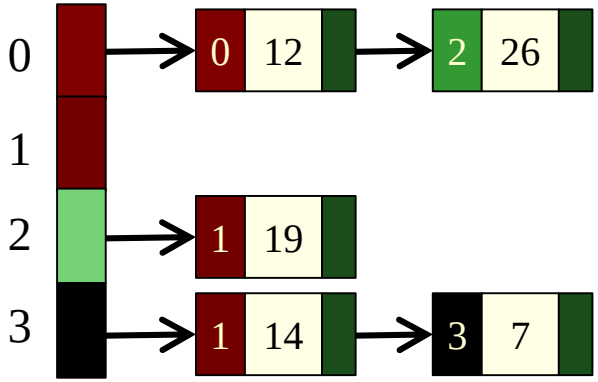
Compressed sparse row (CSR) = cache-efficient adjacency lists



Graph representations

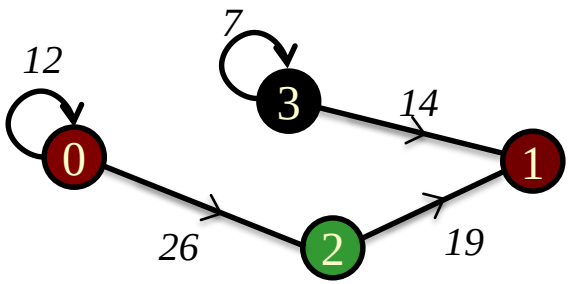
Compressed sparse row (CSR) = cache-efficient adjacency lists

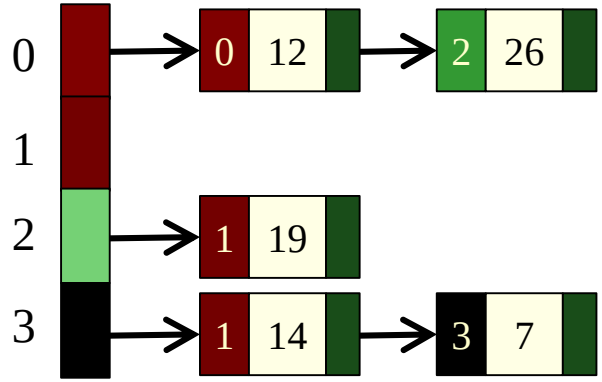


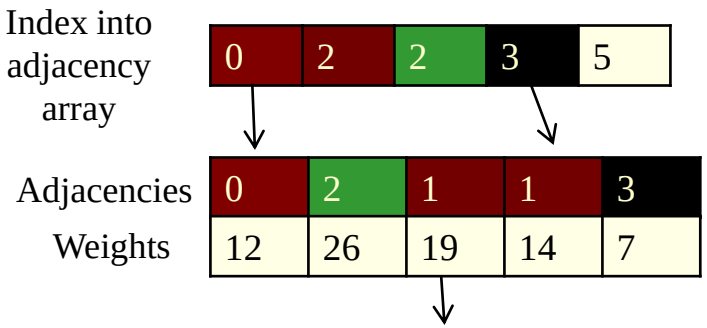


Graph representations

Compressed sparse row (CSR) = cache-efficient adjacency lists







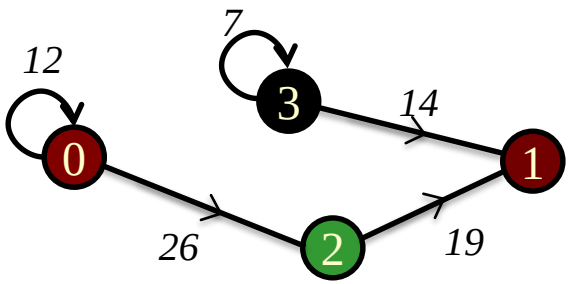
(row starts in CSR)

(column ids in CSR)

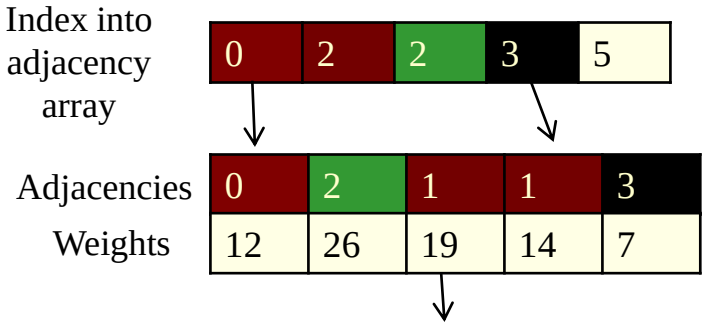
(numerical values in CSR)

Graph representations

Compressed sparse row (CSR) = compressed version of dense adjacency matrix



	0	1	2	3
0	12	0	26	0
1	0	0	0	0
2	0	19	0	0
3	0	14	0	7



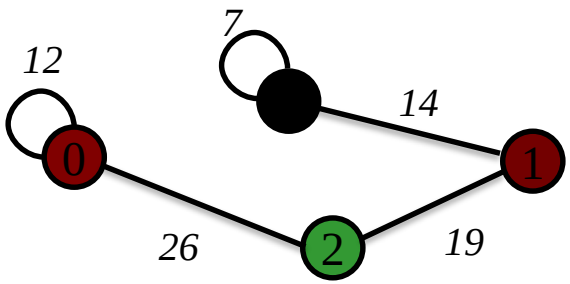
(row starts in CSR)

(column ids in CSR)

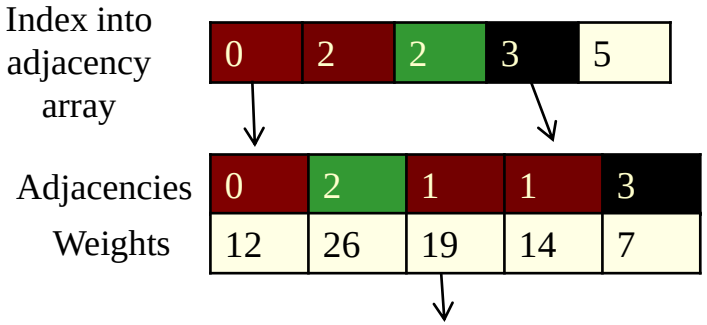
(numerical values in CSR)

Directed vs Undirected

An undirected graph corresponds to a symmetric matrix – need only store ~half (triangle)



	0	1	2	3
0	12	0	26	0
1	0	0	19	14
2	26	19	0	0
3	0	14	0	7

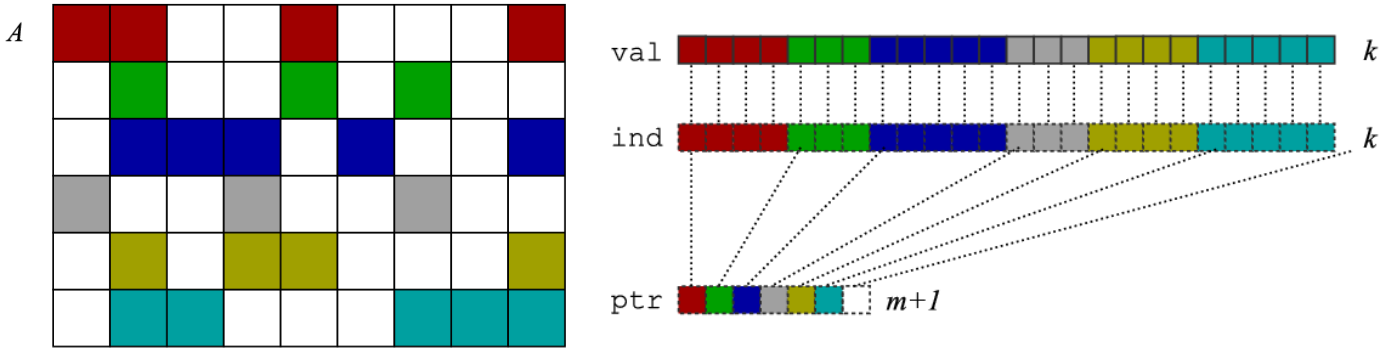


(row starts in CSR)

(column ids in CSR)

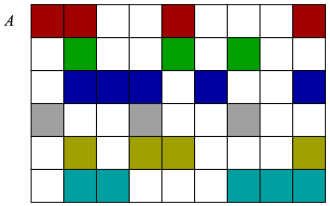
(numerical values in CSR)

Compressed Sparse Row (CSR) Storage



- **CSR has:**
  - Size nnz = number of nonzeros
  - Array of the nonzero values (val) of size nnz
  - Array of the column indices for each value of size nnz
  - Array of row start pointers of size n = number of rows

Other Storage Formats



- **Compressed Sparse Row (CSR) is most common**

Others include

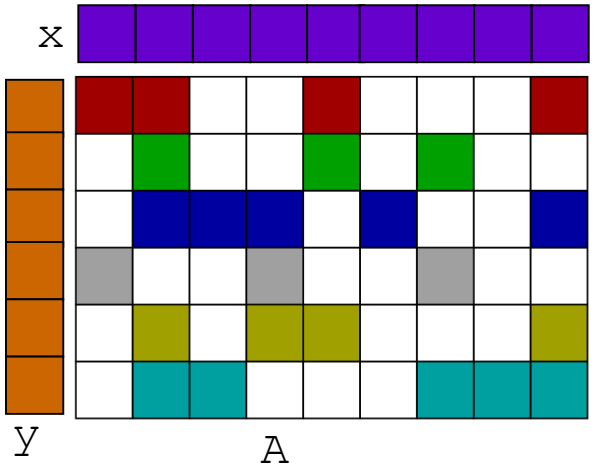
- **Compressed Sparse Column (CSC)**
- **Coordinate (COO): row + column index per nonzero (easy to build / modify)**
- **Diagonal (DIAG): store main diagonal as 1D array; or diagonal bands as 2D (padded)**
- **Symmetric: store only ½ the array (indexing more complicated)**
- **Blocked: store each block contiguously**
  - Register blocked: blocks a small and dense, avoid indexes within blocks
  - Cache blocked: blocks are large and themselves sparse

And many more specialized ones!

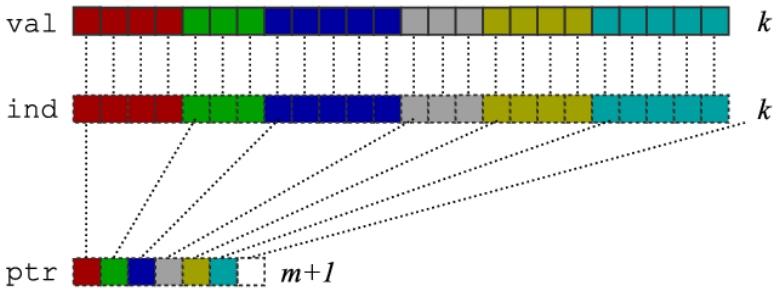
Sparse Matrix-Vector Multiply (SpMV)

- **Sparse matrix-(dense) vector multiplication (SpMV) used in:**
  - Solving linear systems
  - Eigenvalue problems
  - Optimization algorithms
  - Machine learning, etc.
- **Sparse matrix, sparse vector (SpMSpV)**
  - e.g., graph algorithms: breadth-first search, bipartite graph matching, and maximal independent sets
- **Sparse matrix, sparse matrix (SpGEMM)**
  - e.g., graph algorithms
  - Common special case: $A * A^T$
- **Sparse matrix, dense matrix (SpDM3)**
  - Machine learning

SpMV in CSR: Serial



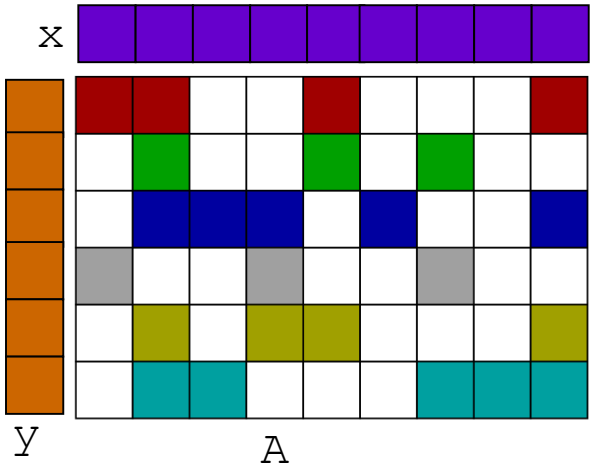
Representation of A



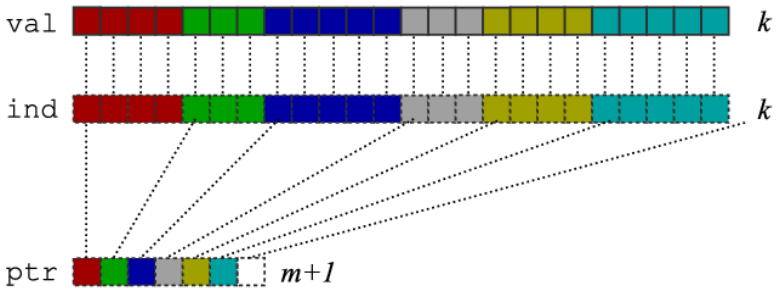
SpMV: Sparse Matrix-Vector Multiplication

CSR: Compressed Sparse Row

SpMV in CSR: Serial

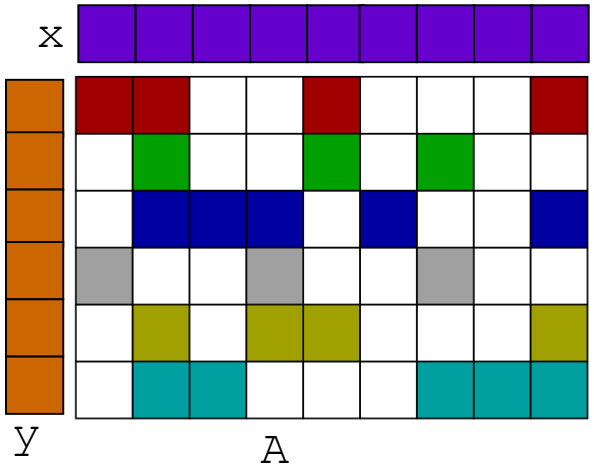


Representation of A

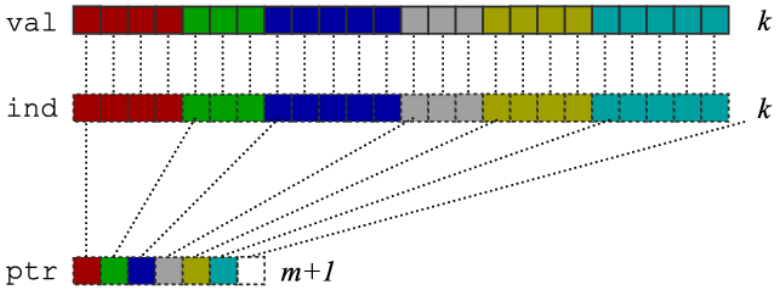


Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j)*x(j)$

SpMV in CSR: Serial



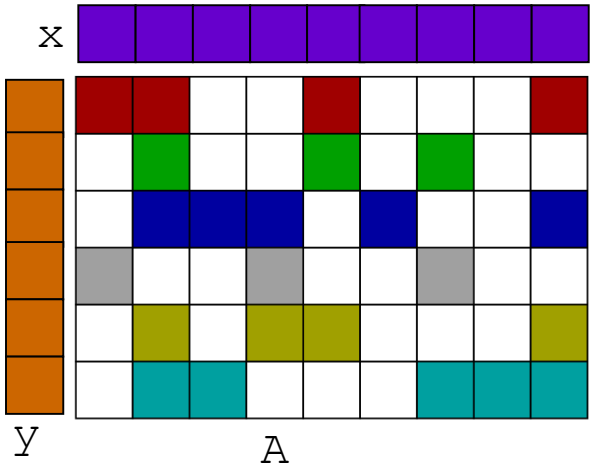
Representation of A



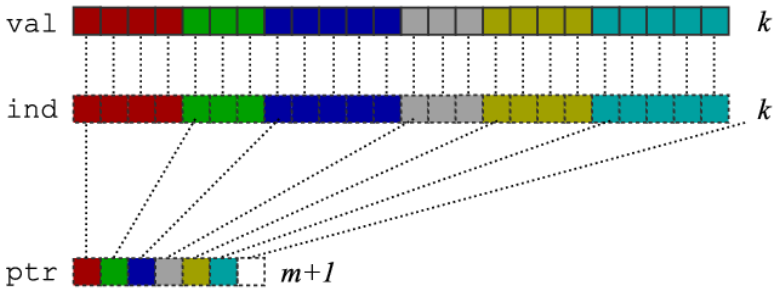
Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j)*x(j)$

```
for each row i
  for k=ptr[i] to ptr[i+1] do
    y[i] = y[i] + val[k] * x[ind[k]]
```

SpMV in CSR: Serial



Representation of A



Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j)*x(j)$

```
for each row i
  for k=ptr[i] to ptr[i+1] do
    y[i] = y[i] + val[k] * x[ind[k]]
```

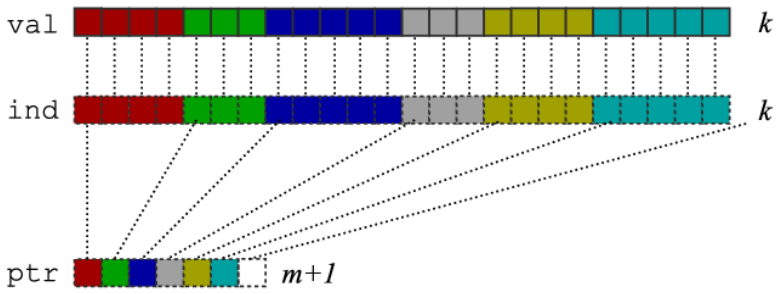
- BLAS2 not BLAS3, so no reuse in A
- Maximum reuse of y (full row) as written
- Re-use of x?

Outline for today

- Sparse matrices in the world
- Sparse matrix formats and serial SpMV
- Parallel and distributed SpMV
- Register / cache blocking and autotuning SpMV
- CA iterative solvers
- Sparse matmul (SpGEMM, SPMM,...)

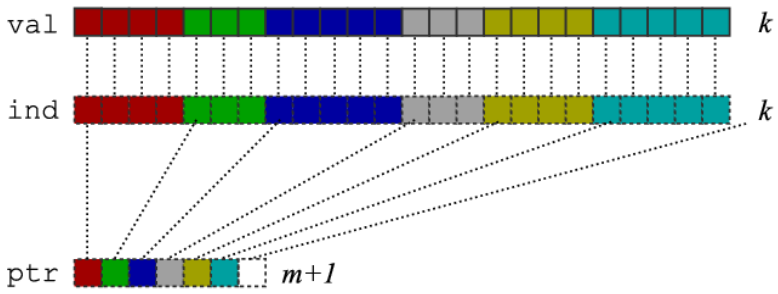
SpMV in CSR: OpenMP Parallel

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private(j, i, tmp) schedule(static)
    for (int i=0; i<m; i++)  {
        for (j = ptr[i]; j < ptr[i+1]; j++)  {
            tmp = ind[j];
            y[i] += val[j] * x[tmp];
        }
    }
}
```



SpMV in CSR: OpenMP Parallel

```
#pragma omp parallel num_threads(thread_num)
{
#pragma omp for private(j, i, tmp) schedule(dynamic)
    for (int i=0; i<m; i++)  {
        for (j = ptr[i]; j < ptr[i+1]; j++)  {
            tmp = ind[j];
            y[i] += val[j] * x[tmp];
        }
    }
}
```



SpMV for 8-wide SIMD

```
void avx2_csr_spmv( float *A, int32_t *nIdx, int32_t **indices, float
*x, int32_t m, float *y) {
  int32_t A_offset = 0;
  for(int32_t i = 0; i < m; i++) {
      int32_t nElem = nIdx[i]; float t = 0.0f;
      __m256 vT =
_mm256_set_ps(0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f);
      int32_t smLen = nElem - (nElem & 7);
      for(int32_t j = 0; j < smLen; j+=8) {
      __m256i vIdx = _mm256_loadu_si256((__m256i*)&(indices[i][j]));
      __m256 vX = _mm256_i32gather_ps((float const*)x,vIdx,4);
      __m256 vA = _mm256_loadu_ps(&A[A_offset + j]);
      vT = _mm256_add_ps(vT, _mm256_mul_ps(vX,vA));
    }
    t += sum8(vT);
    for(int32_t j = smLen; j < nElem; j++) {
      int32_t idx = indices[i][j];
      t += x[idx]*A[A_offset + j];
    }
    y[i] = t;
    A_offset += nElem;
  }
}
```

For each row

For each 8 nonzeros

Gather X, load A
multiply, add to Y

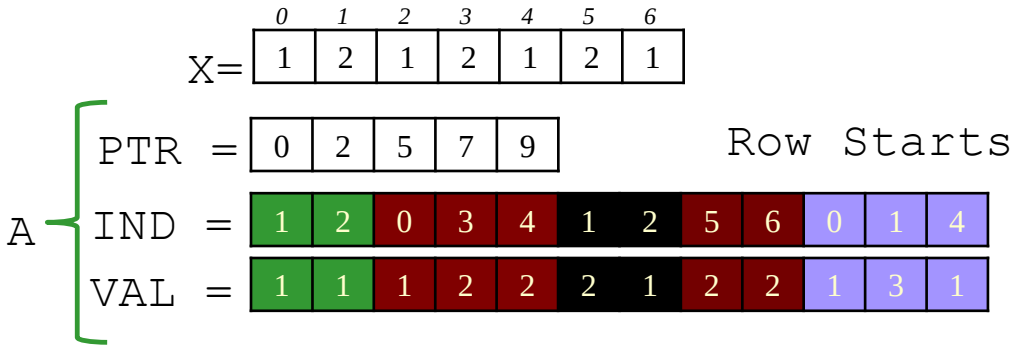
In case # cols
does not divide
the vector width

Code courtesy of David Sheffield
(TA @ Berkeley CS267 S21)

SpMV with CSR in CUDA

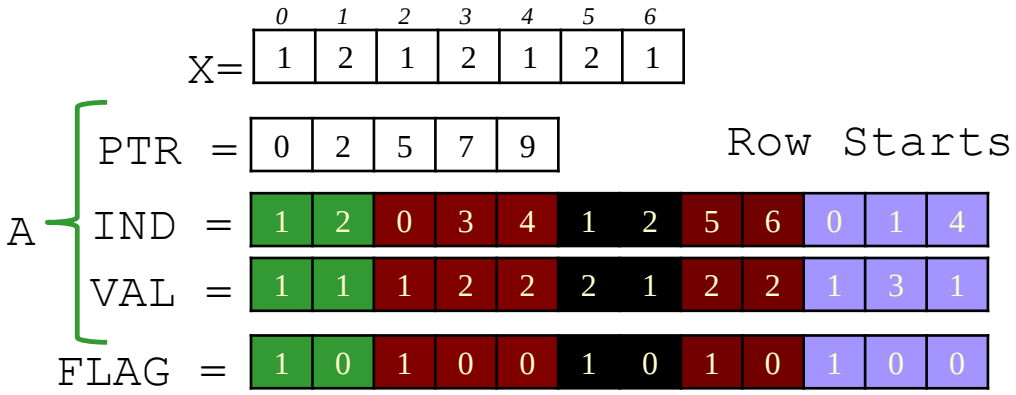
```
// Parallel SpMV using CSR format
__global__ void spmv(int *ptr, int *ind, float *val,
                     int m, float *x, float *y) {

    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
         i < m; i += blockDim.x * gridDim.x) {
        float yi = 0;
        for (int j = ptr[i]; j < ptr[i+1]; j++) {
            yi += values[j] * x[col_ind[j]];
        }
        y[i] = yi;
    }
}
```

SpMV (Segmented Suffix Scan)



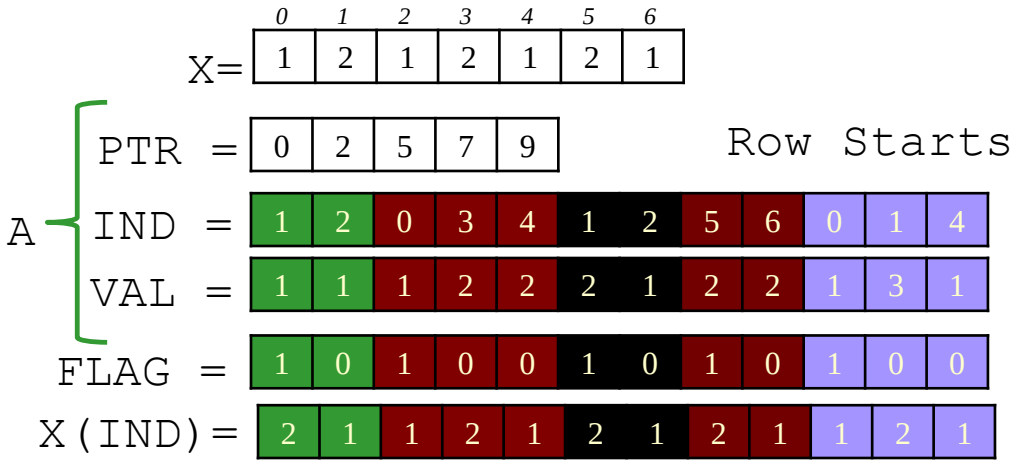
Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

SpMV (Segmented Suffix Scan)



FLAG = ZEROS + ONES(PTR)

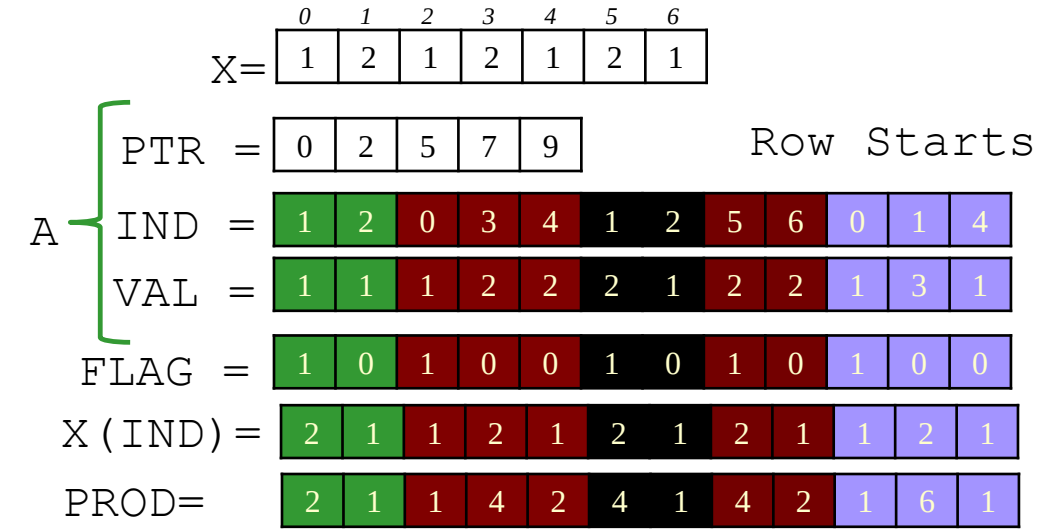
SpMV (Segmented Suffix Scan)




```
FLAG = ZEROS + ONES(PTR)
PROD = VAL * X(IND)
```

Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

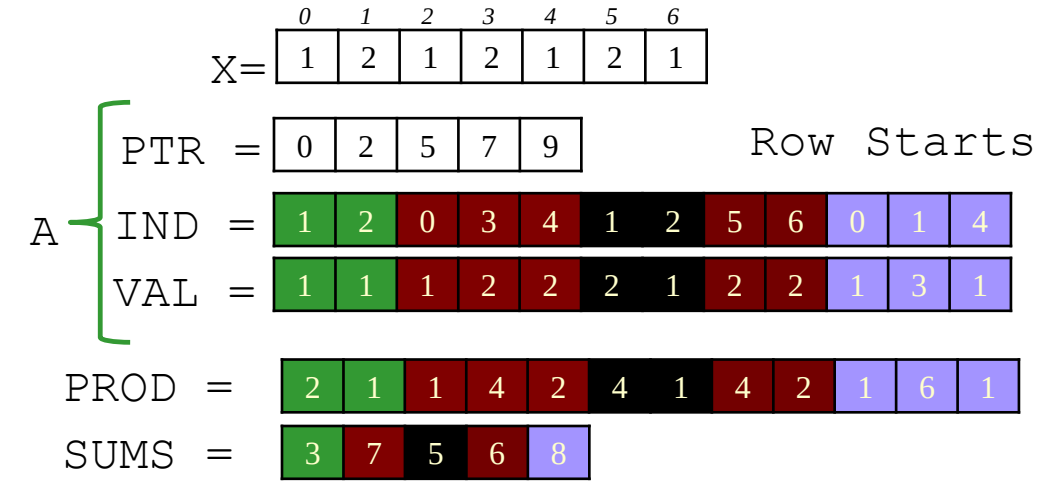
SpMV (Segmented Suffix Scan)




```
FLAG = ZEROS + ONES(PTR)
PROD = VAL * X(IND)
```

Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

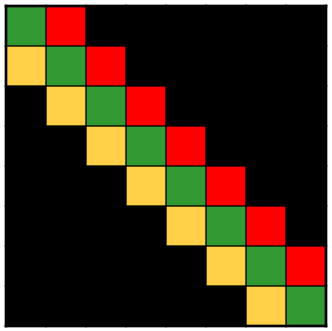
SpMV (Segmented Suffix Scan)

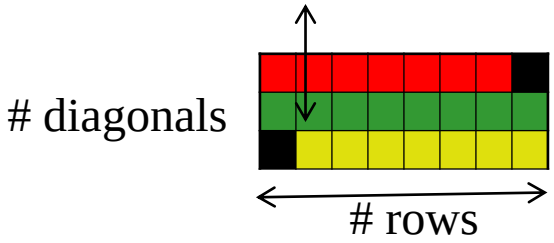



```
FLAG = ZEROS + ONES(PTR)
PROD = VAL * X(IND)
SUMS = SUM_SCAN(PROD,SEGMENT=SEGS
                     PREFIX(FLAG))
Y = SUMS(PTR)
```

Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors . Guy E. Blelloch, Michael A. Heroux, and Marco Zagha. CMU-CS-93-173

SpMV Diagonal format in OpenMP





+1
0
-1

columnoffset

val

Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j)*x(j)$

```
for each diagonal k do
  #pragma omp parallel for
  for each row i do
    column = i + columnoffset[k]
    if (column >= 0 && column < n)
      y[i] = y[i] + val[k][column] * x[column]
```

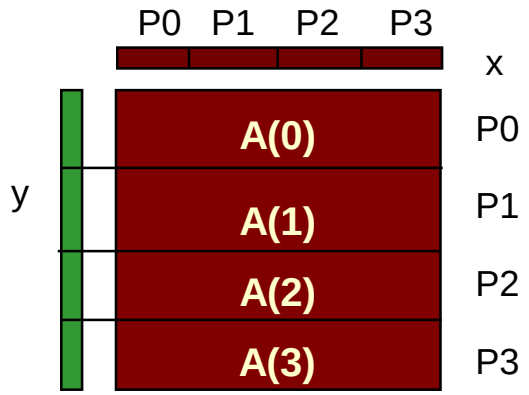
Distributed Dense Matrix-Vector Product (Row Major)

- **Compute y = y + A*x, where A is a dense matrix**

Warmup on dense case

- **Layout:** 1D row blocked
- **Algorithm:**

```
foreach processor p
  Broadcast x[p] chunk owned by p
for all local i
  for all j
    Compute y[i] += A[i,j]*x[j] locally
```

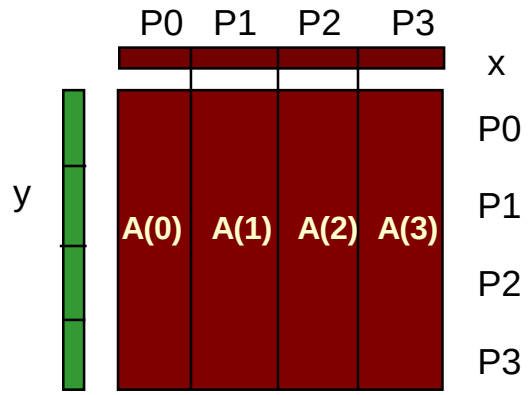


Broadcast + local dot product

Distributed Dense Matrix-Vector Product (Column Major)

- **Compute y = y + A*x, where A is a dense matrix**
- **Layout: 1D column blocked**
- **Algorithm:**

```
foreach processor p
  make a temp vector of size n
  for all local j
    for all i
      Compute temp[i] = A[i,j]*x[j] locally
 Reduce across all rows
  y(i) += SumReduce (temp(i))
```

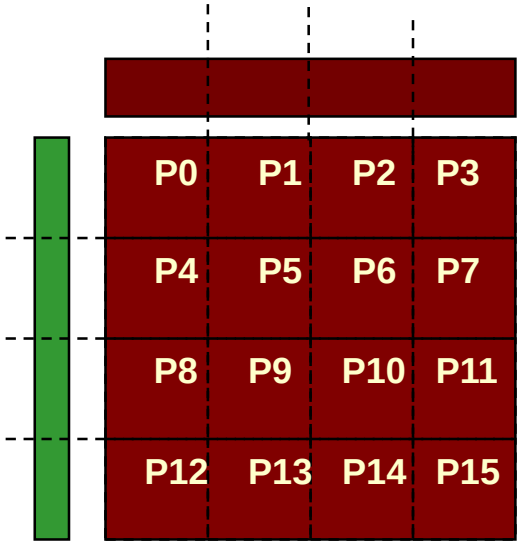


Local daxpy and parallel reduction

Distributed Dense Matrix-Vector Product (2D Blocked)

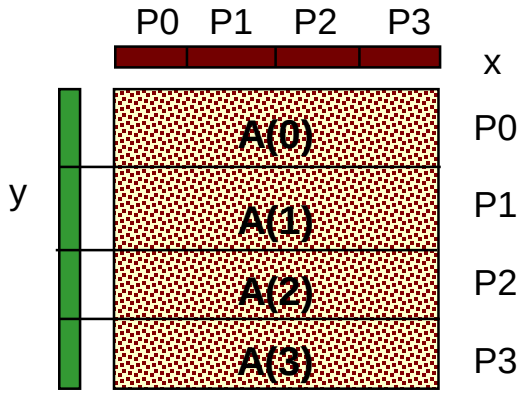
- **A 2D blocked layout uses a broadcast and reduction**
- **Both on a subset of processors**
  - sqrt(p) for square processor grid
  - Can use other rectangular shapes

    p_row * p_col = p



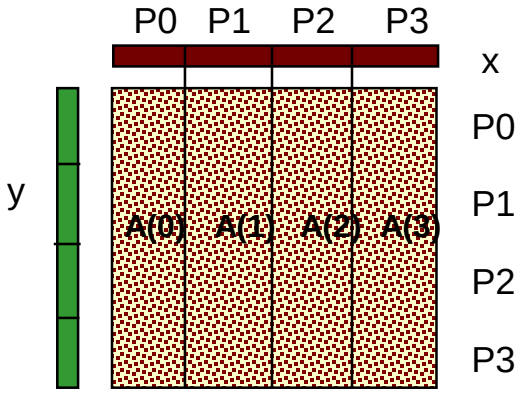
Distributed SpMV

1. **Row parallelism (y & A partitioned)**
   - Replicate x across processors
   - Or exchange only necessary elements
   - Are nonzeros clustered, e.g., near diagonal?



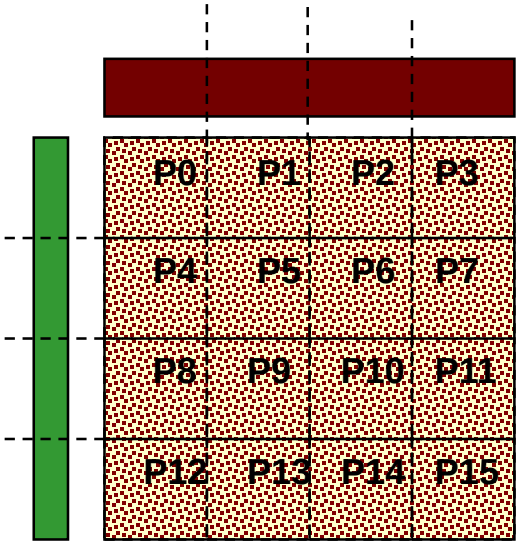
Distributed SpMV

1. **Row parallelism (y & A partitioned)**
   - Replicate x across processors
   - Or exchange only necessary elements
   - Are nonzeros clustered, e.g., near diagonal?
2. **Column parallelism (x & A partitioned)**
   - Make temporary temp_y = [0,…] on all processors;
   - Update that; and (sparse?) sum-reduce over processors



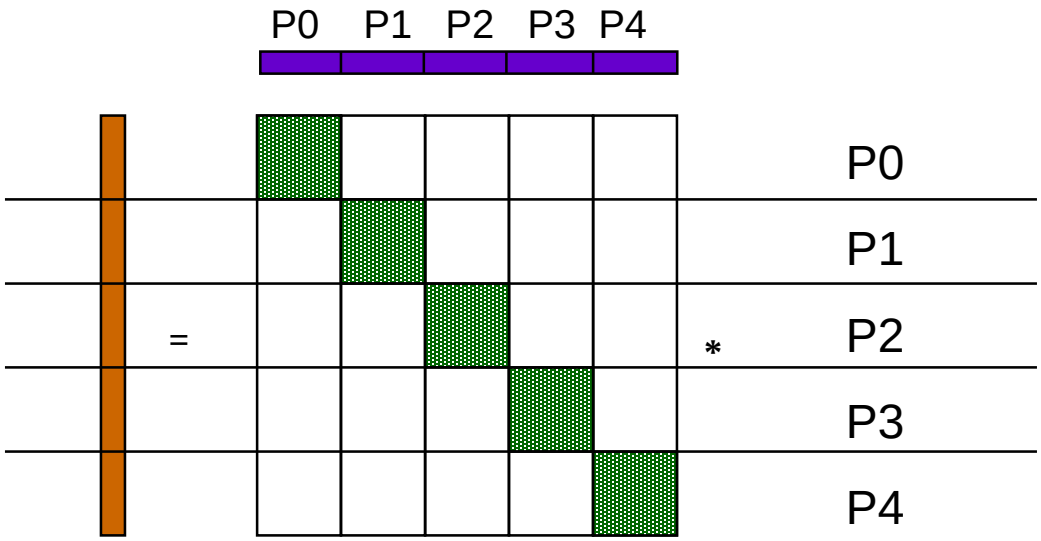
Distributed SpMV

1. **Row parallelism (y & A partitioned)**
   - Replicate x across processors
   - Or exchange only necessary elements
   - Are nonzeros clustered, e.g., near diagonal?
2. **Column parallelism (x & A partitioned)**
   - Make temporary temp_y = [0,…] on all processors;
   - Update that; and (sparse?) sum-reduce over processors
3. **2D parallelism for large p and when nonzeros are uniform**
   - Divide processors into p1 x p2 (e.g., square grid)
   - Hybrid of Row and Column parallelism using teams
   - NAS CG benchmark does this (random nonzero pattern)
   - Bad load balance for clustered nonzeros



Ideal Sparse Structure: P diagonal Blocks

- **"Ideal" matrix structure for parallelism: block diagonal**
  - If p_i holds x_i and y_i, blocks, no vectors to communicate
  - If no non-zeros outside these blocks, no communication needed



Matrix Reordering via Graph Partitioning

- **"Ideal" matrix structure for parallelism: block diagonal**
- **Can we reorder the rows/columns to get close to this?**
  - Most nonzeros in diagonal blocks, few outside

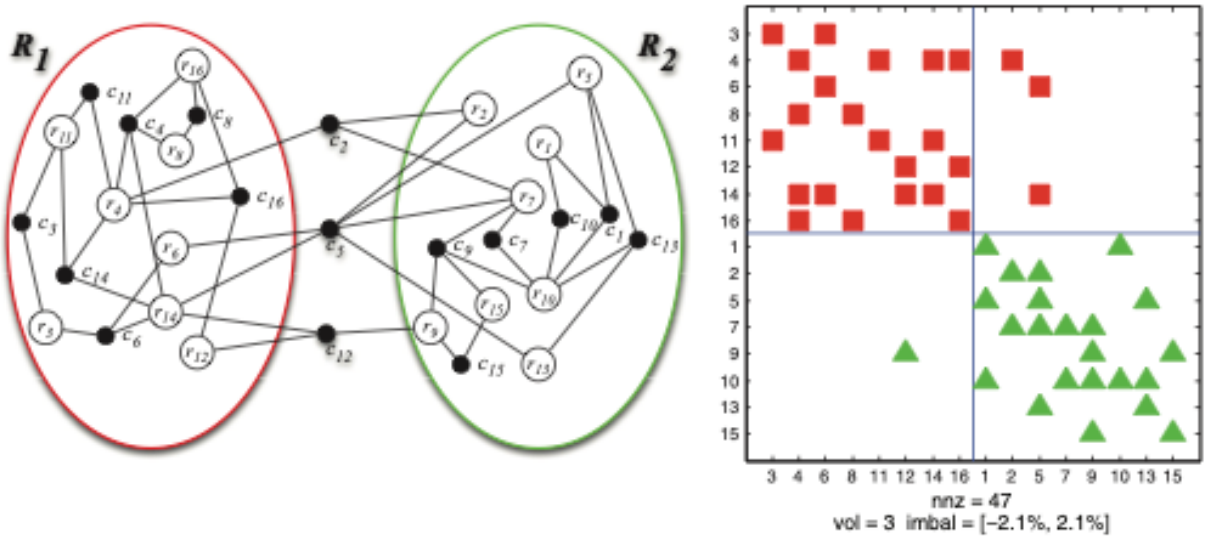


image source: https://faculty.cc.gatech.edu/~umit/projects/hypergraph/

More Matrix Reordering Algorithms

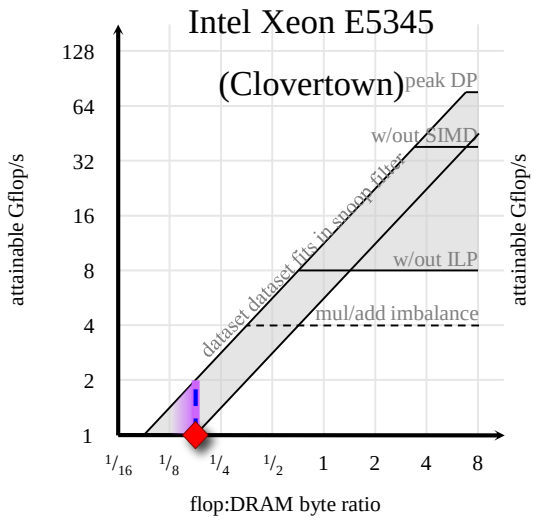
Short Name	Reordering Algorithm	Description
RCM	Reverse Cuthill-McKee	bandwidth reduction via breadth-first graph traversal
AMD	Approximate minimum degree	local greedy strategy to reduce fill by selecting sparsest pivot row
ND	Nested dissection	recursive divide-and-conquer using vertex separators to reduce fill
GP	Graph partitioning	multi-level recursive graph partitioning with METIS using edge-cut objective
HP	Hypergraph partitioning	column-net hypergraph partitioning with PaToH using cut-net metric
Gray	Gray code ordering	splitting of sparse and dense rows and Gray code ordering

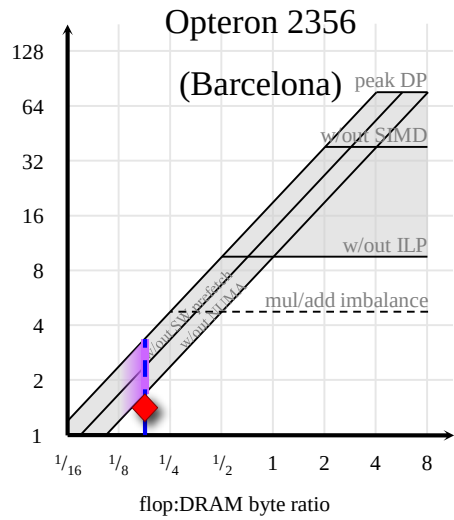
J. D. Trotter et al., “Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Nov. 2023, pp. 1–13.

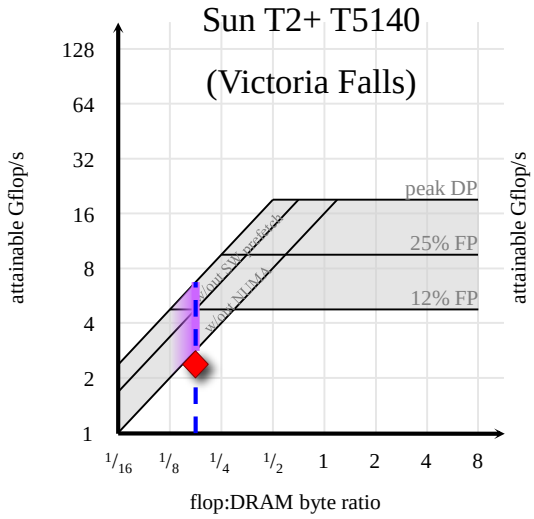
Roofline model for SpMV

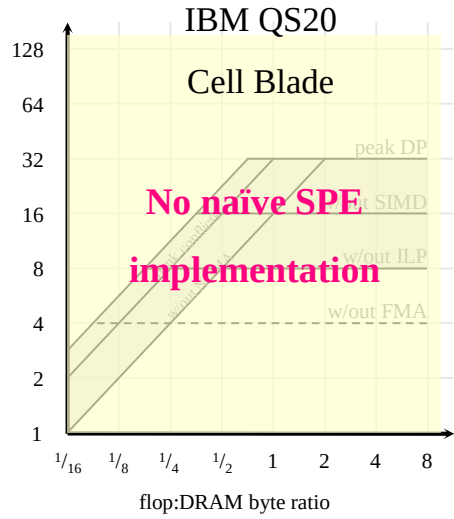
(out-of-the-box parallel)

- **Two unit stride streams**
- **Inherent FMA**
- **No ILP**
- **No DLP**
- **FP is 12-25%**
- **Naïve compulsory flop:byte < 0.166**
- **For simplicity: dense matrix in sparse format**



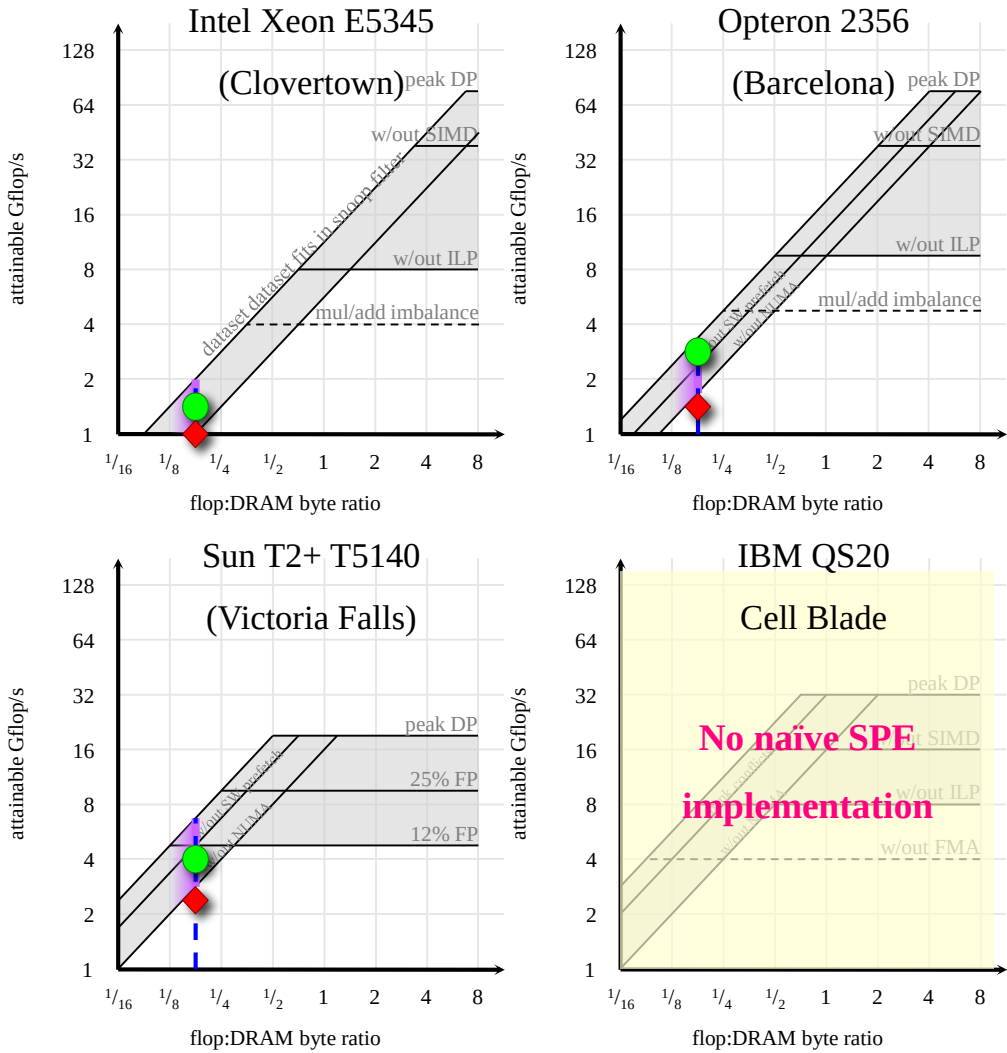






Roofline model for SpMV
(NUMA & SW prefetch)

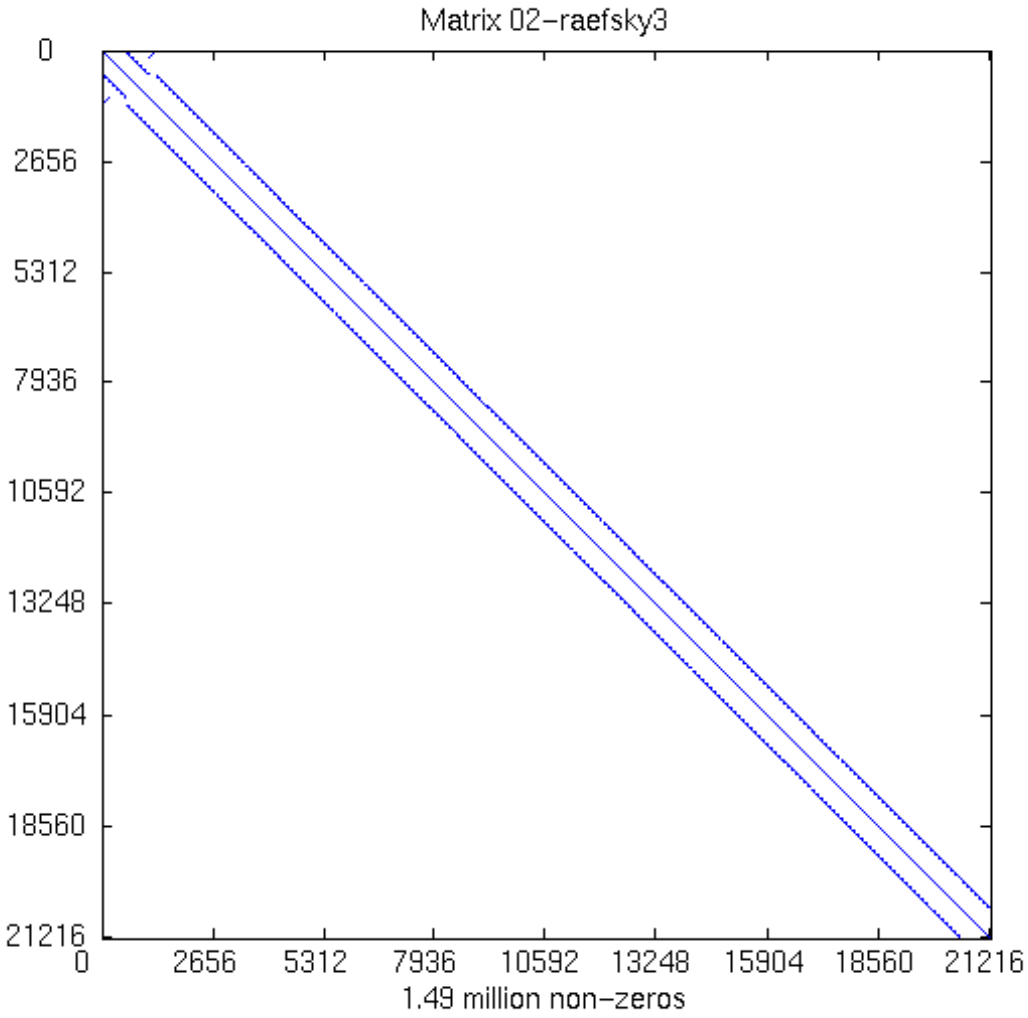
- **compulsory flop:byte ~ 0.166**
- **utilize all memory channels**



Outline

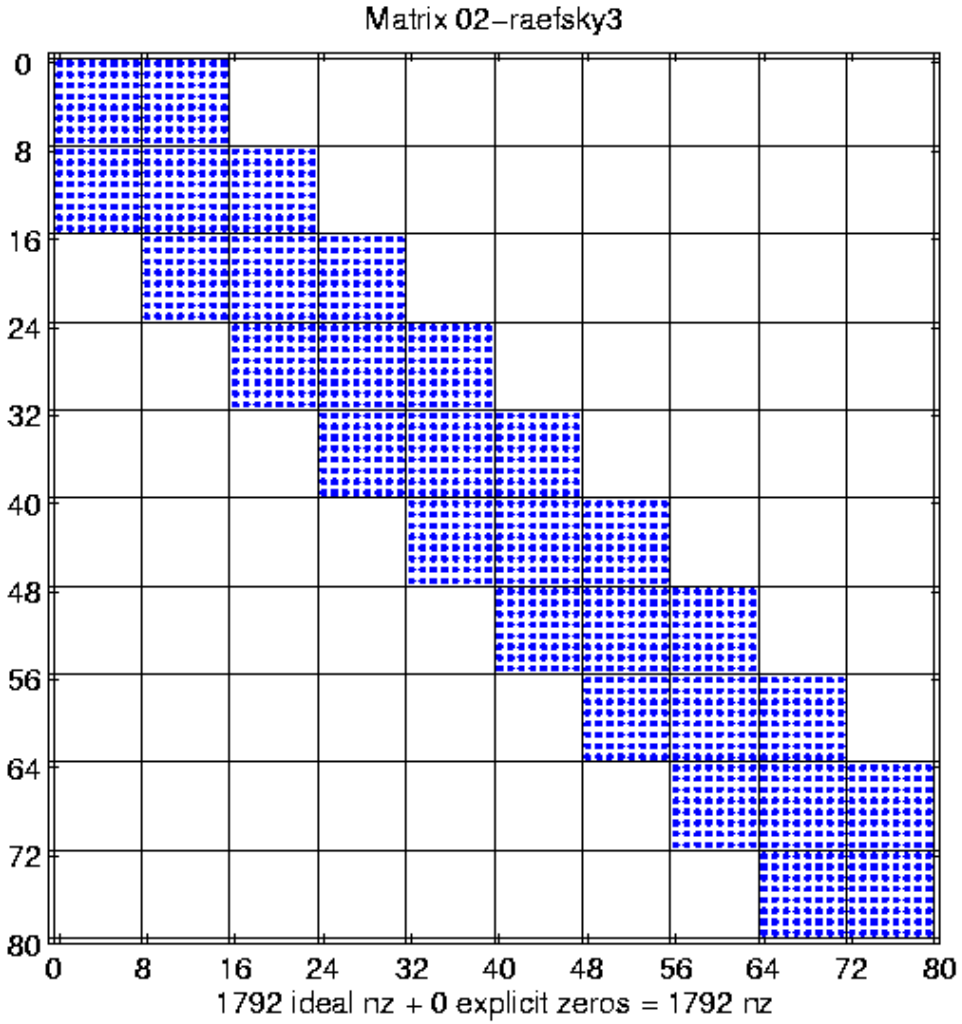
- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- Cache blocking SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

Changing Matrix Format: Blocking



- **n = 21200**
- **nnz = 1.5 M**
- **kernel: SpMV**
- **Source: NASA structural analysis problem**

Changing Matrix Format: Blocking

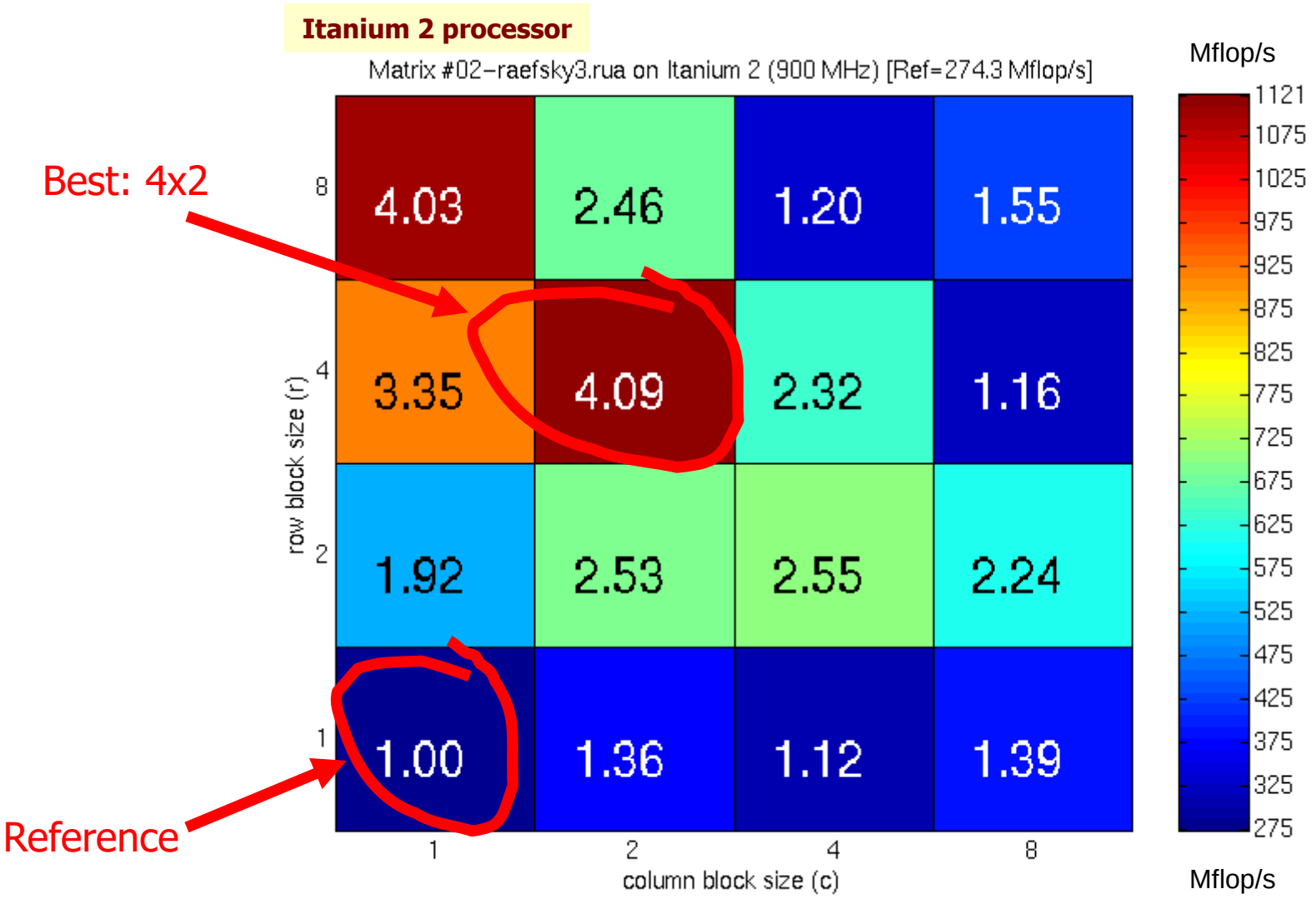


- n = 21200
- nnz = 1.5 M
- kernel: SpMV
- Source: NASA structural analysis problem
- 8x8 dense substructure

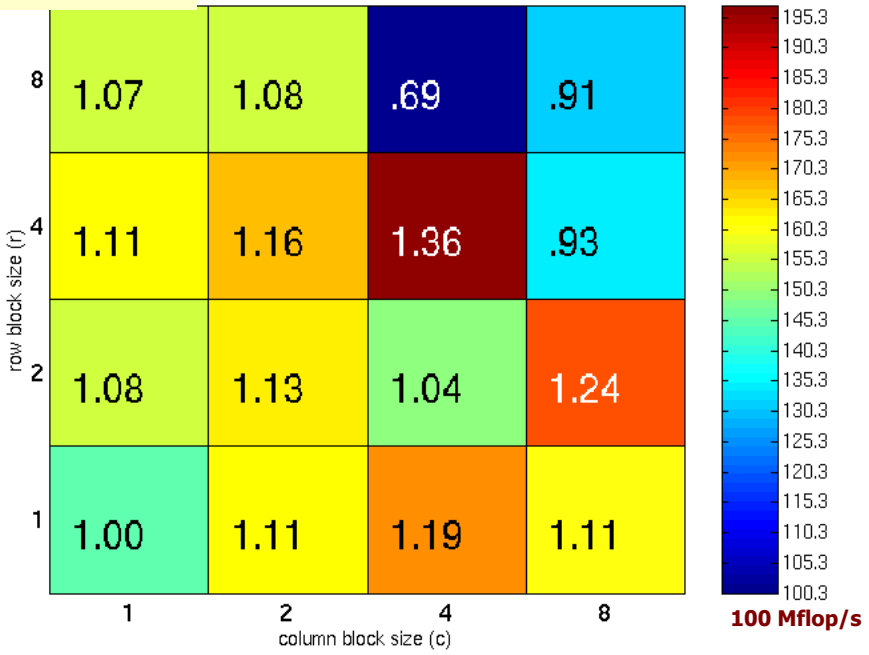
Taking advantage of block structure in SpMV

- **Bottleneck is time to get matrix from memory**
  - Only 2 flops for each nonzero in matrix
  - Fetching at ~1 int (column index) + 1 float (value) for 2 flops
- **Don't store each nonzero with index, instead store each nonzero r-by-c block with 1 column index**
  - As r*c grows, storage drops by up to 2x, for all 32-bit quantities
  - Time to fetch matrix from memory decreases
- **Change both data structure and algorithm**
  - Need to pick r and c
  - Need to change algorithm accordingly
- **In example, is r=c=8 best choice?**
  - Minimizes storage, so looks like a good idea…
- **Consider best case: dense matrix in sparse format**

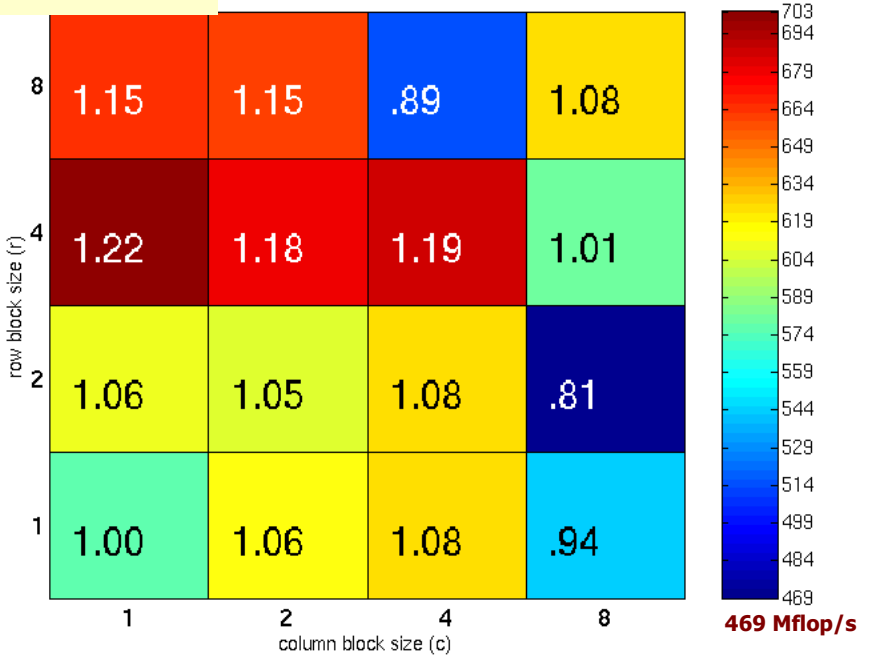
The Need for Search



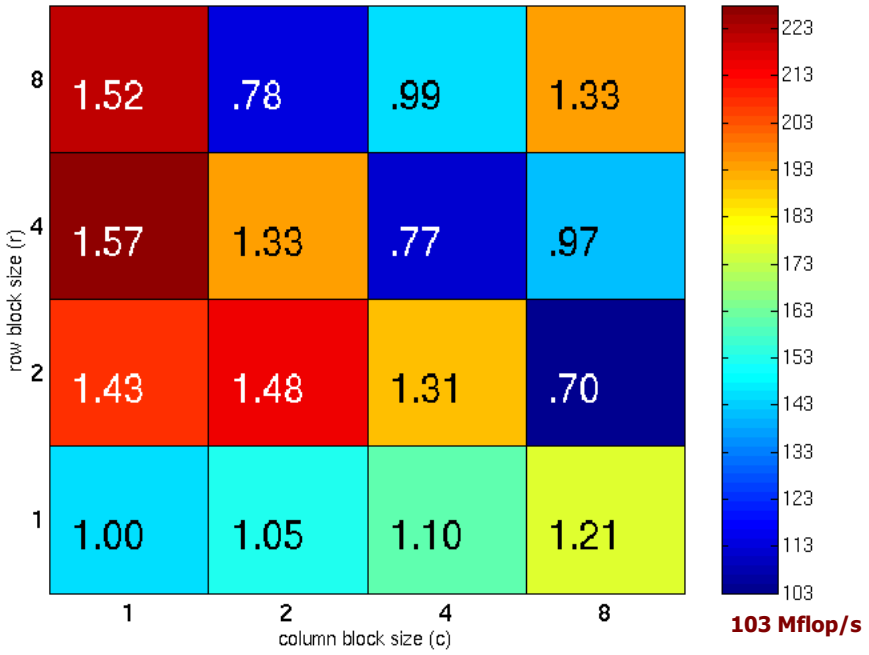
Power3 - 13%



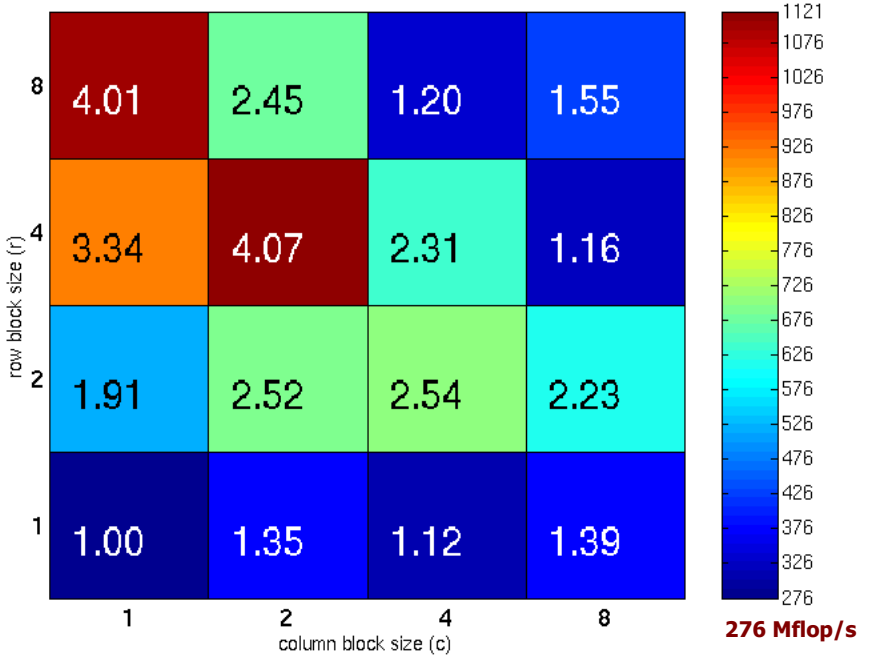
Power4 - 14%



Itanium 1 - 7%

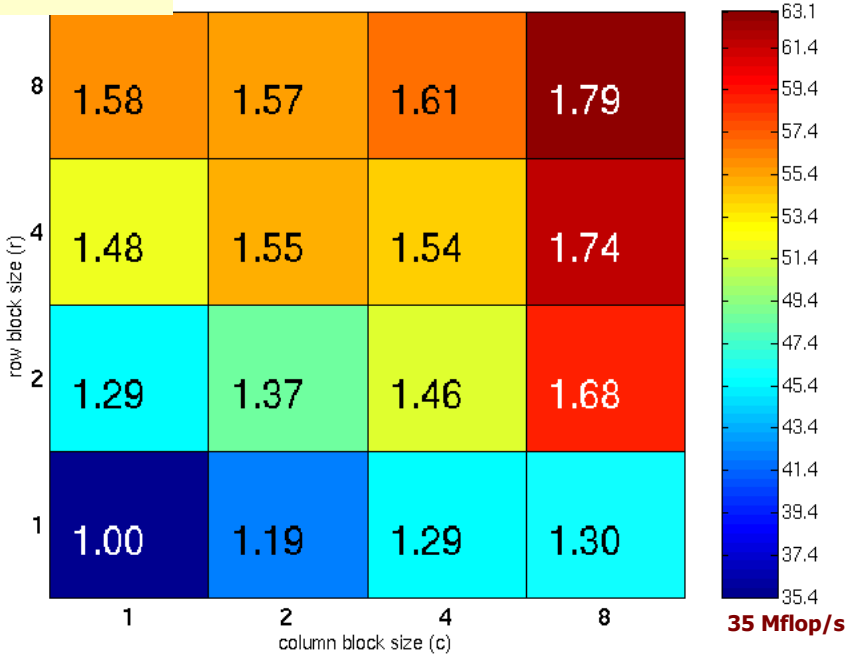


Itanium 2 - 31%



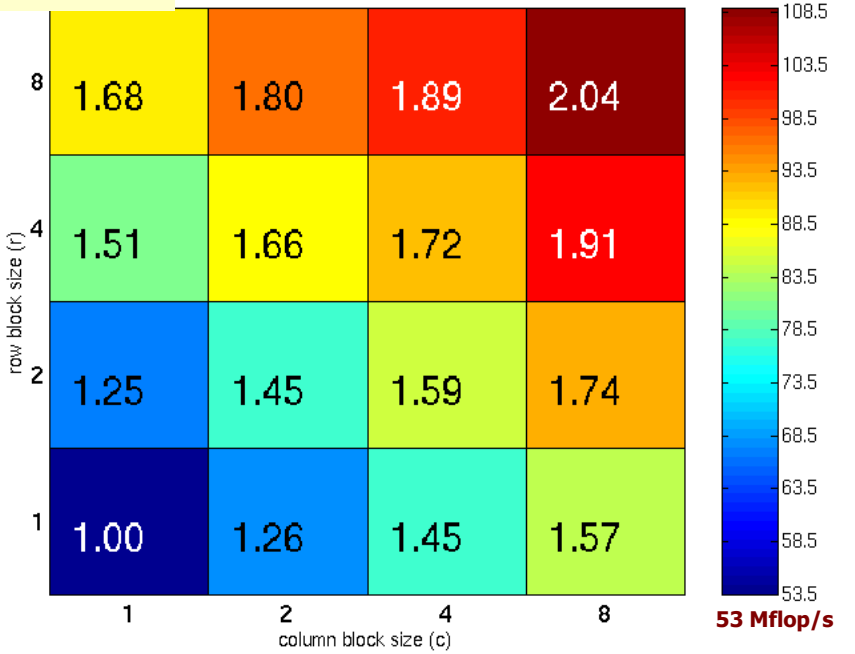
Ultra 2i - 9%

raefsky3.rua [ref=35.3 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]



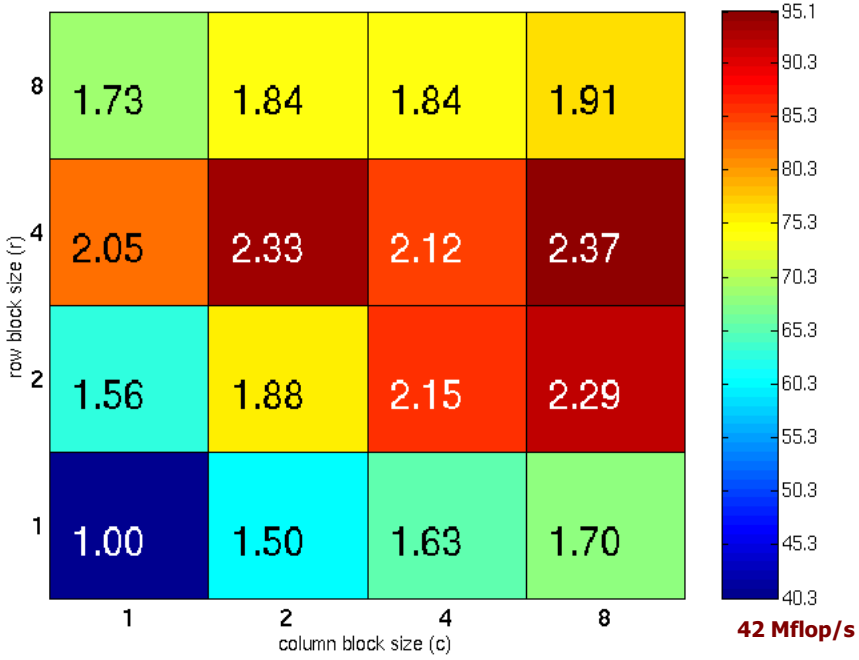
Ultra 3 - 5%

ce: raefsky3.rua [ref=53.5 Mflop/s; 900 MHz Ultra 3, Sun CC v6]



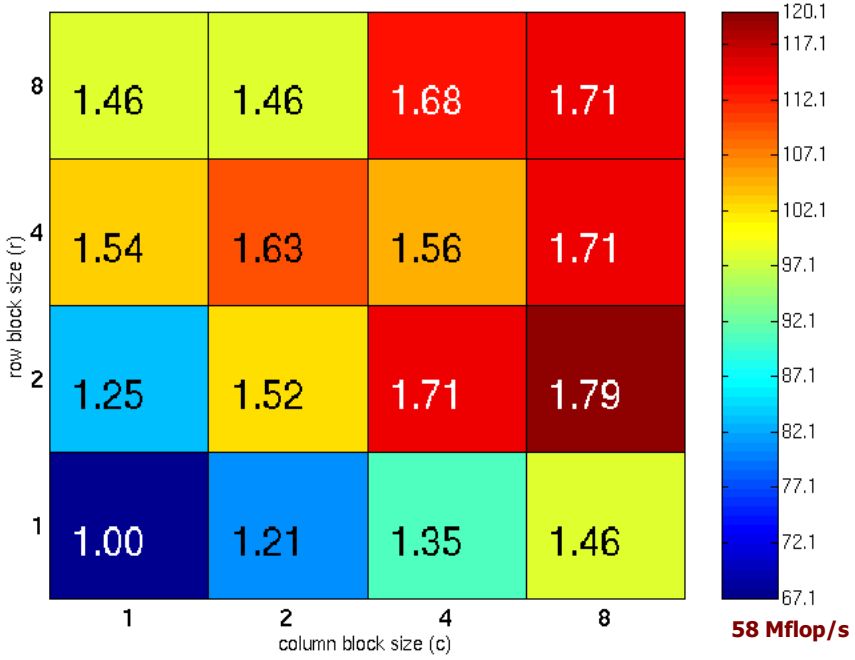
Pentium III - 19%

.rua [ref=40.2 Mflop/s; 500 MHz Pentium III, Intel C v7]

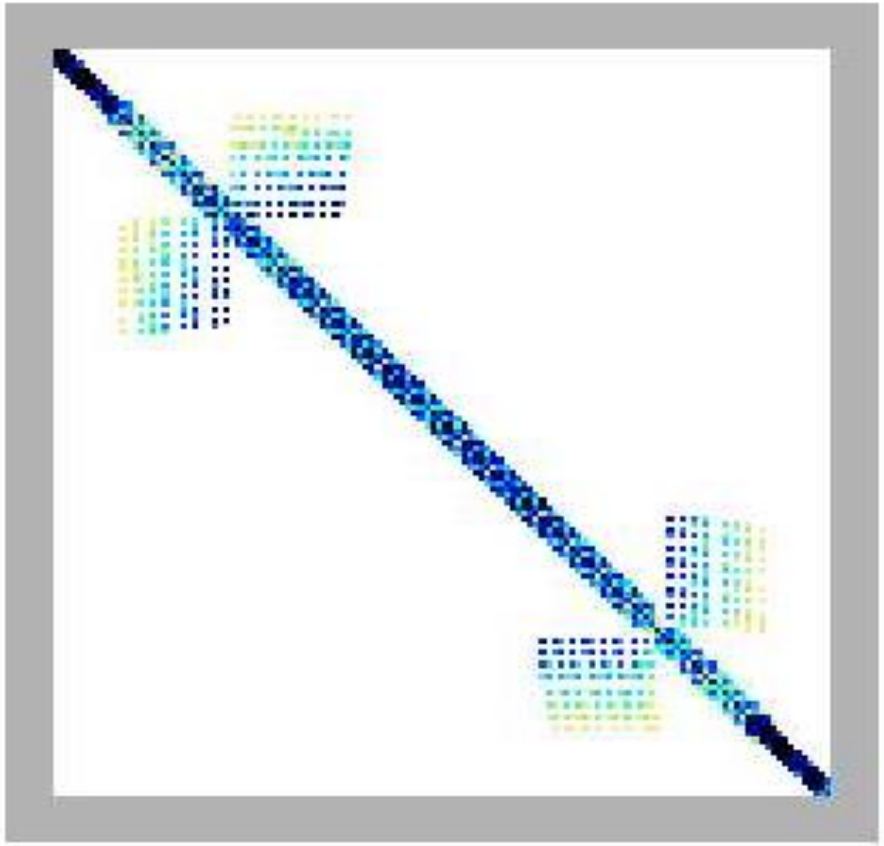


Pentium III-M - 15%

67.1 Mflop/s; 800 MHz Pentium III-M, Intel C v7.0]



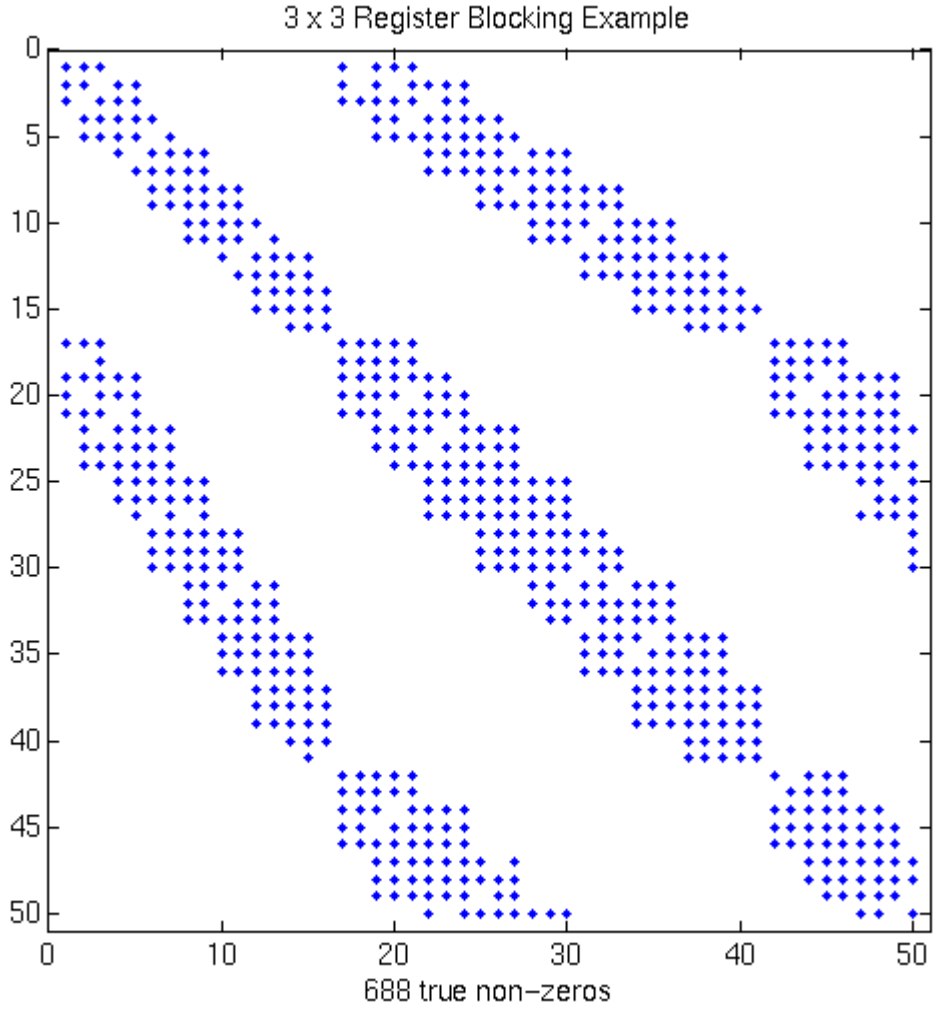
But most matrices don't block so easily



- FEM Fluid dynamics problems
- More complicated non-zero structure in general

- N = 16614
- NNZ = 1.1M

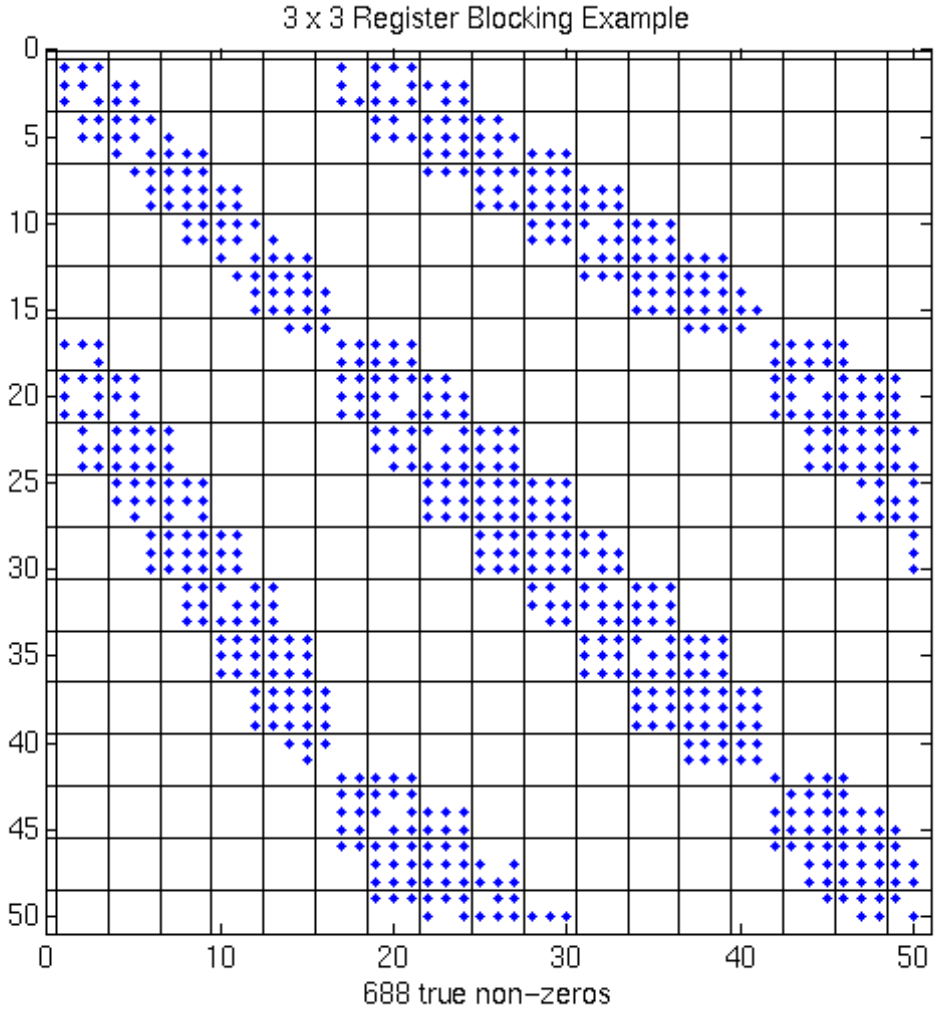
Zoom in to top corner



- More complicated non-zero structure

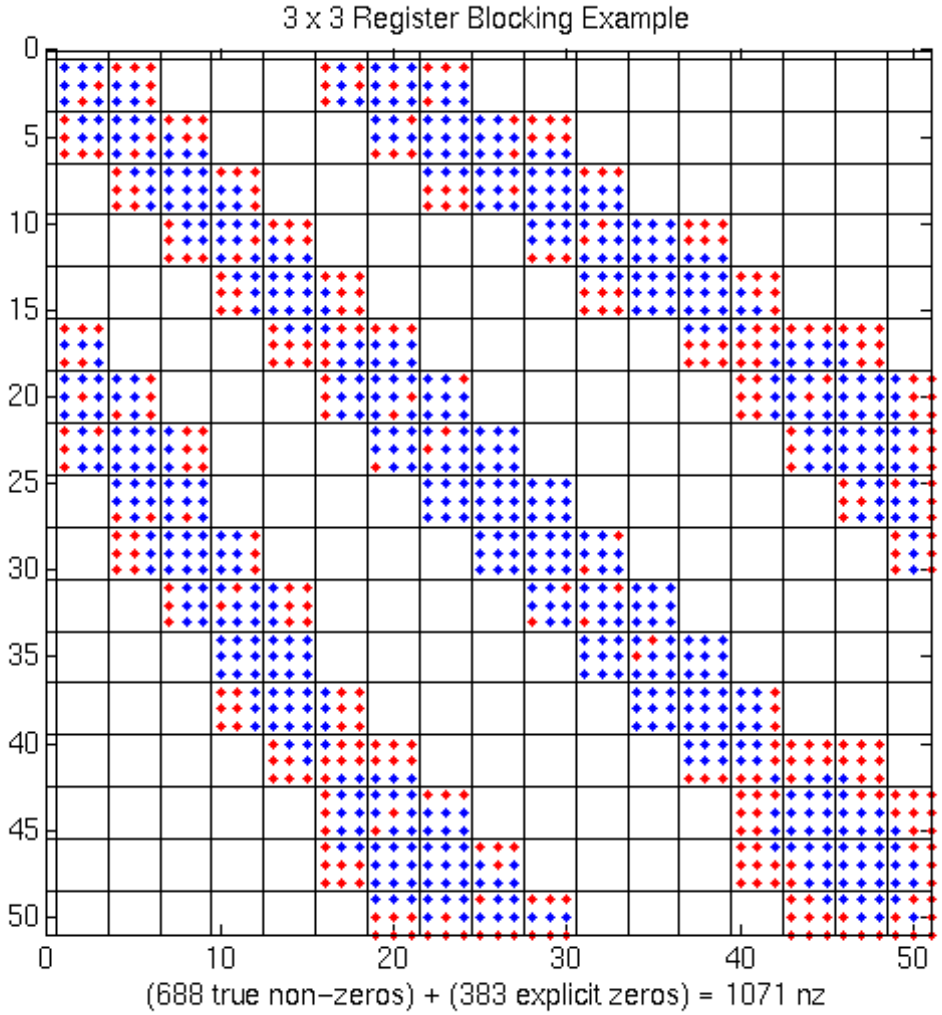
- N = 16614
- NNZ = 1.1M

3x3 blocks look natural, but…



- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Many cell are not full
- N = 16614
- NNZ = 1.1M

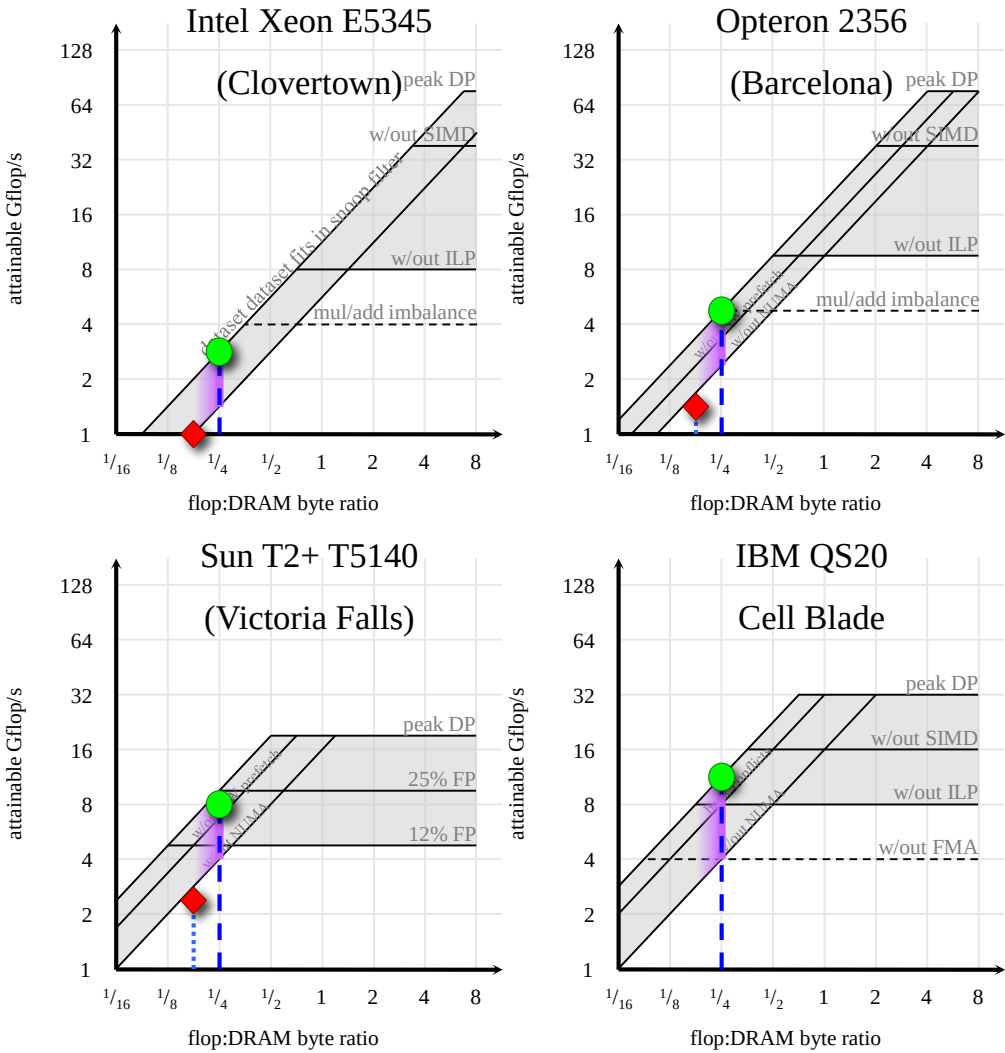
Extra work can improve efficiency



- More complicated non-zero structure
- Example: 3x3 blocks
  - Grid of 3x3 cells
  - Add explicit zeros: 1.5x "fill overhead"
  - Unroll loops
- More work but faster
  - 1.5x faster on PIII
  - 2.2x "theoretical" for dense in sparse format

Roofline model for SpMV
(matrix compression)

- **Inherent FMA**
- **Register blocking improves ILP, DLP, flop:byte ratio, and FP% of instructions**
- **Other forms of matrix compression may also help**
  - **16-bit indices within blocks**
  - **Patterns of repeated nonzeros**



What about sparse matrices?

- **How to build optimized library when:**
  - Formats are not known? Libraries like PETSc and Trilinos will let the user provide format and SpMV
- **How to build optimized matrix kernel library?**
  - Nonzero structure is key to optimization
- **OSKI = Optimized Sparse Kernel Interface**
  - pOSKI for multicore

Automatic Register Block Size Selection

- **Selecting the r x c block size**
  - Off-line benchmark of "register profile"
    - Precompute **Mflops(r,c)** using dense A in sparse format (blocked sparse row) for each r x c
    - Once per machine/architecture
  - Run-time "search"
    - Sample A to estimate **Fill(r,c)** for each r x c
  - Run-time heuristic model
    - Choose r, c to minimize **time ~ Fill(r,c) / Mflops(r,c)**

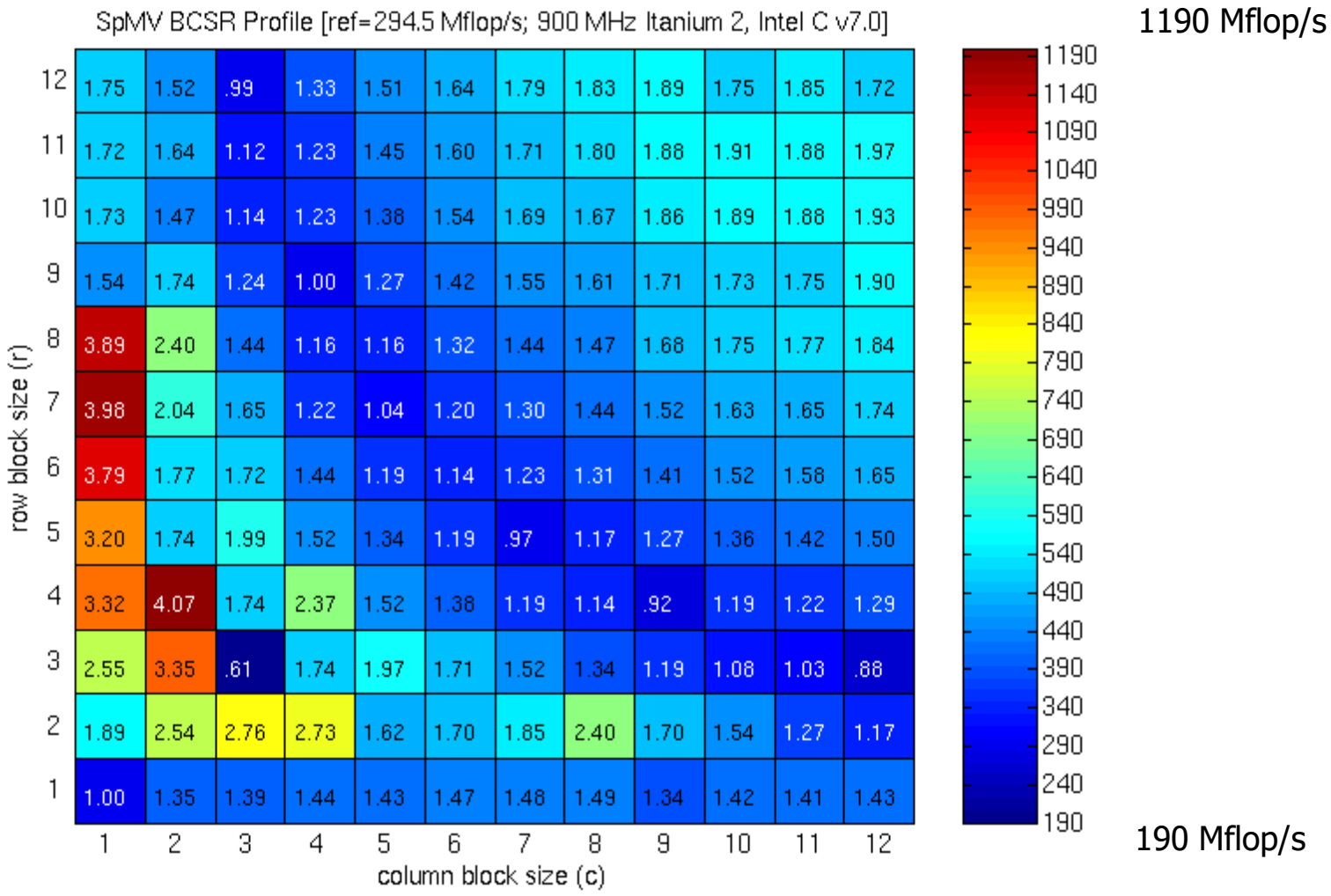
Eun-Jin Im PhD thesis: http://digitalassets.lib.berkeley.edu/techreports/ucb/text/CSD-00-1104.pdf

Rich Vuduc PhD thesis: http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

Machine Learning in Automatic Performance Tuning

- **Autotuning**
  - Vuduc, R.W. (2011). Autotuning. In: Padua, D. (eds) Encyclopedia of Parallel Computing. Springer.
  - Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. “Statistical Models for Empirical Search-Based Performance Tuning”, Int. J. High Perform. Comput. Appl. 18, 1 (February 2004), pp. 65–94.
  - A. S. Ganapathi. 2009. “Predicting and optimizing system utilization and performance via statistical machine learning.” PhD dissertation. UC Berkeley. Advisor(s) D. Patterson.
  - J. Bergstra, N. Pinto and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," Innovative Parallel Computing (InPar) 2012.
  - Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. “Practical Bayesian optimization of machine learning algorithms”, NIPS 2012.
  - J. Ansel et al., "OpenTuner: An extensible framework for program autotuning," PACT 2014.
- **Related techniques**
  - hyperparameter optimization (HBO)
  - black box optimization (BBO)

Register Profile: dense matrix in sparse format

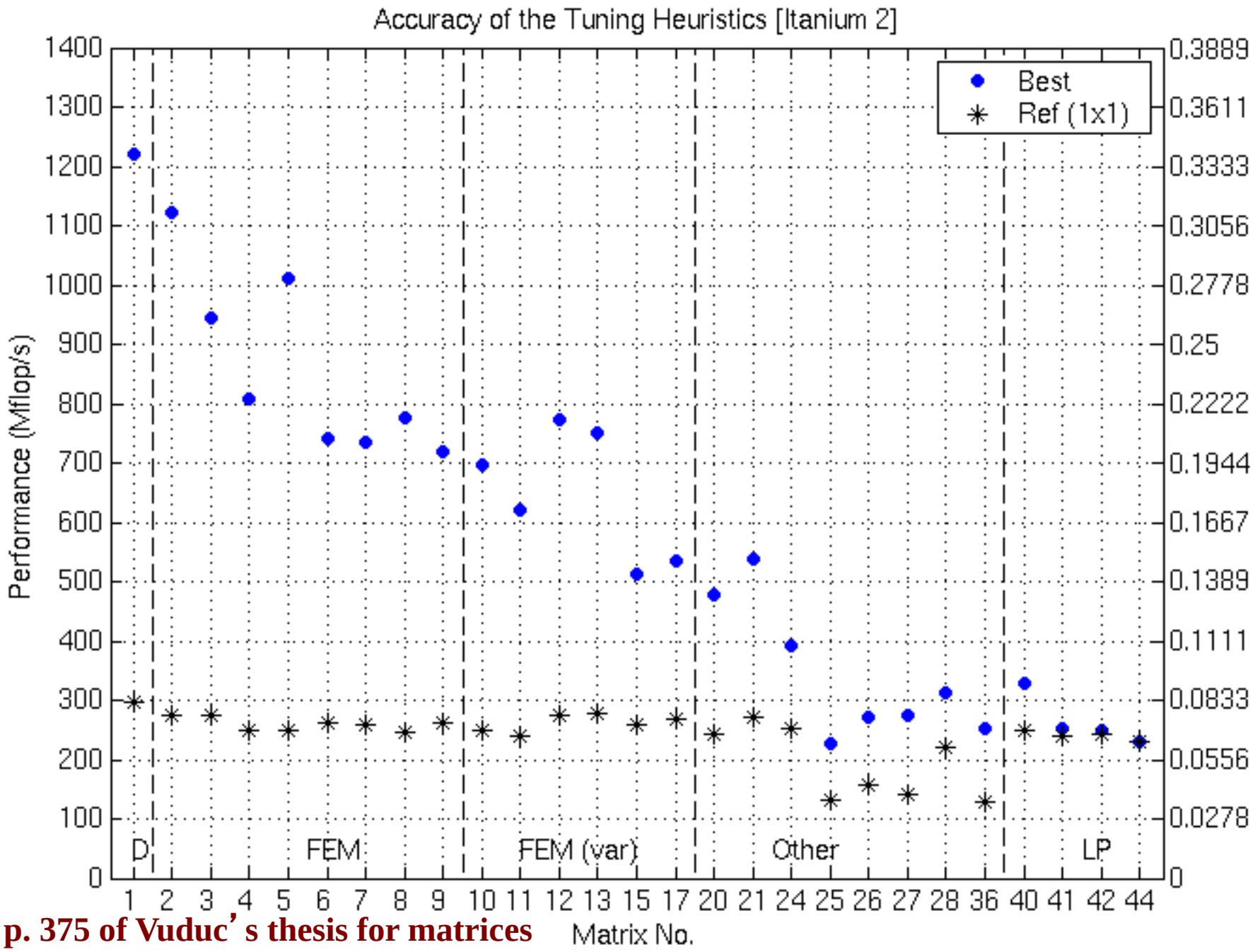


1190 Mflop/s

190 Mflop/s

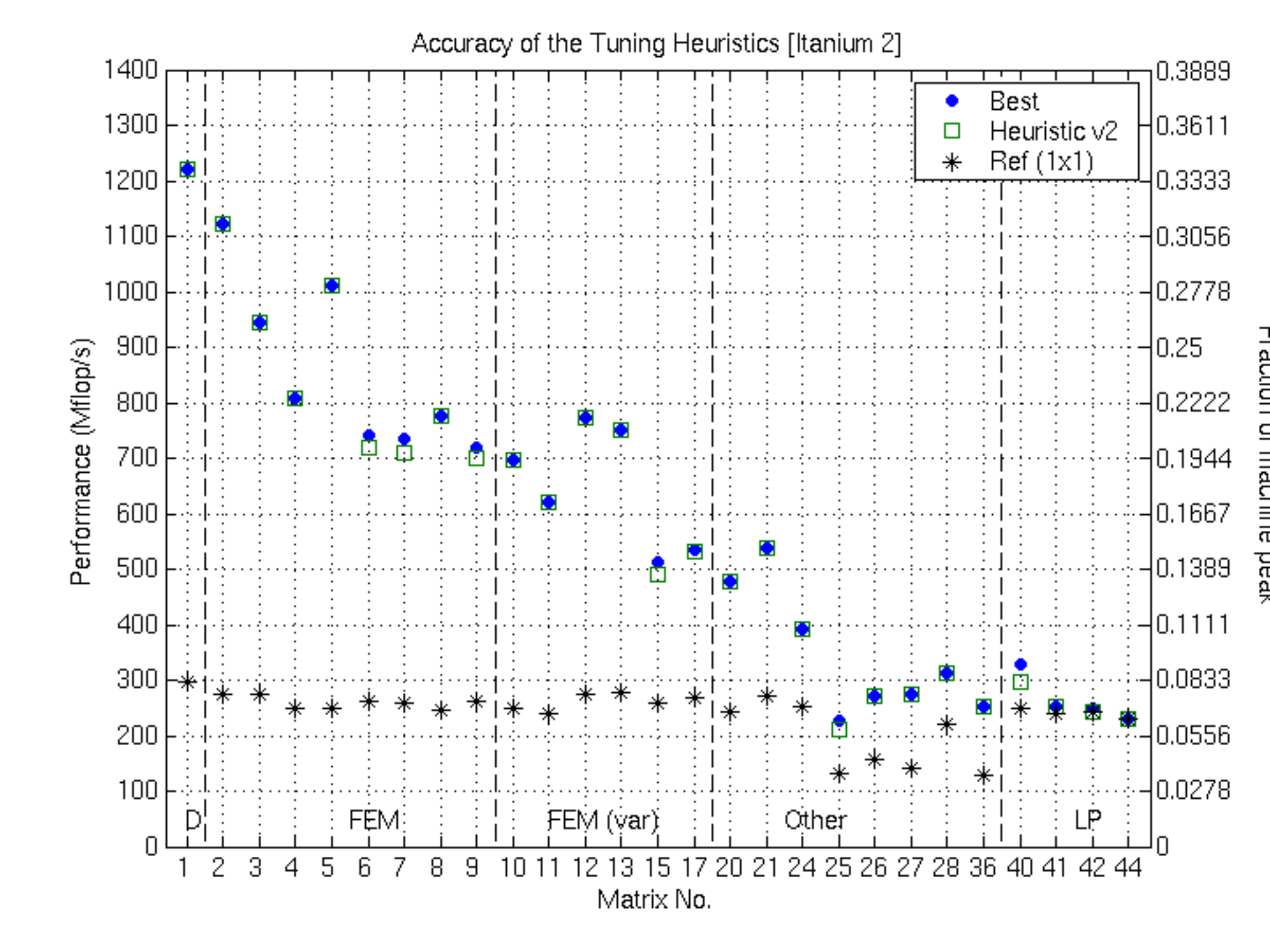
Accurate and Efficient Adaptive Fill Estimation

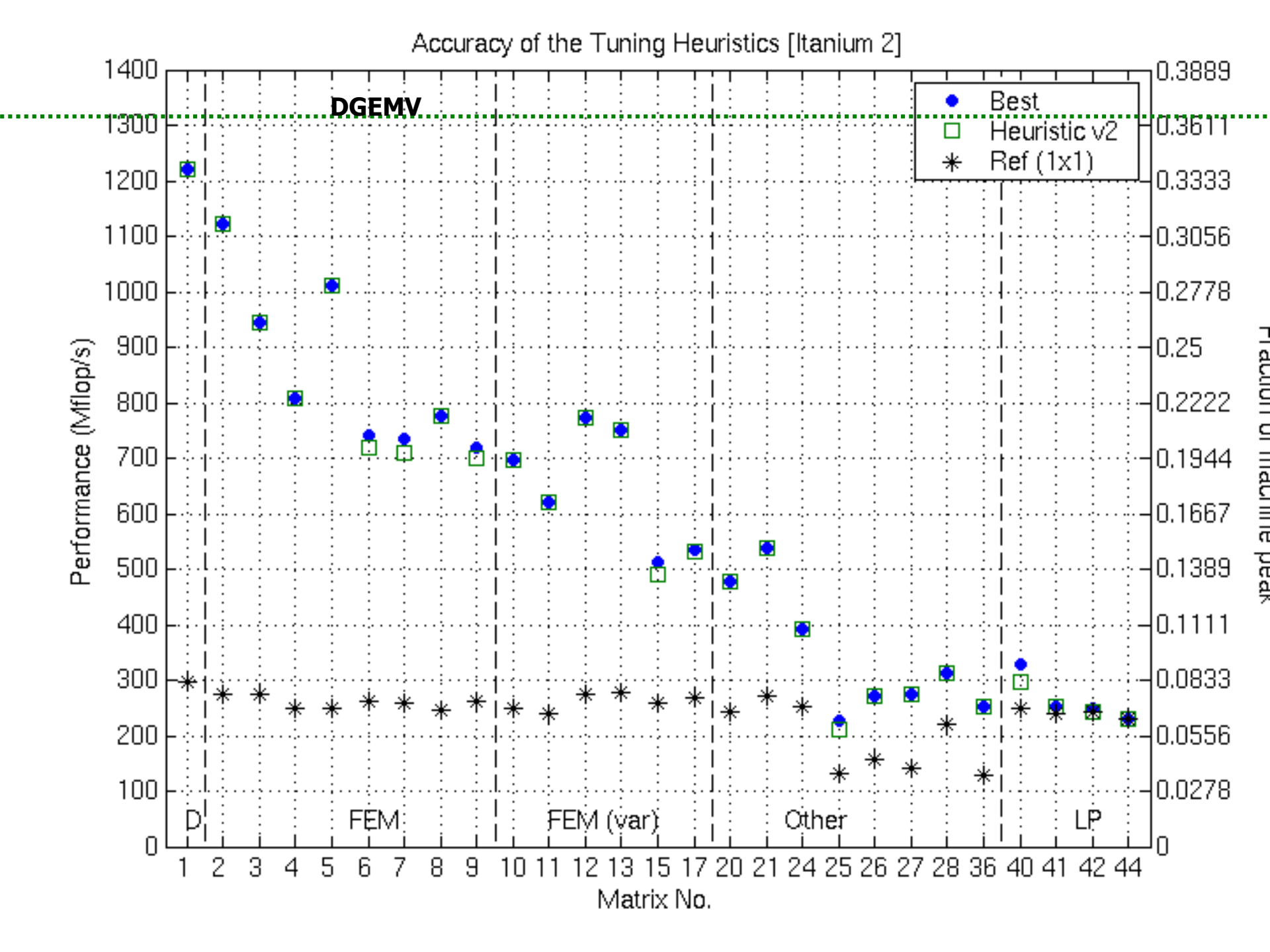
- **Idea: Sample matrix**
  - Fraction of matrix to sample: s Î [0,1]
  - Cost ~ O(s * nnz)
  - Control cost by controlling s
    - Search at run-time: the constant matters!
- **Control s automatically by computing statistical confidence intervals**
  - Idea: Monitor variance
- **Cost of tuning**
  - Lower bound: convert matrix in 5 to 40 unblocked SpMVs
  - Heuristic: 1 to 11 SpMVs



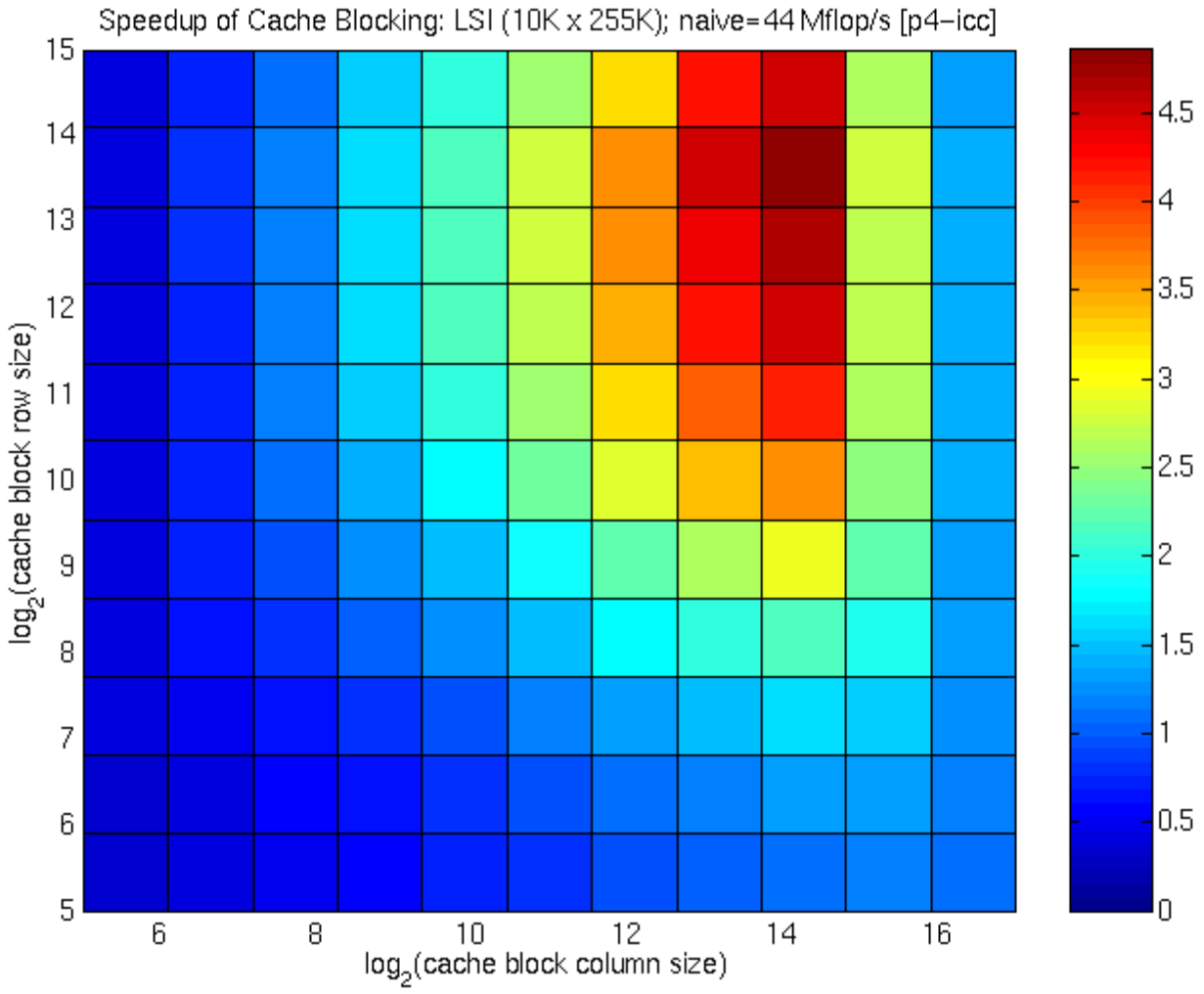
See p. 375 of Vuduc's thesis for matrices

NOTE: "Fair" flops used (ops on explicit zeros not counted as "work")


Accuracy of the Tuning Heuristics [Itanium 2]
Performance (Mflop/s)
Fraction of machine peak
Matrix No.
Best
Heuristic v2
Ref (1x1)
D
FEM
FEM (var)
Other
LP
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
0.3889
0.3611
0.3333
0.3056
0.2778
0.25
0.2222
0.1944
0.1667
0.1389
0.1111
0.0833
0.0556
0.0278
0
1 2 3 4 5 6 7 8 9 10 11 12 13 15 17 20 21 24 25 26 27 28 36 40 41 42 44


Accuracy of the Tuning Heuristics [Itanium 2]
DGEMV
Best
Heuristic v2
Ref (1x1)
Performance (Mflop/s)
Fraction of Machine Peak
Matrix No.
D
FEM
FEM (var)
Other
LP
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
0.3889
0.3611
0.3333
0.3056
0.2778
0.25
0.2222
0.1944
0.1667
0.1389
0.1111
0.0833
0.0556
0.0278
0
1 2 3 4 5 6 7 8 9 10 11 12 13 15 17 20 21 24 25 26 27 28 36 40 41 42 44

Cache Blocking on LSI Matrix: Pentium 4



A

10k x 255k

3.7M non-zeros

Baseline:

44 Mflop/s

Best block size & performance:

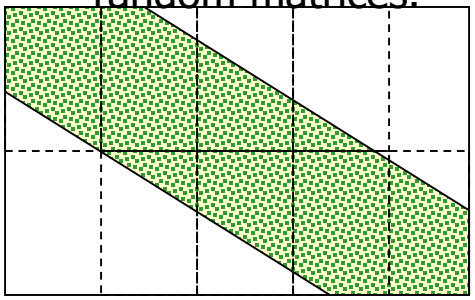
16k x 16k

210 Mflop/s

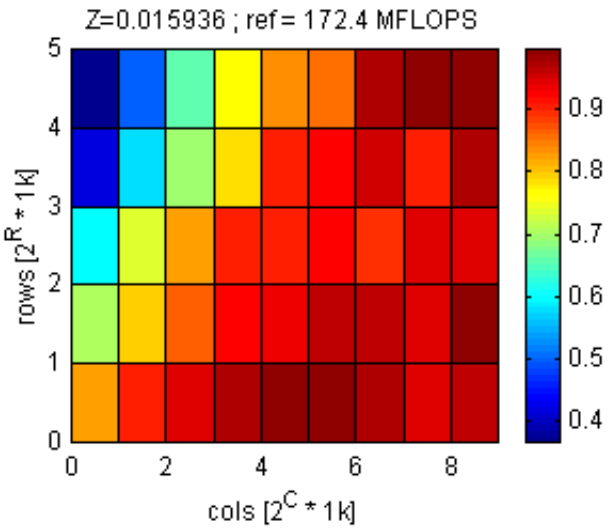
Nishtala, et al (2007). When cache blocking of sparse matrix vector multiply works and why.

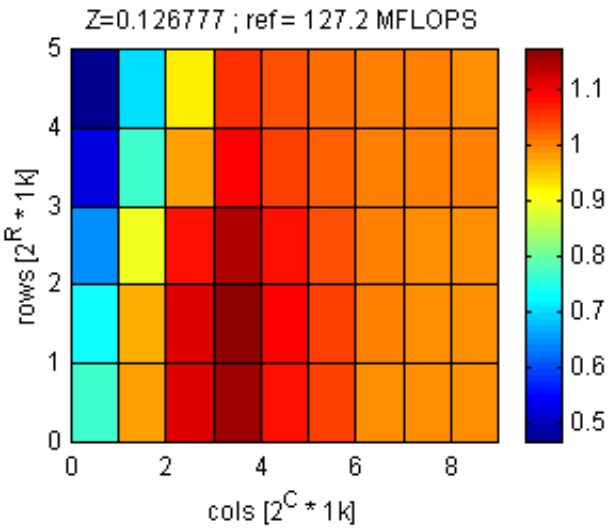
Cache Blocking on Random Matrices: Itanium

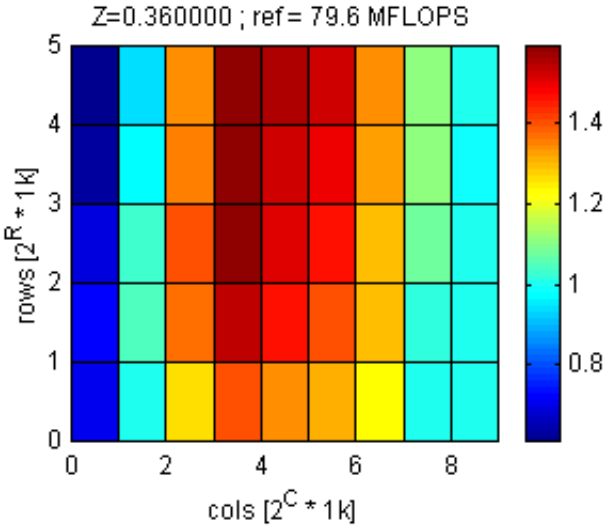
Speedup on four banded random matrices.

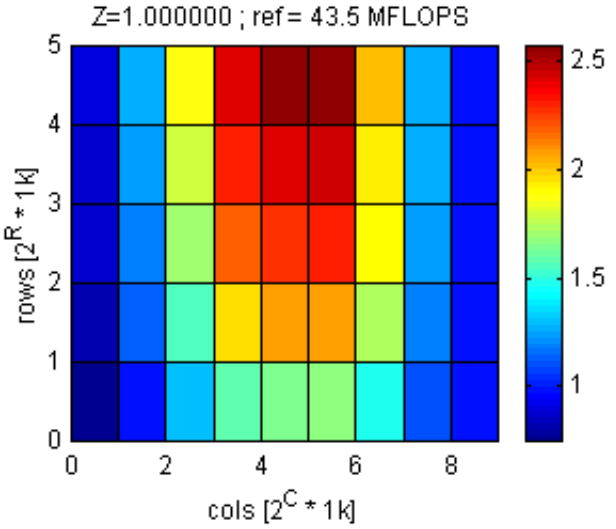


Nishtala, et al (2007). When cache blocking of sparse matrix vector multiply works and why.





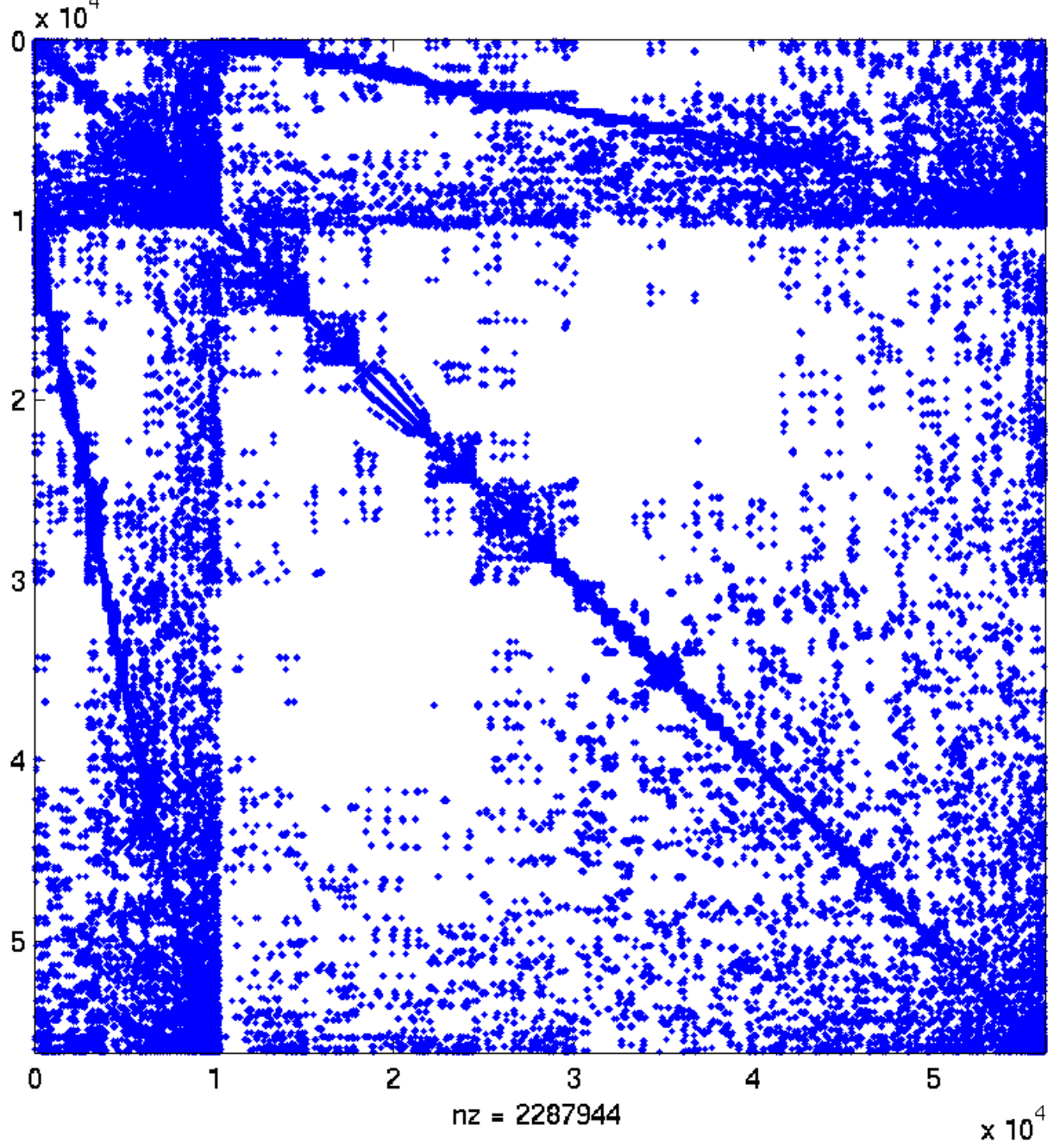


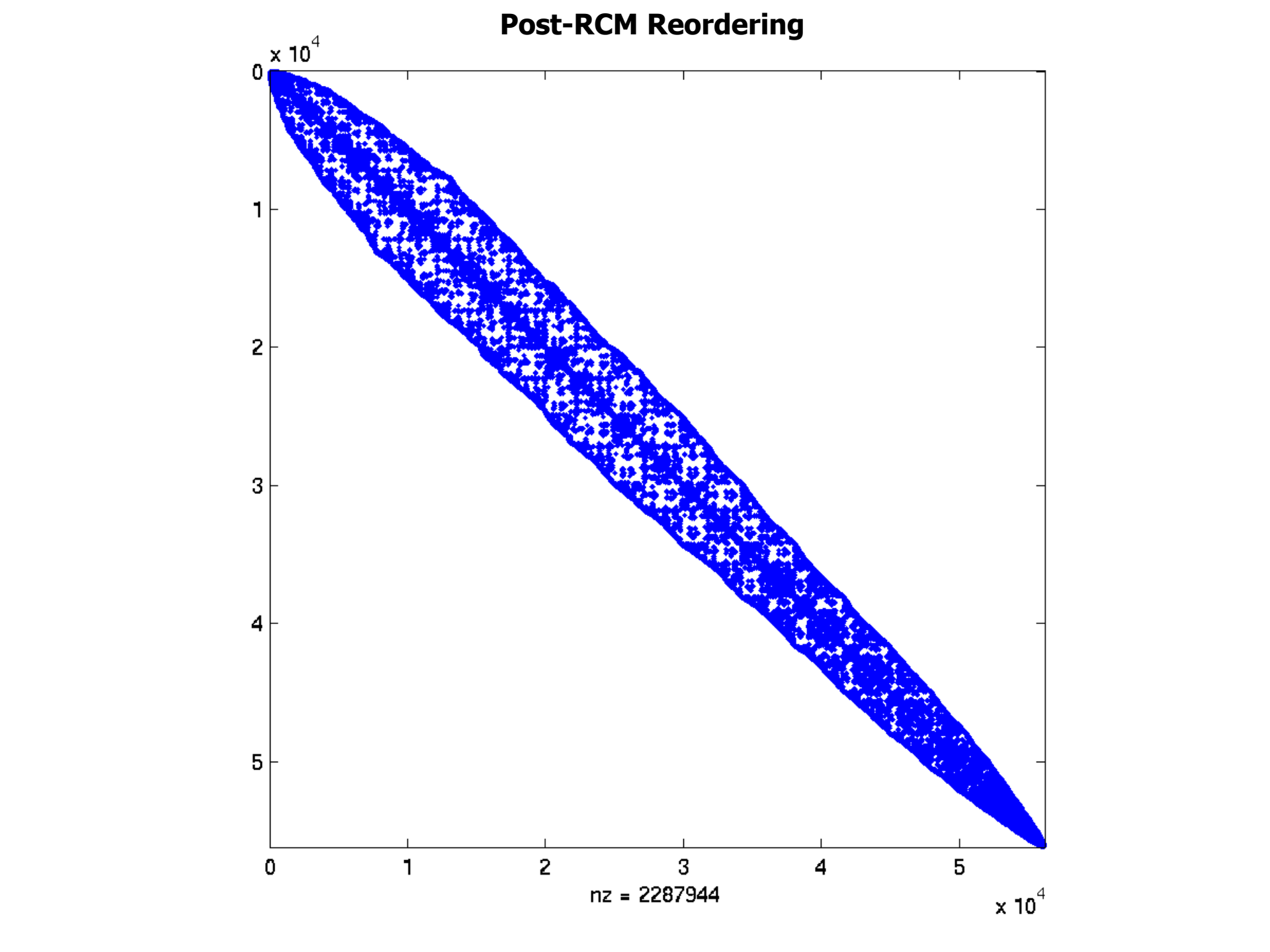


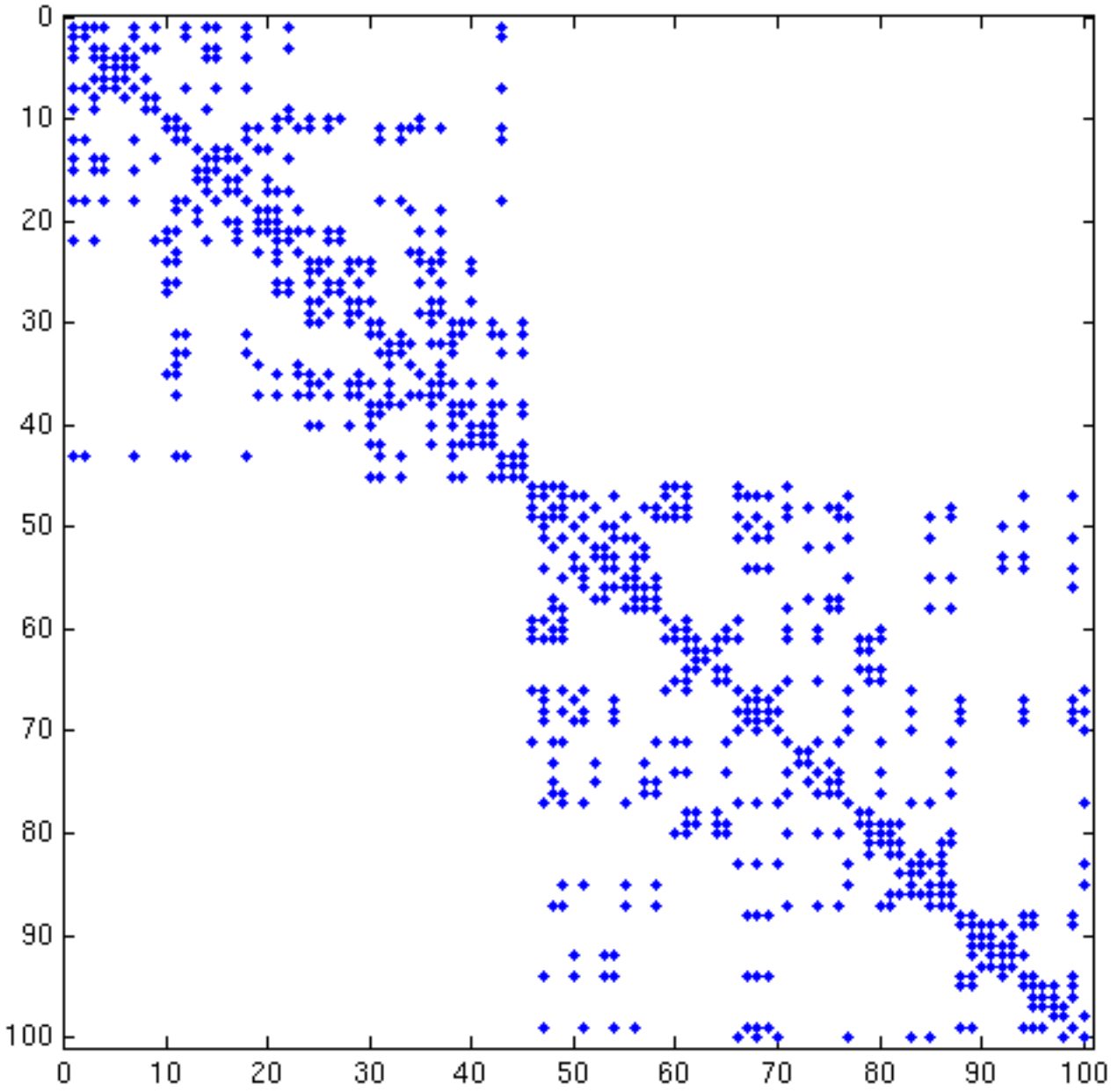
Matrix Reordering: Case study

- **Application: accelerator cavity design [Ko]**
- **Relevant optimization techniques**
  - Symmetric storage
  - Register blocking
  - Reordering, to create more dense blocks
    - Reverse Cuthill-McKee ordering to reduce bandwidth
      - Do Breadth-First-Search, number nodes in reverse order visited
    - Traveling Salesman Problem-based ordering to create blocks
      - Nodes = columns of A
      - Weights(u, v) = no. of nonzeros u, v have in common
      - Tour = ordering of columns
      - Choose maximum weight tour
      - See [Pinar & Heath ’ 97]
- **2.1x speedup on Power 4**

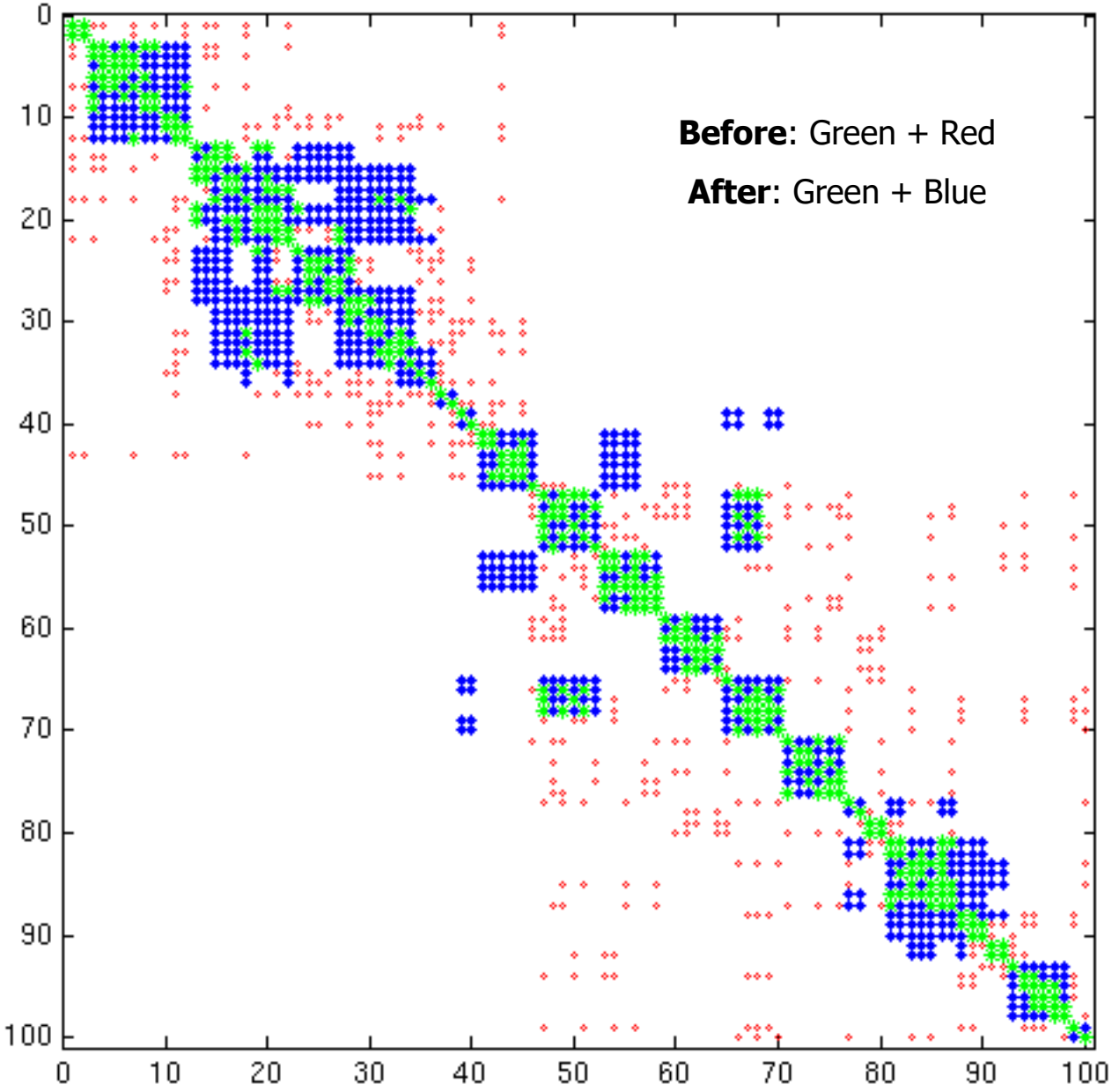
Source: Accelerator Cavity Design Problem (Ko via Husbands)



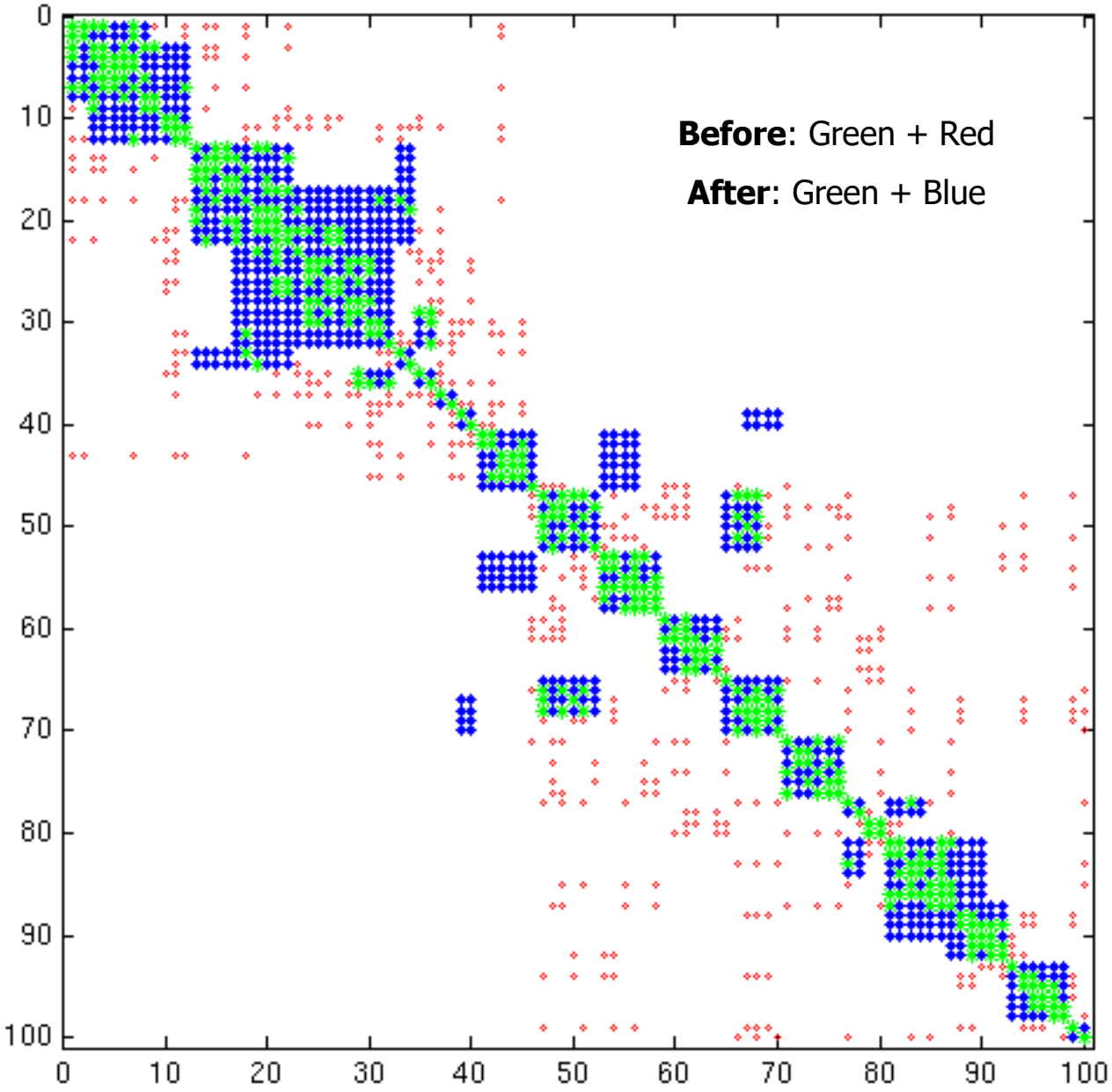

Post-RCM Reordering
x 10^4
0
1
2
3
4
5
0
1
2
3
4
5
nz = 2287944
x 10^4

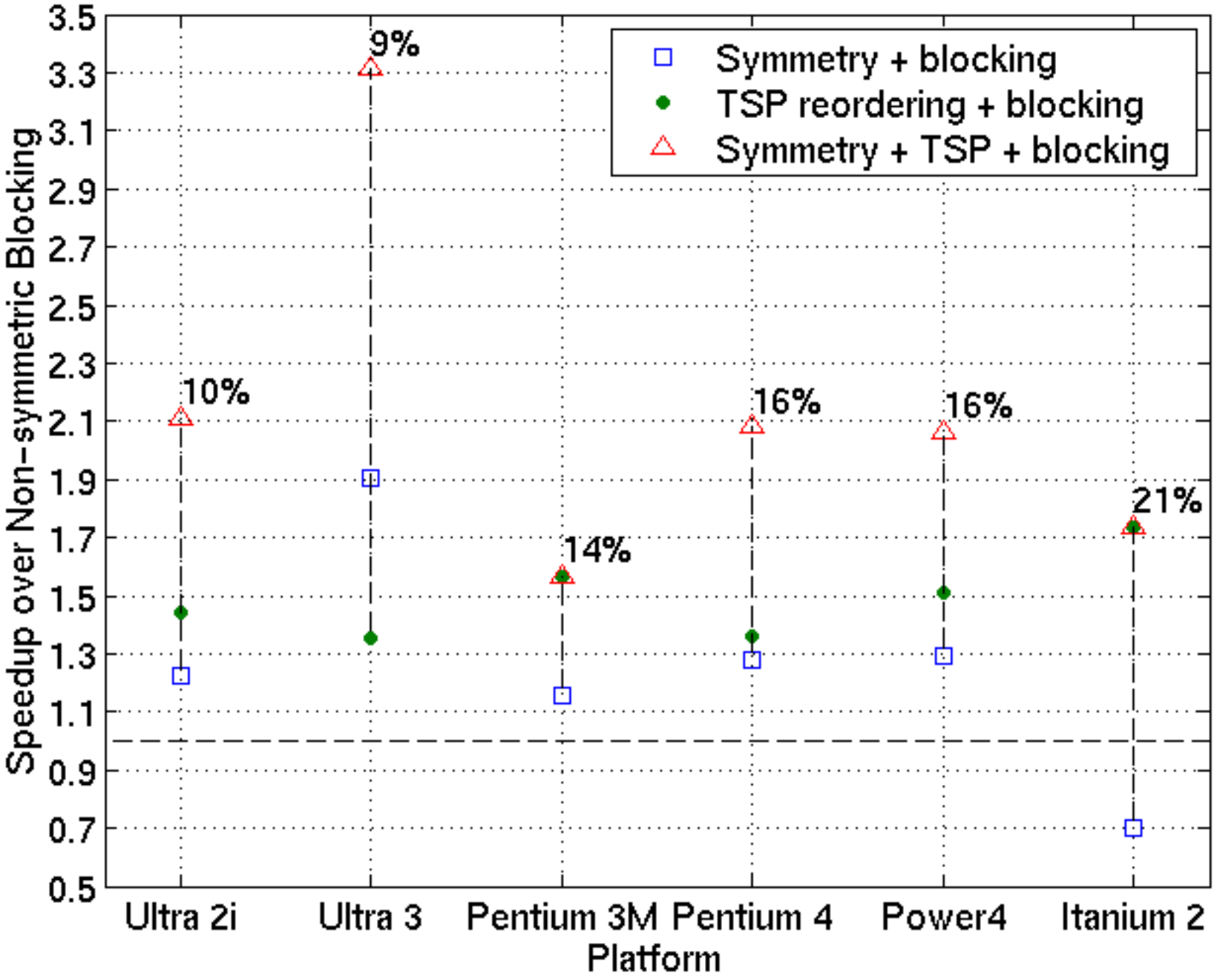

100x100 Submatrix Along Diagonal
0
10
20
30
40
50
60
70
80
90
100
0
10
20
30
40
50
60
70
80
90
100

“Microscopic” Effect of RCM Reordering



“Microscopic” Effect of Combined RCM+TSP Reordering




SpMV Speedups: SLAC Matrix (Omega3P)
Symmetry + blocking
TSP reordering + blocking
Symmetry + TSP + blocking
Speedup over Non-symmetric Blocking
3.5
3.3
3.1
2.9
2.7
2.5
2.3
2.1
1.9
1.7
1.5
1.3
1.1
0.9
0.7
0.5
9%
10%
14%
16%
16%
21%
Ultra 2i
Ultra 3
Pentium 3M
Pentium 4
Power4
Itanium 2
Platform

How do permutations affect algorithms?

- **A = original matrix, A^P = A with permuted rows, columns**
- **Naïve approach: permute x, multiply $y=A^Px$, permute y**
- **Faster way to solve Ax=b**
  - Write $A^P = P^TAP$ where P is a permutation matrix
  - Solve $A^Px^P = P^Tb$ for x^P, using SpMV with A^P, then let $x = Px^P$
  - Only need to permute vectors twice, not twice per iteration
- **Faster way to solve $Ax=\lambda x$**
  - A and A^P have same eigenvalues, no vectors to permute!
  - $A^Px^P = \lambda x^P$ implies $Ax = \lambda x$ where $x = Px^P$
- **Where else do optimizations change higher level algorithms? More later…**

Summary of Other Sequential Performance Optimizations

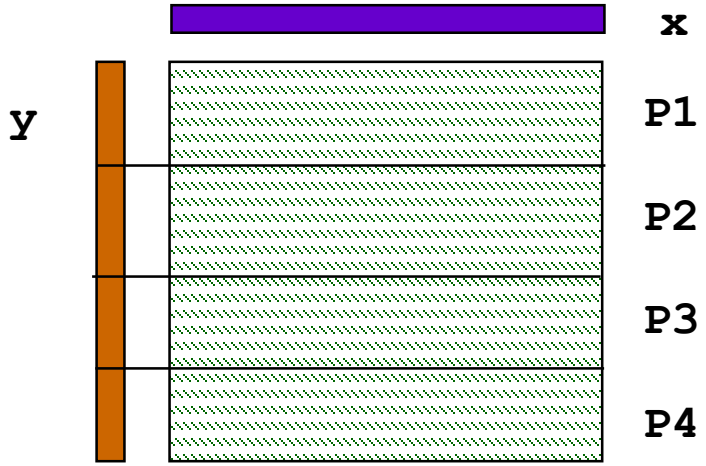
- **Optimizations for SpMV**
  - Register blocking (RB): up to 4x over CSR
  - Variable block splitting: 2.1x over CSR, 1.8x over RB
  - Diagonals: 2x over CSR
  - Reordering to create dense structure + splitting: 2x over CSR
  - Symmetry: 2.8x over CSR, 2.6x over RB
  - Cache blocking: 2.8x over CSR
  - Multiple vectors (SpMM): 7x over CSR
  - And combinations…
- **Sparse triangular solve**
  - Hybrid sparse/dense data structure: 1.8x over CSR
- **Higher-level kernels**
  - $A \cdot A^T \cdot x$, $A^T \cdot A \cdot x$: 4x over CSR, 1.8x over RB
  - $A^2 \cdot x$: 2x over CSR, 1.5x over RB
  - $[A \cdot x, A^2 \cdot x, A^3 \cdot x, .. , A^k \cdot x]$

Outline

- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

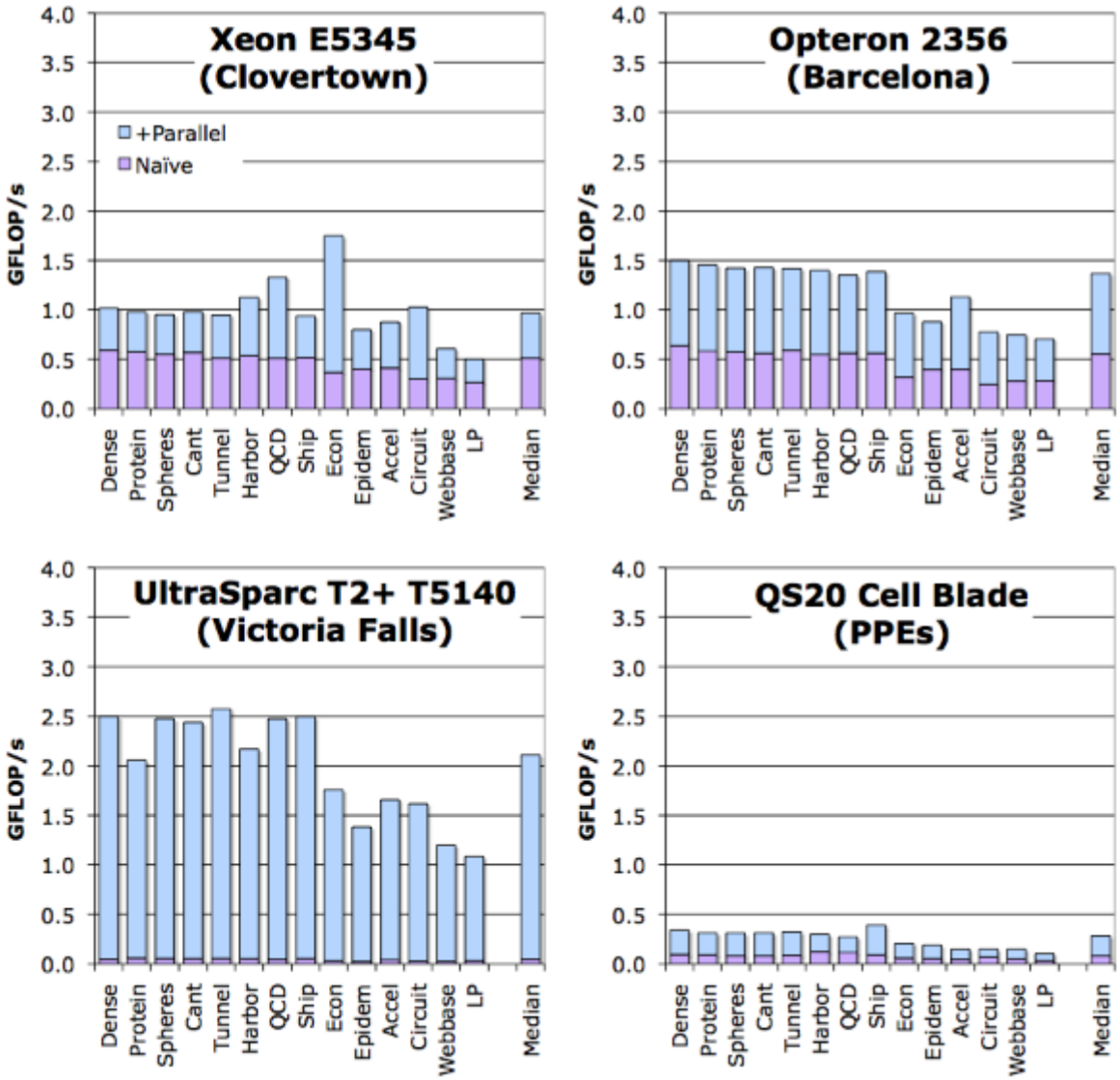
Row parallelism in SpMV

- **y = A*x, where A is a sparse matrix**
  - In iterative solvers, y is often used to compute next x
- **Row parallelism**
  - Random access to x
  - No inter-thread dependences,
  - so no races / locks
  - Any performance issues?



- **Load balancing**
  - Divide number of nonzeros ~evenly, not number of rows
- **Compare to column parallelism (probably in CSC):**
  - Both random access read & write to y
  - 2x bandwidth and need to synchronize
  - But combined row and column gives more potential parallelism

SpMV Performance
(simple parallelization)



- Out-of-the box SpMV performance on a suite of 14 matrices
- Simplest solution = parallelization by rows

- Scalability isn't great
- Can we do better?

Naïve Pthreads

Naïve

Source: Sam Williams

Summary of Multicore Optimizations

- **NUMA - Non-Uniform Memory Access**
  - pin submatrices to memories close to cores assigned to them
- **Prefetch – values, indices, and/or vectors**
  - use exhaustive search on prefetch distance
- **Matrix Compression – not just register blocking (BCSR)**
  - 32 or 16-bit indices, Block Coordinate format for submatrices
- **Cache-blocking**
  - 2D partition of matrix, so needed parts of x,y fit in cache

79

Block Compressed Sparse Row (BCSR)

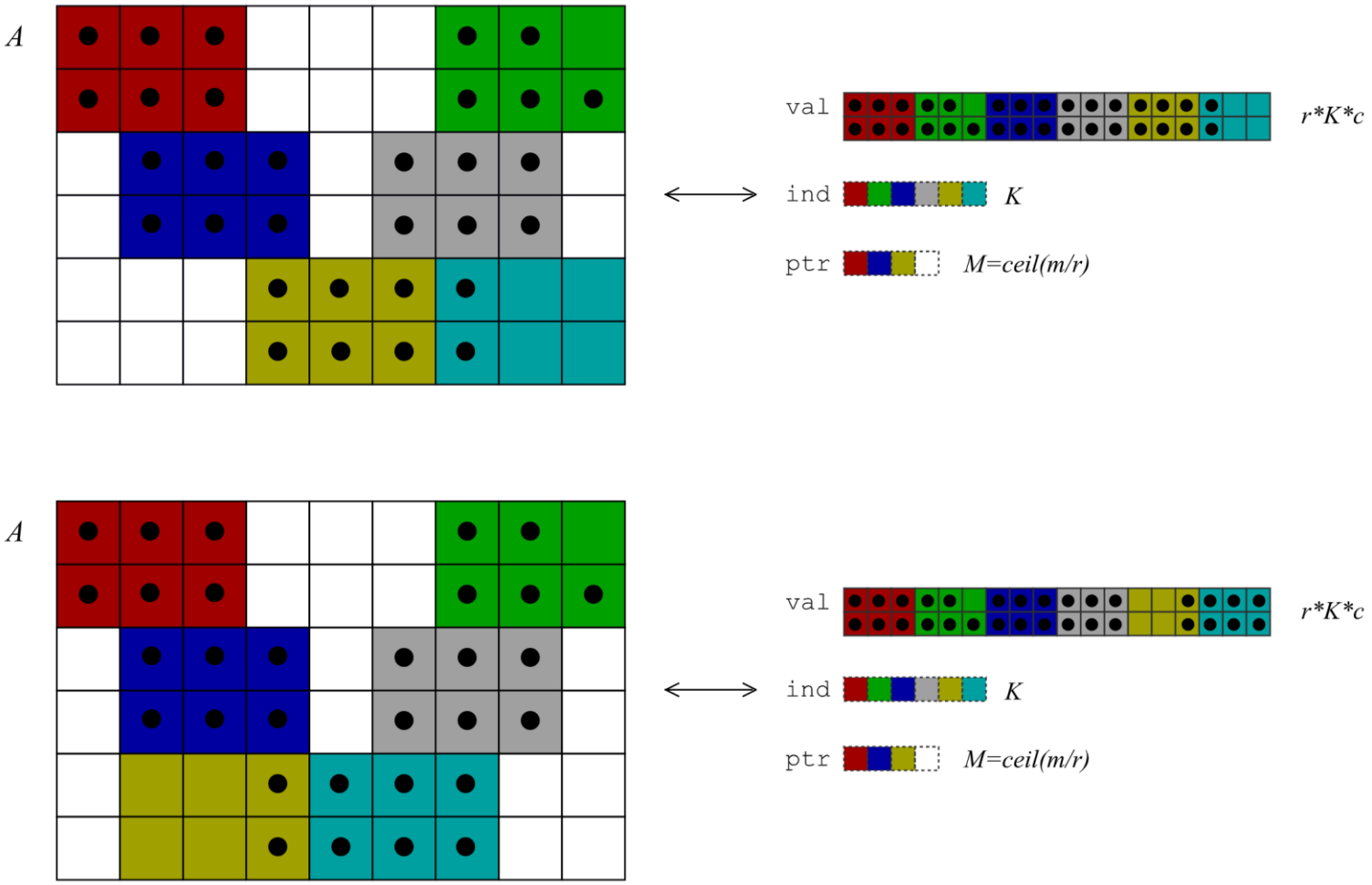
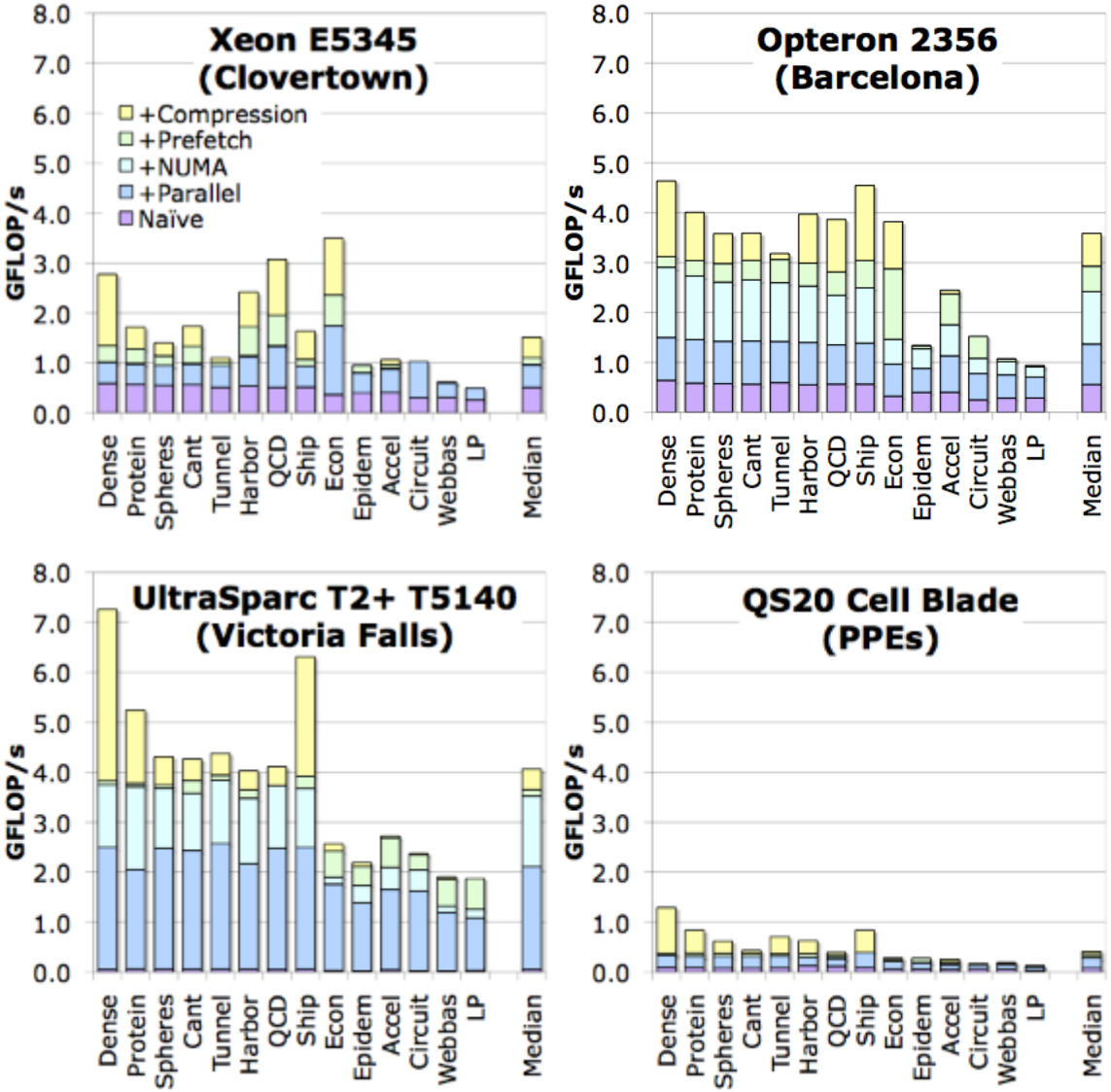


image source: http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf

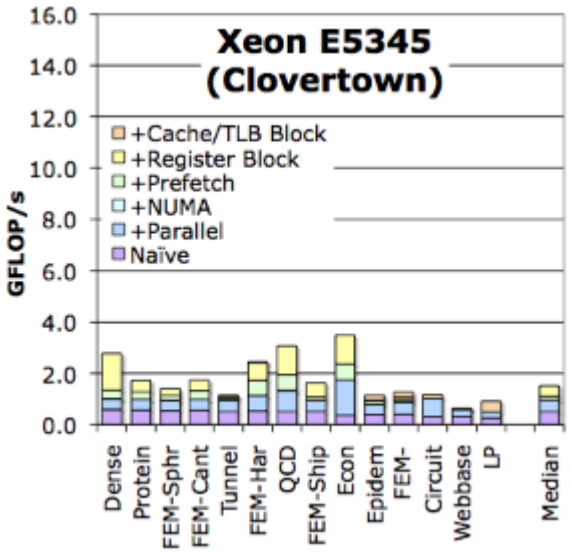
SpMV Performance
(Matrix Compression)

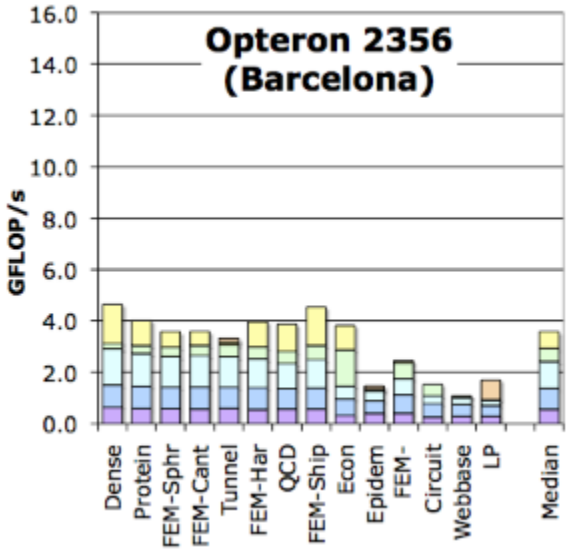


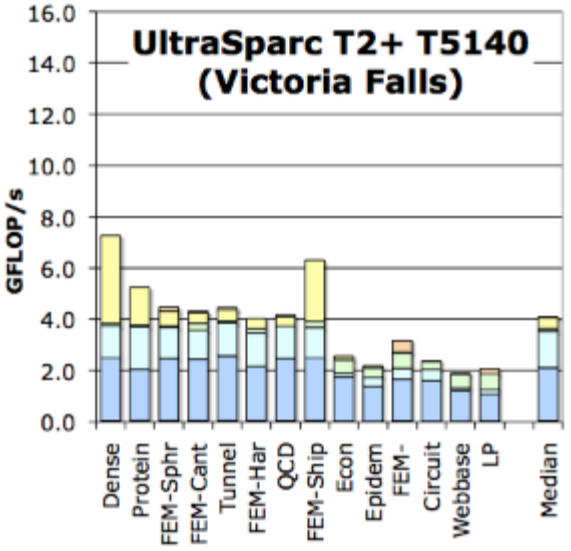
- After maximizing memory bandwidth, the only hope is to minimize memory traffic.
- Compression: exploit
  - register blocking
  - other formats
  - smaller indices
- Use a traffic minimization **heuristic** rather than search
- Benefit is matrix-dependent.
- Register blocking enables efficient software prefetching (one per cache line)

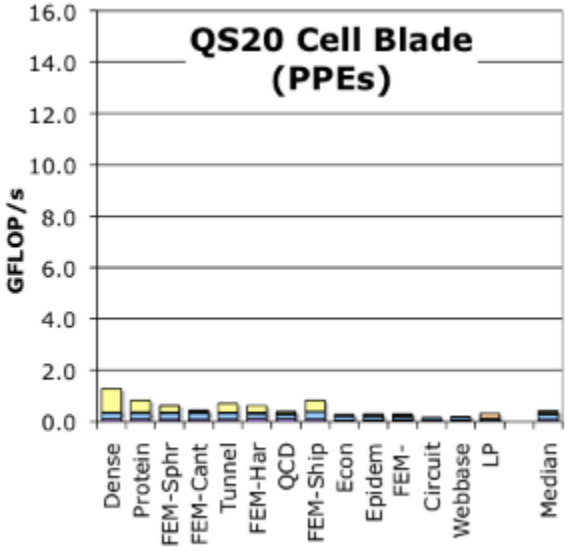
Source: Sam Williams

Auto-tuned SpMV Performance
(cache and TLB blocking)









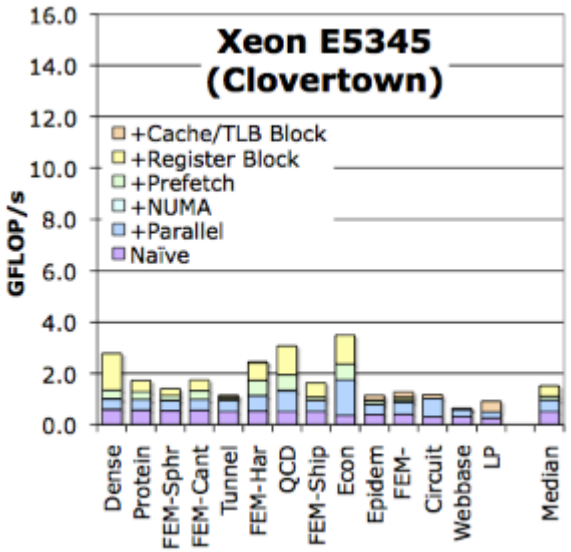
- Fully auto-tuned SpMV performance across the suite of matrices
- Why do some optimizations work better on some architectures?
- matrices with naturally small working sets
- architectures with giant caches

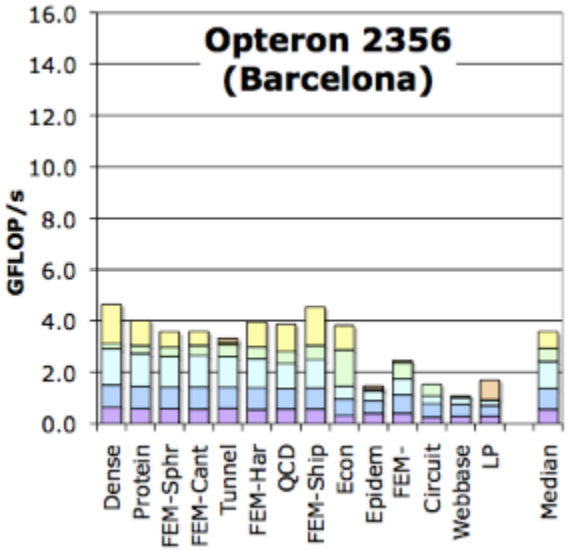
+Cache/LS/TLB Blocking
+Matrix Compression
+SW Prefetching
+NUMA/Affinity
Naïve Pthreads
Naïve

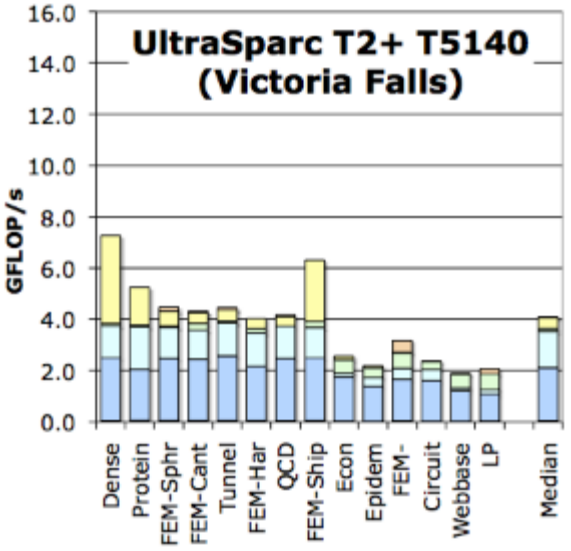
Source: Sam Williams

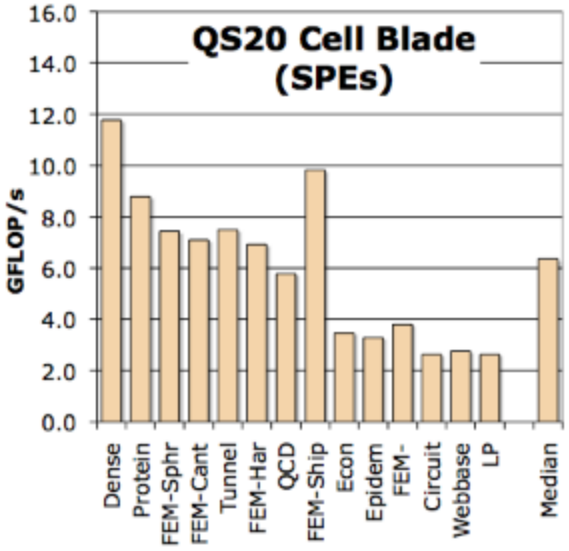
Auto-tuned SpMV Performance

(architecture specific optimizations)

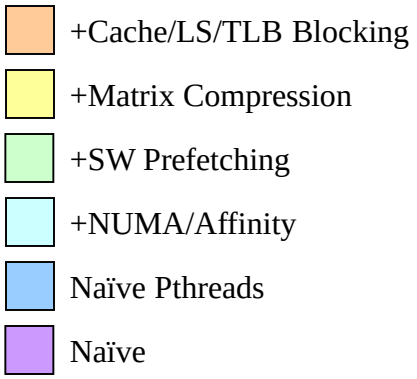






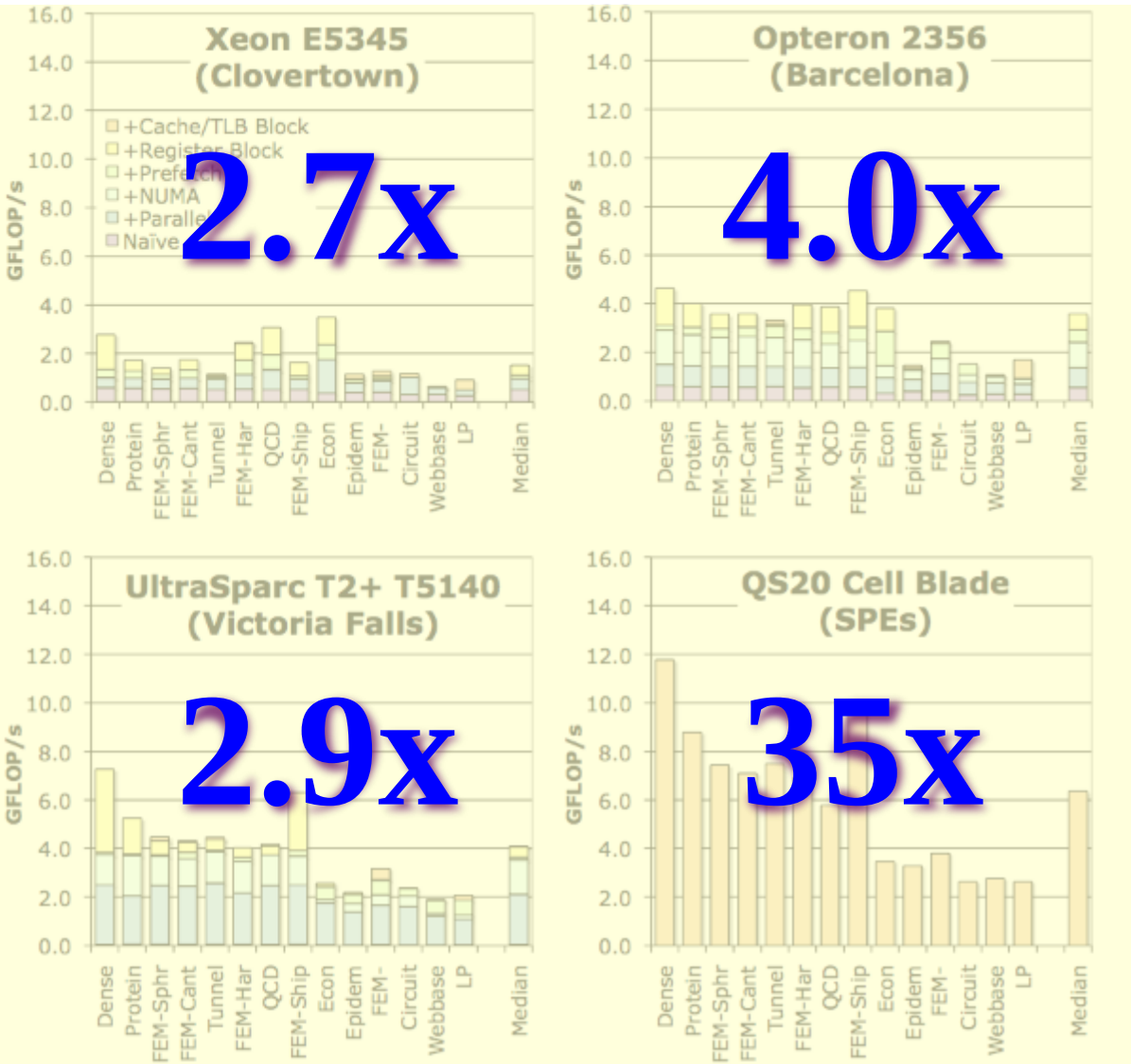


- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?



Source: Sam Williams

Auto-tuned SpMV Performance
(max speedup)



- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?

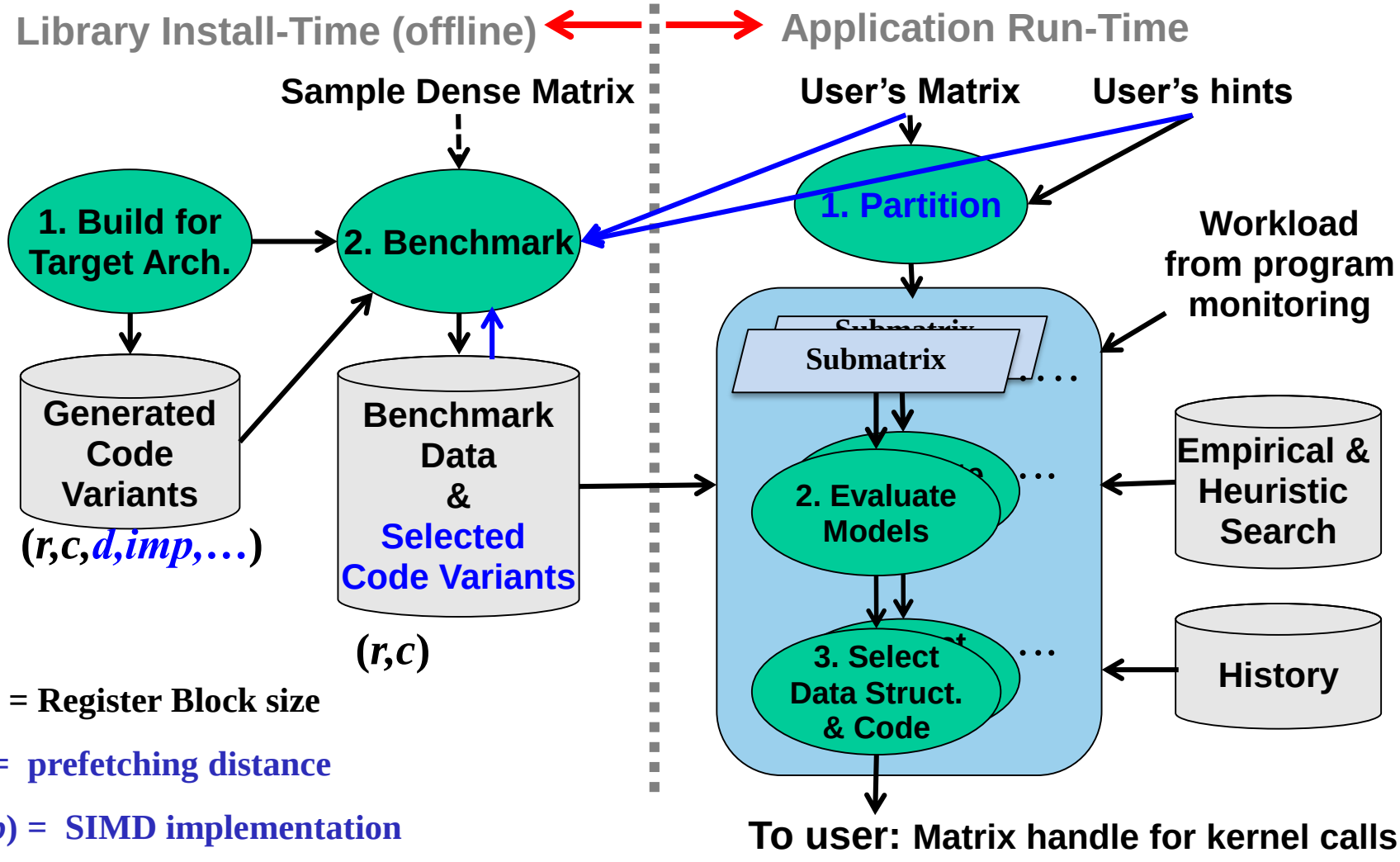
+Cache/LS/TLB Blocking
+Matrix Compression
+SW Prefetching
+NUMA/Affinity
Naïve Pthreads
Naïve

Source: Sam Williams

Optimizations in parallel OSKI (pOSKI)

- **Fully automatic heuristics for**
  - Sparse matrix-vector multiply (Ax, ATx)
    - Register-level blocking, Thread-level blocking
    - SIMD, software prefetching, software pipelining, loop unrolling
    - NUMA-aware allocations
- **"Plug-in" extensibility**
  - Very advanced users may write their own heuristics, create new data structures/code variants and dynamically add them to the system, using embedded scripting language Lua
- **Other optimizations that could be added**
  - Cache-level blocking, Reordering (RCM, TSP), variable block structure, index compressing, Symmetric storage, etc.

How the pOSKI Tunes (Overview)

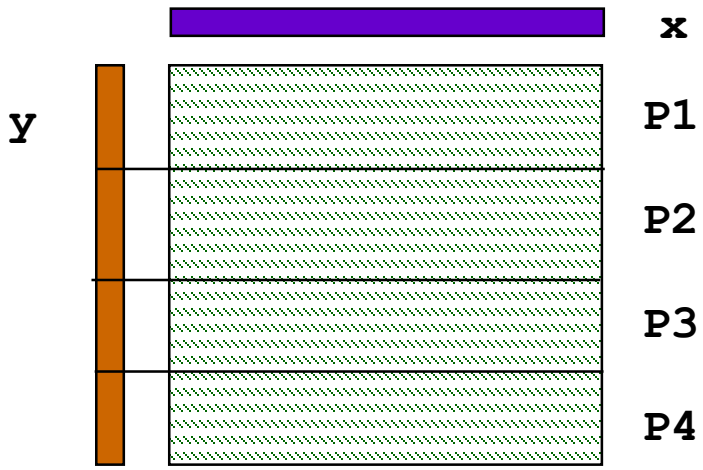


Outline

- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- SpMV on multicore
- Distributed memory SpMV
- Sparse matmult
- CA iterative solvers

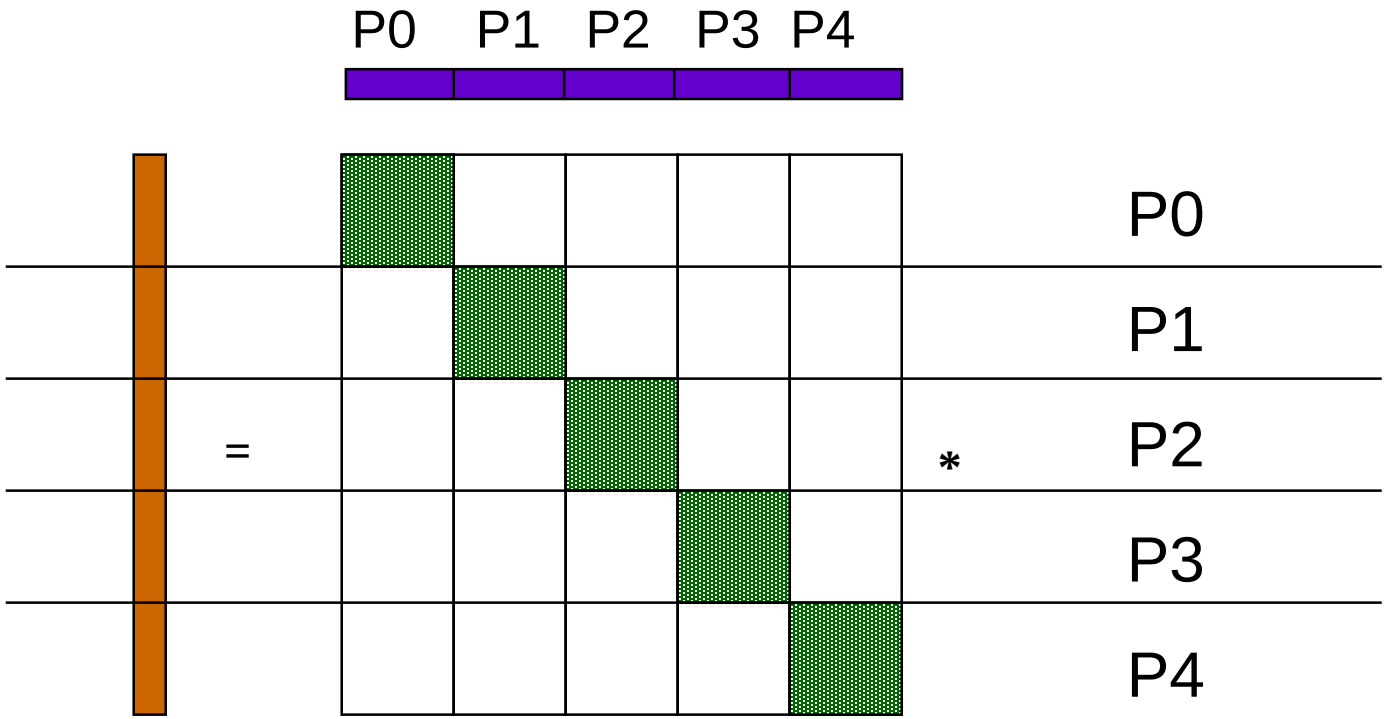
Parallelism in Distributed SpMV

- **y = A*x, where A is a sparse matrix**
- **Row parallelism (y & A partitioned)**
  - Replicate x across processors
  - Or exchange only necessary elements
  - Are nonzeros clustered, e.g., near diagonal?
- **Column parallelism (x & A partitioned)**
  - Make temporary delta_y = [0,…] on all processors;
  - Update that; and add-reduce across processors
- **2D parallelism for large p and when nonzeros uniform**
  - Divide processors into p1 x p2 (e.g., square grid)
  - Hybrid of Row and Column parallelism using teams
  - NAS CG benchmark does this (random nozero pattern)
  - Bad load balance for clustered nonzeros



Matrix Reordering via Graph Partitioning

- **"Ideal" matrix structure for parallelism: block diagonal**
  - p (number of processors) blocks, can all be computed locally.
  - If no non-zeros outside these blocks, no communication needed
- **Can we reorder the rows/columns to get close to this?**
  - Most nonzeros in diagonal blocks, few outside

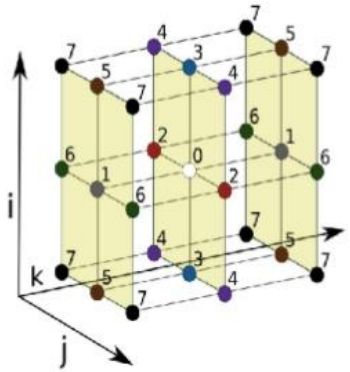


HPCG Benchmark

http://tiny.cc/hpcg

Model Problem Description

- Synthetic discretized 3D PDE (FEM, FVM, FDM).
- Single DOF heat diffusion model.
- Zero Dirichlet BCs, Synthetic RHS s.t. solution = 1.
- Local domain: $(n_x \times n_y \times n_z)$
- Process layout: $(np_x \times np_y \times np_z)$
- Global domain: $(n_x * np_x) \times (n_y * np_y) \times (n_z * np_z)$
- Sparse matrix:
  - 27 nonzeros/row interior.
  - 7 – 18 on boundary.
  - Symmetric positive definite.



27-point stencil operator

HPCG Results (Nov 2017)

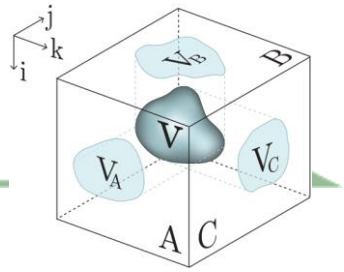
Rank	Site	Computer	Cores	HPL (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Advanced Institute for Computational Science, Japan	**K computer** – , SPARC64 VIIIfx 2.0GHz, Tofu interconnect, Fujitsu	705,024	10.51	10	0.603	5.30%
2	NSCC / Guangzhou, China	**Tianhe-2 (MilkyWay-2)** – TH-IVB-FEP Cluster, Intel Xeon 12C 2.2GHz, TH Express 2, Intel Xeon Phi 31S1P 57-core, NUDT	3,120,000	33.863	2	0.58	1.10%
3	DOE/NNSA/LANL/SNL, USA	**Trinity** – Cray XC40, Intel Xeon E5-2698 v3 300160C 2.3GHz, Aries, Cray	979,072	14.137	7	0.546	1.80%
4	Swiss National Supercomputing Centre (CSCS), Switzerland	**Piz Daint** – Cray XC50, Intel Xeon E5-2690v3 12C 2.6GHz, Cray Aries, NVIDIA Tesla P100 16GB, Cray	361,760	19.59	3	0.486	1.90%
5	National Supercomputing Center in Wuxi, China	**Sunway TaihuLight** – Sunway MPP, SW26010 260C 1.45GHz, Sunway, NRCPC	10,649,600	93.015	1	0.481	0.40%
6	Joint Center for Advanced High Performance Computing, Japan	**Oakforest-PACS** – PRIMERGY CX600 M1, Intel Xeon Phi Processor 7250 68C 1.4GHz, Intel Omni-Path Architecture, Fujitsu	557,056	13.555	9	0.385	1.50%
7	DOE/SC/LBNL/NERSC, USA	**Cori** – XC40, Intel Xeon Phi 7250 68C 1.4GHz, Cray Aries, Cray	632,400	13.832	8	0.355	1.30%
8	DOE/NNSA/LLNL, USA	**Sequoia** – IBM BlueGene/Q, PowerPC A2 1.6 GHz 16-core, 5D Torus, IBM	1,572,864	17.173	6	0.33	1.60%
9	DOE/SC/Oak Ridge Nat Lab, USA	**Titan** – Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x, Cray	560,640	17.59	5	0.322	1.20%
10	GSIC Center, Tokyo Institute of Technology, Japan	**TSUBAME3.0** – SGI ICE XA, Intel Xeon E5-2680 2.9GHz, Intel Omni-Path, NVIDIA TESLA P100 SXM2 with NVLink, HPE	136,080	8.125	13	0.189	1.60%

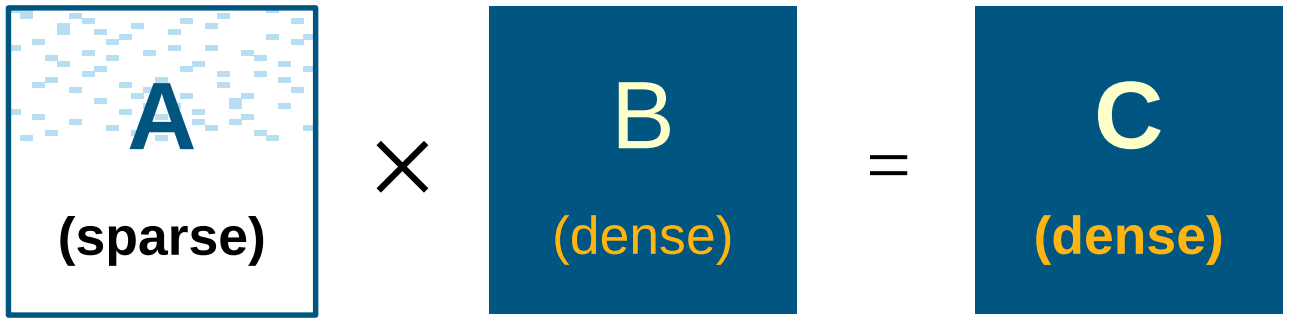
Outline

- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- Cache blocking SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

Sparse ⨯ Dense Matmul

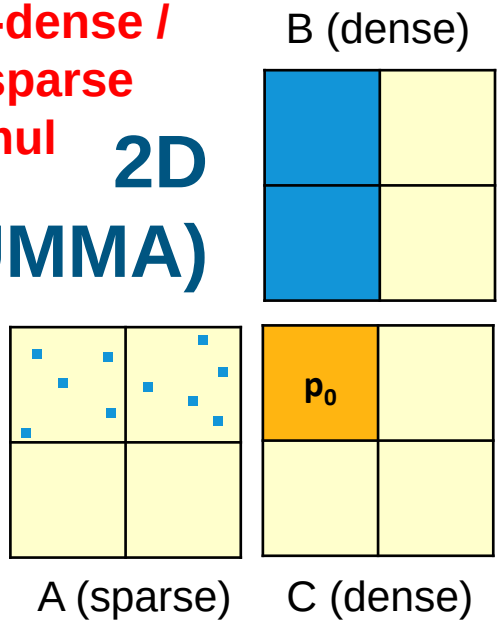
These are not necessarily square



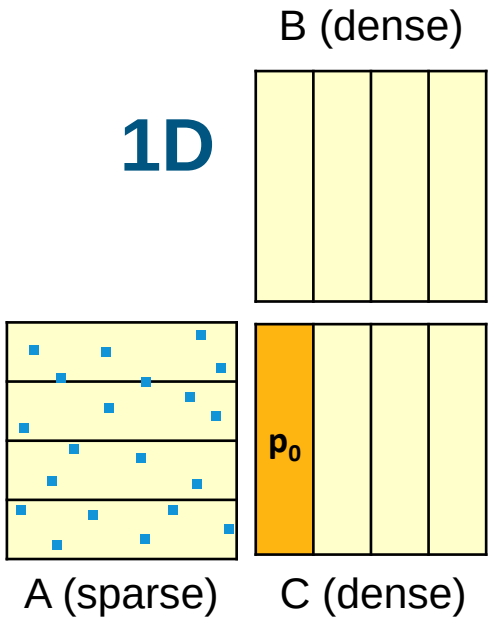


2D/2.5D/3D only optimal for dense-dense / sparse-sparse matmul

2D (SUMMA)



1D



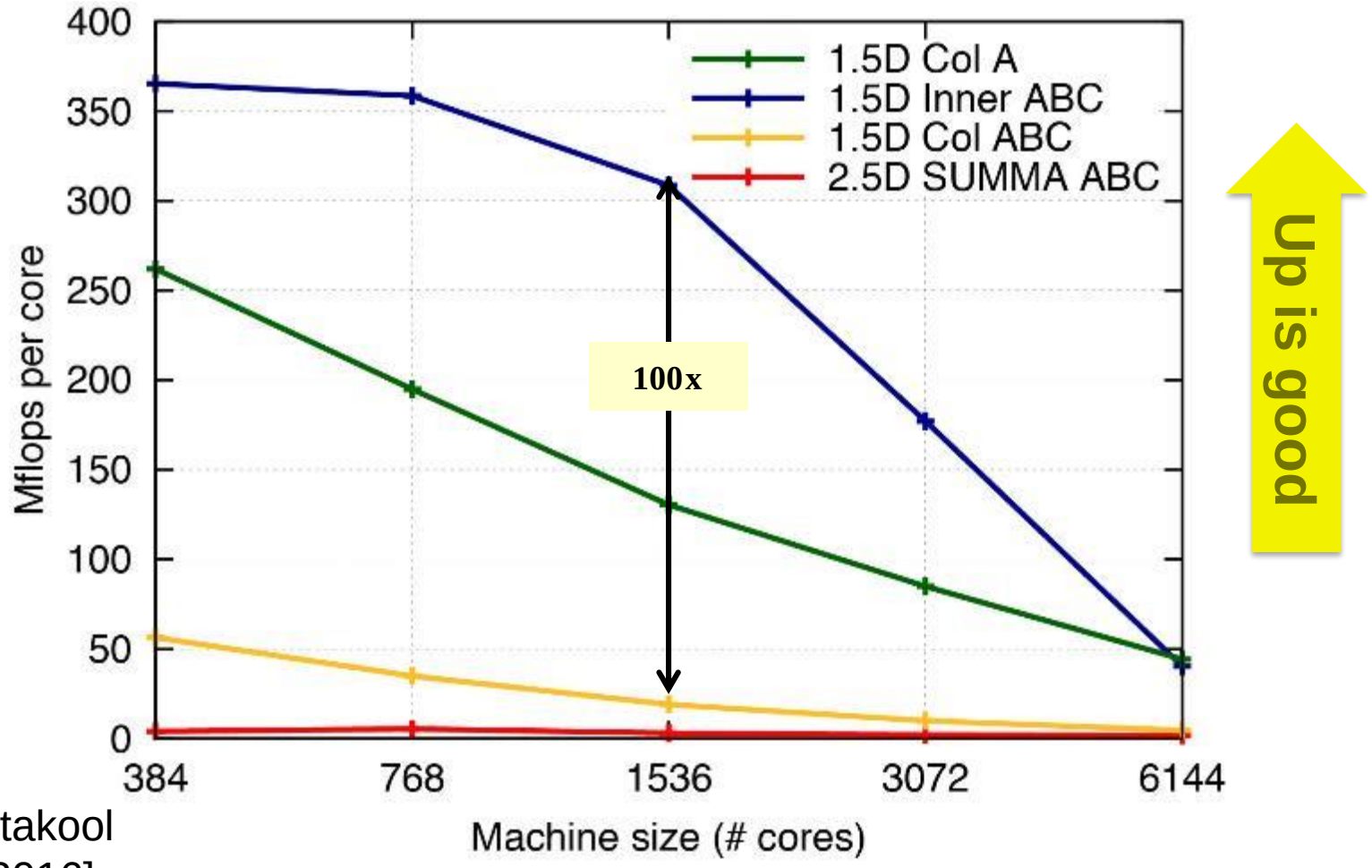
+ replication

= 1.5D

[Koanantakool et al. 2016]

100x Improvement for the right algorithm

- $A^{66k \times 172k}$, $B^{172k \times 66k}$, 0.0038% nnz, Cray XC30

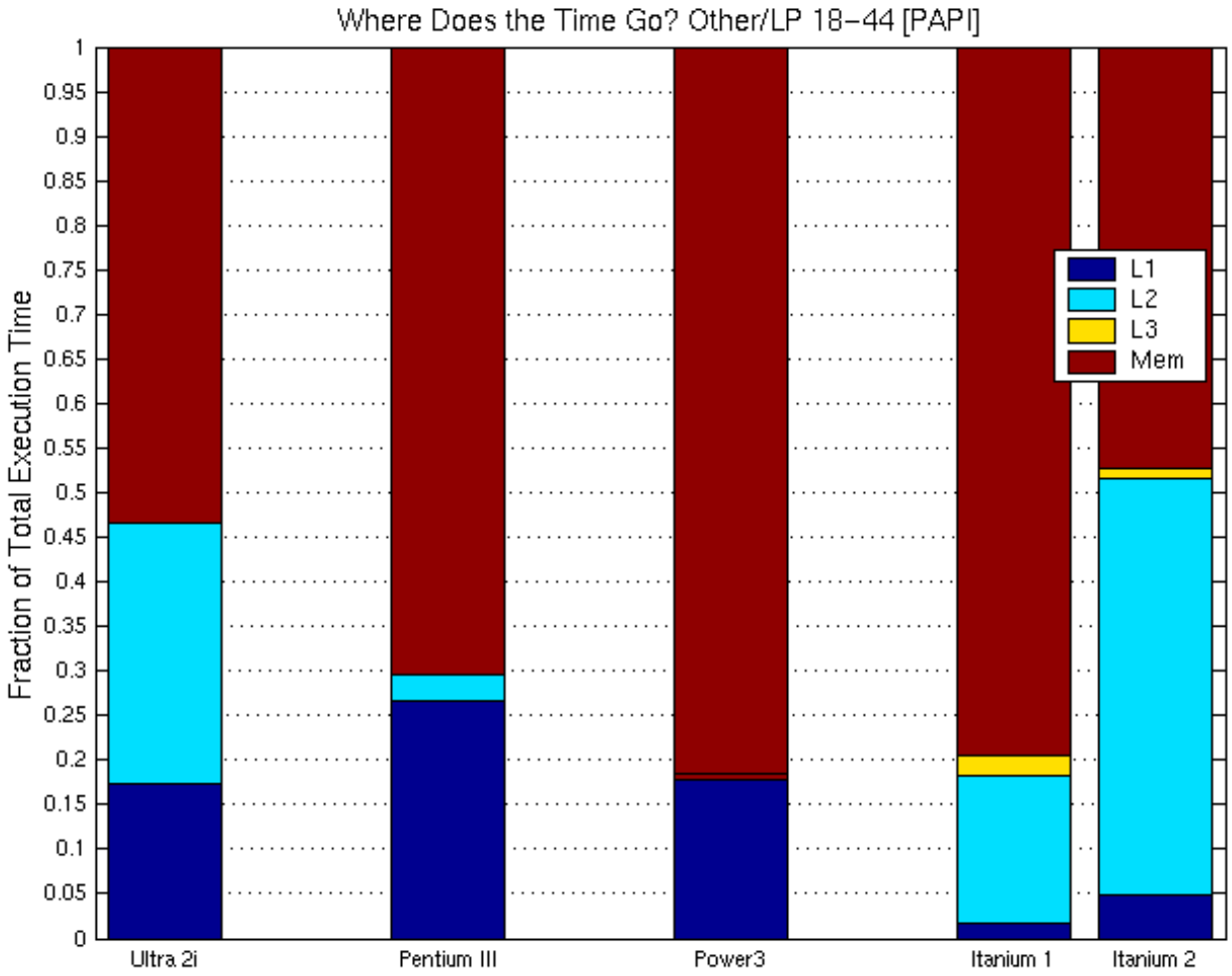


[Koanantakool et al. 2016]

Outline

- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- Cache blocking SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

Execution Time Breakdown in SpMV



Matrix 40
(PAPI counters)

Is tuning SpMV all we can do?

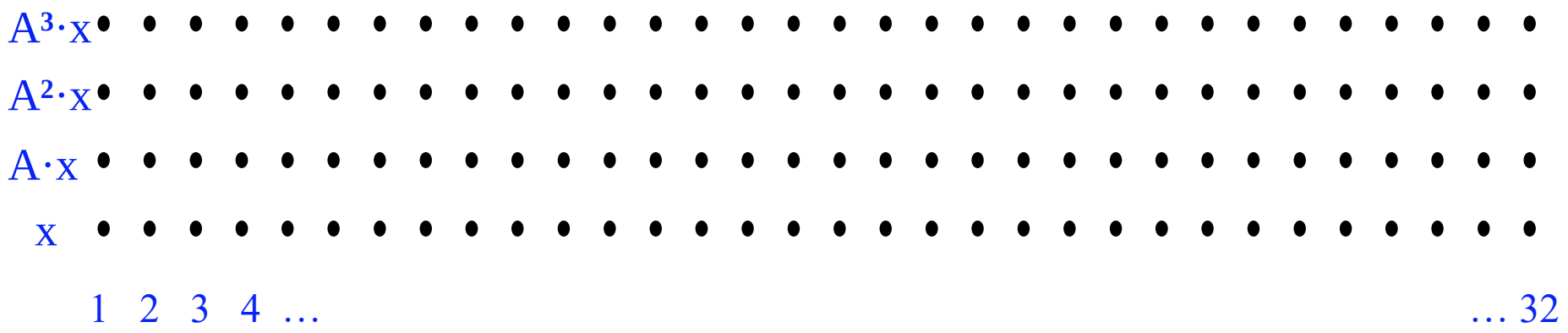
- **Iterative methods all depend on it**
- **But speedups are limited**
  - Just 2 flops per nonzero
  - Communication costs dominate
- **Can we beat this bottleneck?**
- **Need to look at next level in stack:**
  - What do algorithms that use SpMV do?
  - Can we reorganize them to avoid communication?
- **Only way significant speedups will be possible**

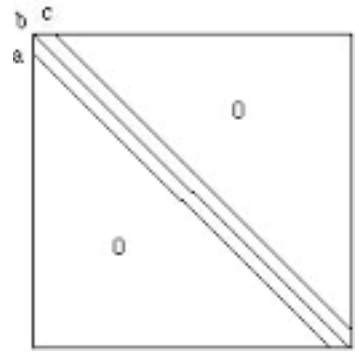
Tuning Higher Level Algorithms than SpMV

- **We almost always do many SpMVs, not just one**
  - "Krylov Subspace Methods" (KSMs) for Ax=b, Ax = λx
    - Conjugate Gradients, GMRES, Lanczos, …
  - Do a sequence of k SpMVs to get vectors [x1 , … , xk]
  - Find best solution x as linear combination of [x1 , … , xk]
- **Main cost is k SpMVs**
- **Since communication usually dominates, can we do better?**
- **Goal: make communication cost independent of k**
  - Parallel case: O(log P) messages, not O(k log P) - optimal
    - same bandwidth as before
  - Sequential case: O(1) messages and bandwidth, not O(k) - optimal
- **Achievable when matrix partitionable with**
- **low surface-to-volume ratio**

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]

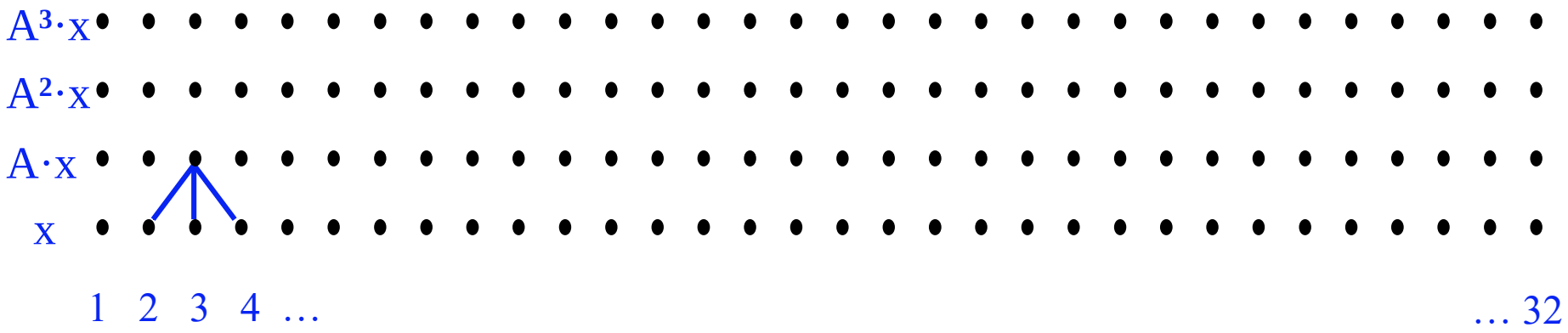




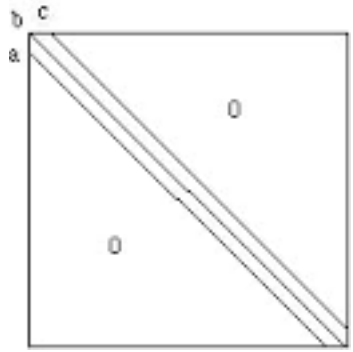
- Example: A tridiagonal, n=32, k=3
- Works for any “well-partitioned” A

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]

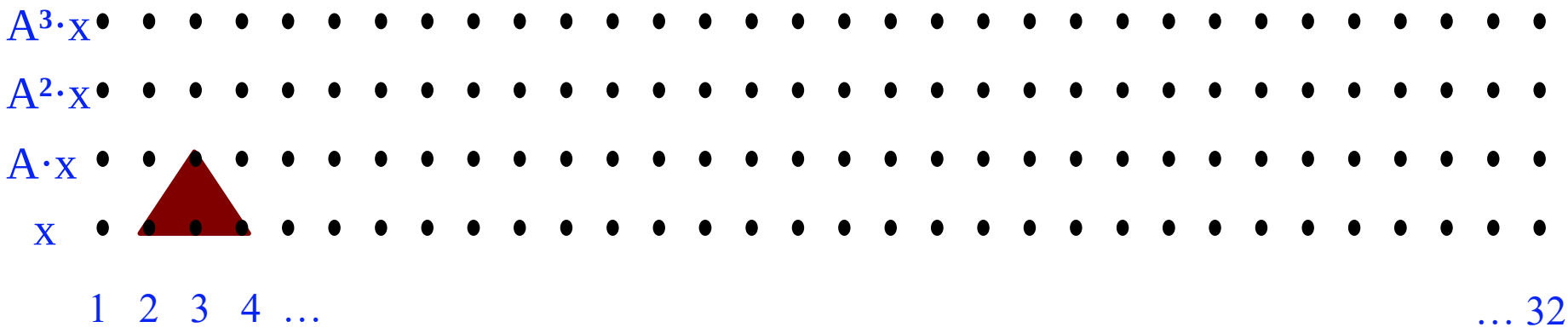


- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

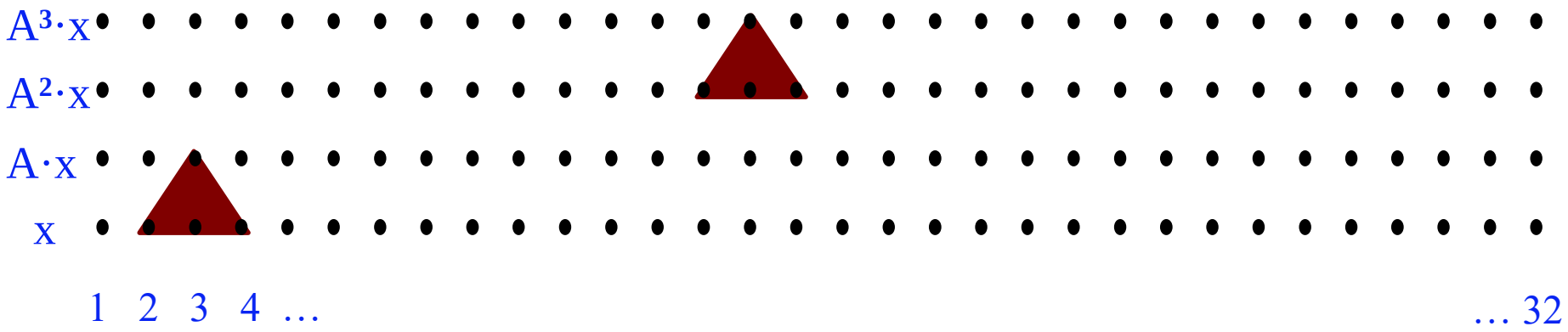
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

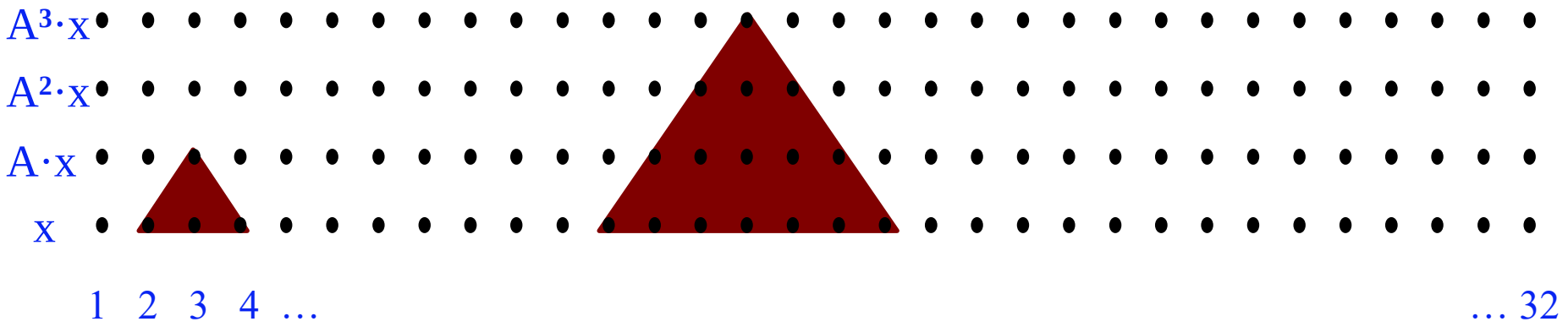
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

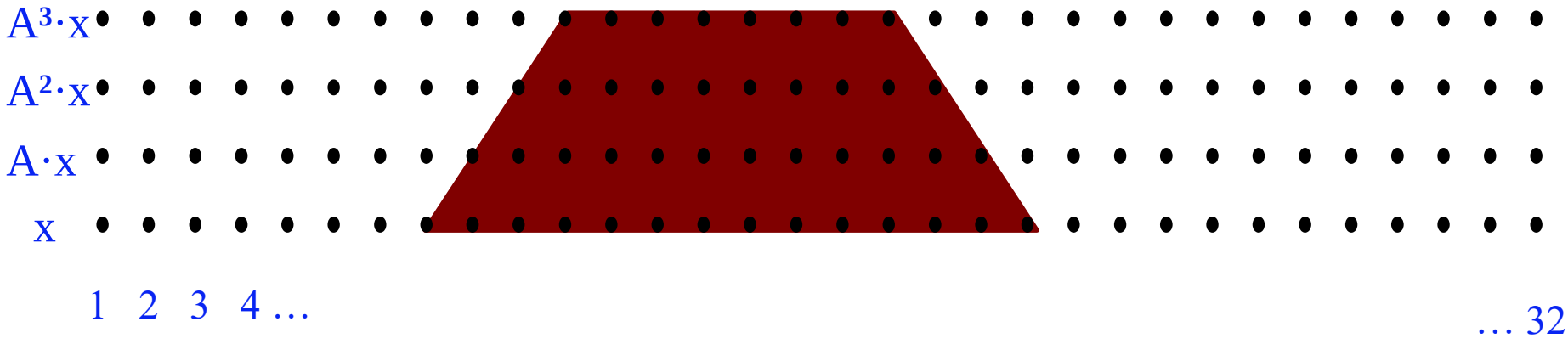
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

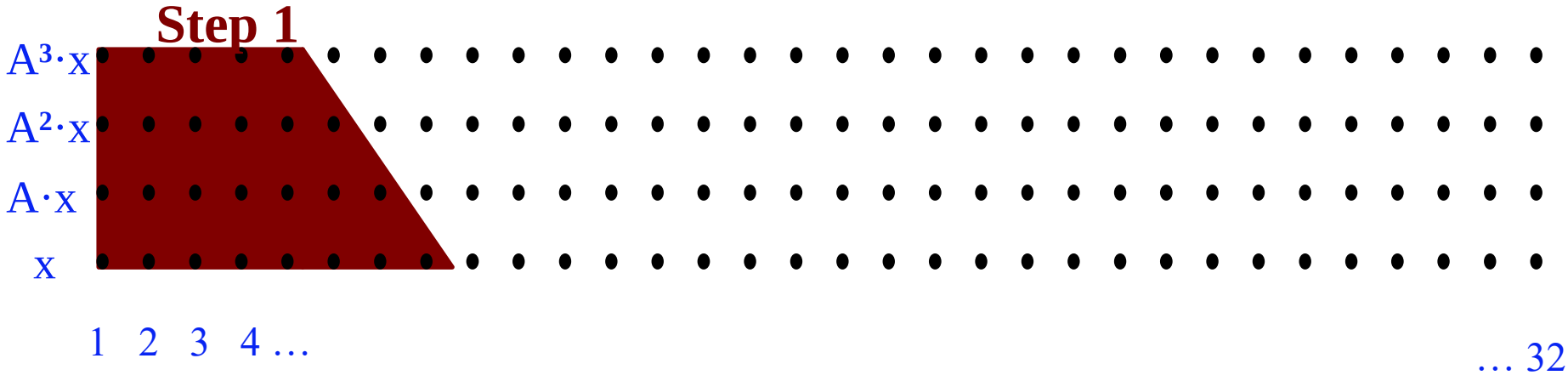
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

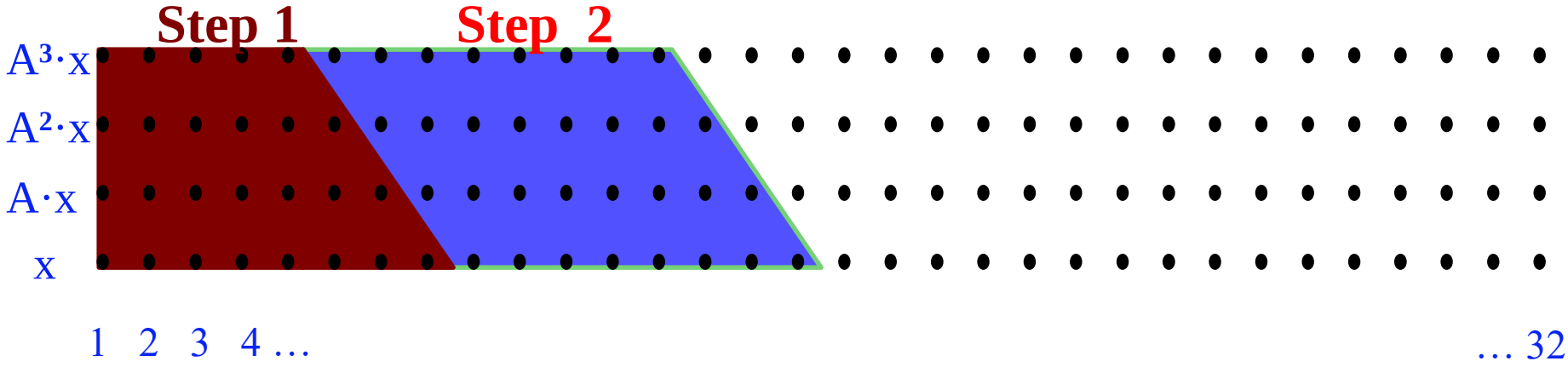
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Sequential Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

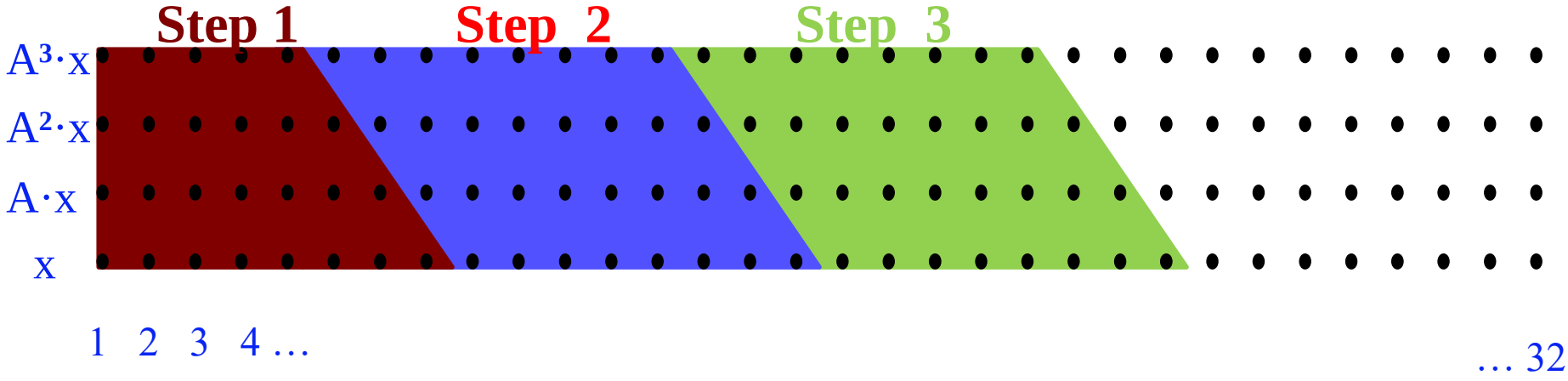
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Sequential Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

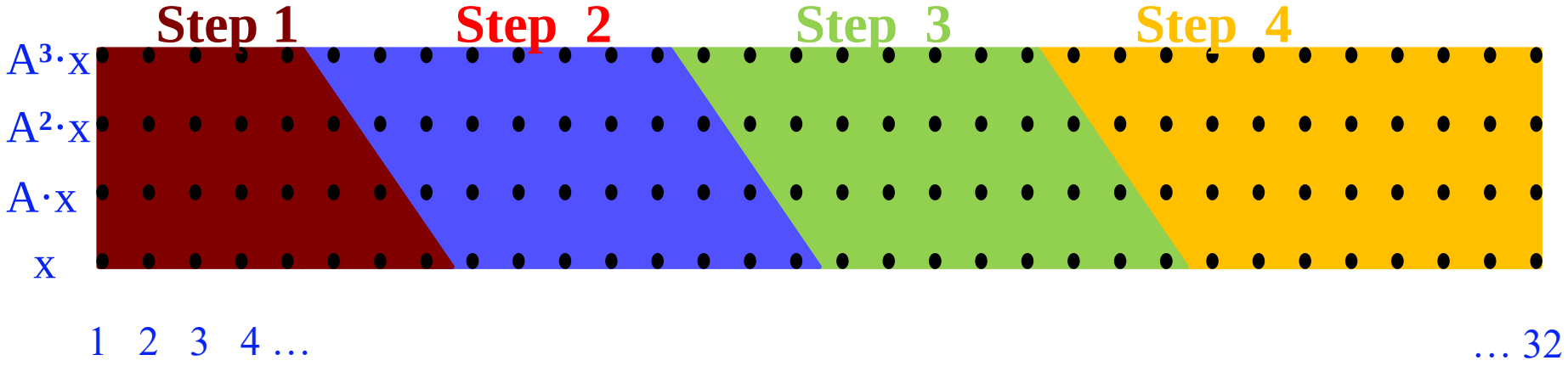
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Sequential Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

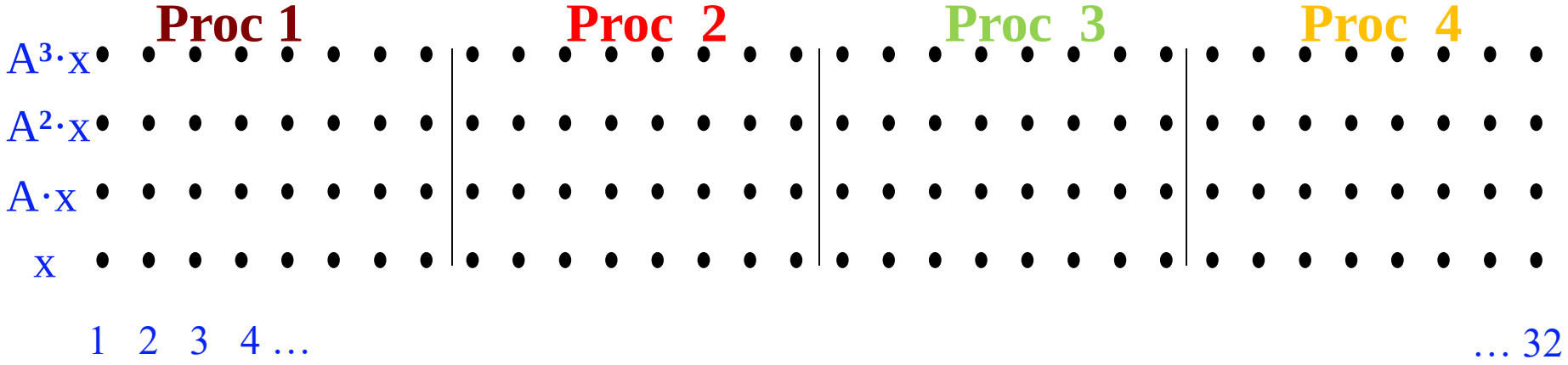
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Sequential Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

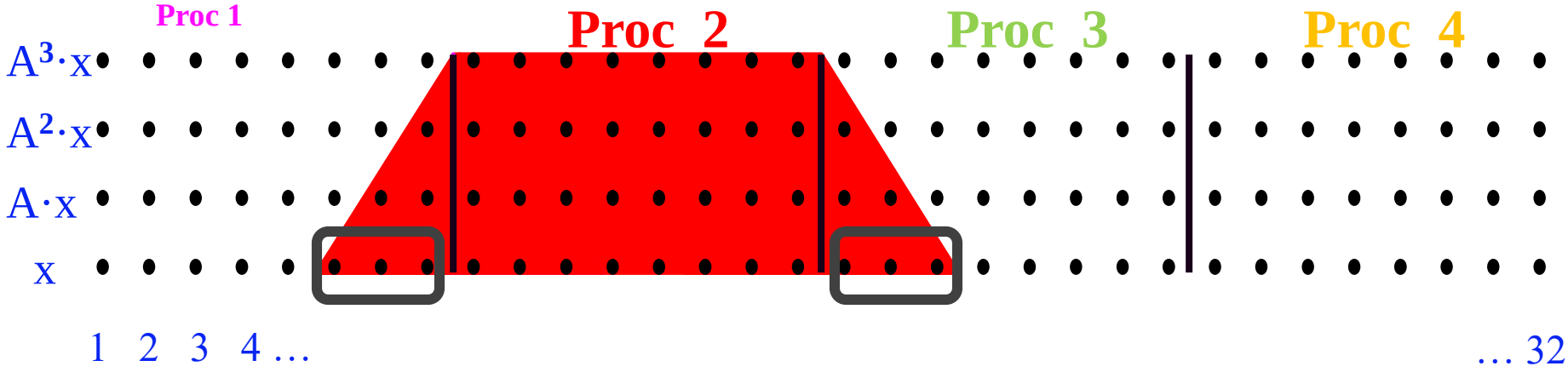
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Parallel Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

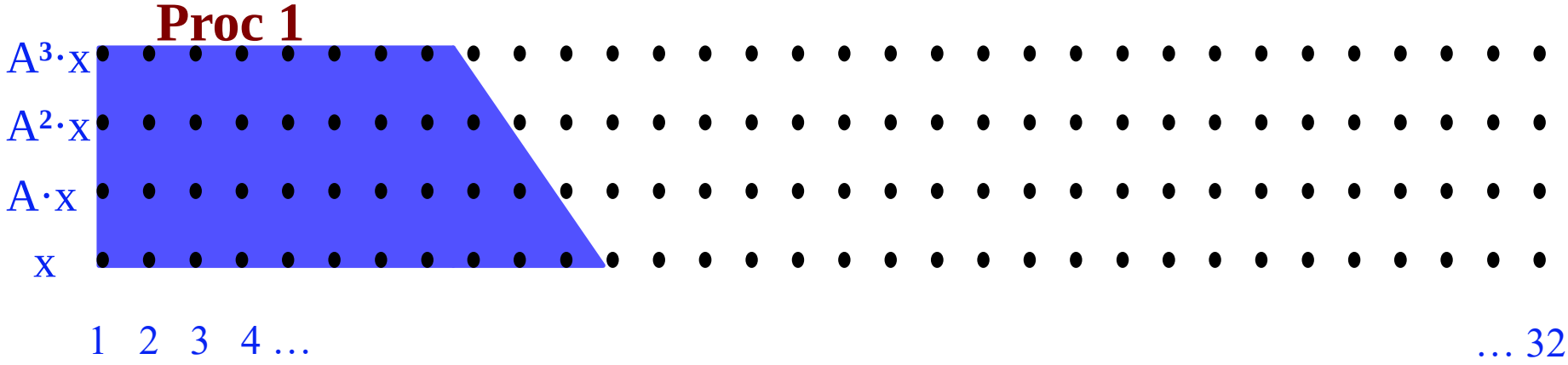
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Parallel Algorithm
- Example: A tridiagonal, n=32, k=3
- Each processor communicates once with neighbors

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

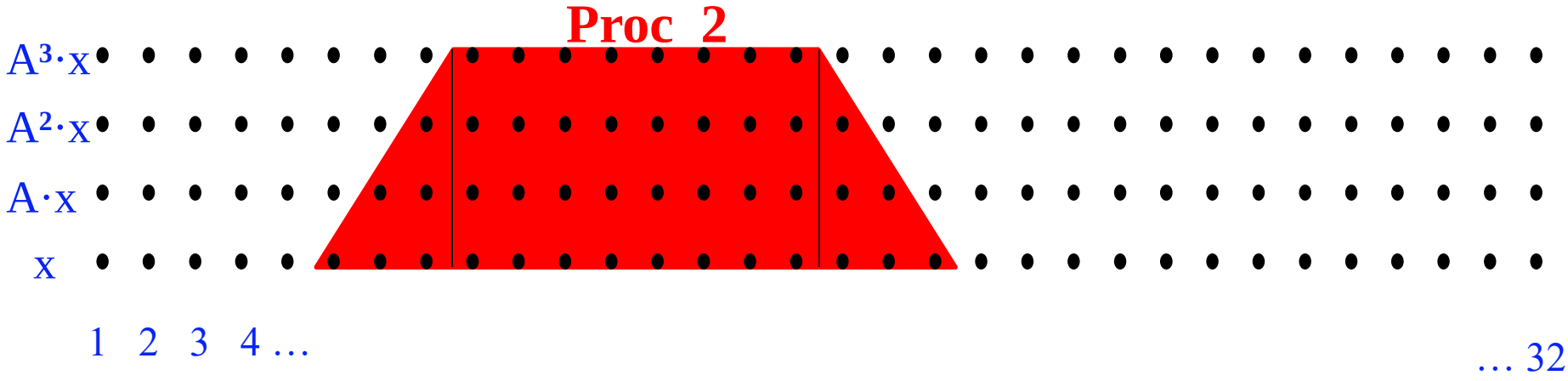
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Parallel Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

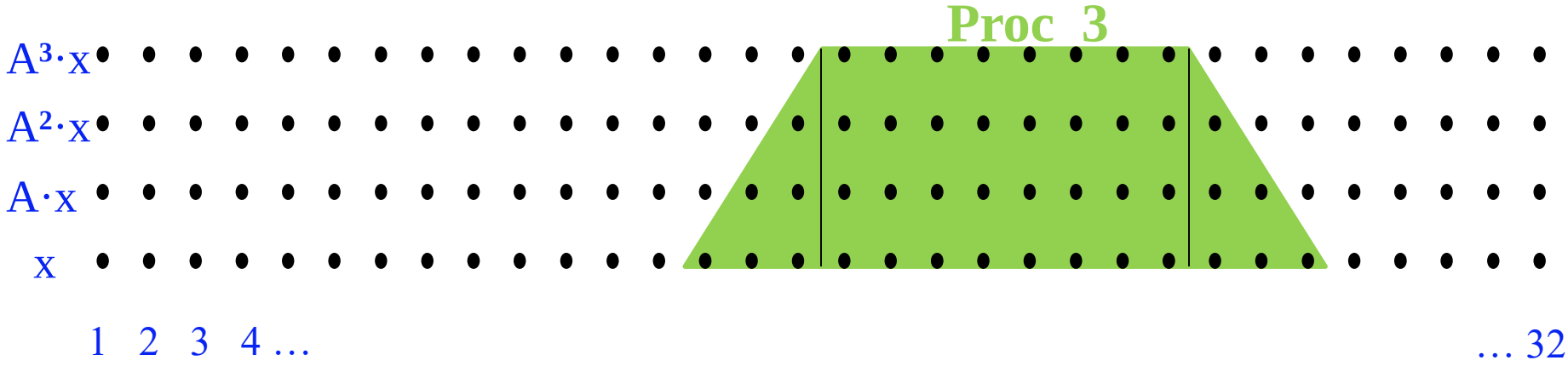
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Parallel Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

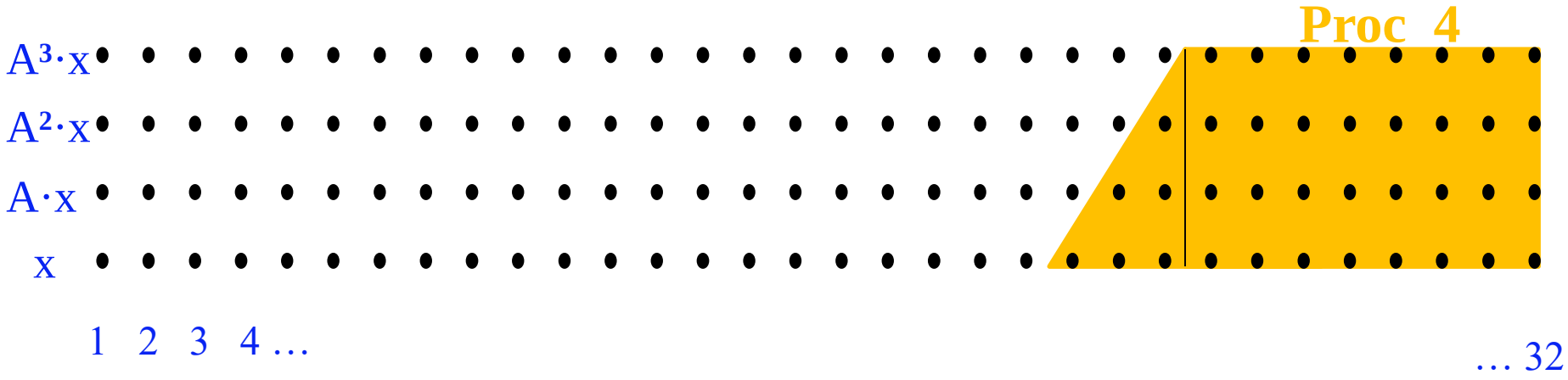
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Parallel Algorithm
- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

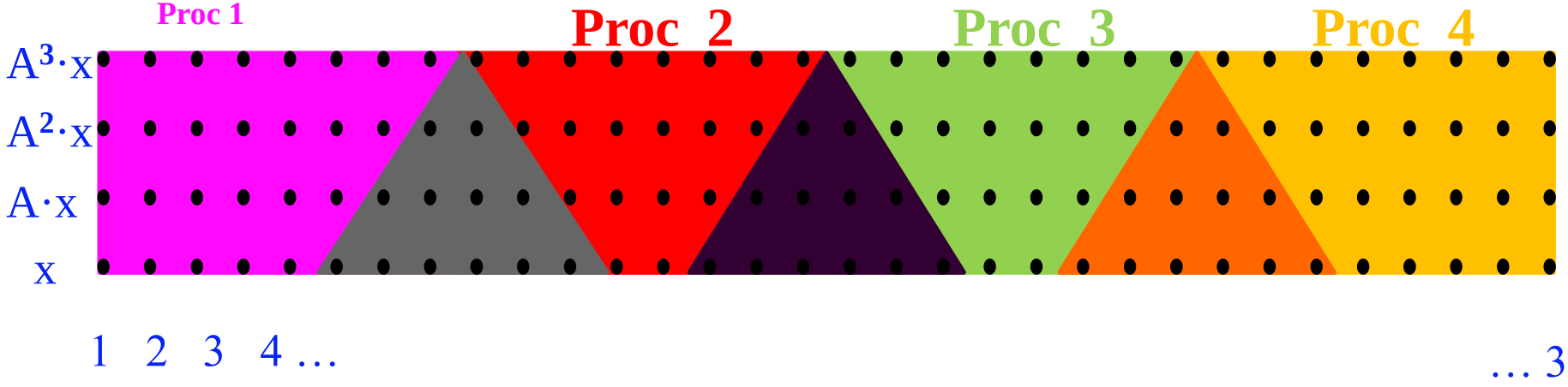
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Parallel Algorithm
- Example: A tridiagonal, n=32, k=3

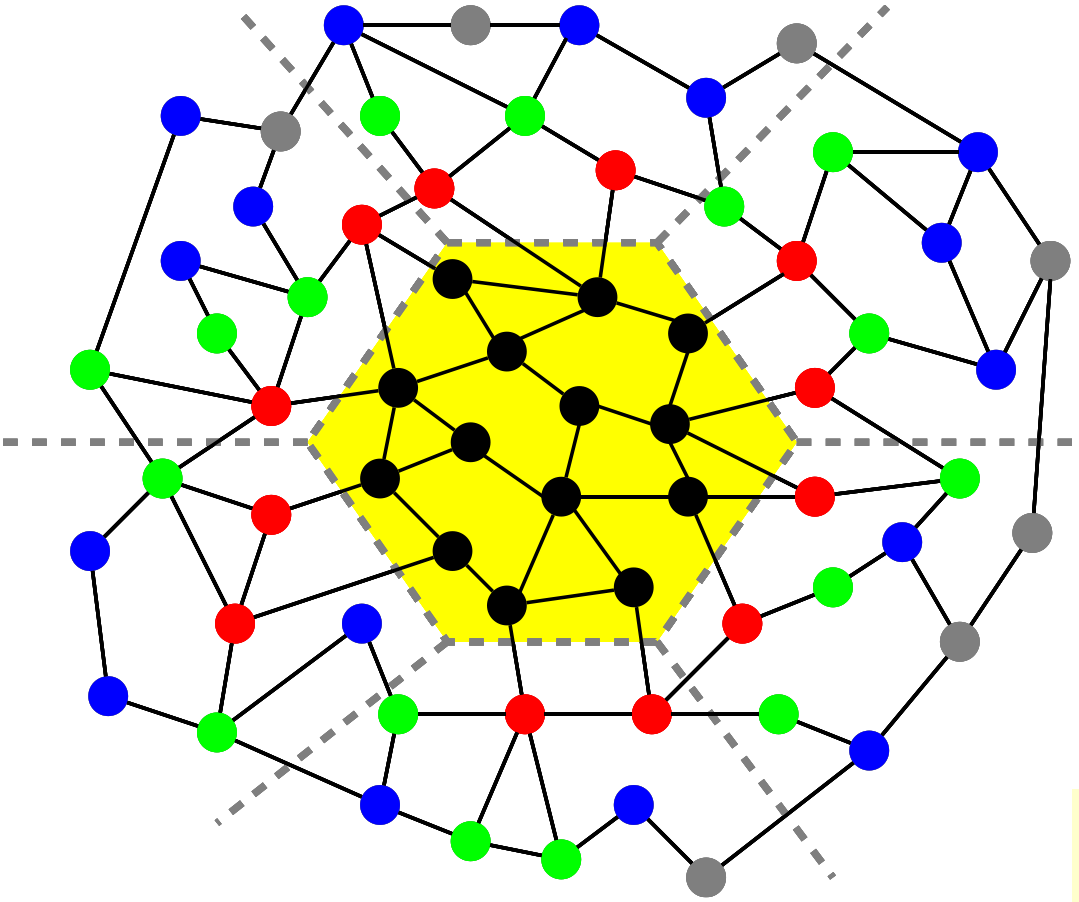
Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- Parallel Algorithm
- Example: A tridiagonal, n=32, k=3
- Each processor works on (overlapping) trapezoid

Matrix Powers Kernel on a General Matrix

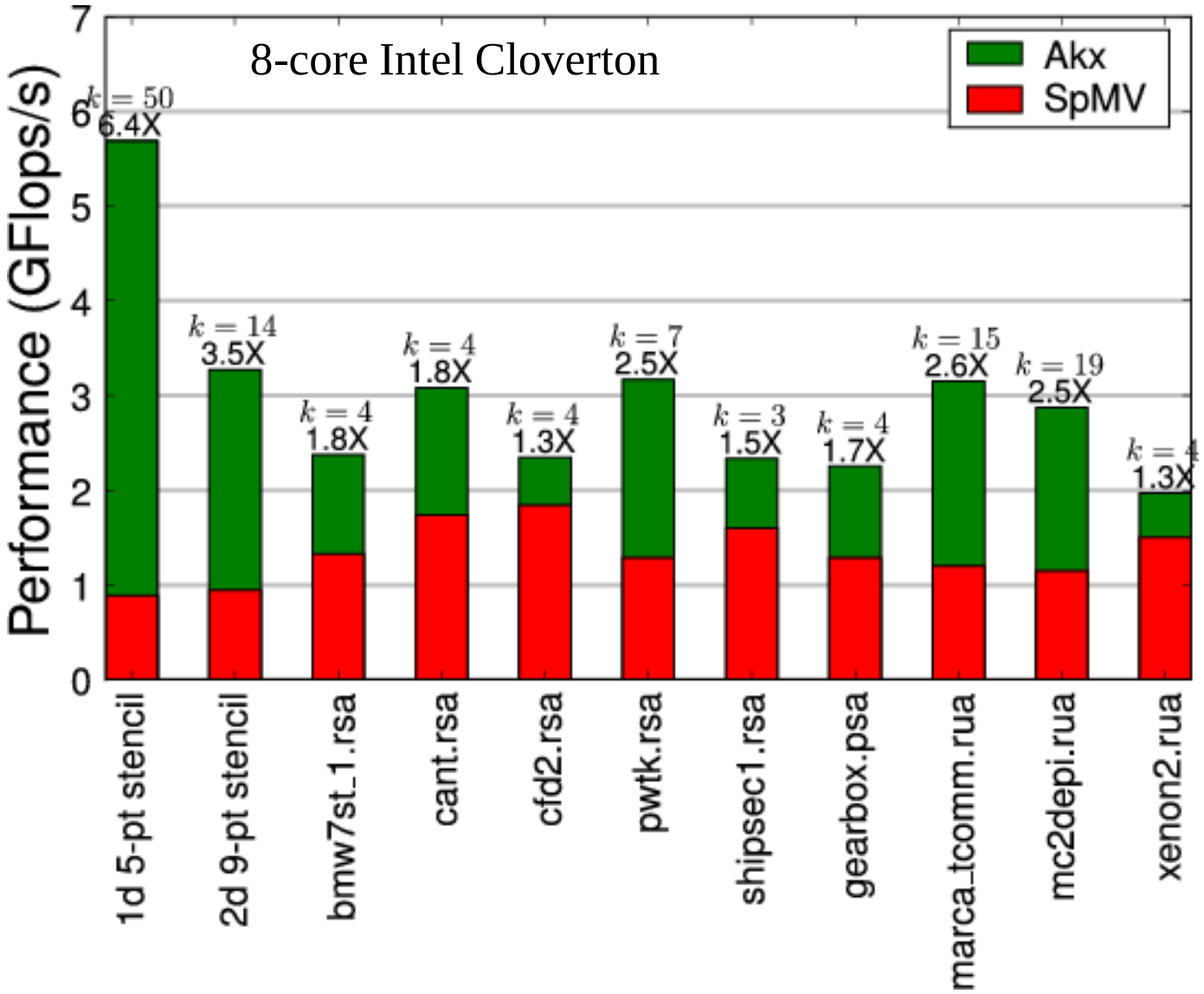


- *Need hypergraph partitioning*
- *For implicit memory management (caches) uses a TSP algorithm for layout*

See paper by Jim Demmel, Mark Hoemman, Marghoob Mohiyuddin, Kathy Yelick

- **Saves communication for "well partitioned" matrices**
  - **Serial memory bandwidth: O(1) moves of data moves vs. O(k)**
  - **Parallel message latency: O(log p) messages vs. O(k log p)**

Multicore Speedups



Performance Results

- Measured Multicore (Clovertown) speedups up to 6.4x
- Measured/Modeled sequential OOC speedup up to 3x
- Modeled parallel Petascale speedup up to 6.9x
- Modeled parallel Grid speedup up to 22x

- Sequential speedup due to bandwidth, works for many problem sizes
- Parallel speedup due to latency, works for smaller problems on many processors
- Multicore results used both techniques

Avoiding Communication in Iterative Linear Algebra

- **k-steps of typical iterative solver for sparse Ax=b or Ax=λx**
  - Does k SpMVs with starting vector
  - Finds "best" solution among all linear combinations of these k+1 vectors
  - Many such "Krylov Subspace Methods"
    - Conjugate Gradients, GMRES, Lanczos, Arnoldi, …
- **Goal: minimize communication in Krylov Subspace Methods**
  - Assume matrix "well-partitioned," with modest surface-to-volume ratio
  - Parallel implementation
    - Conventional: O(k log p) messages, because k calls to SpMV
    - New: O(log p) messages - optimal
  - Serial implementation
    - Conventional: O(k) moves of data from slow to fast memory
    - New: O(1) moves of data – optimal
- **Lots of speed up possible (modeled and measured)**
  - Price: some redundant computation
- **Much prior work**
  - See theses of Mark Hoemmen, Erin Carson, other papers at bebop.cs.berkeley.edu

Minimizing Communication of GMRES to solve Ax=b

- **GMRES: find x in span{b,Ab,…,Akb} minimizing || Ax-b ||2**
- **Cost of k steps of standard GMRES vs new GMRES**

Standard GMRES

```
for i=1 to k
  w = A · v(i-1)
  MGS(w, v(0),…,v(i-1))
  update v(i), H
endfor
solve LSQ problem with H
```

Sequential: #words_moved =
O(k·nnz) from SpMV
+ O(k^2·n) from MGS
Parallel: #messages =
O(k) from SpMV
+ O(k^2 · log p) from MGS

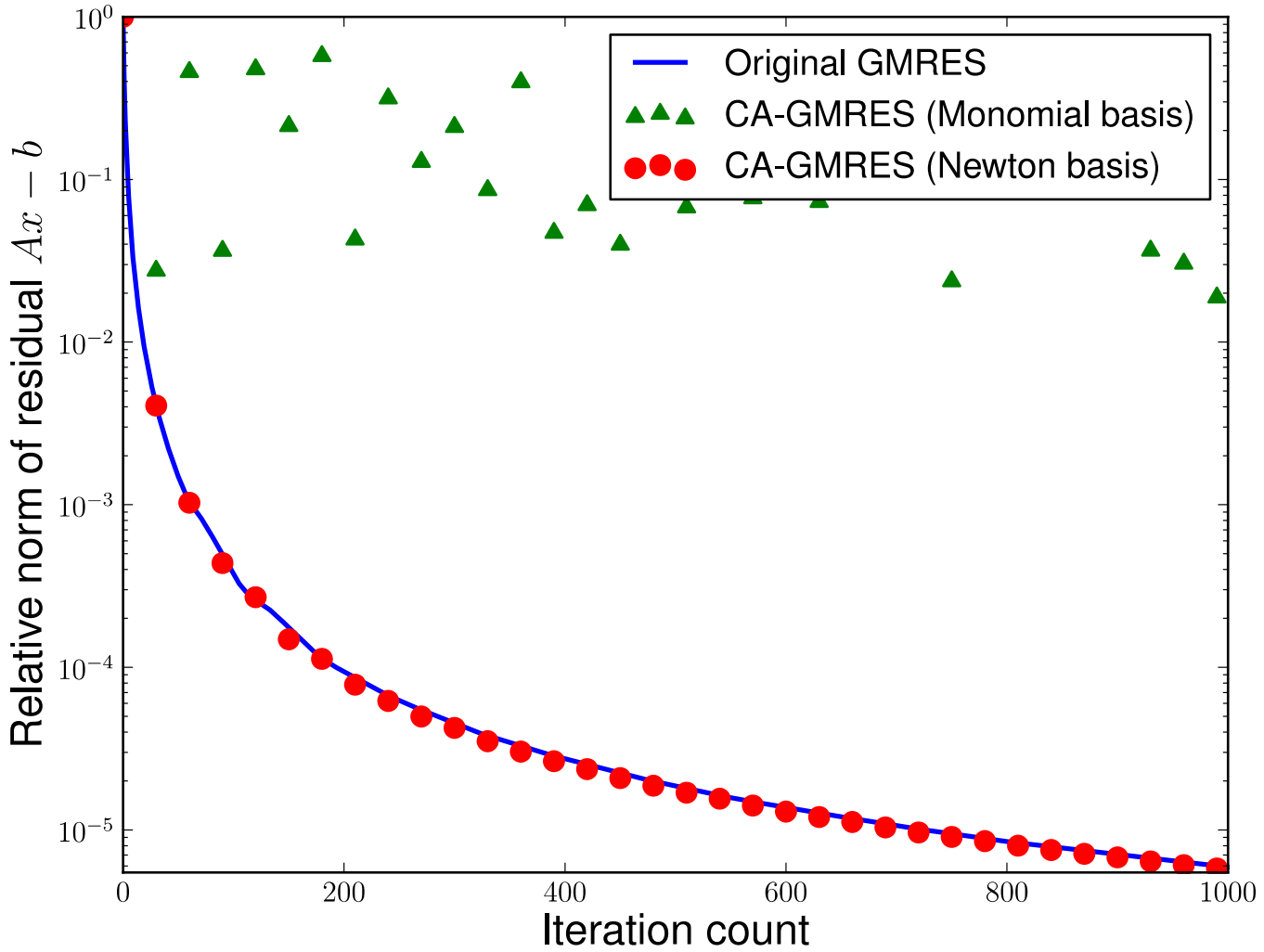
Communication-avoiding GMRES

```
W = [ v, Av, A2v, … , Akv ]
[Q,R] = TSQR(W)  … "Tall Skinny QR"
Build H from R, solve LSQ problem
```

Sequential: #words_moved =
O(nnz) from SpMV
+ O(k·n) from TSQR
Parallel: #messages =
O(1) from computing W
+ O(log p) from TSQR

- Oops – W from power method, precision lost!

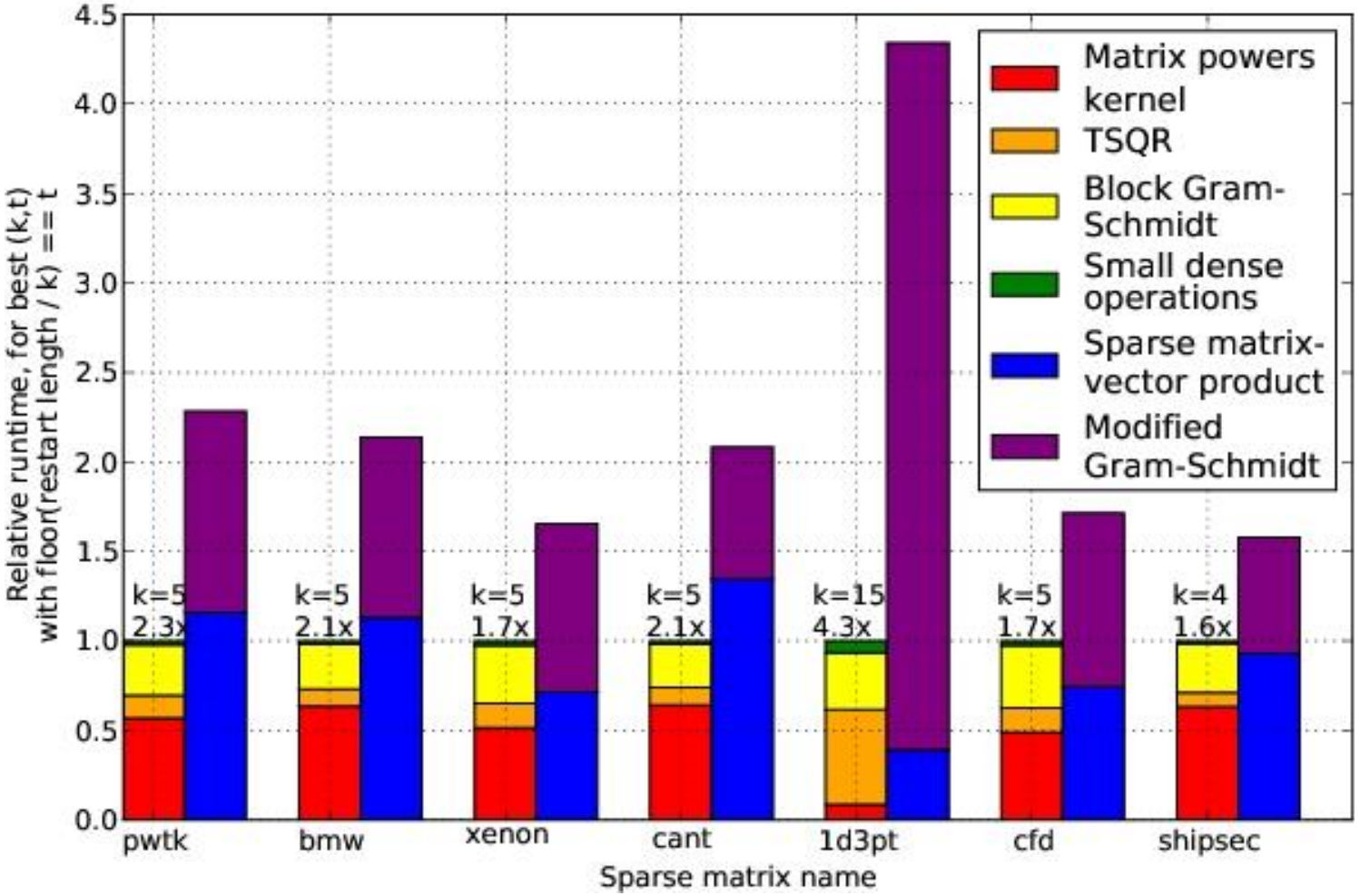
Matrix Powers Kernel (and TSQR) in an iterative solver (GMRES)



Jim Demmel, Mark Hoemmen, Marghoob Mohiyuddin, Kathy Yelick

Communication-Avoiding Krylov Method (GMRES)

Performance on 8 core Clovertown



Takeaway messages

- Tuning for modern processors is hard
- Sparse matrices: tuning harder
- SpMV: benefits lower due to low Computational Intensity (you need to read the matrix)
- Usual low level tuning (prefetch, etc.) have some benefit
- Compressing the matrix can be a big win
- Reordering (including graph partitioning) improves locality
- After tuning SpMV should be memory bandwidth limited
- Optimizing at a high level (across iterations) can improve reuse, but it does affect numerics