Sparse-Matrix-Vector-Multiplication (SpMV) and Sparse Iterative Solvers

Kathy Yelick

https://sites.google.com/lbl.gov/cs267-spring-2018/

Outline for today

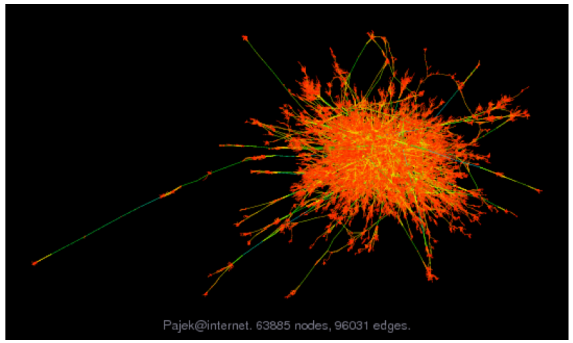
- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- Cache blocking SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

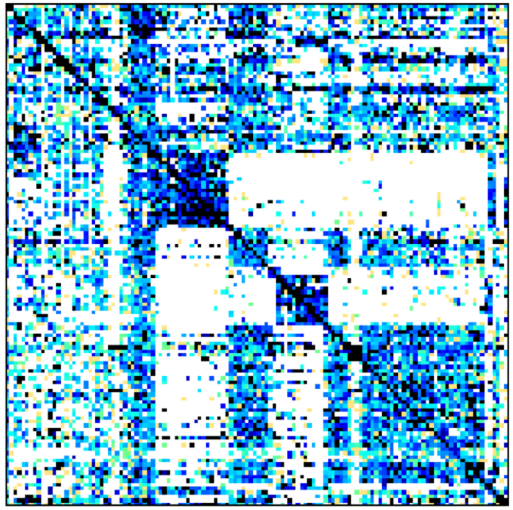
Outline for today

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

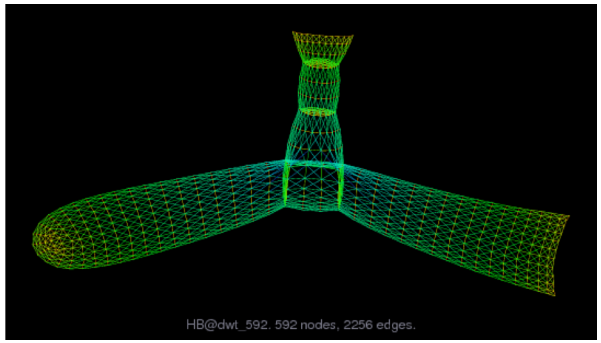
Sparse matrices are everywhere

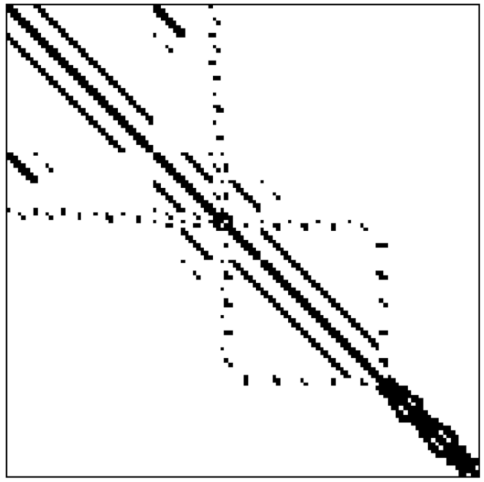
Internet connectivity



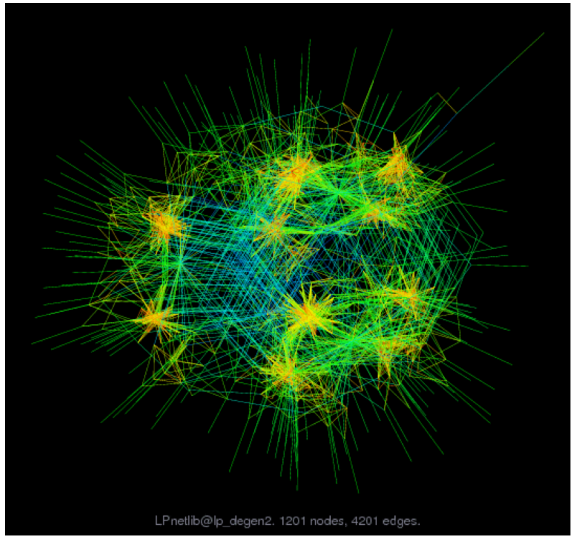


Structural design





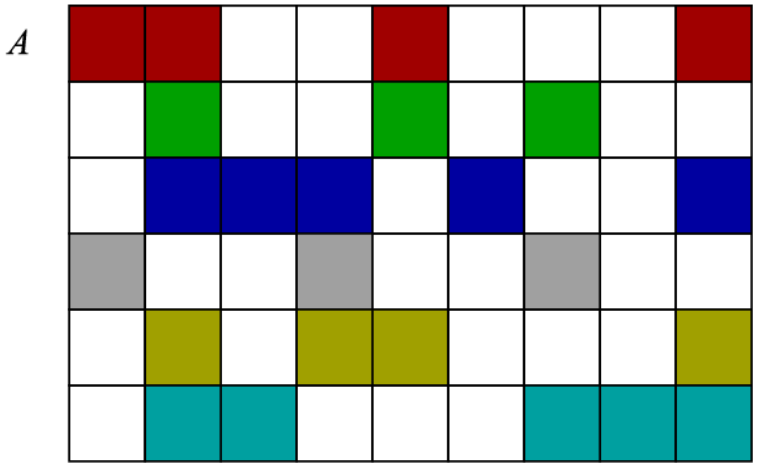
Linear Programming

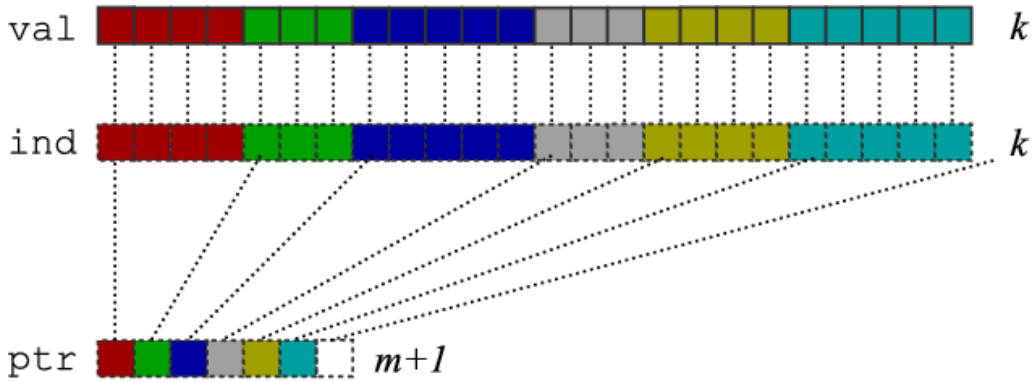


Sparse Matrix-Vector Multiply (SpMV)

- 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 ($SpDM^3$)
  - Machine learning

Compressed Sparse Row (CSR) Storage

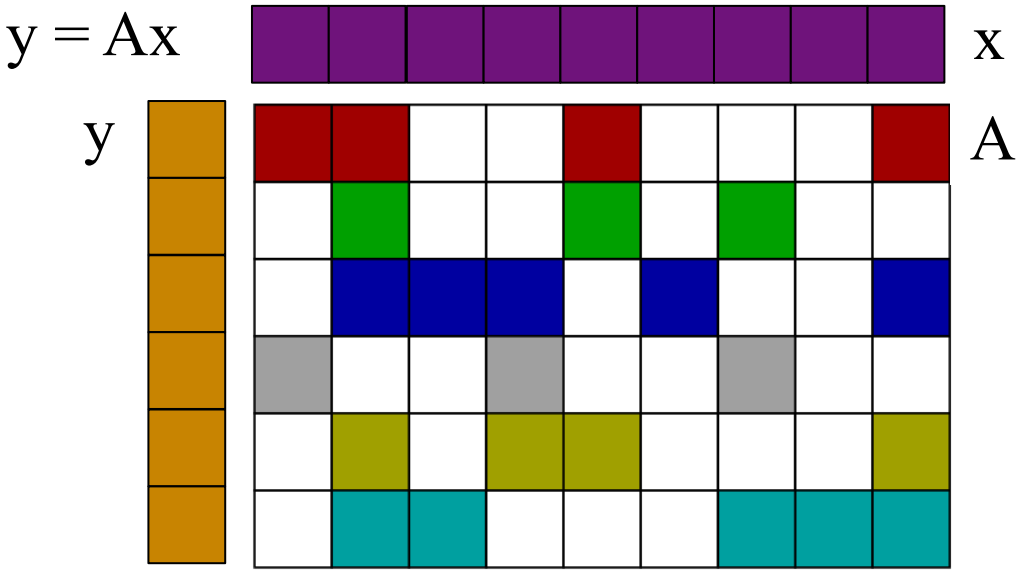


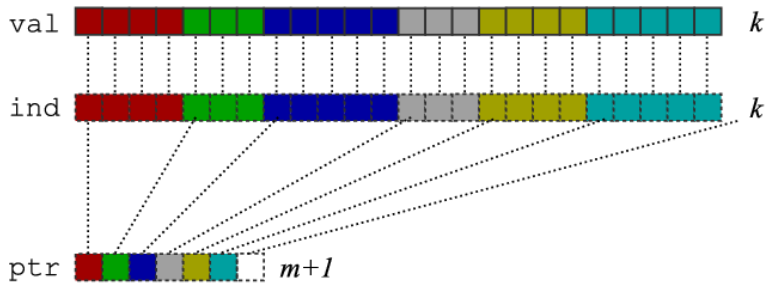


- CSR has:
  - Array of the nonzero values (val) of size nnz = number of nonzeros
  - Array of the column indices for each value of size nnz
  - Array of row start pointers of size n = number of rows
- Other common formats (plus blocking)
  - Compressed sparse column (CSC)
  - Coordinate (COO): row + column index per nonzero (easy to build)

And many more specialized ones!

SpMV with Compressed Sparse Row (CSR)





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

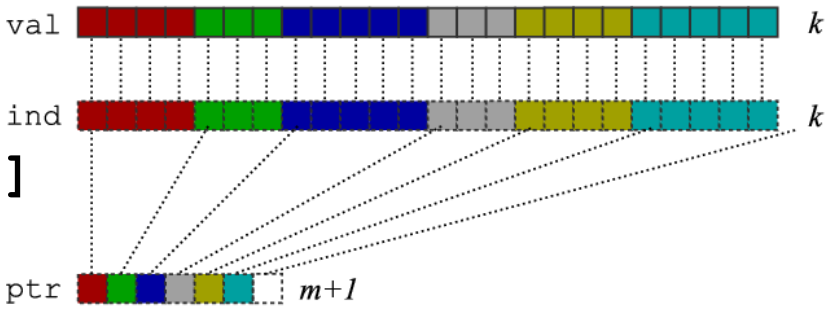
```
for each row i
   for k=ptr[i] to ptr[i+1]-1 do
       y[i] = y[i] + val[k]*x[ind[k]]
```

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

SpMV with Compressed Sparse Row (CSR)

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]-1 do
    y[i] = y[i] + val[k]*x[ind[k]]
```



Possible optimizations:

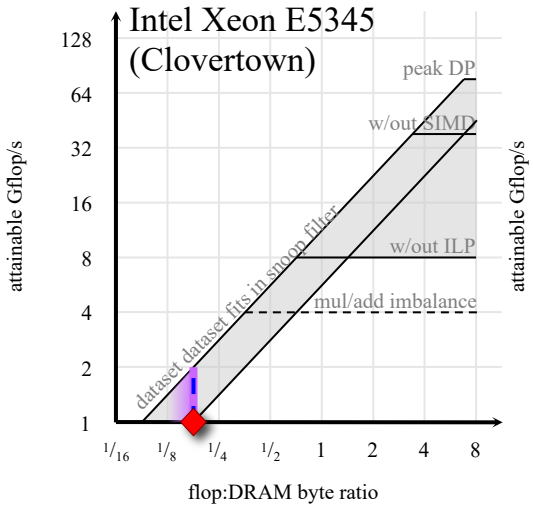
1) Unroll the k loop → need # non-zeros per row
2) Hoist y[i] → OK absent aliasing
3) Eliminate ind[i] → need to know non-zero pattern
4) Reuse elements of x → need good non-zero pattern

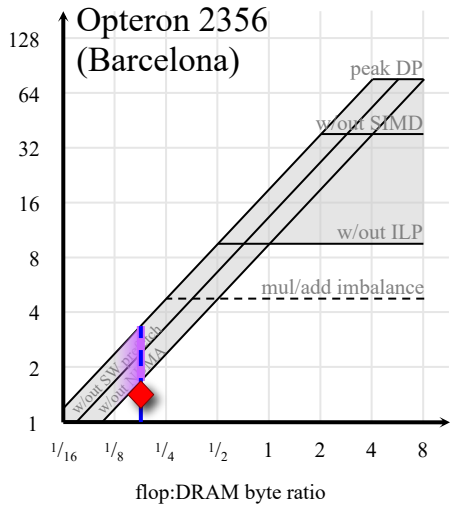
Cache: need nonzeros in nearby rows and same-cache-line columns

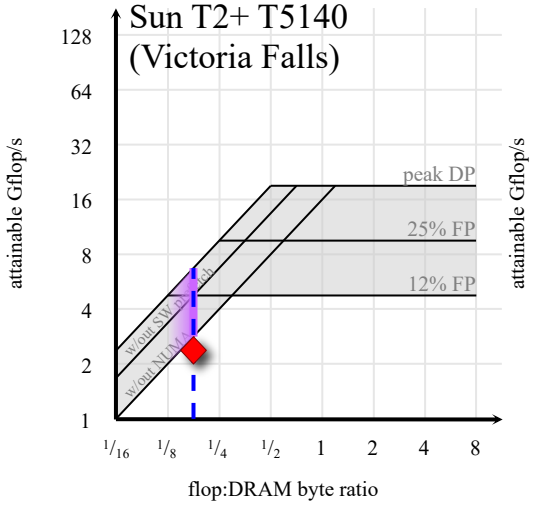
Register: need to know where these nonzeros are to save x[i]

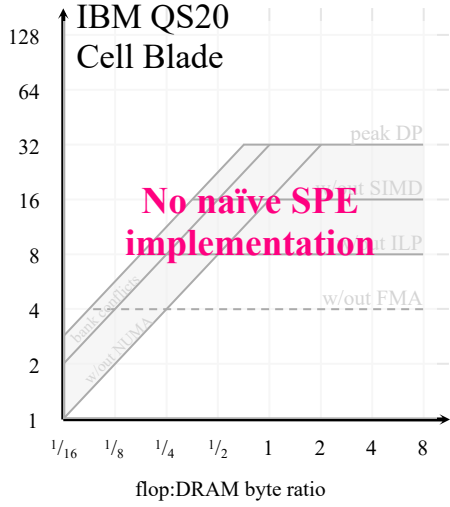
Roofline model for SpMV
(out-of-the-box parallel)

- 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



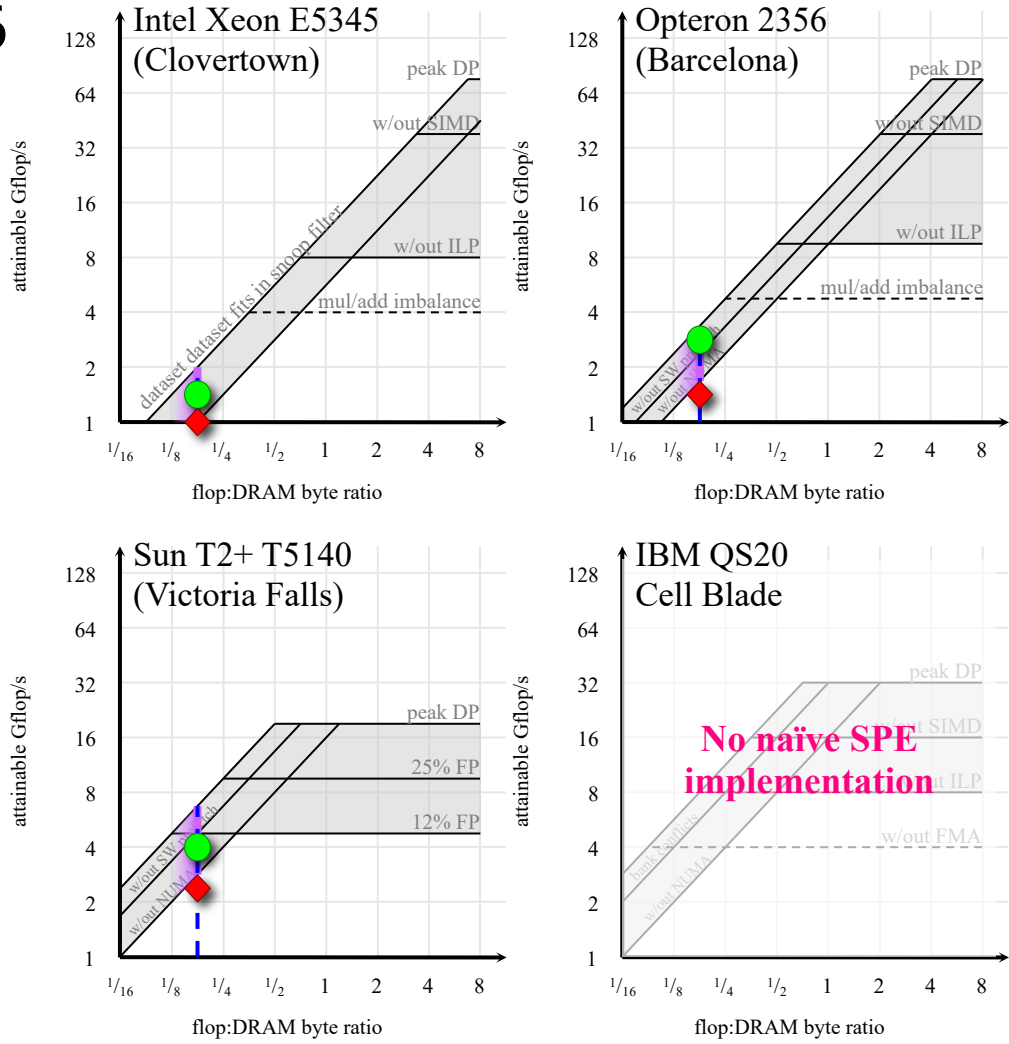






Roofline model for SpMV
(NUMA & SW prefetch)

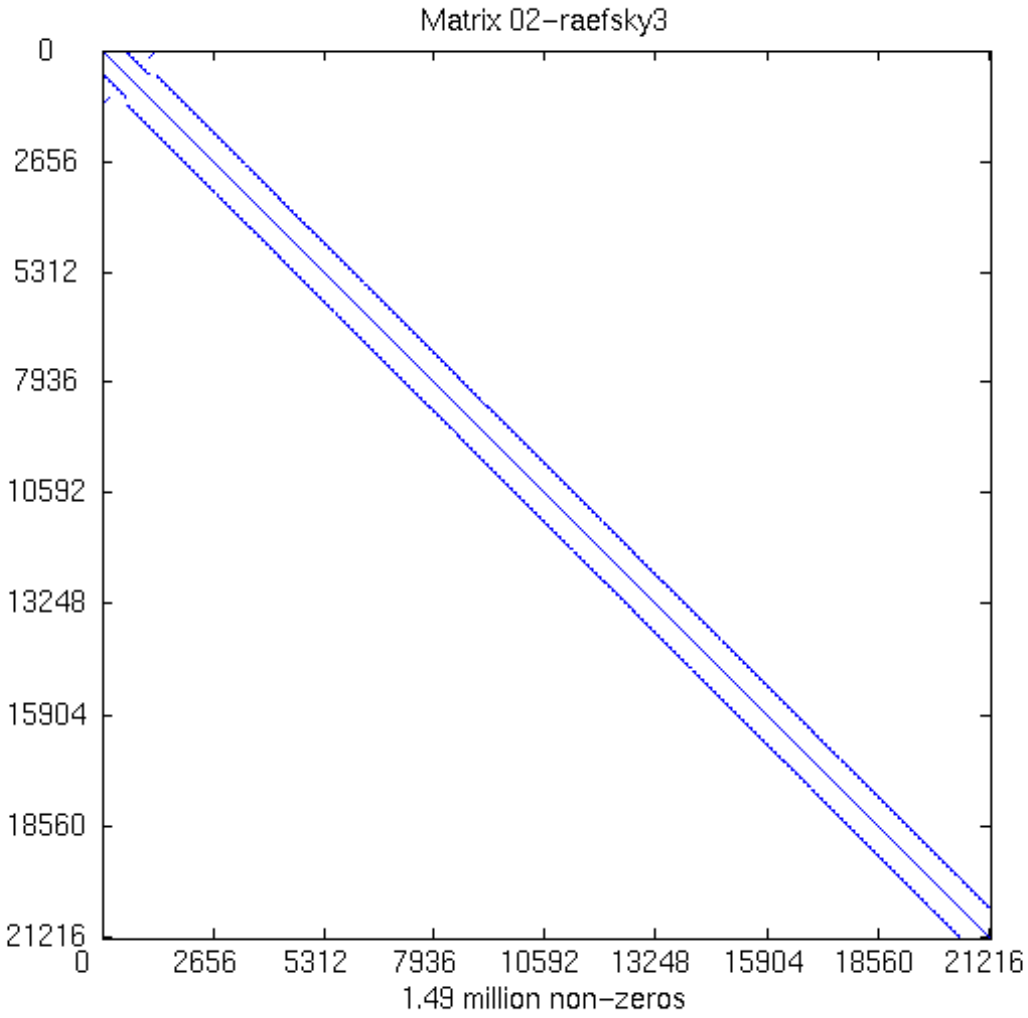
- compulsory flop:byte ~ 0.166
- utilize all memory channels



Outline for today

- Homework 1 results
- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- Cache blocking SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

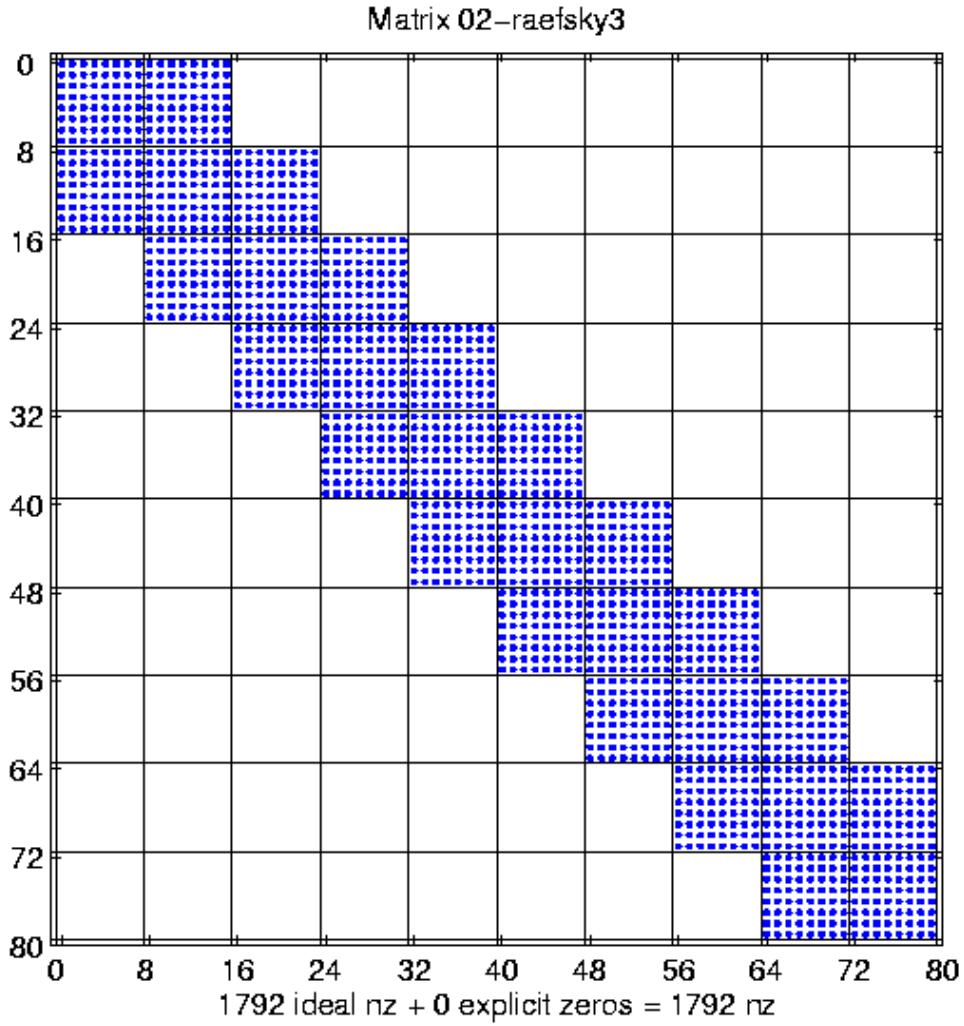
Changing Matrix Format: Blocking



- 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

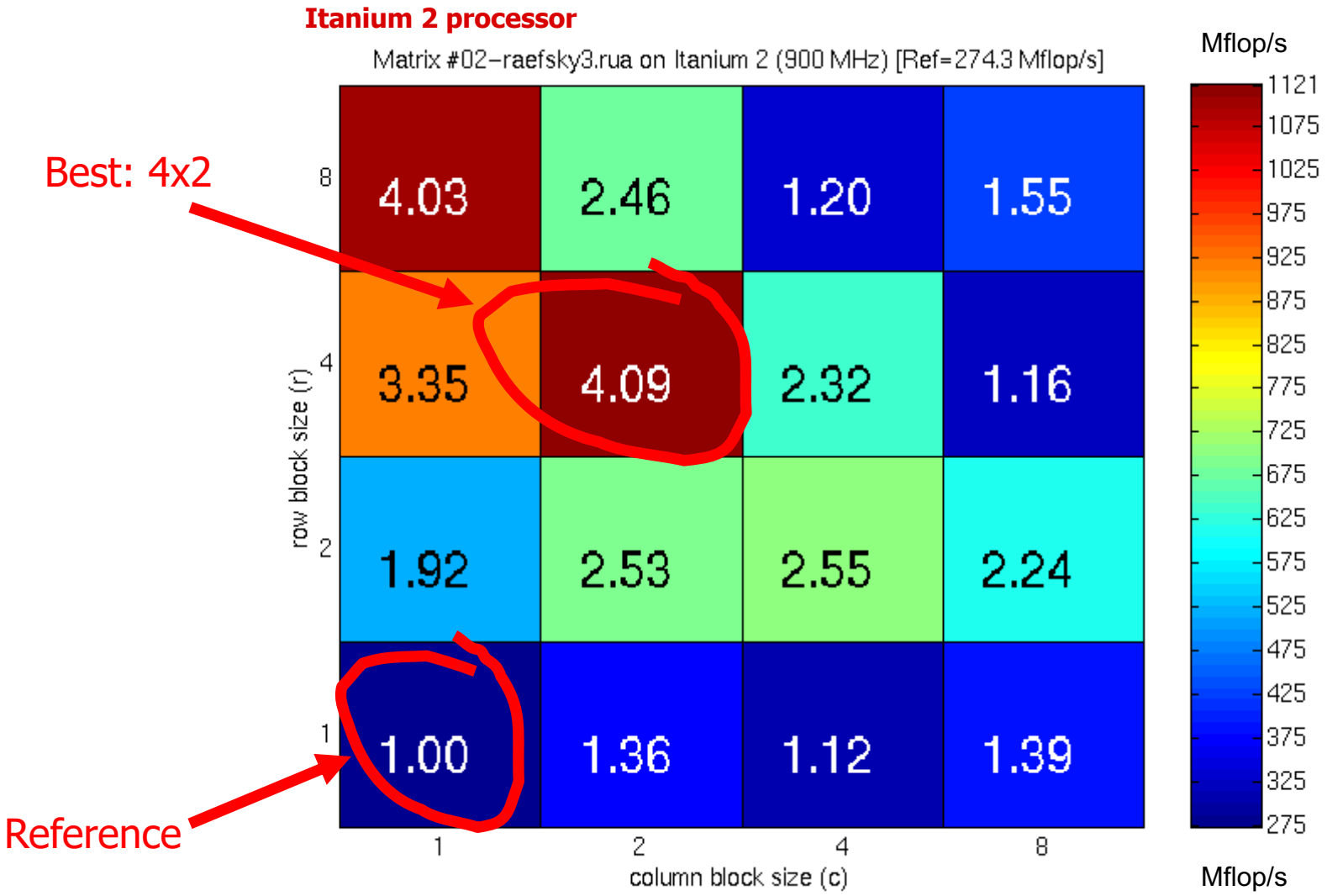
- Source: NASA structural analysis problem

- **8x8** dense substructure

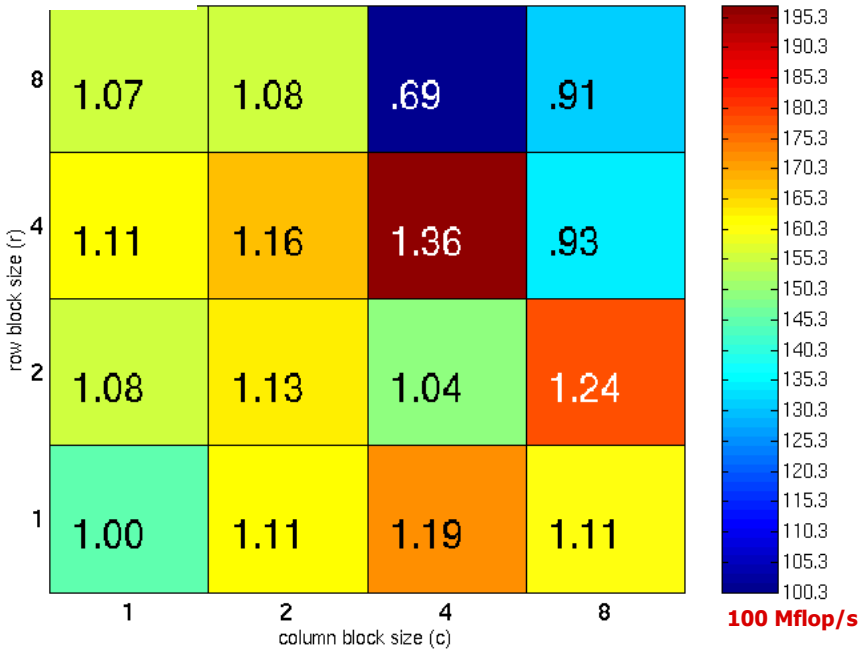
Taking advantage of block structure in SpMV

- 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

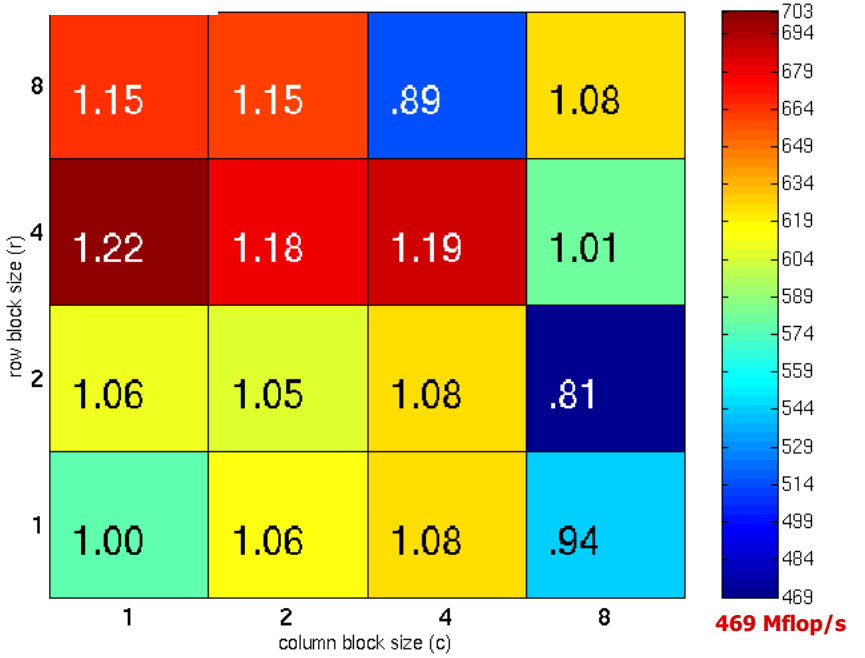
The Need for Search



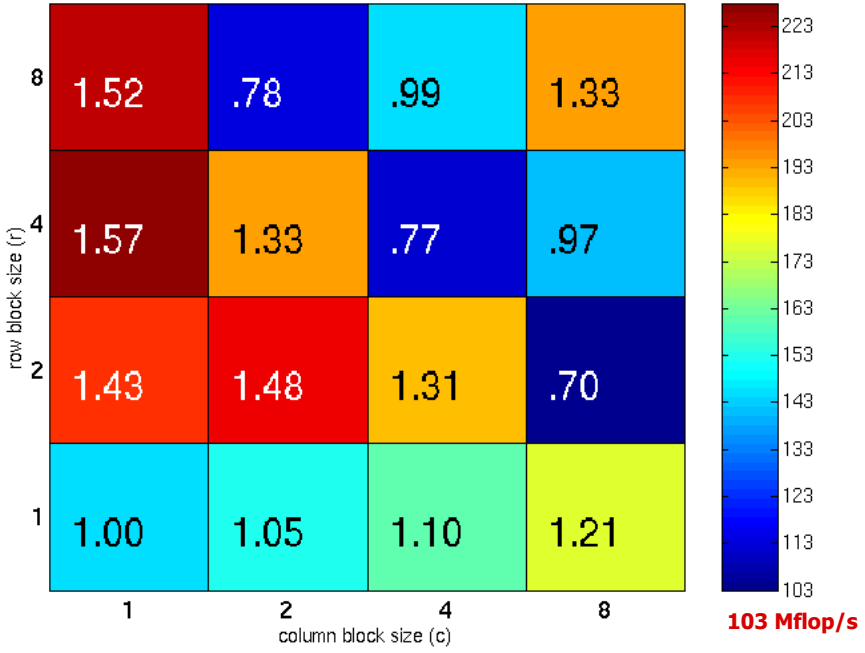
Power3 - 13%



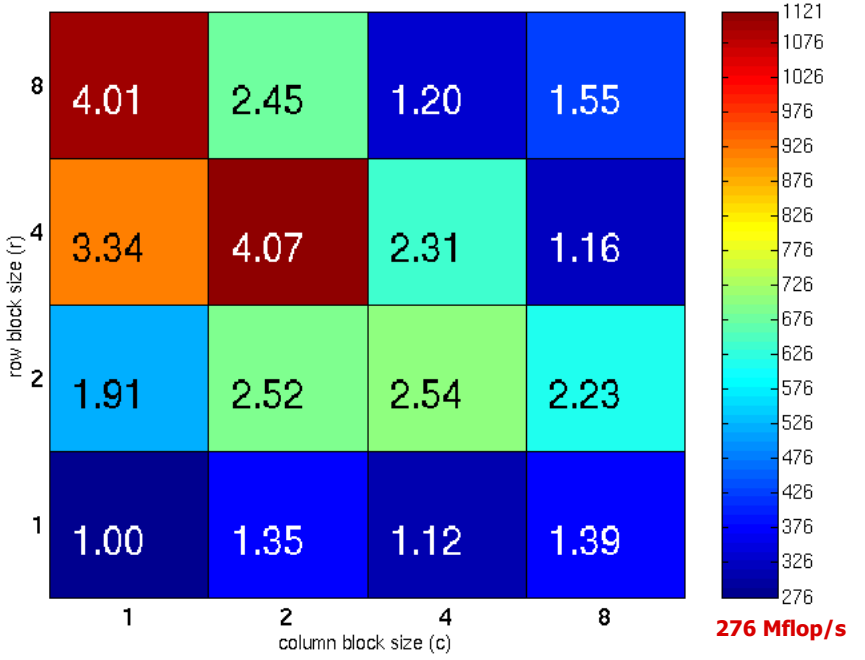
Power4 - 14%



Itanium 1 - 7%

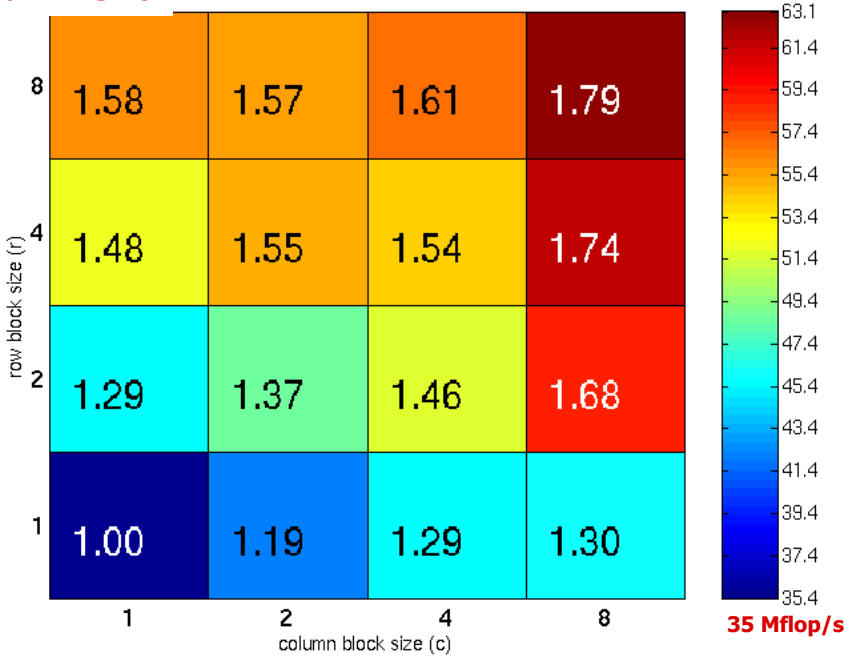


Itanium 2 - 31%



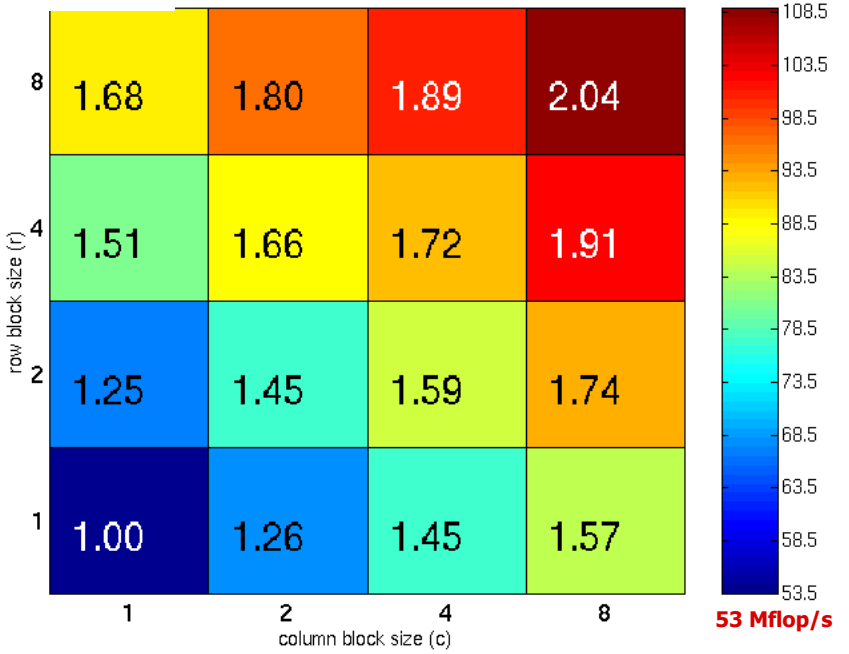
Ultra 2i - 9%

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



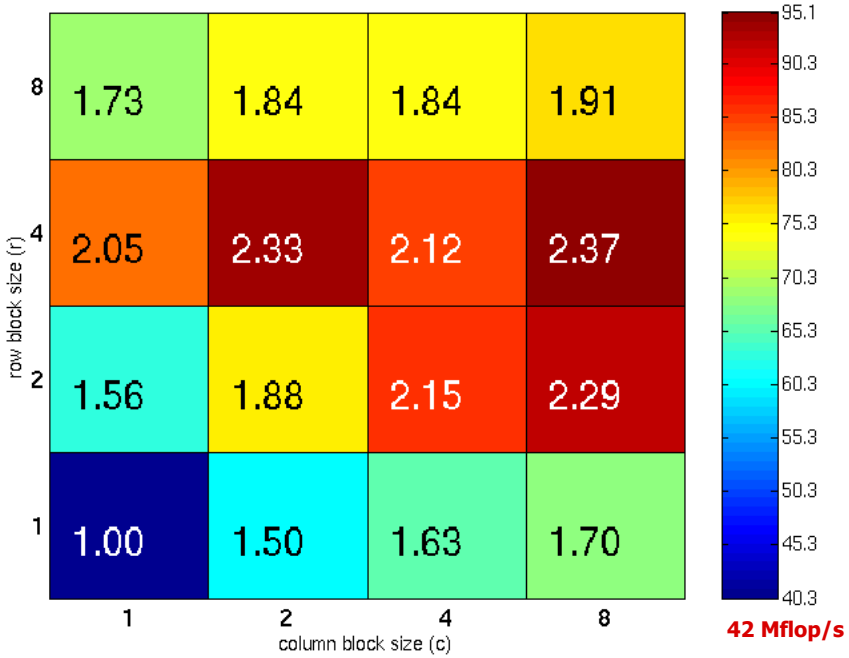
Ultra 3 - 5%

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



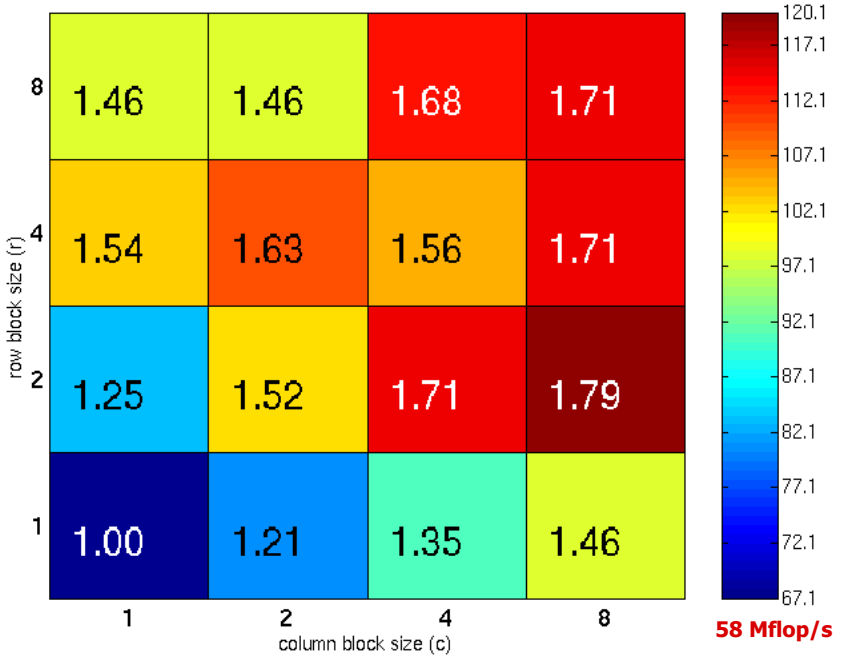
Pentium III - 19%

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

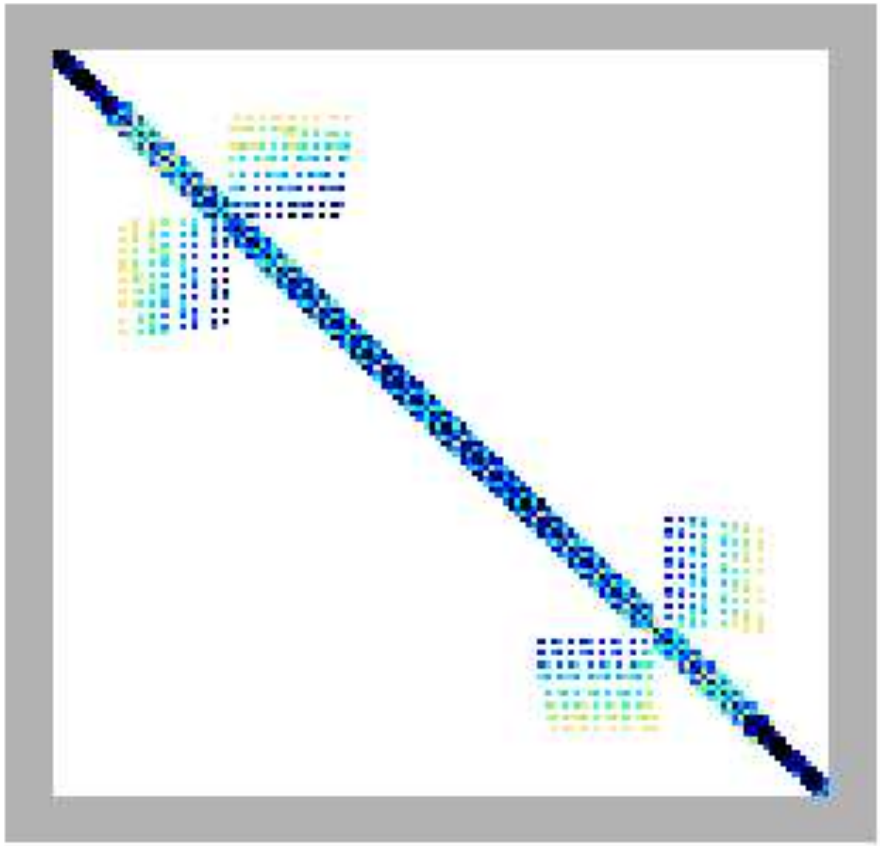


Pentium III-M - 15%

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



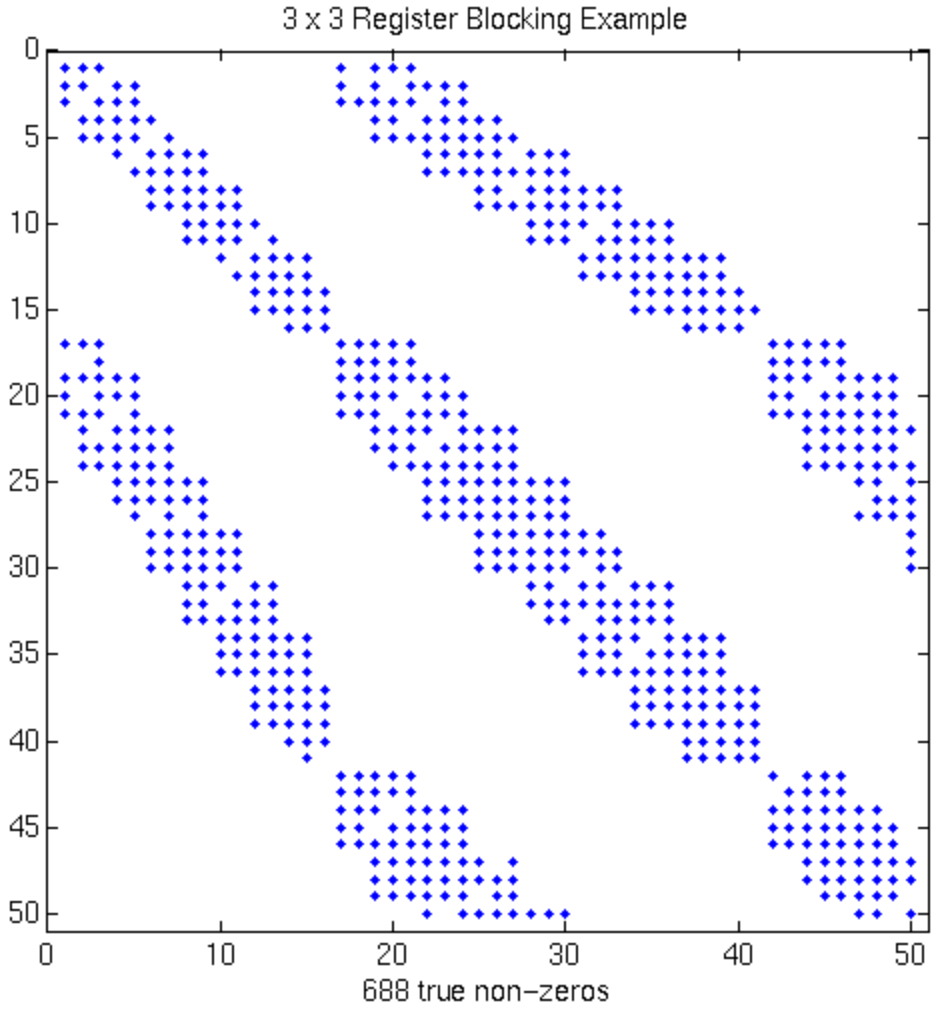
But most matrices don't block so easily



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

- N = 16614
- NNZ = 1.1M

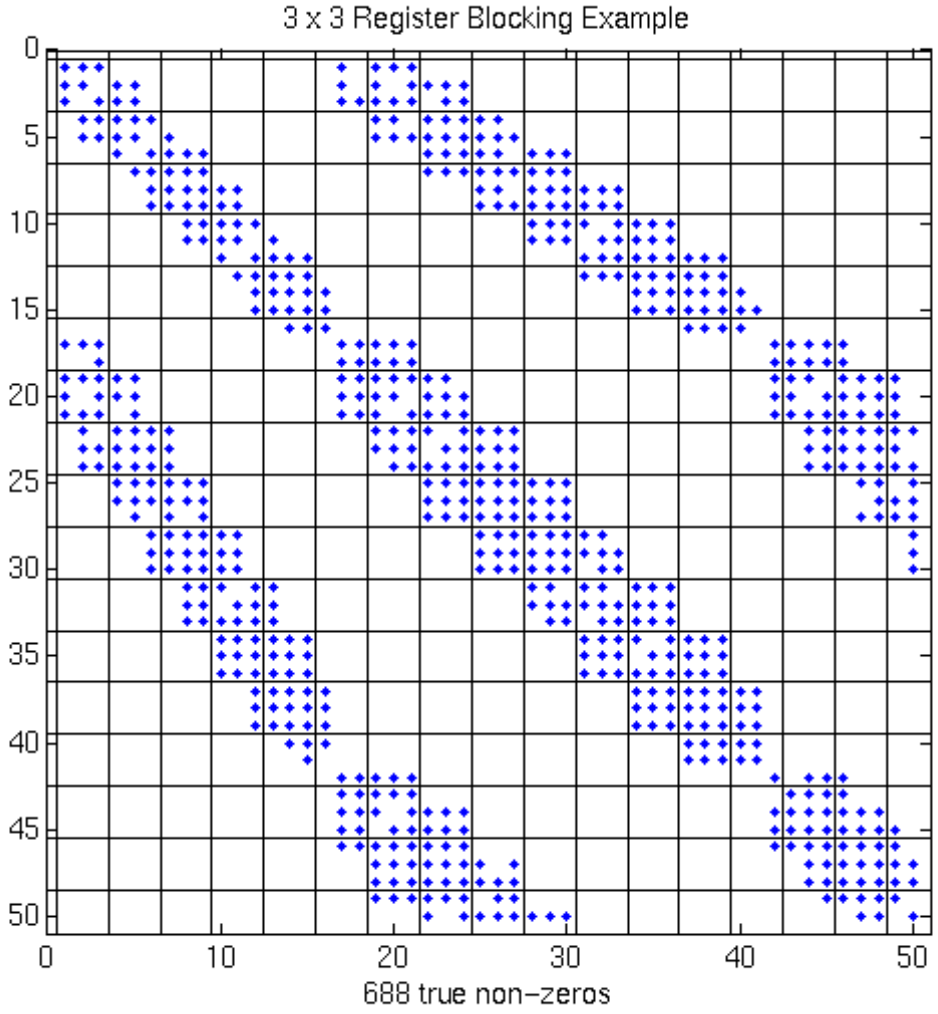
Zoom in to top corner



- More complicated non-zero structure

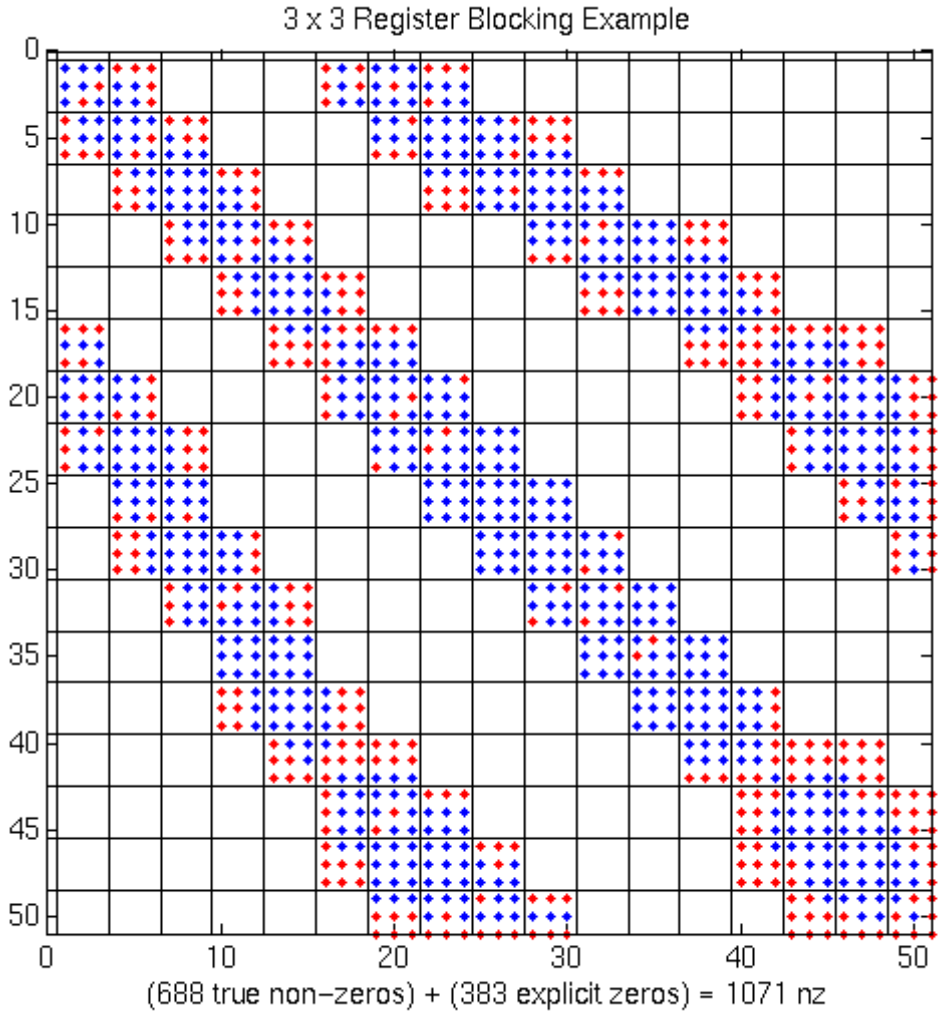
- N = 16614
- NNZ = 1.1M

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

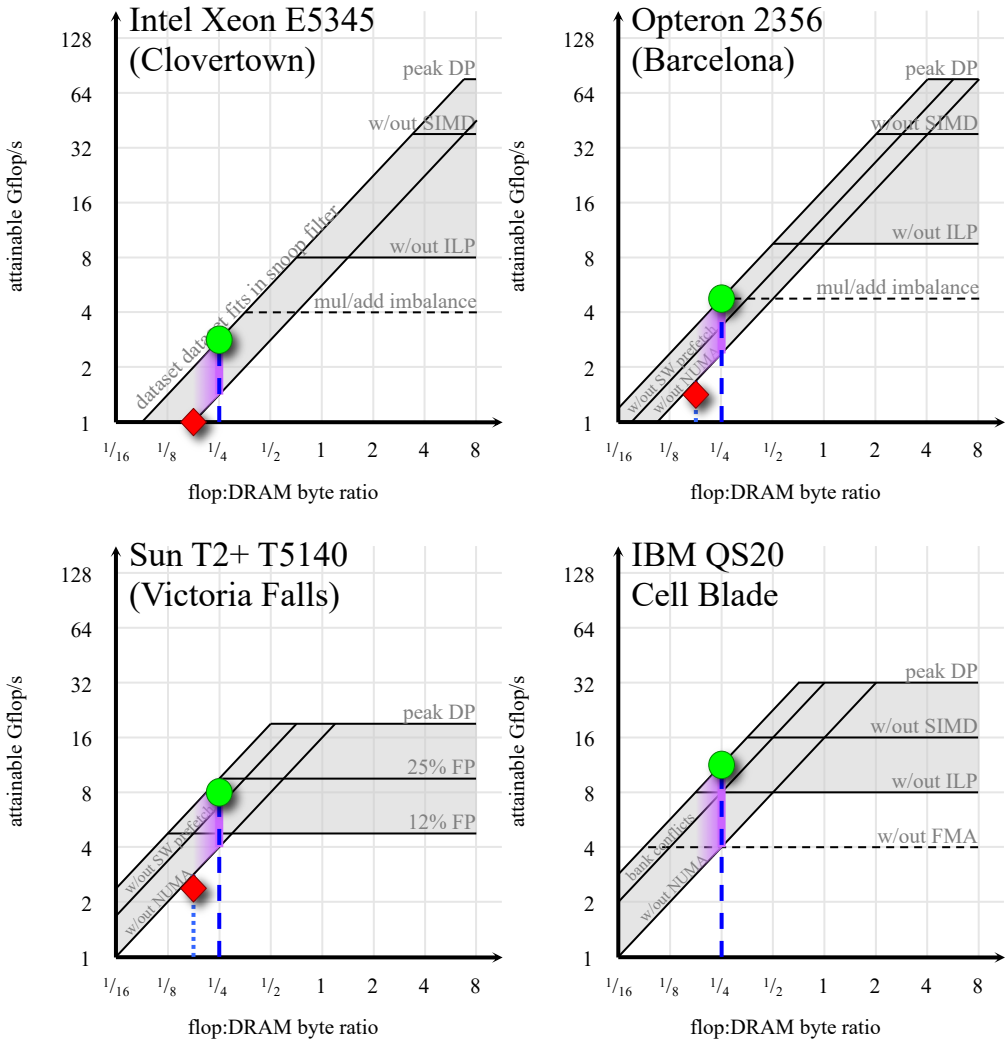
Extra work can improve efficiency



- 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?

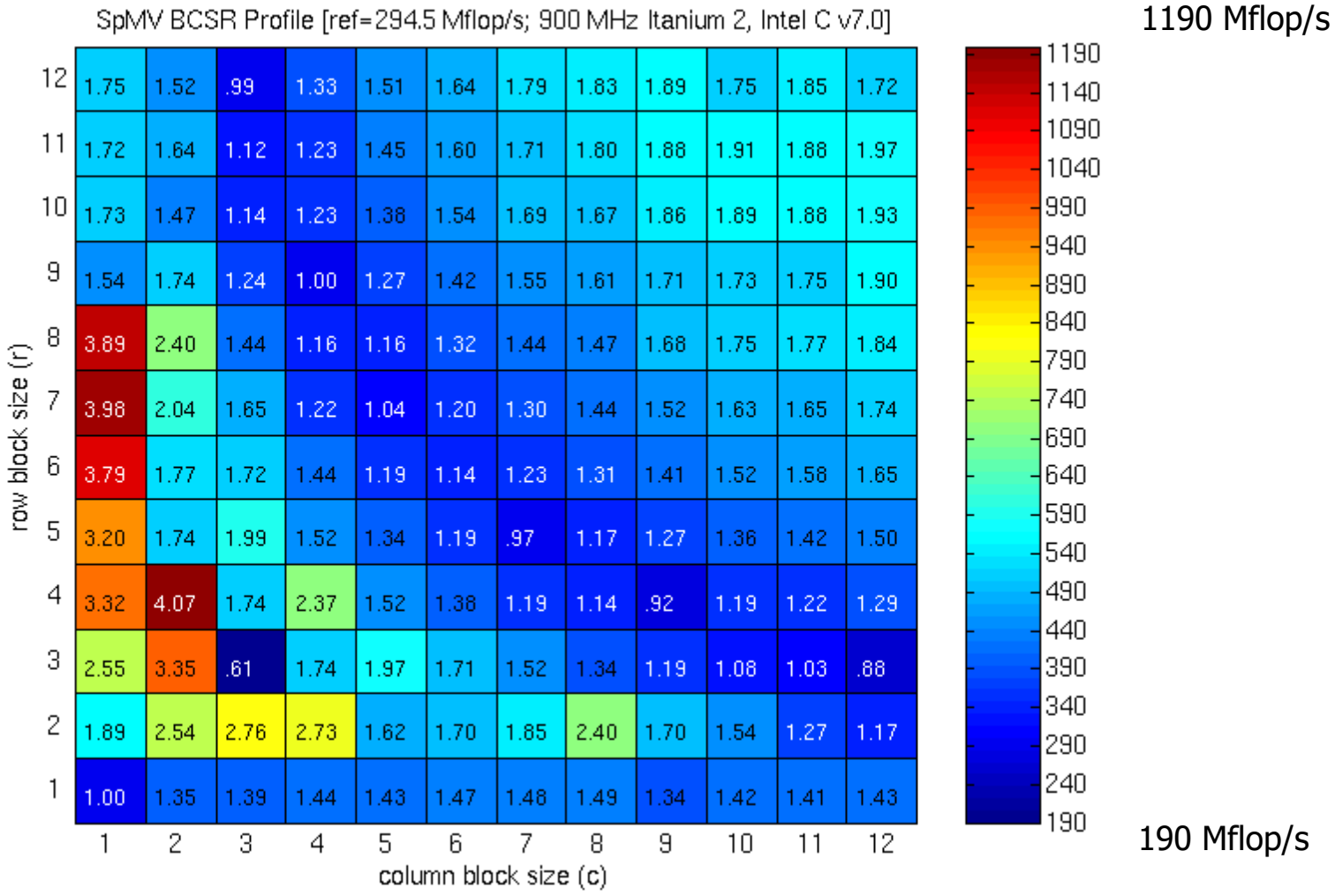
- How to build optimized library when:
  - Formats are not known? Libraries like PETSc and Trilinos will let the user provide format and SpMV
- How to build optimized matrix kernel library?
  - Nonzero structure is key to optimization
- OSKI = Optimized Sparse Kernel Interface
  - pOSKI for multicore
- BeBOP: Berkeley Benchmarking and Optimization Group
  - Many results shown from current and former members
  - Meet weekly (contact Jim or Kathy if interested)

Automatic Register Block Size Selection

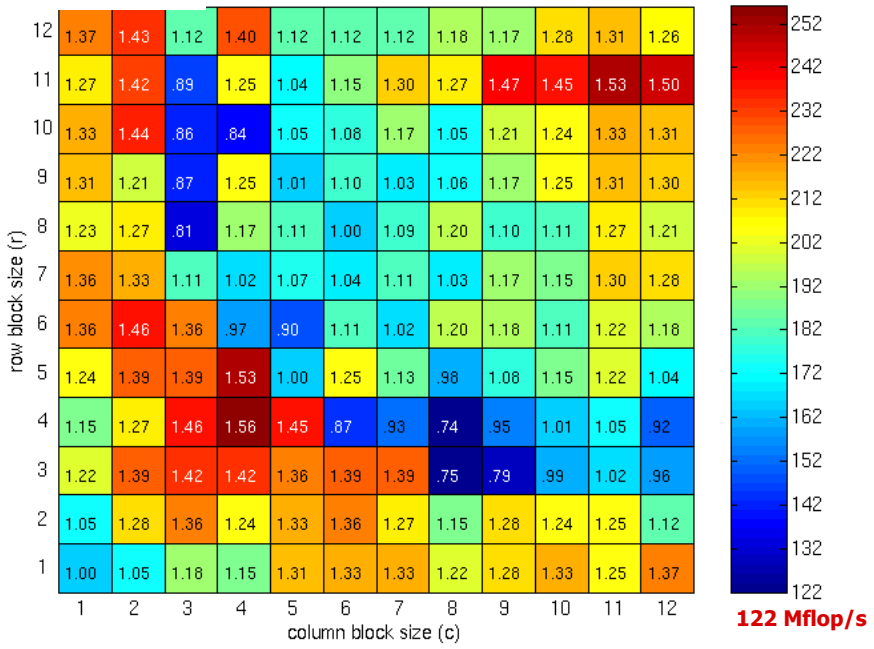
- Selecting the r x c block size
  - **Off-line benchmark of "register profile"**
    - Precompute **Mflops(r,c)** using *dense A in sparse format (blocked sparse row)* for each r x c
    - Once per machine/architecture
  - **Run-time "search"**
    - Sample *A* to estimate **Fill(r,c)** for each r x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

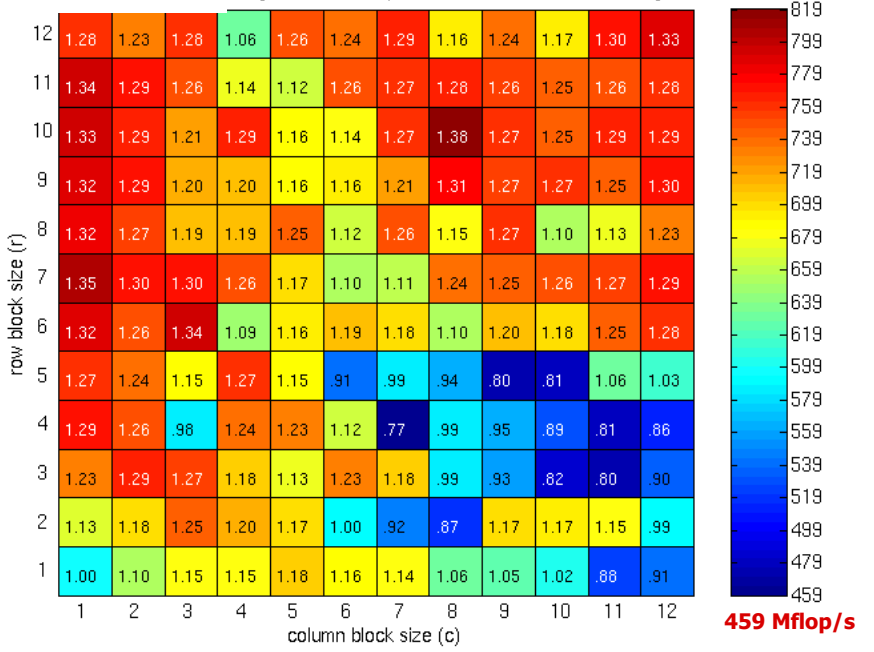
Register Profile: dense matrix in sparse format



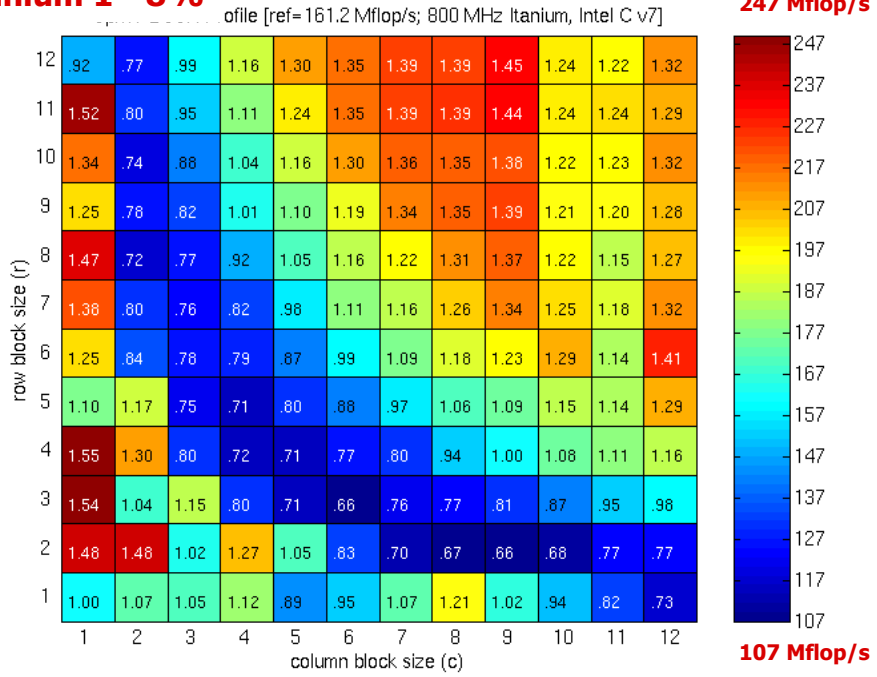
Power3 - 17%



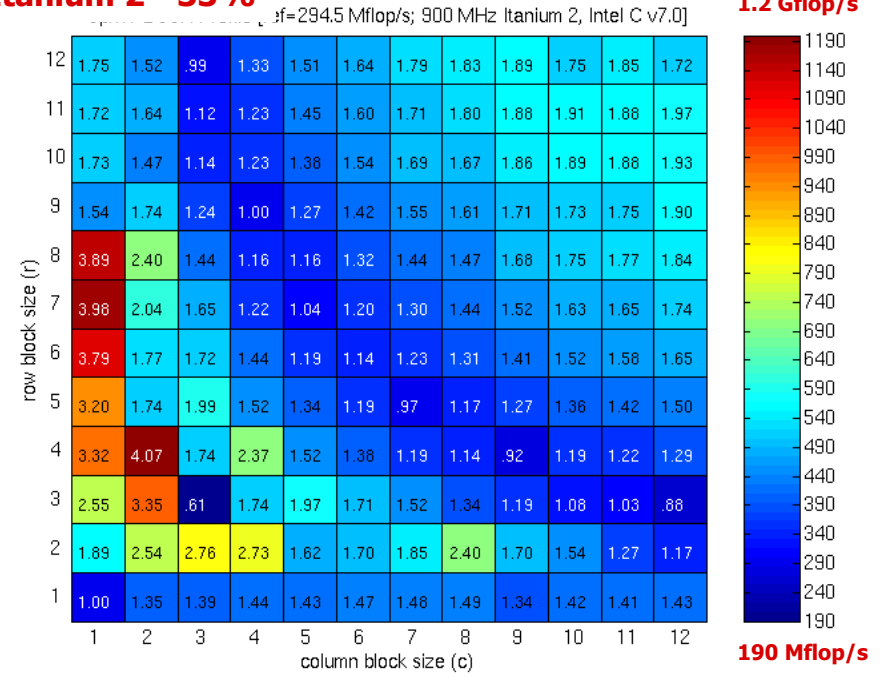
Power4 - 16%



Itanium 1 - 8%

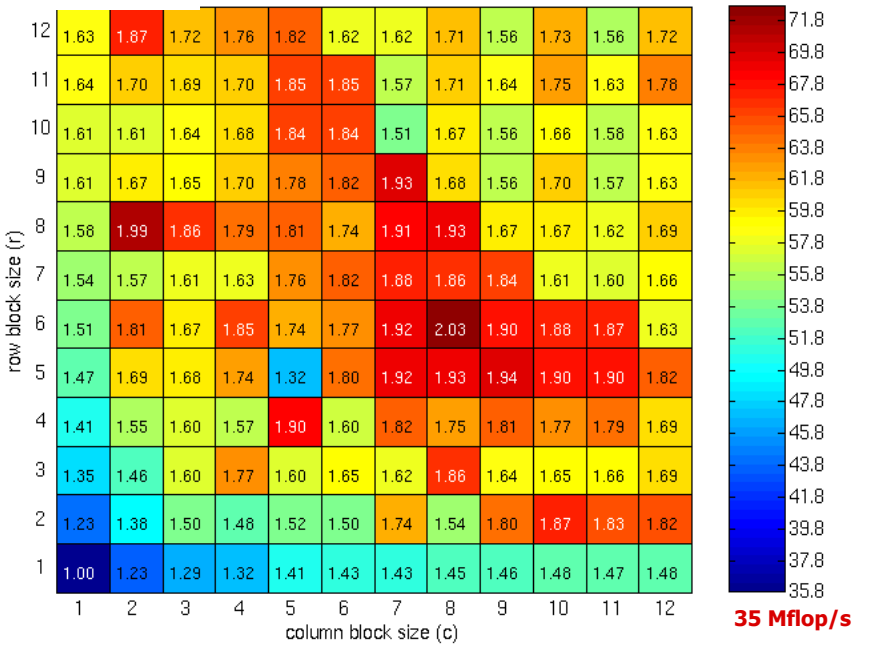


Itanium 2 - 33%



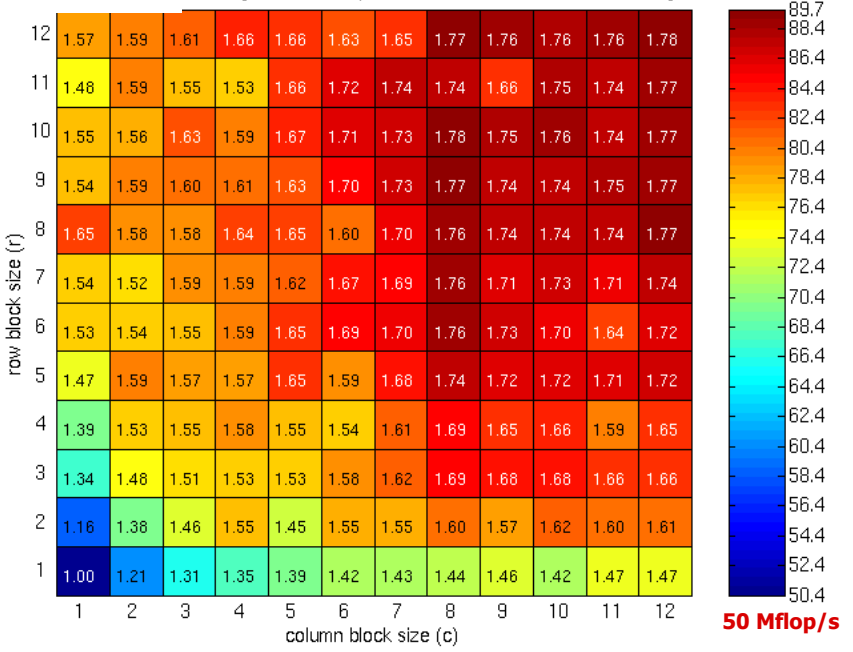
Ultra 2i - 11%

ofile [ref=35.8 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]



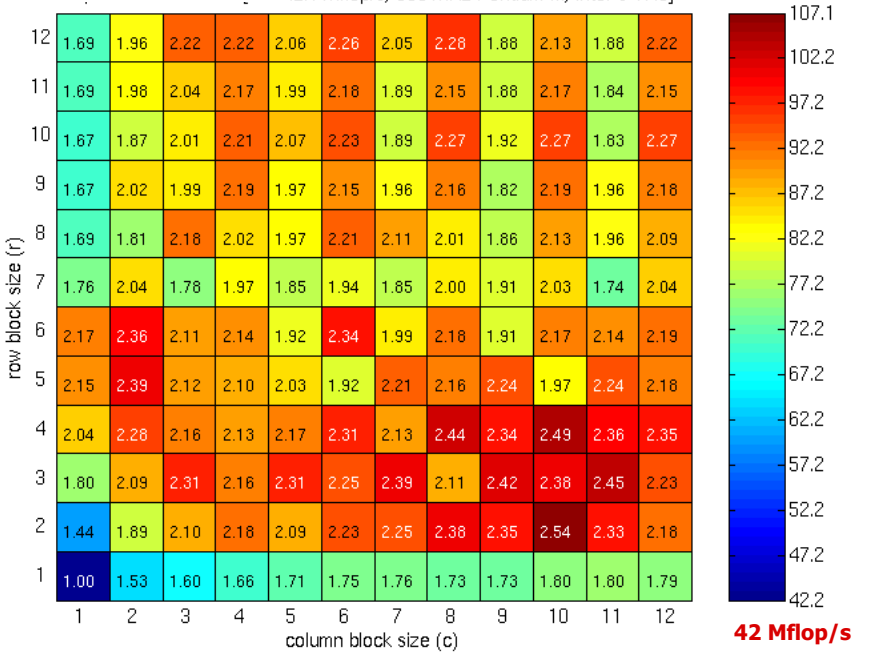
Ultra 3 - 5%

Profile [ref=50.3 Mflop/s; 900 MHz Sun Ultra 3, Sun C v6.0]



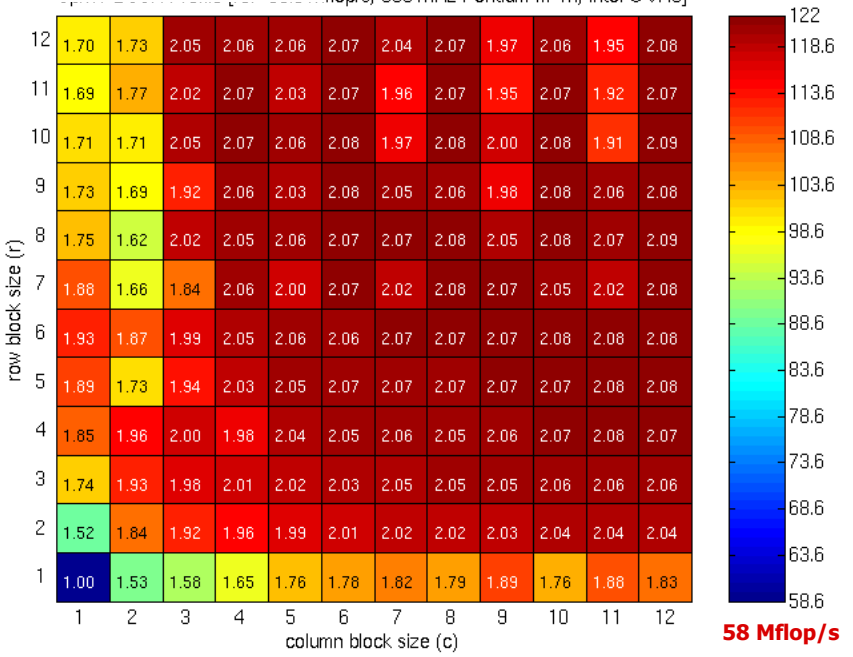
Pentium III - 21%

=42.1 Mflop/s; 500 MHz Pentium III, Intel C v7.0]



Pentium III-M - 15%

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

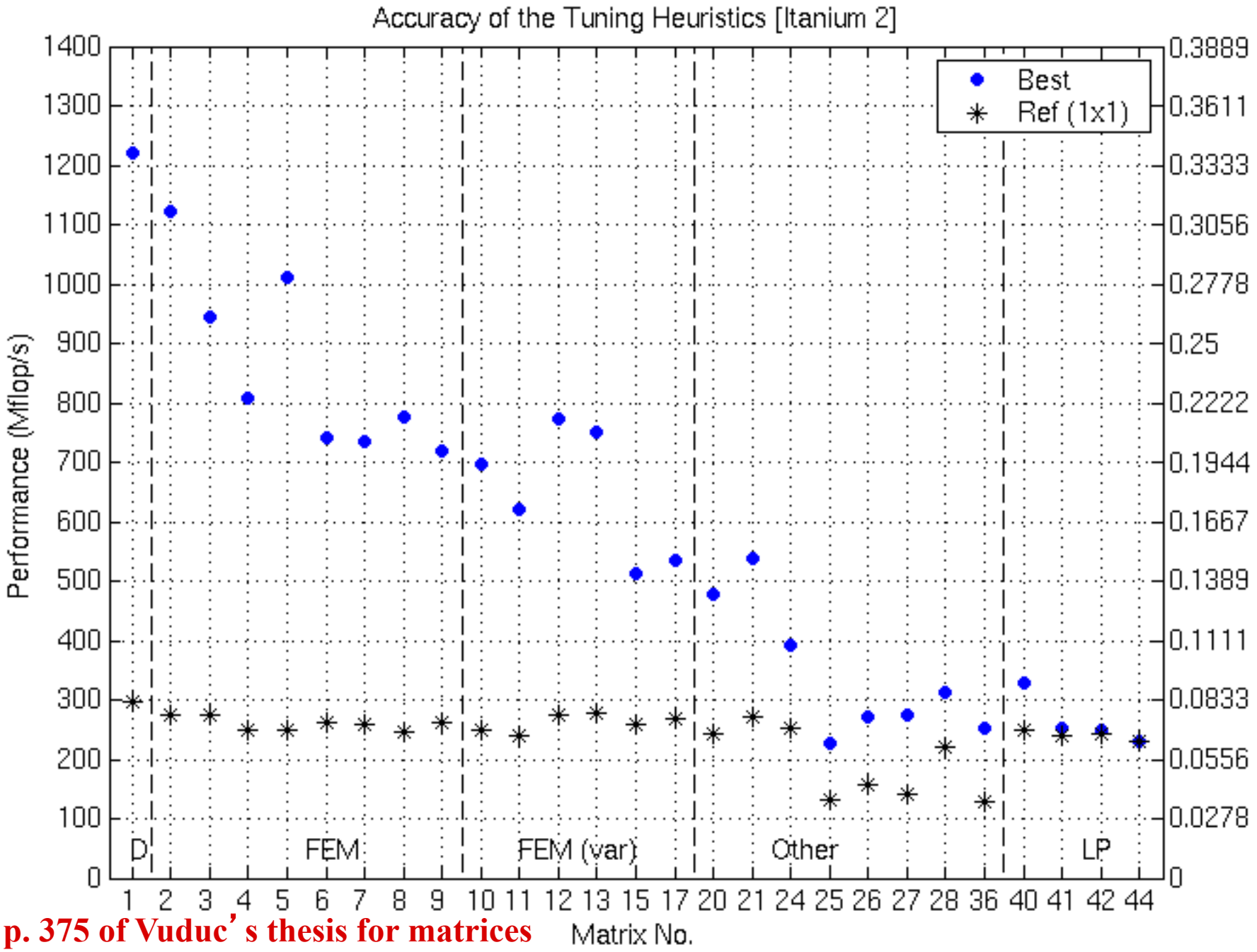


Accurate and Efficient Adaptive Fill Estimation

- Idea: Sample matrix
  - Fraction of matrix to sample: $s \in [0,1]$
  - 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

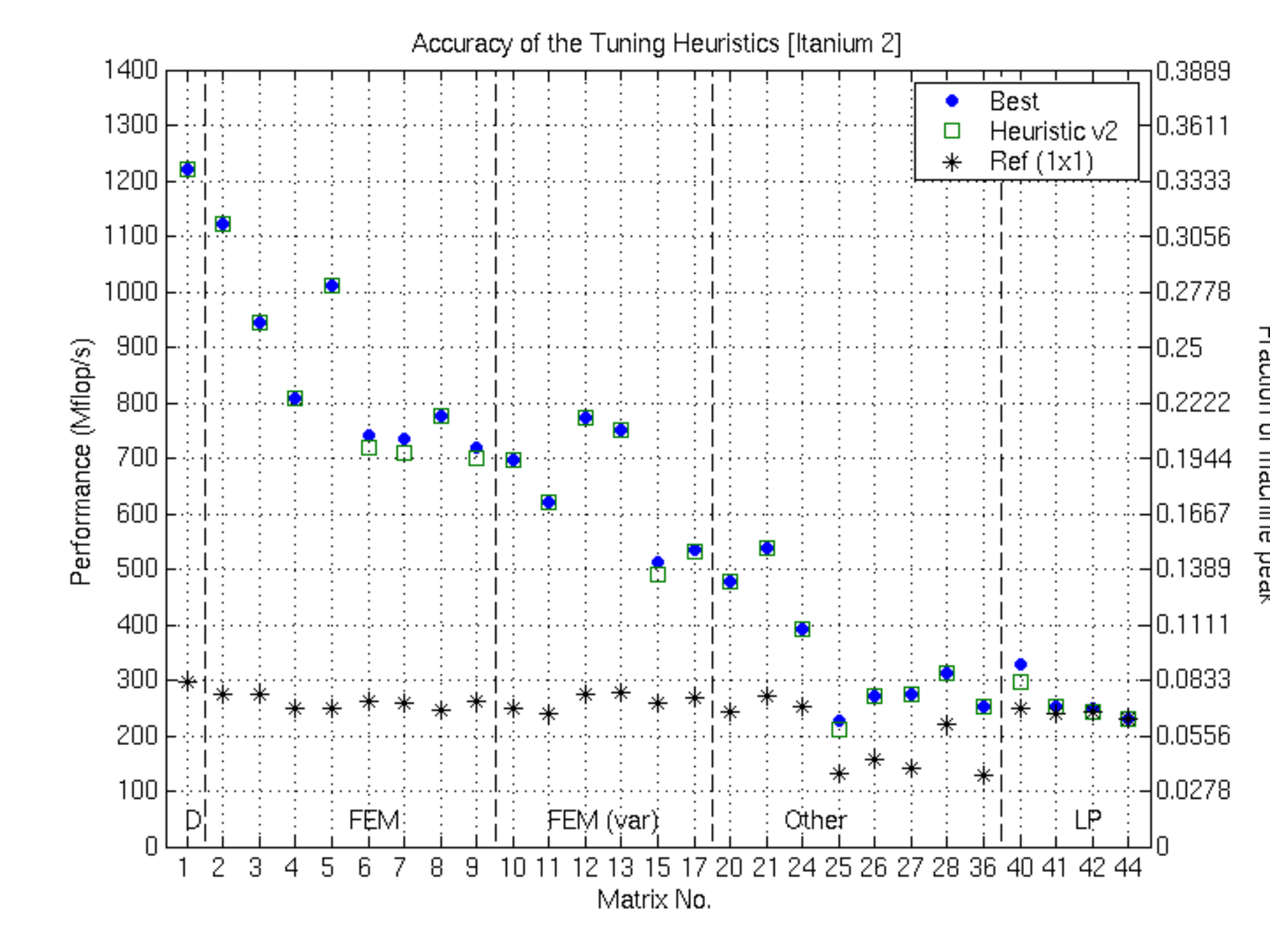
Machine Learning in Automatic Performance Tuning

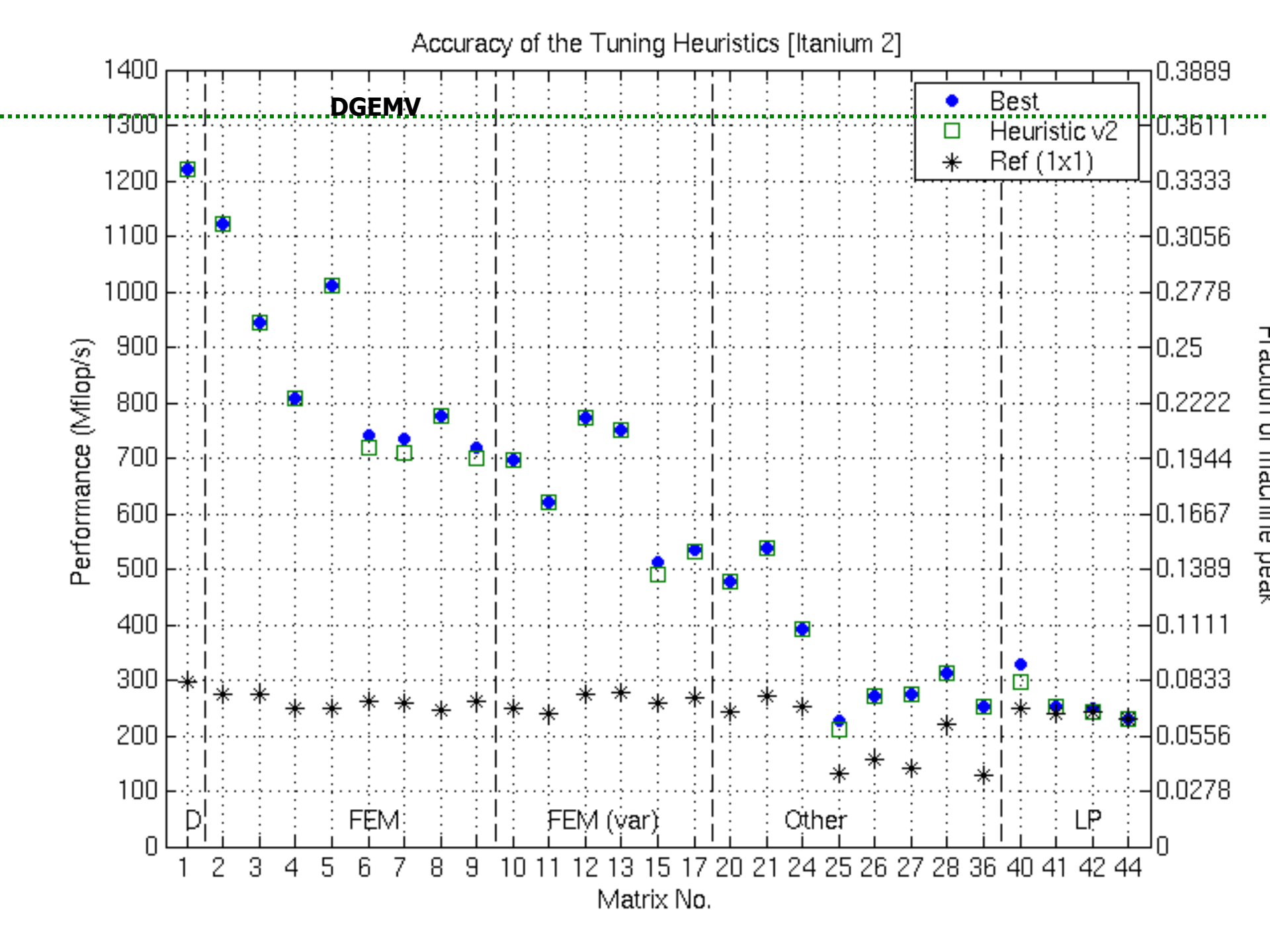
- **Statistical Models for Empirical Search-Based Performance Tuning** (*International Journal of High Performance Computing Applications*, 18 (1), pp. 65-94, February 2004)
  Richard Vuduc, J. Demmel, and Jeff A. Bilmes.
- **Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning** (Computer Science PhD Thesis, University of California, Berkeley. UCB//EECS-2009-181) Archana Ganapathi
- **Machine Learning for Predictive Autotuning with Boosted Regression Trees,** (*Innovative Parallel Computing, 2012*) J. Bergstra et al.
- **Practical Bayesian Optimization of Machine Learning Algorithms,** (*NIPS 2012*) J. Snoek et al
- **OpenTuner: An Extensible Framework for Program Autotuning**, (*dspace.mit.edu/handle/1721.1/81958*) S. Amarasinghe et al



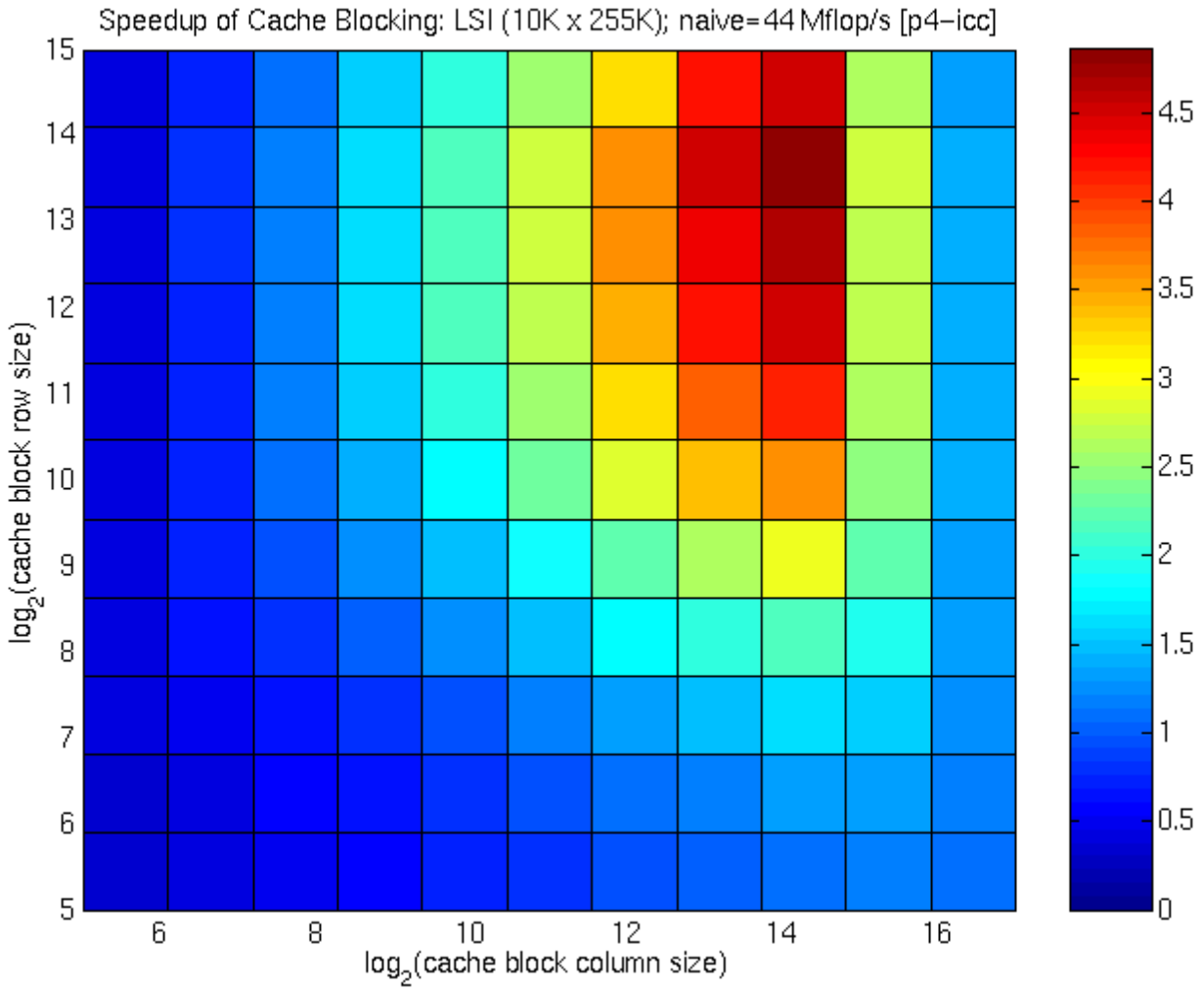
See p. 375 of Vuduc's thesis for matrices

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

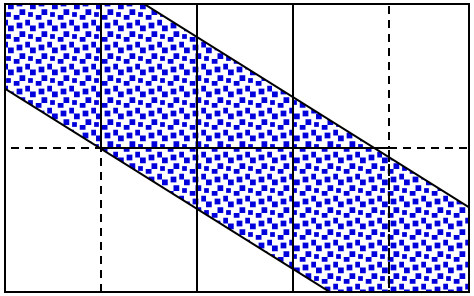
Baseline:
44 Mflop/s

Best block size & performance:
16k x 16k
210 Mflop/s

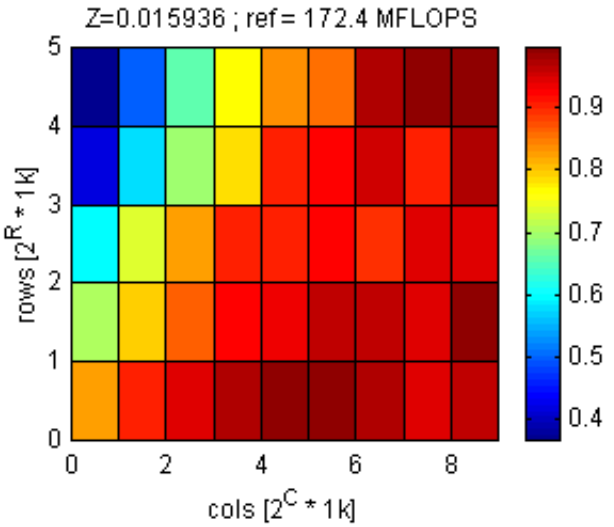
Nishtala, et al (2007). When cache blocking of sparse matrix vector multiply works and why.

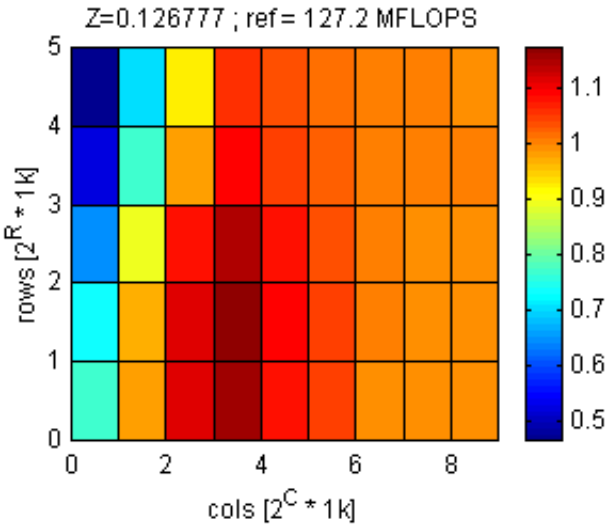
Cache Blocking on Random Matrices: Itanium

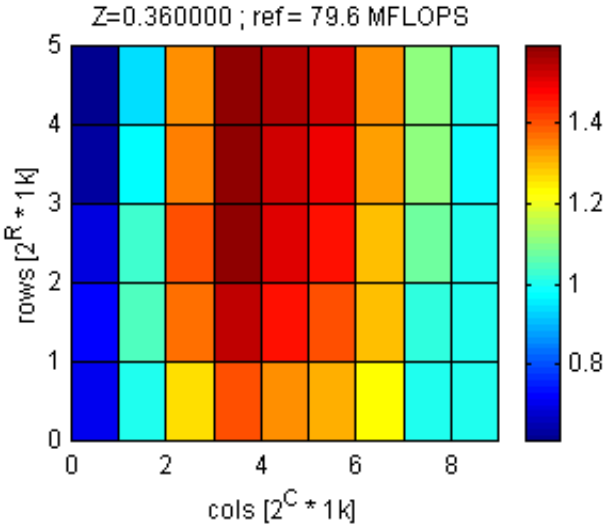
Speedup on four banded random matrices.

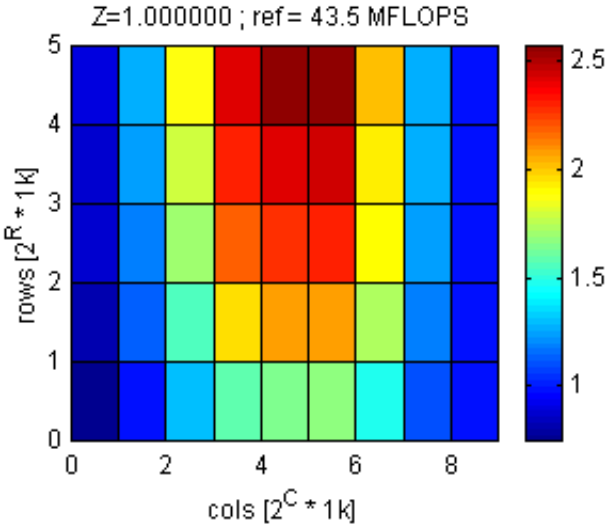


Nishtala, et al (2007). When cache blocking of sparse matrix vector multiply works and why.





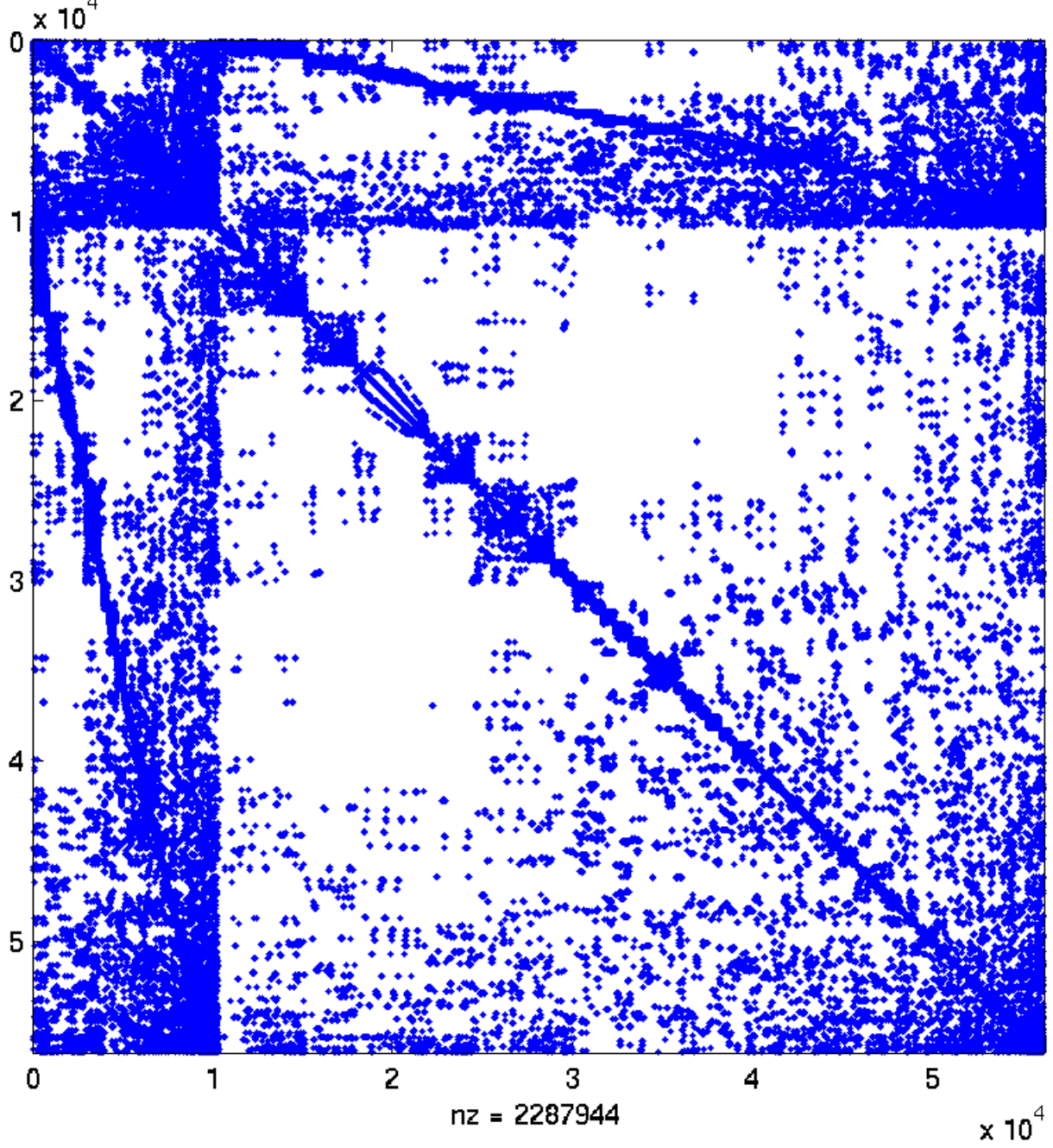


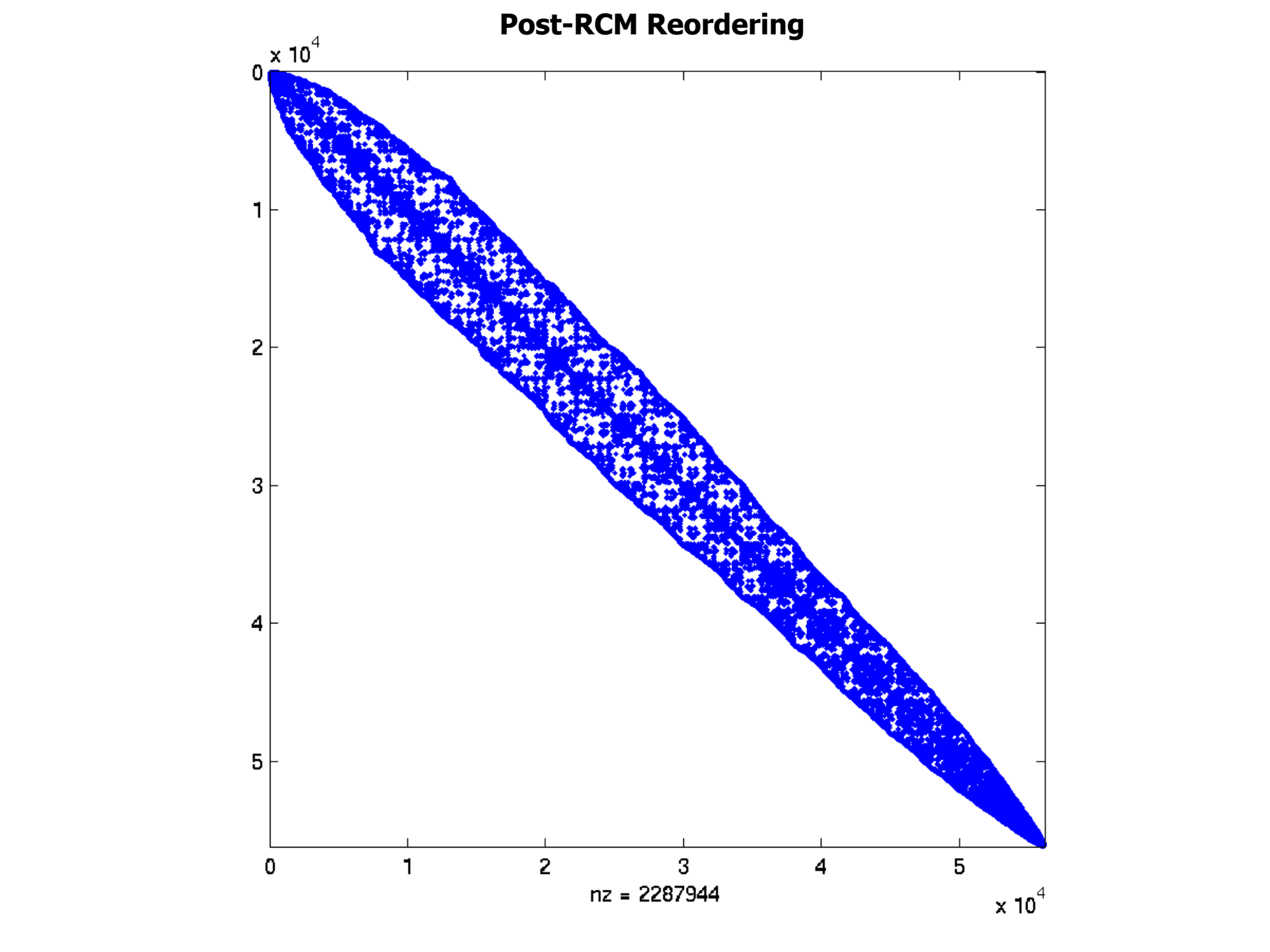


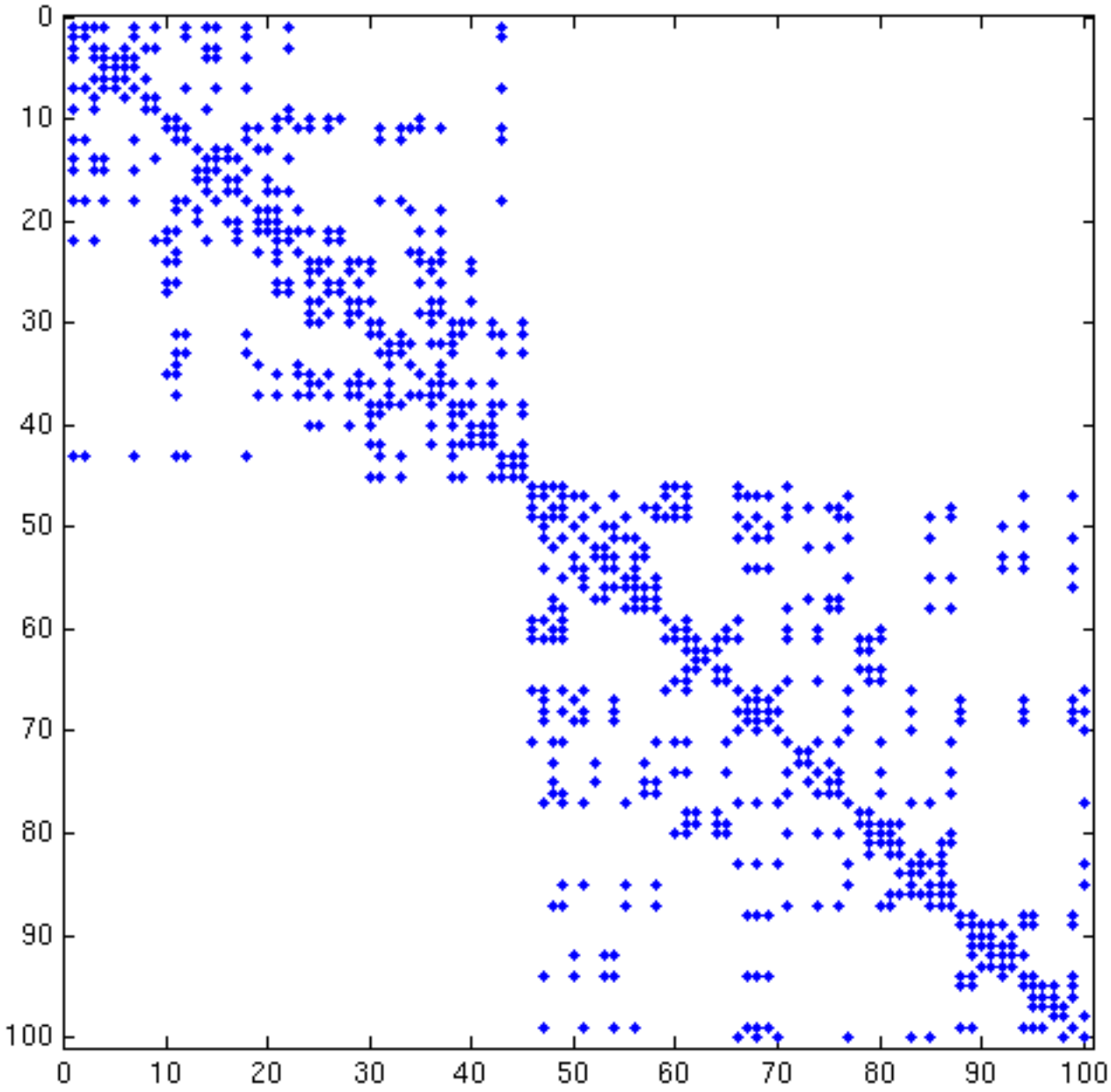
Matrix Reordering: Case study

- 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

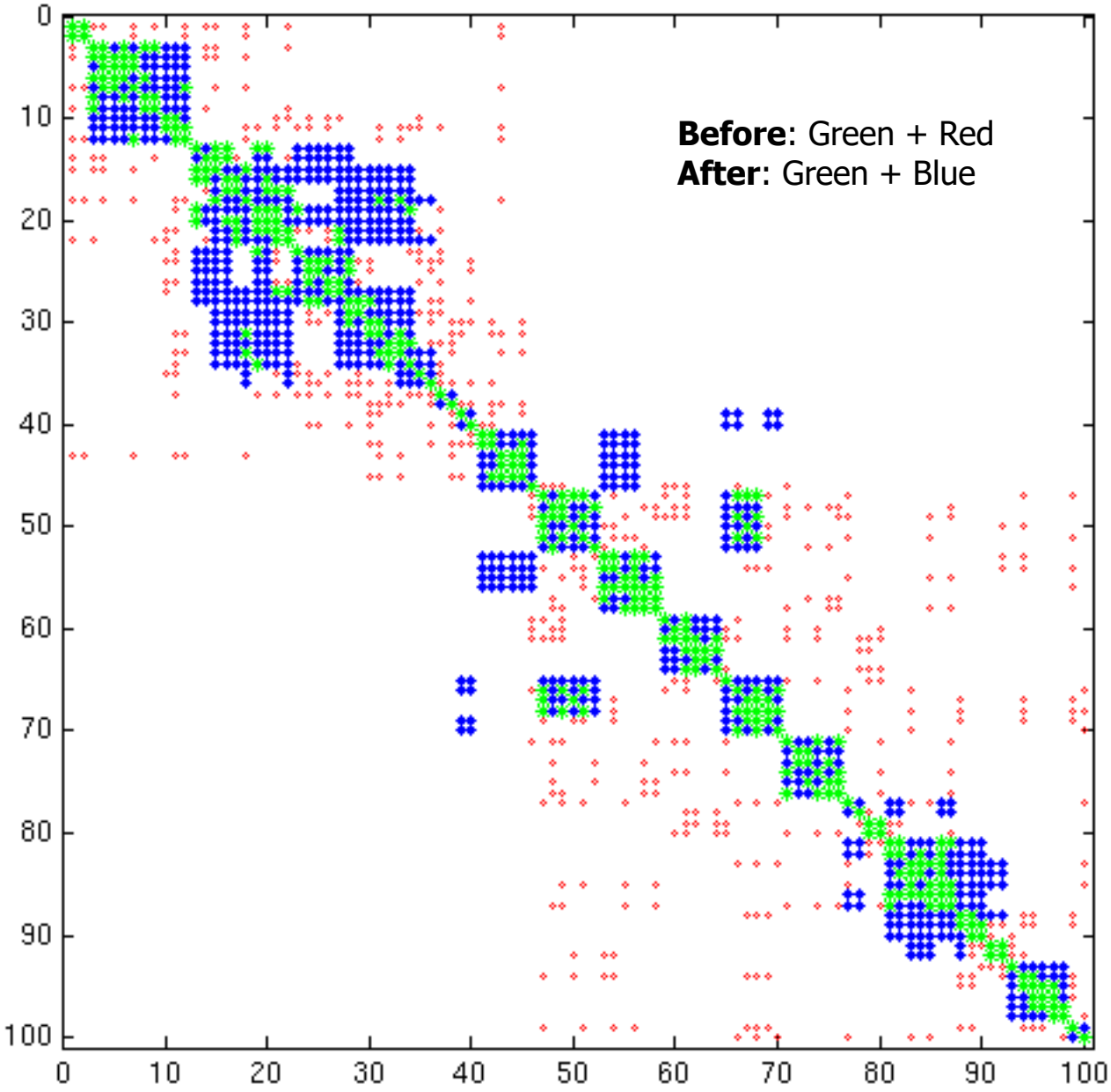
Source: Accelerator Cavity Design Problem (Ko via Husbands)



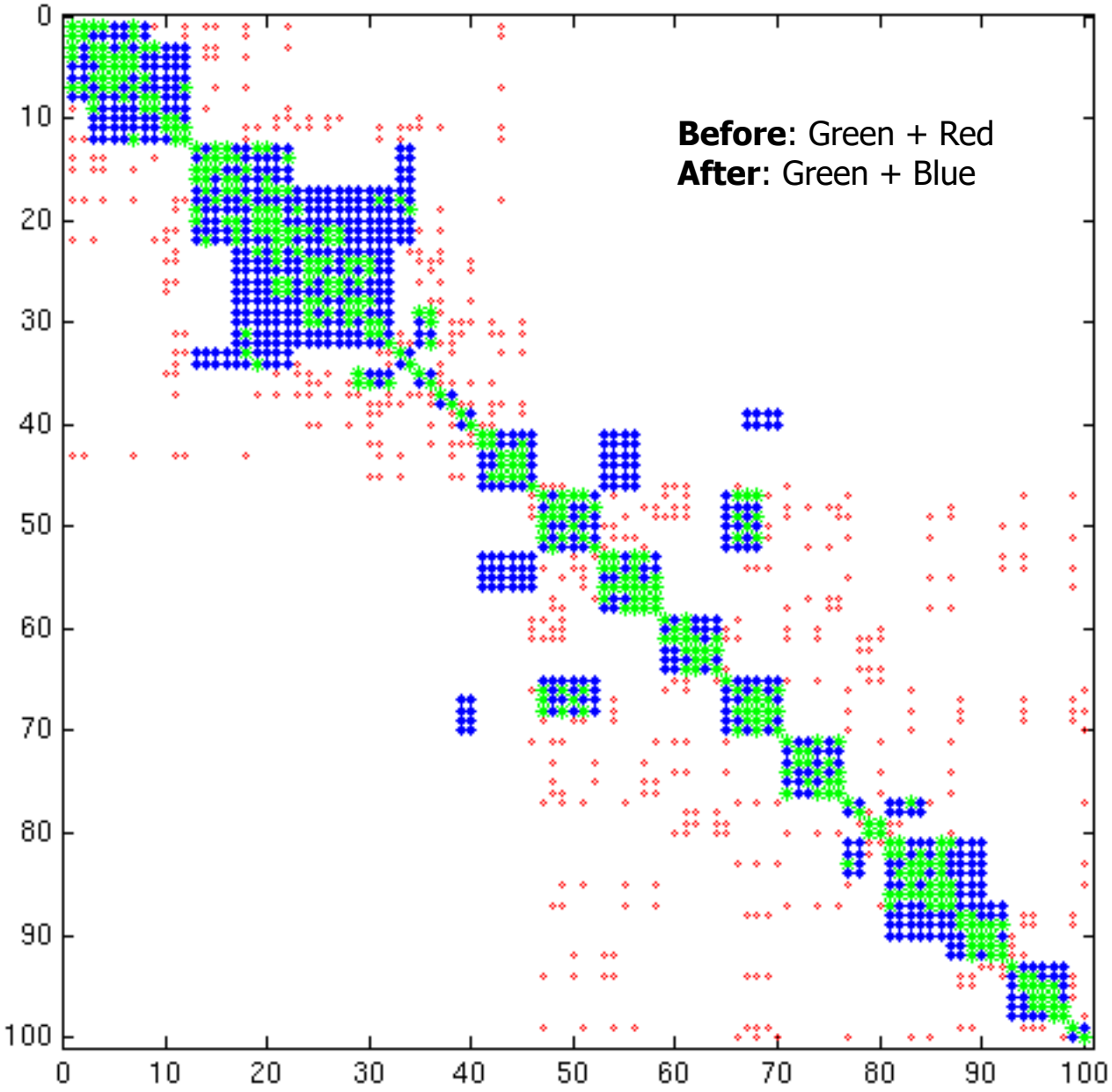

Post-RCM Reordering
x 10^4
0
1
2
3
4
5
0
1
2
3
4
5
nz = 2287944
x 10^4

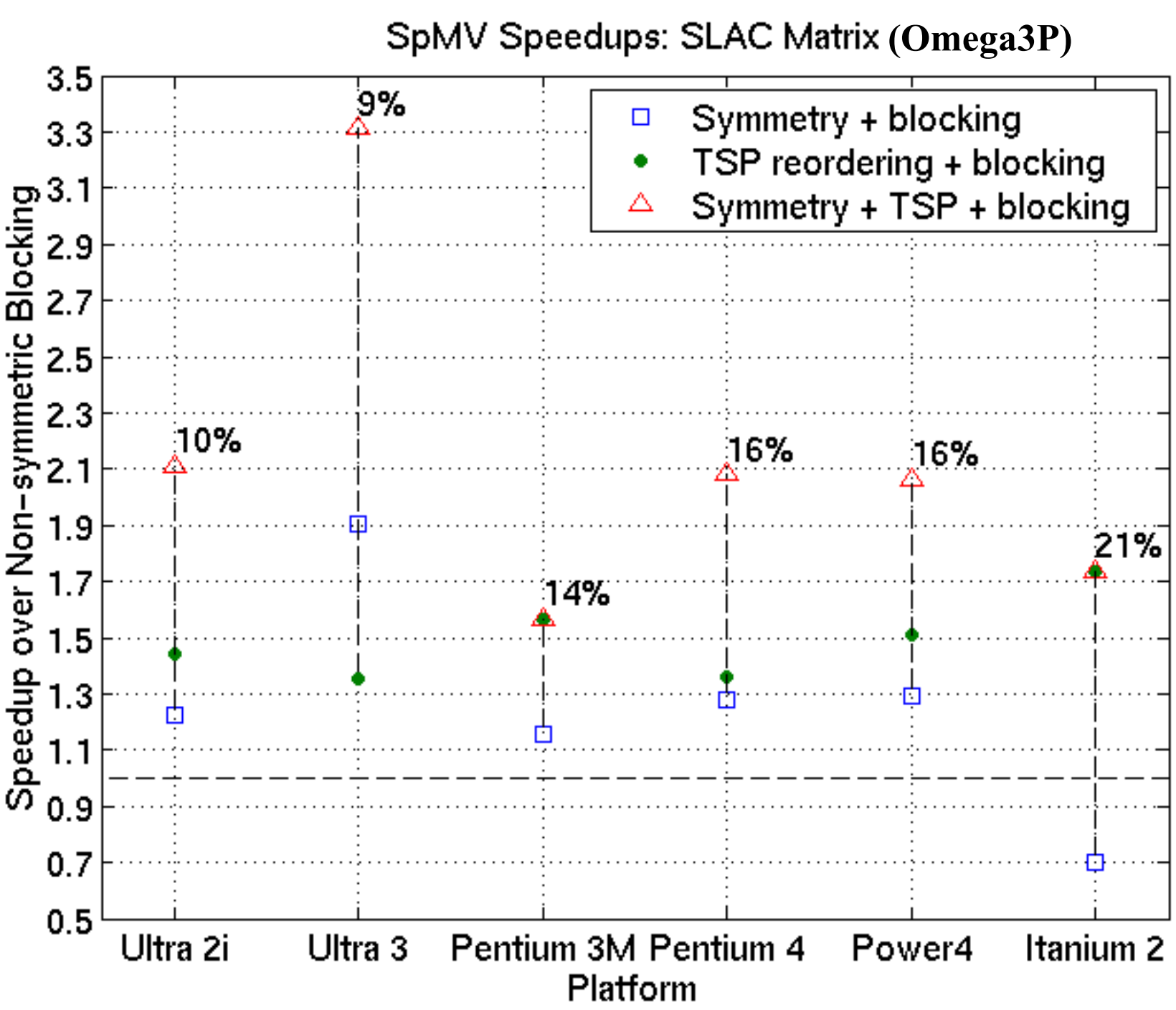

100x100 Submatrix Along Diagonal
0
10
20
30
40
50
60
70
80
90
100
0
10
20
30
40
50
60
70
80
90
100

“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

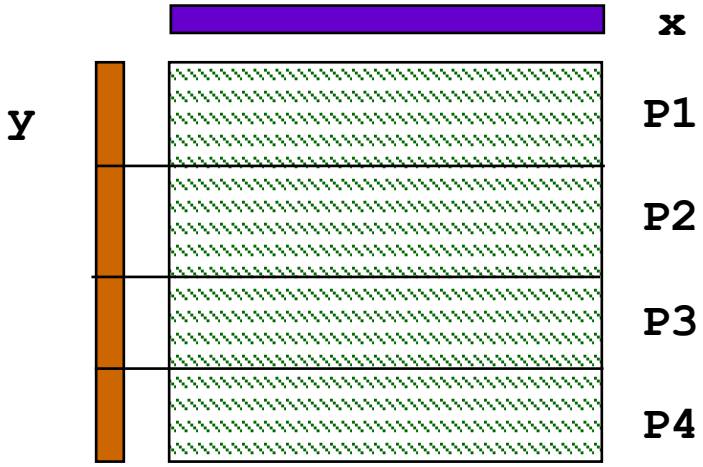
- Optimizations for SpMV
  - **Register blocking (RB)**: up to **4x** over CSR
  - **Variable block splitting**: **2.1x** over CSR, 1.8x over RB
  - **Diagonals**: **2x** over CSR
  - **Reordering** to create dense structure + **splitting**: **2x** over CSR
  - **Symmetry**: **2.8x** over CSR, 2.6x over 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
  - $\mathbf{A \cdot A^T \cdot x}$, $\mathbf{A^T \cdot A \cdot x}$: **4x** over CSR, 1.8x over RB
  - $\mathbf{A^2 \cdot x}$: **2x** over CSR, 1.5x over RB
  - $[A \cdot x, A^2 \cdot x, A^3 \cdot x, .. , A^k \cdot x]$

Outline for today

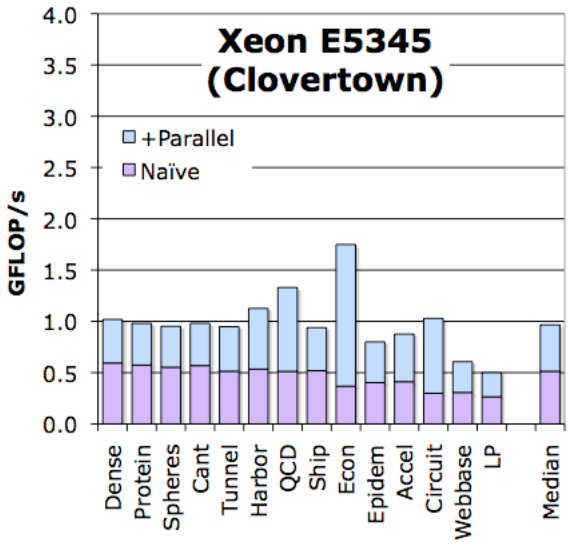
- Homework 1 results
- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

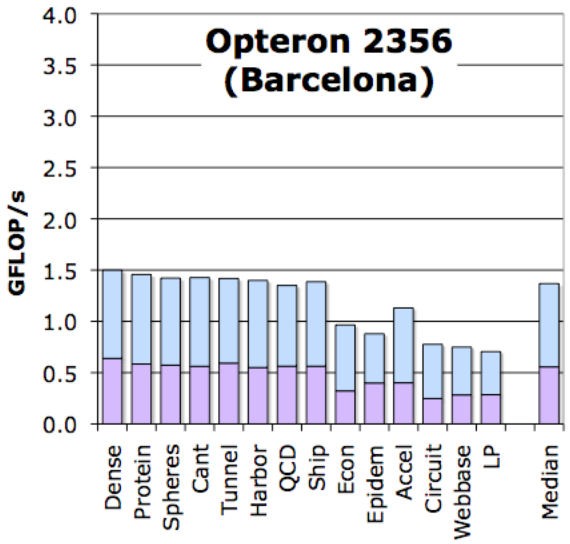
Row parallelism in SpMV

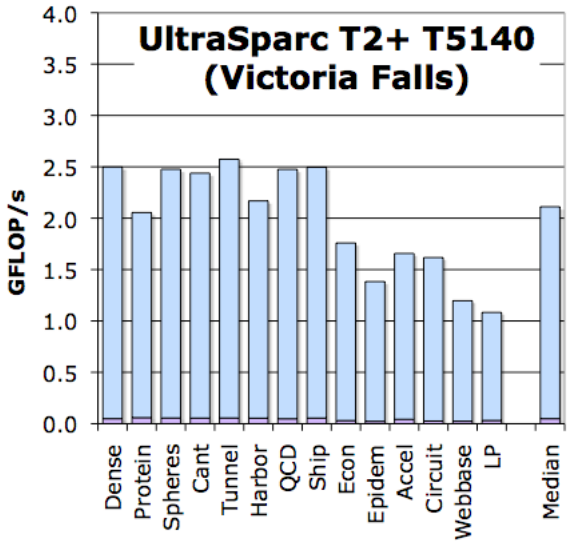
- y = A*x, where A is a sparse matrix
  - In iterative solvers, y is often used to compute next x
- Row parallelism
  - Random access to x
  - No inter-thread dependences, so no races / locks
  - Any performance issues?
- Load balancing
  - Divide number of nonzeros ~evenly, not number of rows
- Compare 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

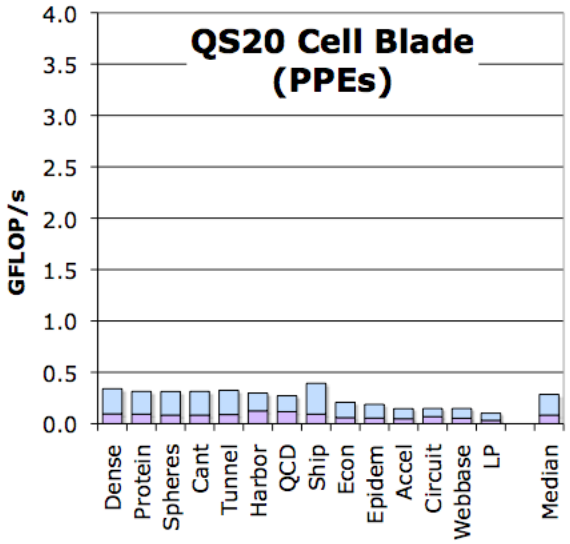


SpMV Performance
(simple parallelization)









- 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

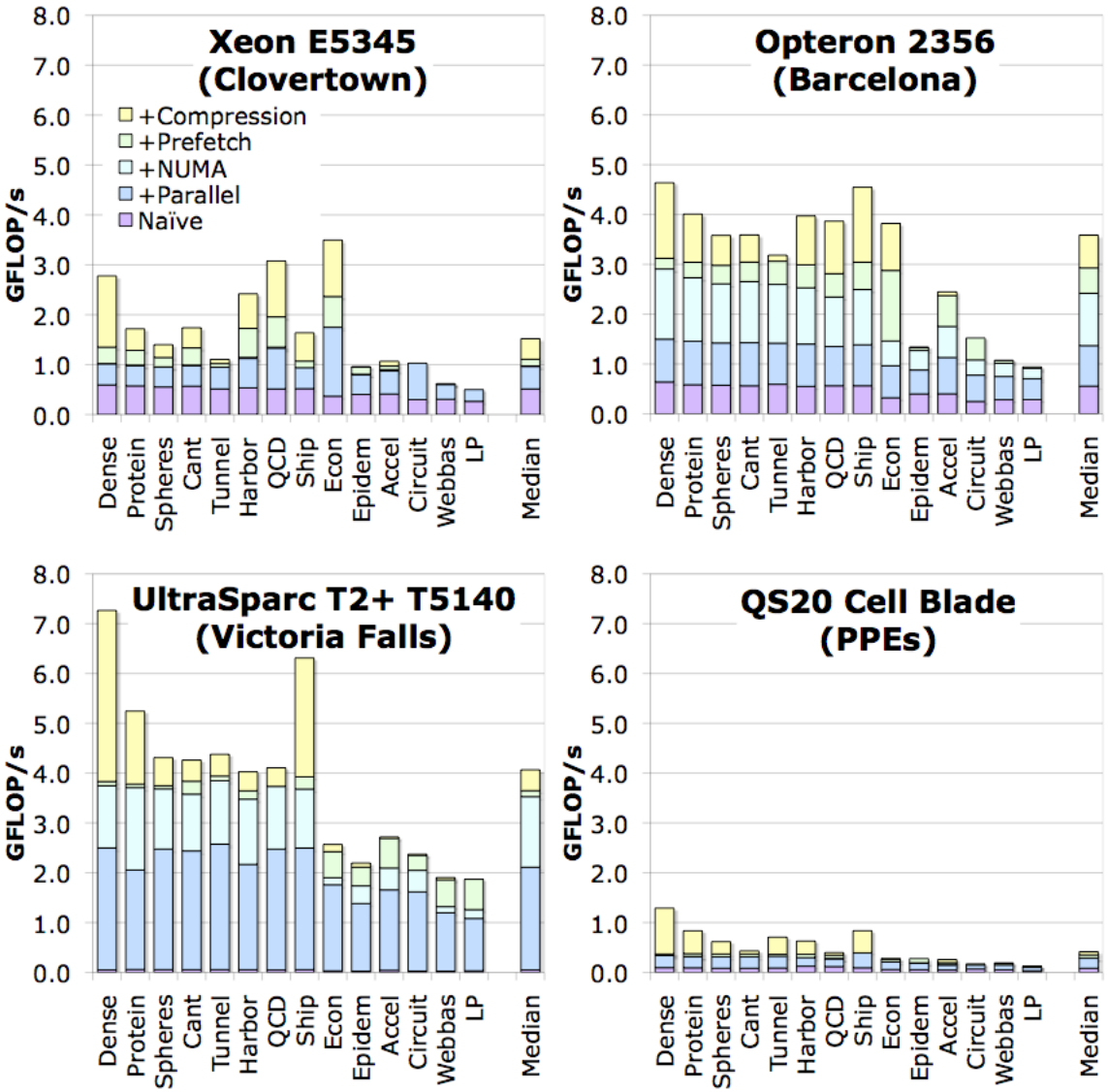
Naïve

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

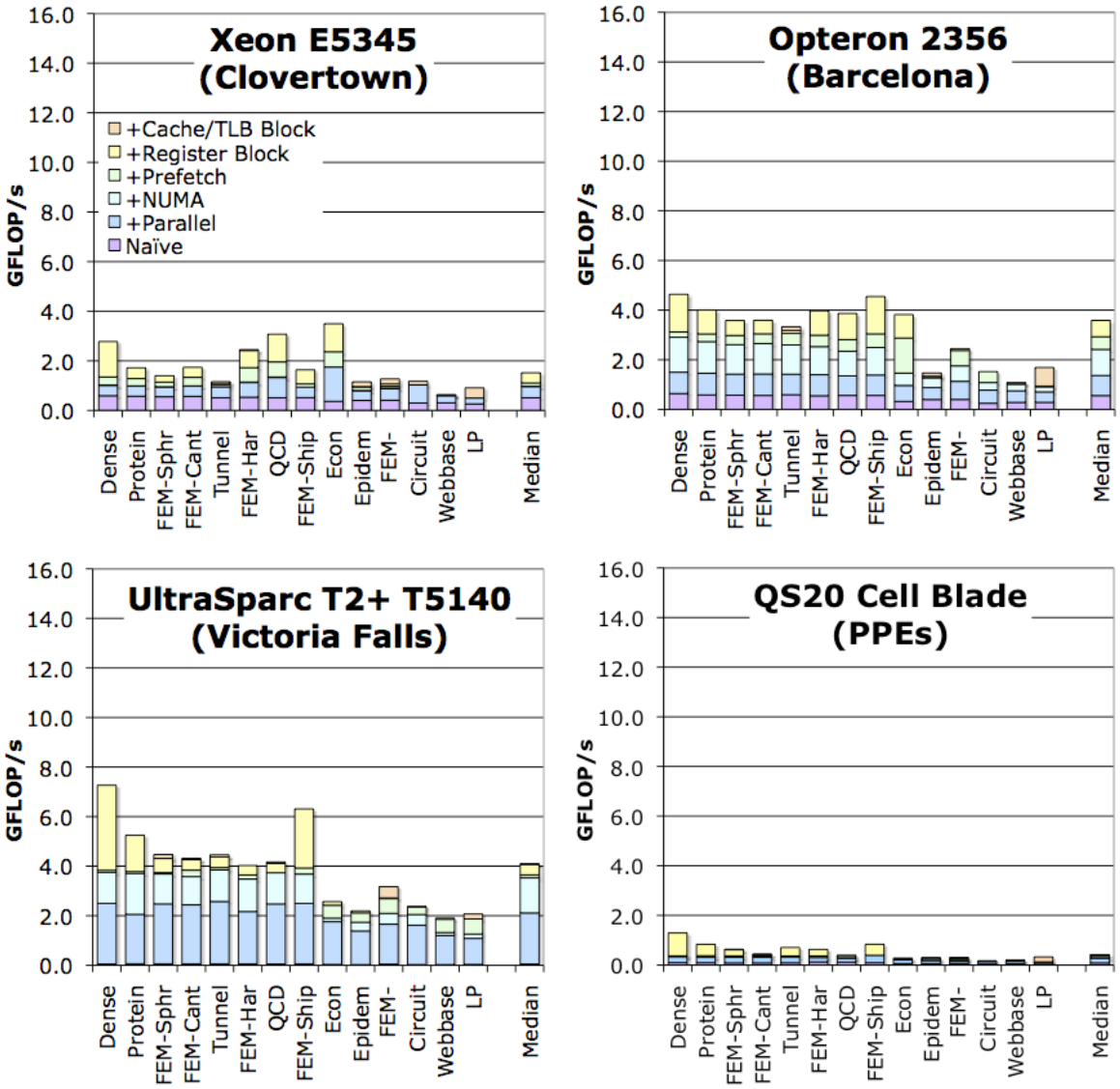
SpMV Performance
(Matrix Compression)



- 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)

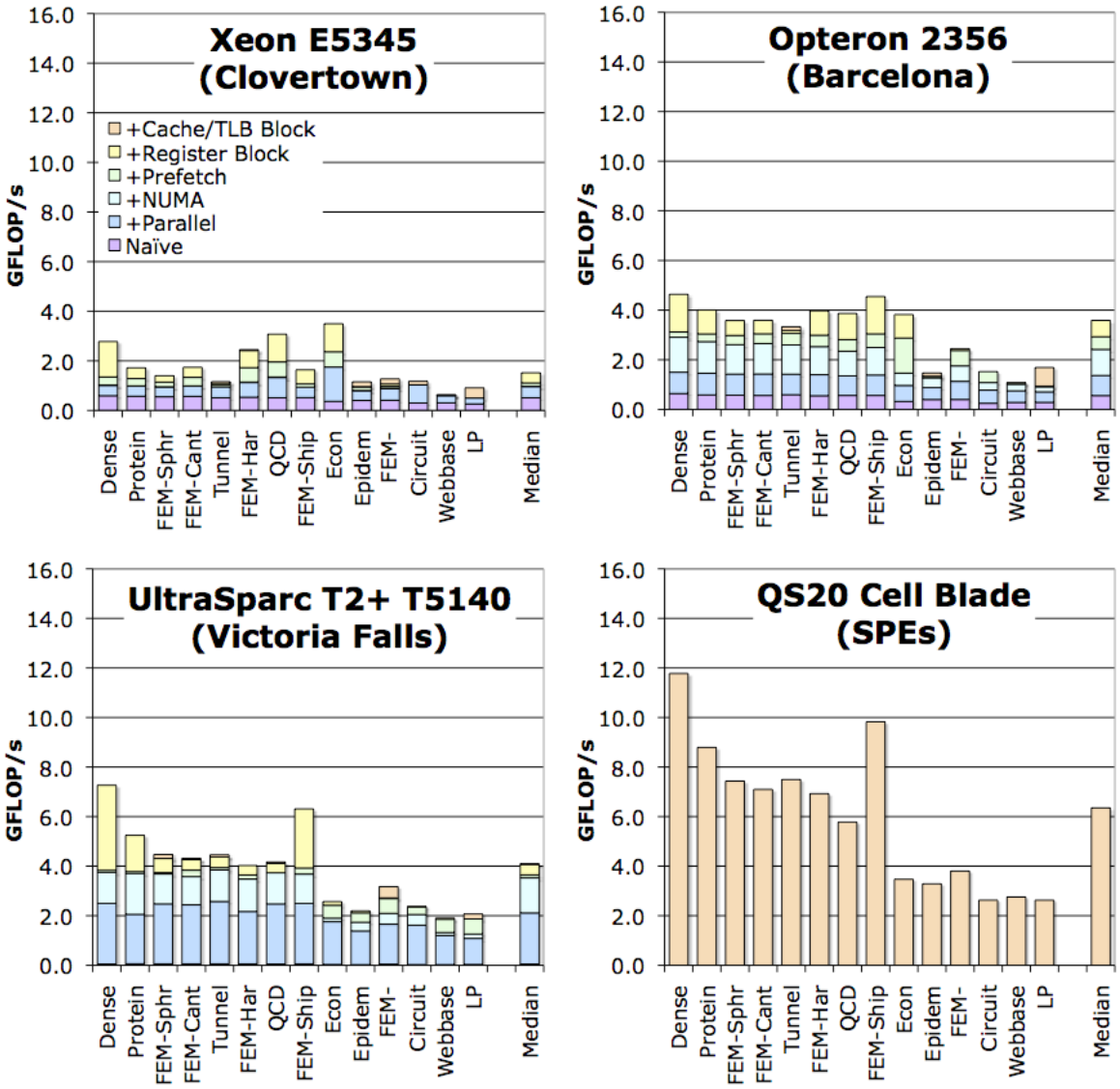


- Fully auto-tuned SpMV performance across the suite of matrices
- Why do some optimizations work better on some architectures?
- **matrices with naturally small working sets**
- **architectures with giant caches**

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

Source: Sam Williams

Auto-tuned SpMV Performance
(architecture specific optimizations)

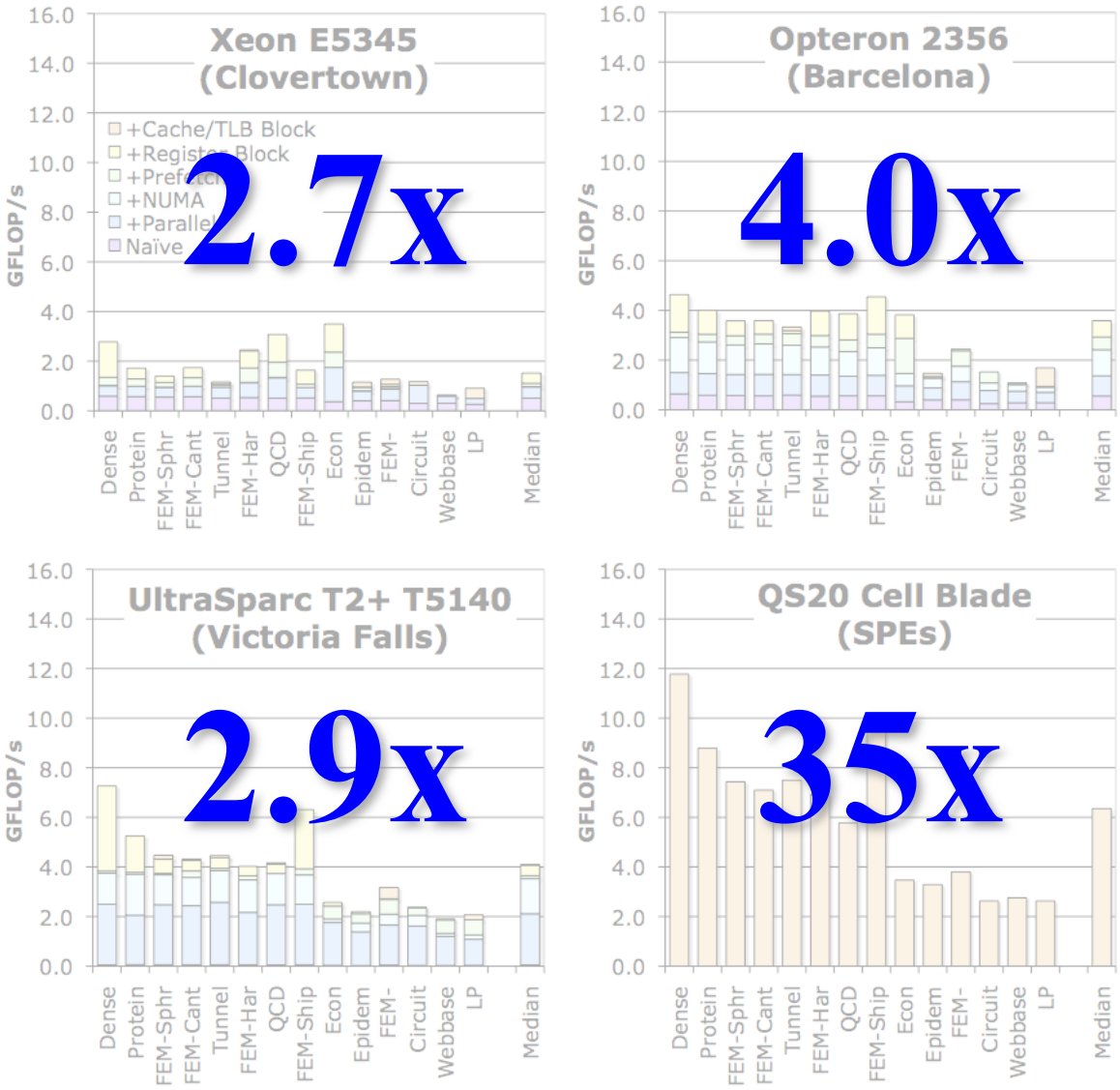


- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?

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

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?

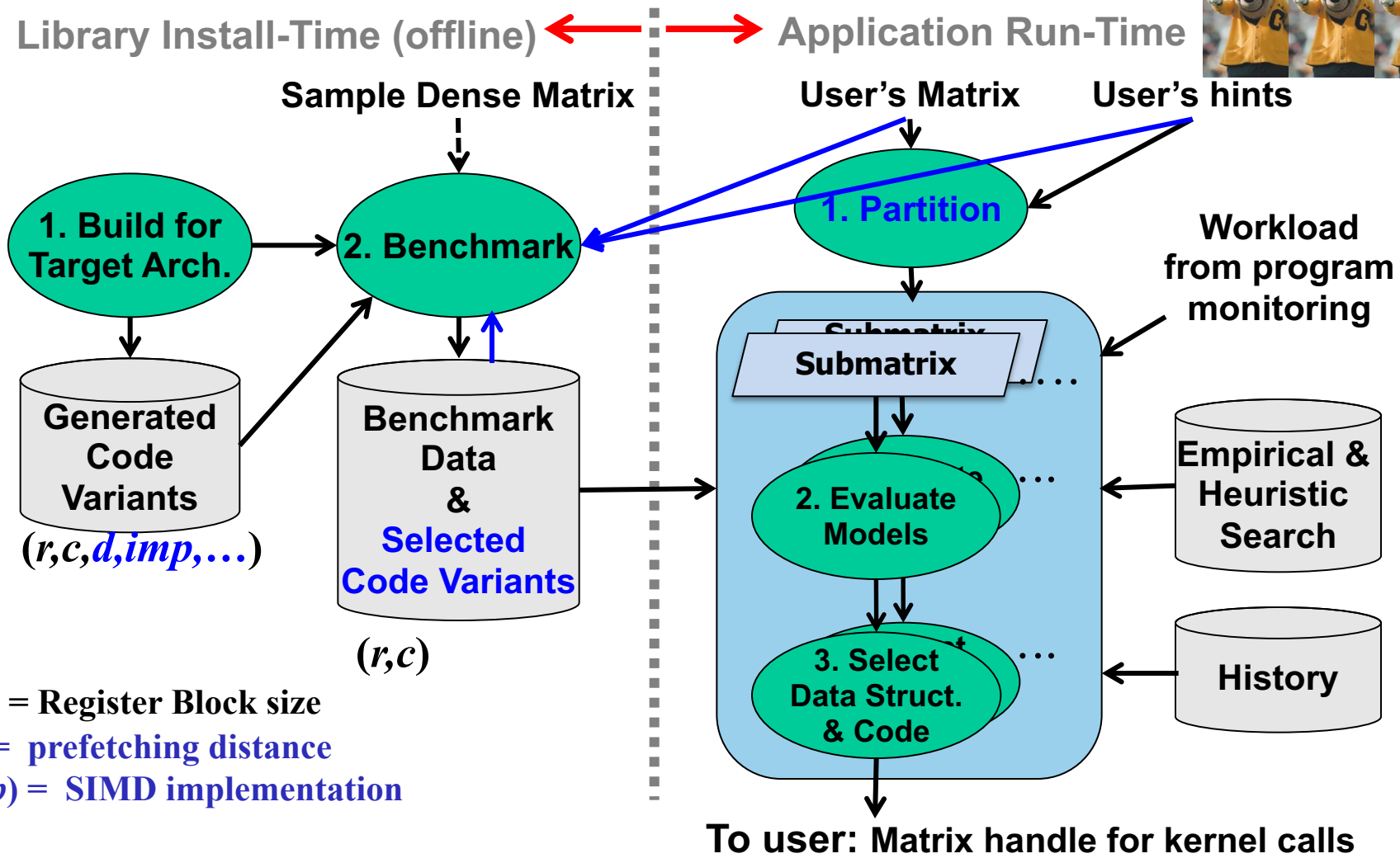
+Cache/LS/TLB Blocking
+Matrix Compression
+SW Prefetching
+NUMA/Affinity
Naïve Pthreads
Naïve

Source: Sam Williams

Optimizations in parallel OSKI (pOSKI)

- Fully automatic heuristics for
  - Sparse matrix-vector multiply (Ax, A^TAx)
    - 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)



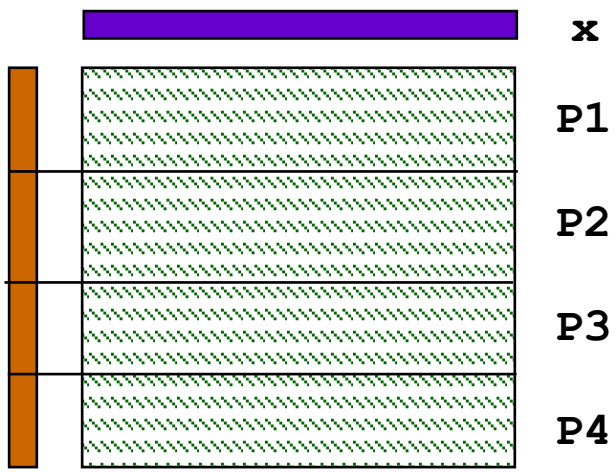
Outline for today

- Homework 1 results
- 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

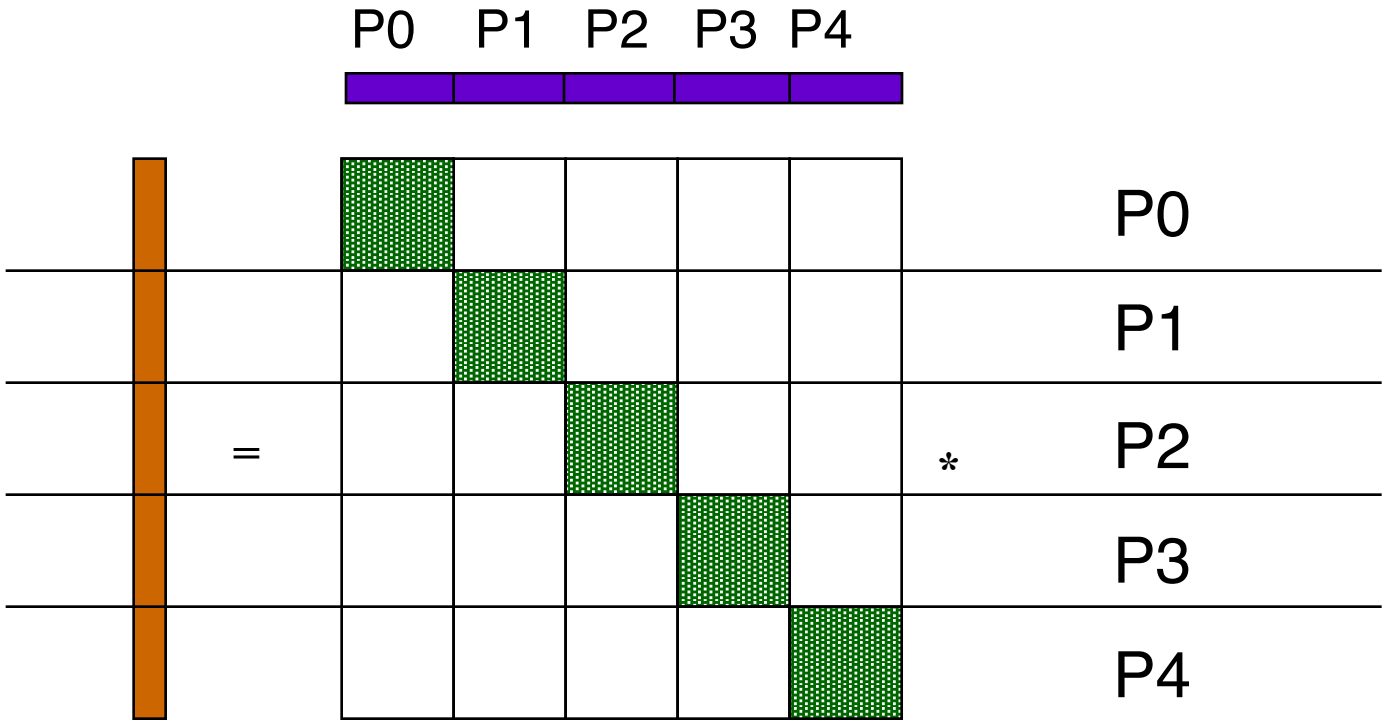
1. y = A*x, where A is a sparse matrix
2. Row parallelism (y & A partitioned)
   1. Replicate x across processors
   2. Or exchange only necessary elements

   Are nonzeros clustered, e.g., near diagonal?
3. Column parallelism (x & A partitioned)
   - Make temporary delta_y = [0,…] on all processors;
   - Update that; and add-reduce across processors
4. 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

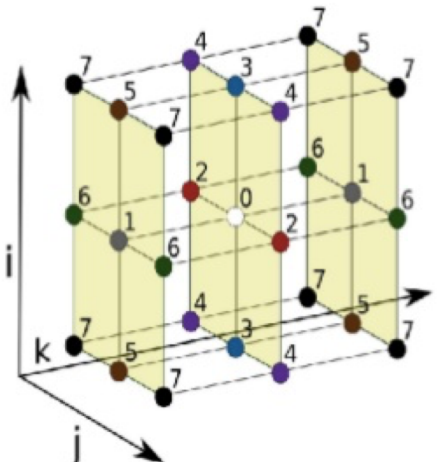
- "Ideal" matrix structure for parallelism: block diagonal
  - p (number of processors) blocks, can all be computed locally.
  - If no non-zeros outside these blocks, no communication needed
- Can we reorder the rows/columns to get close to this?
  - Most nonzeros in diagonal blocks, few outside



HPCG Benchmark

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.



HPCG Results (Nov 2017)

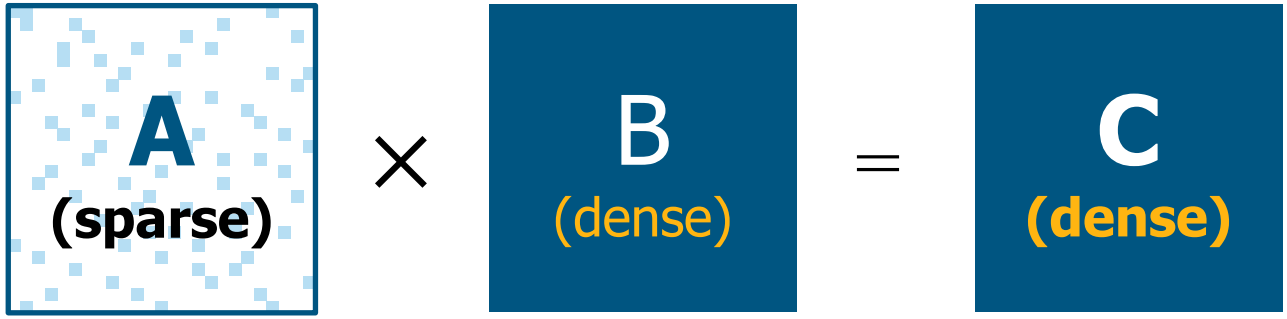
Rank	Site	Computer	Cores	HPL (Pflop/s)	TOP500 Rank	HPCG (Pflop/s)	Fraction of Peak
1	RIKEN Advanced Institute for Computational Science Japan	**K computer** – , SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10.51	10	0.603	5.30%
2	NSCC / Guangzhou China	**Tianhe-2 (MilkyWay-2)** – TH-IVB-FEP 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 for today

- Homework 1 results
- Sparse matrix formats and basic SpMV
- Register blocking and autotuning SpMV
- Cache blocking SpMV on multicore
- Distributed memory
- Sparse matmult
- CA iterative solvers

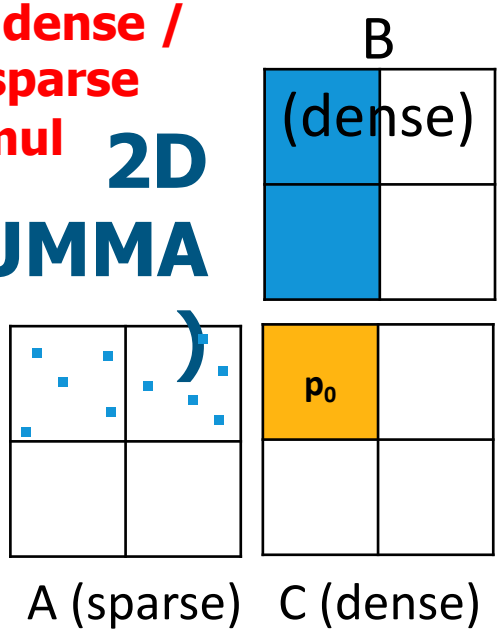
Sparse × Dense Matmul

These are not necessarily square

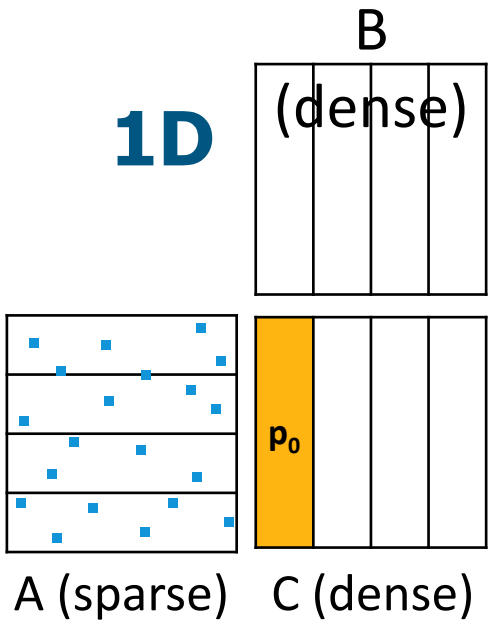


2D/2.5D/3D only optimal for dense-dense / sparse-sparse matmul

2D (SUMMA)



1D

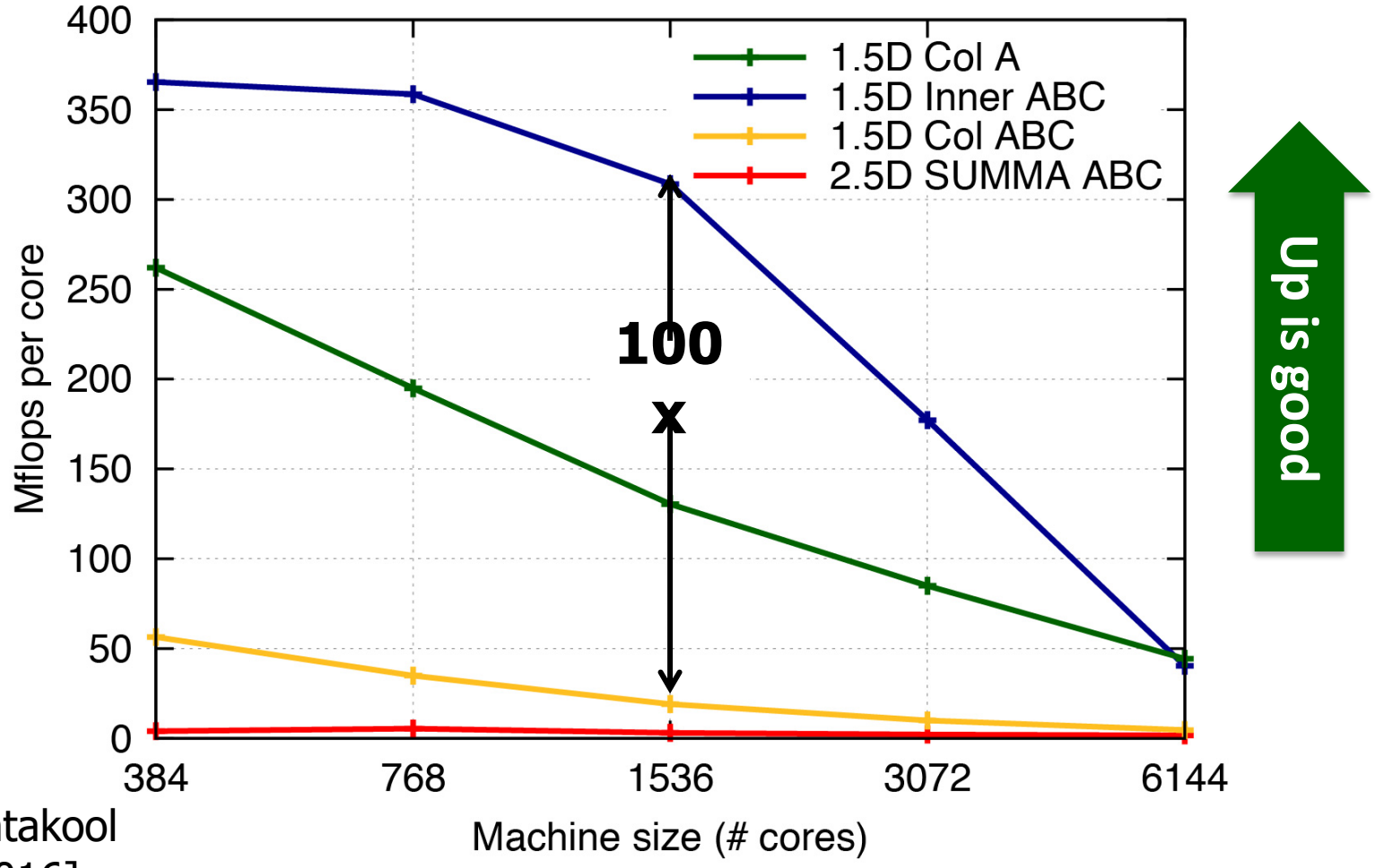


+ replication = 1.5D

[Koanantakool et al. 2016]

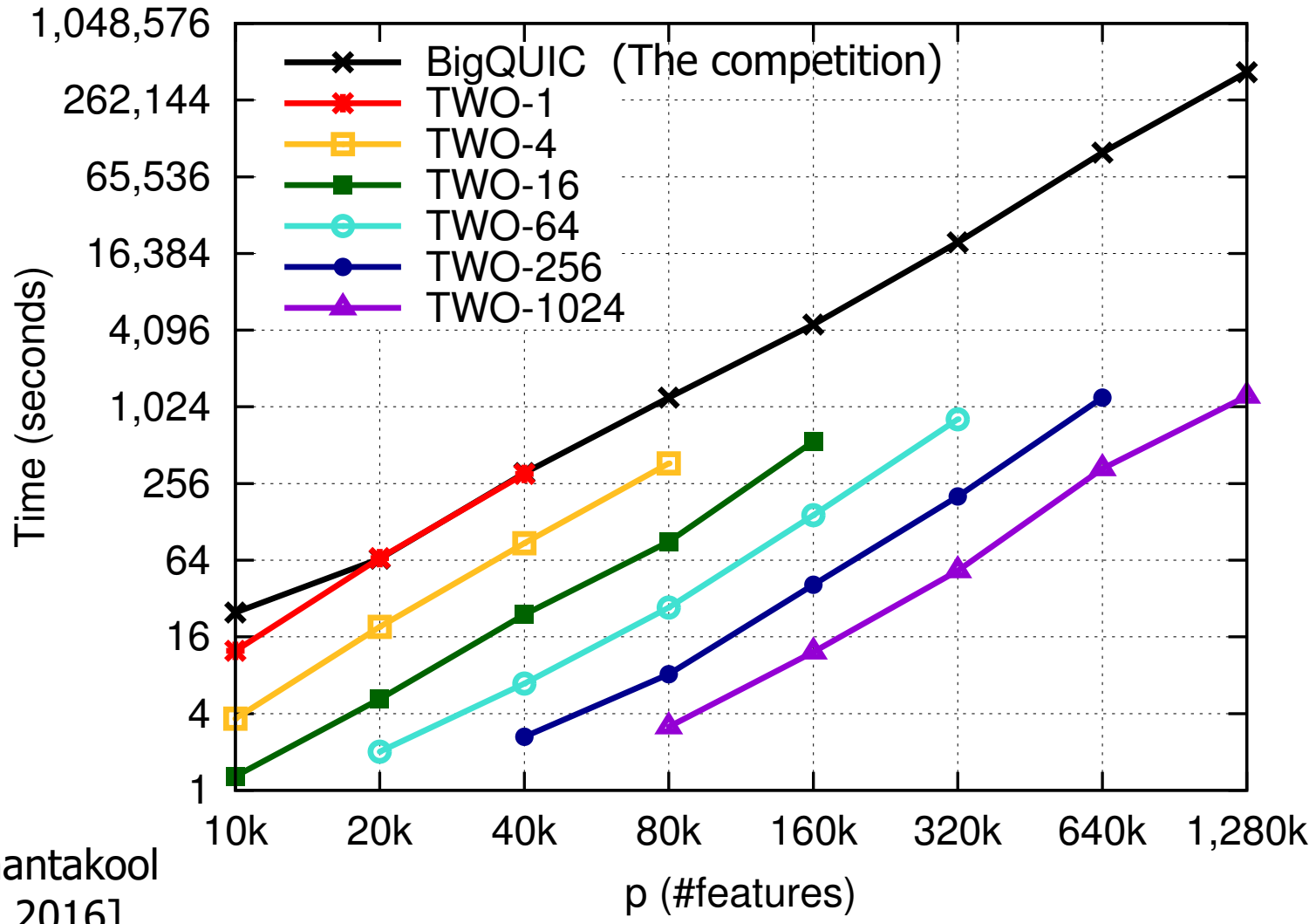
100x Improvement for the right algorithm

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



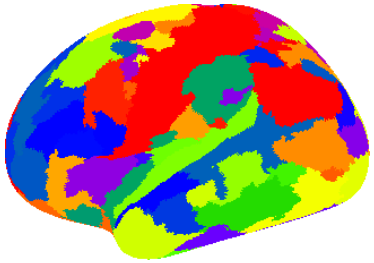
[Koanantakool et al. 2016]

Inverse Covariance Matrix Estimation (CONCORD) using fast SpDM^3 algorithm

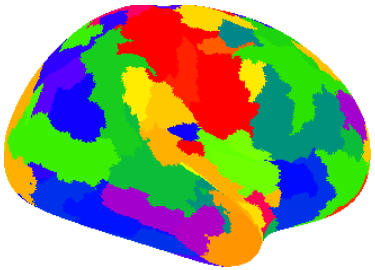


[Koanantakool et al. 2016]

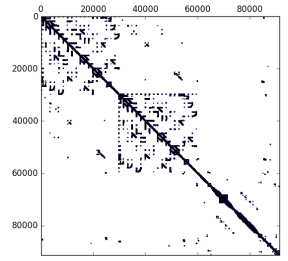
HP-CONCORD on Brain fMRI data



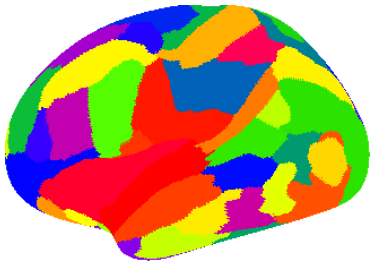
$\lambda_1 = 0.48,\ \lambda_2 = 0.39,\ \epsilon = 3,$
% of best score = 100



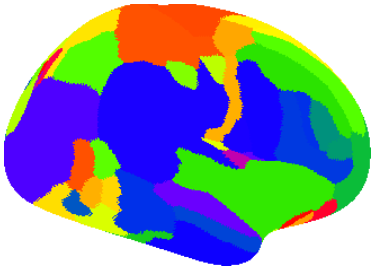
$\lambda_1 = 0.5,\ \lambda_2 = 0.39,\ \epsilon = 3,$
% of best score = 100



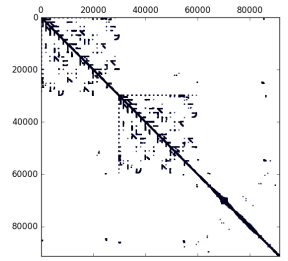
$\lambda_1 = 0.48,\ \lambda_2 = 0.39,\ \epsilon = 3,$
% of best score = 100



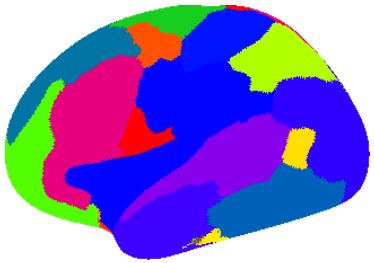
$\lambda_1 = 0.64,\ \lambda_2 = 0.13,\ k = 1,$
% of best score = 75.03



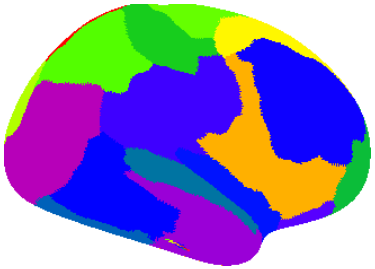
$\lambda_1 = 0.5425,\ \lambda_2 = 0.39,\ k = 0,$
% of best score = 73.45



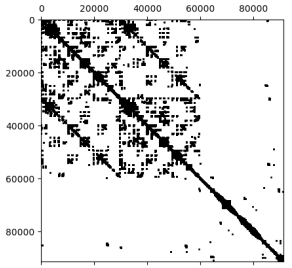
$\lambda_1 = 0.64,\ \lambda_2 = 0.13,\ k = 1,$
% of best score = 75.03



$t = 99.9,\ k = 4,$
% of best score = 32.24



$t = 99.9,\ k = 3,$
% of best score = 32.45



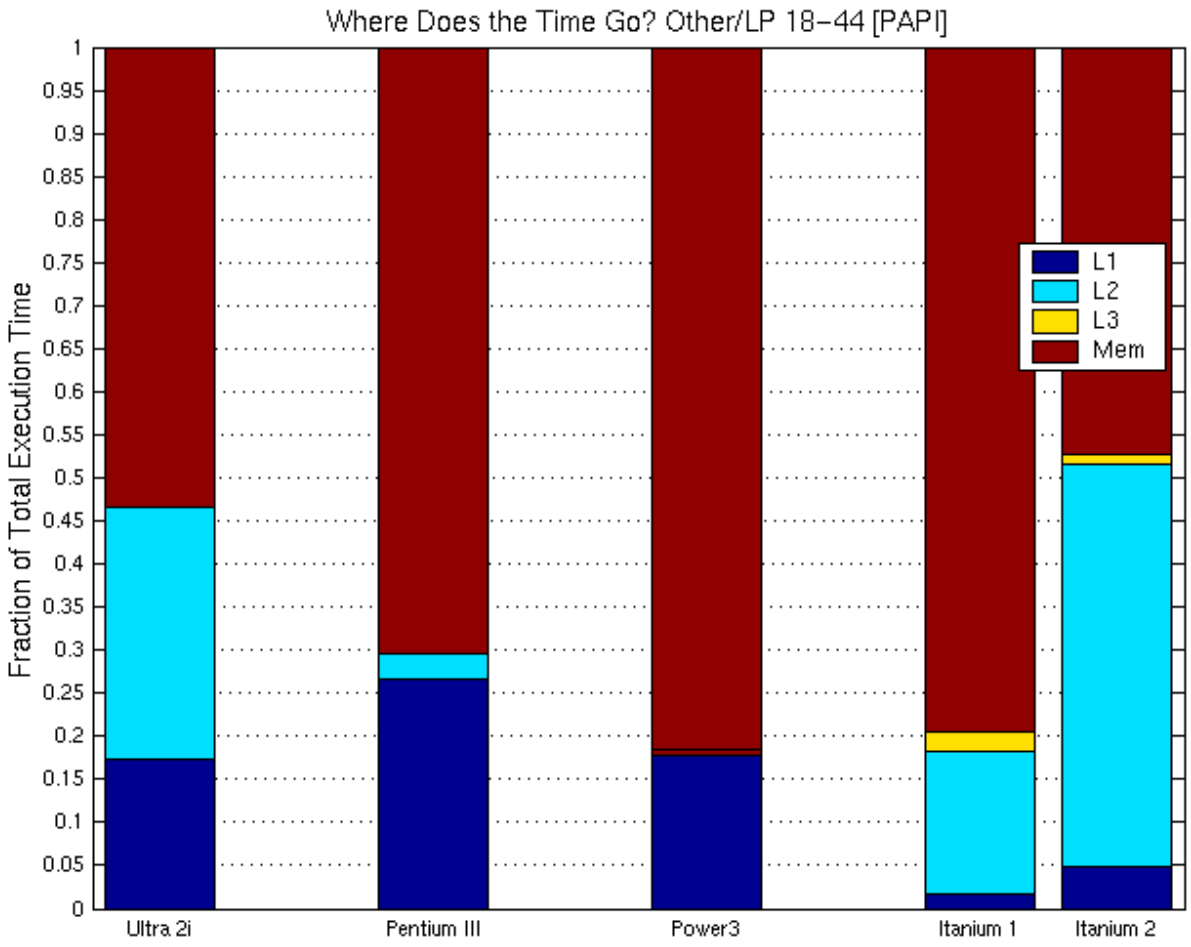
$t = 99.9,\ k = 4$
% of best score = 32.24

[Koanantakool et al. 2018]

Outline for today

- Homework 1 results
- 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?

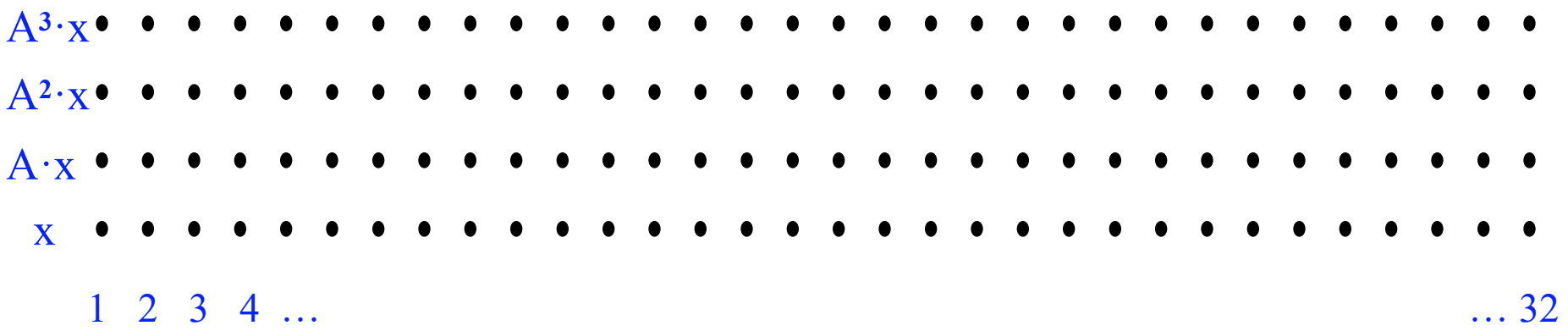
- Iterative methods all depend on it
- But speedups are limited
  - Just 2 flops per nonzero
  - Communication costs dominate
- Can we beat this bottleneck?
- Need to look at next level in stack:
  - What do algorithms that use SpMV do?
  - Can we reorganize them to avoid communication?
- Only way significant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

Tuning Higher Level Algorithms than SpMV

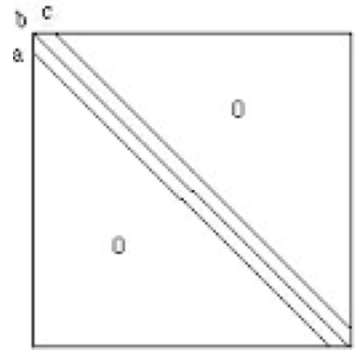
- We almost always do many SpMVs, not just one
  - "Krylov Subspace Methods" (KSMs) for Ax=b, Ax = λx
    - Conjugate Gradients, GMRES, Lanczos, …
  - Do a sequence of k SpMVs to get vectors $[x_1, \ldots, x_k]$
  - Find best solution x as linear combination of $[x_1, \ldots, x_k]$
- Main cost is k SpMVs
- 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]

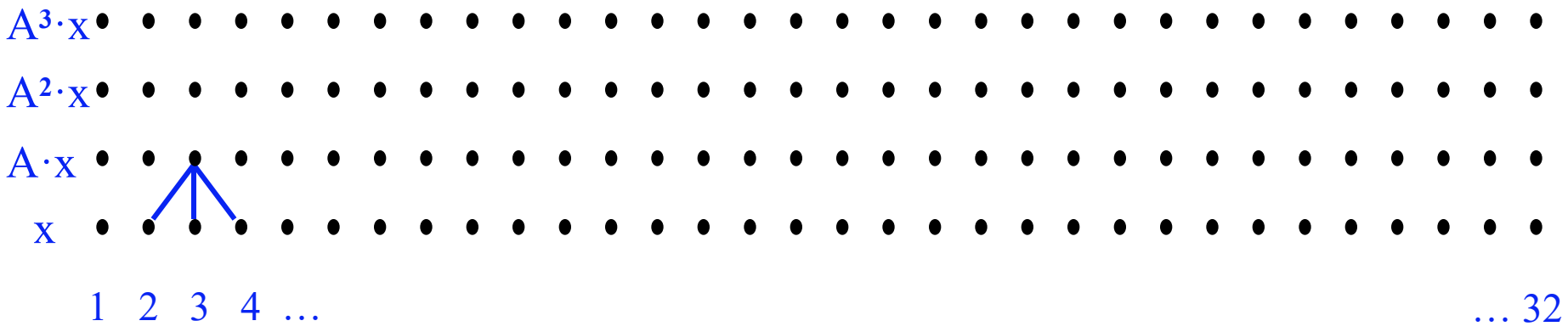


- Example: A tridiagonal, n=32, k=3
- Works for any "well-partitioned" A

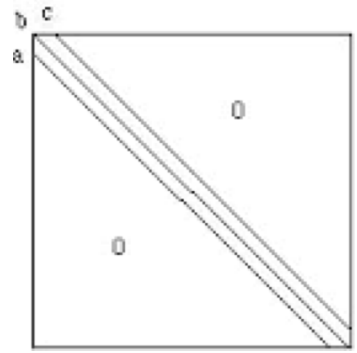


Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

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

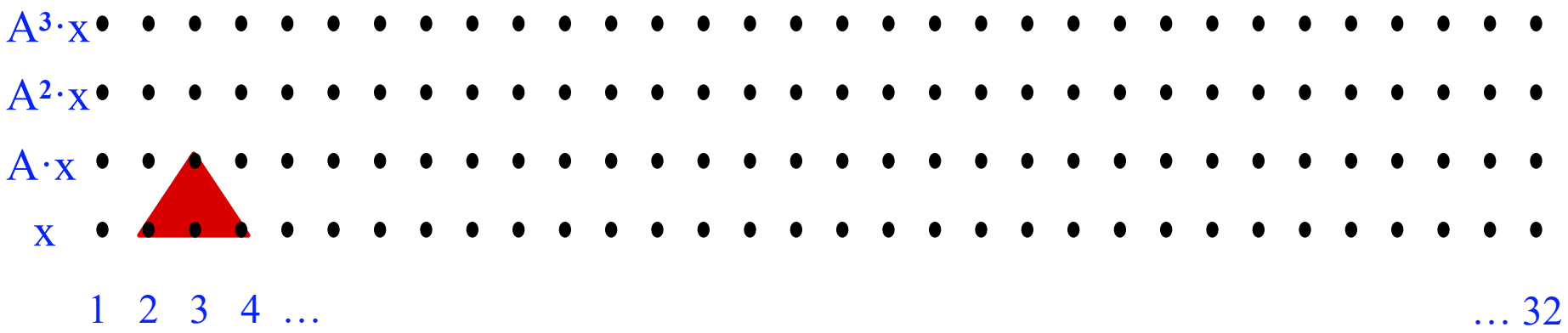


- Example: A tridiagonal, n=32, k=3



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

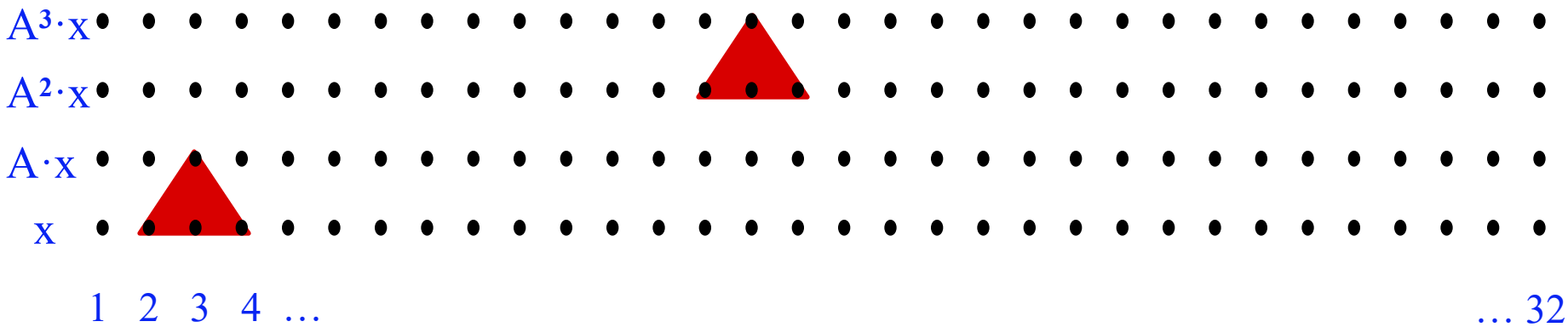
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

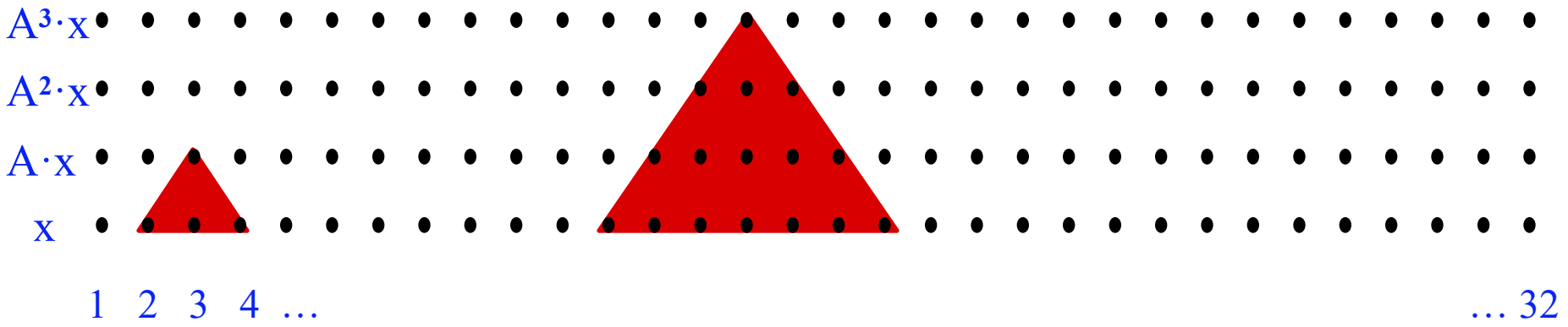
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

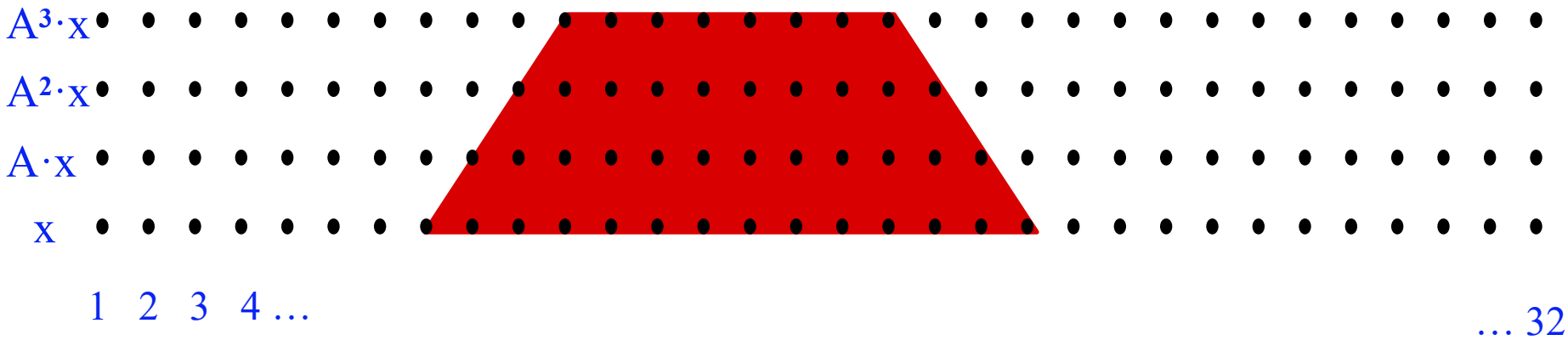
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

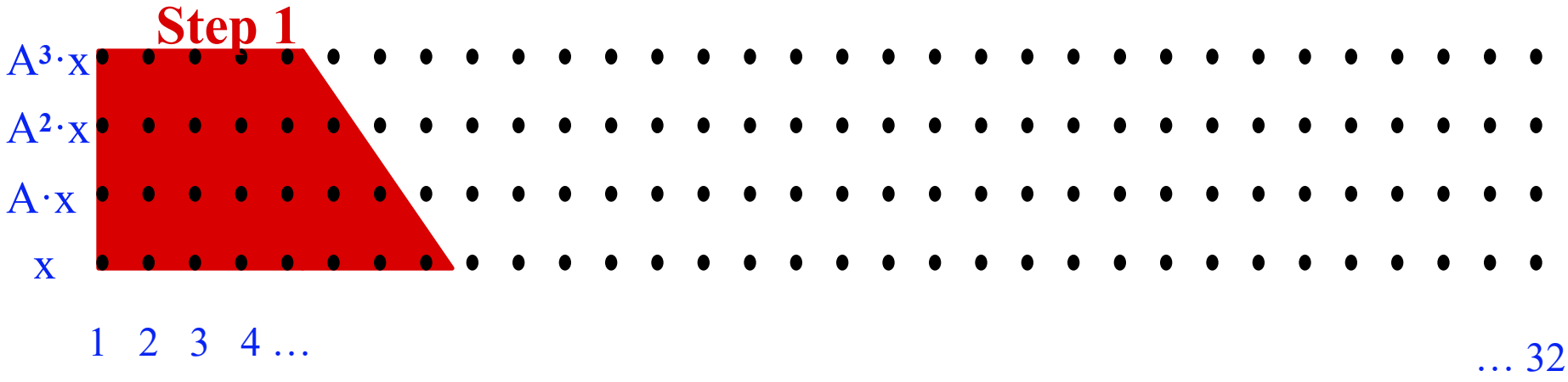
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

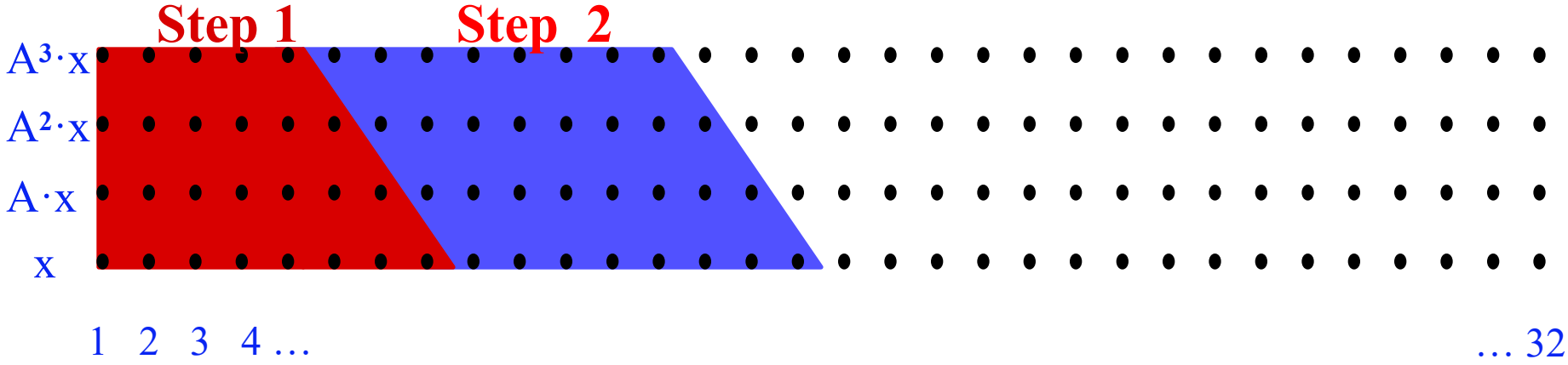
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

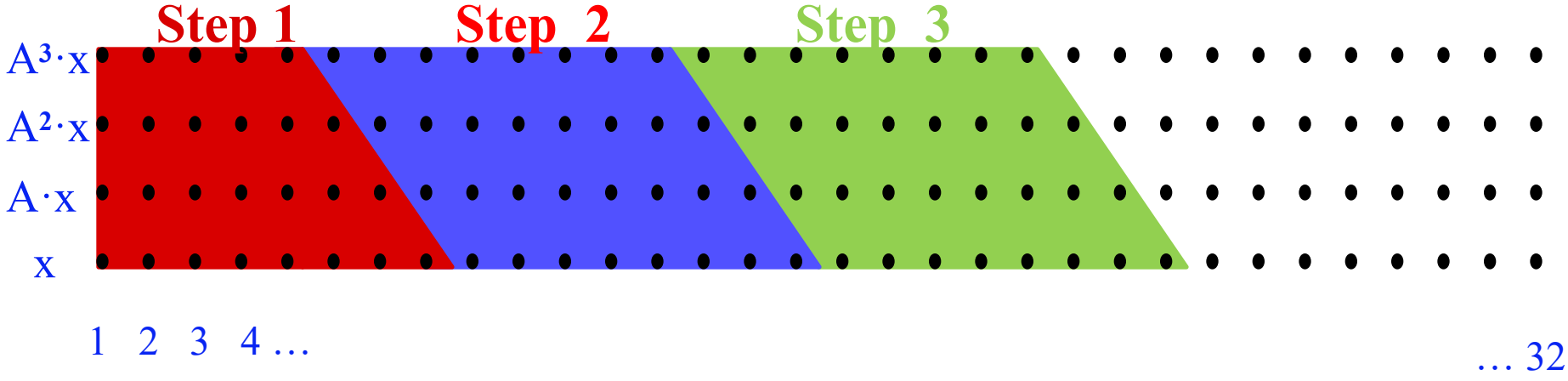
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

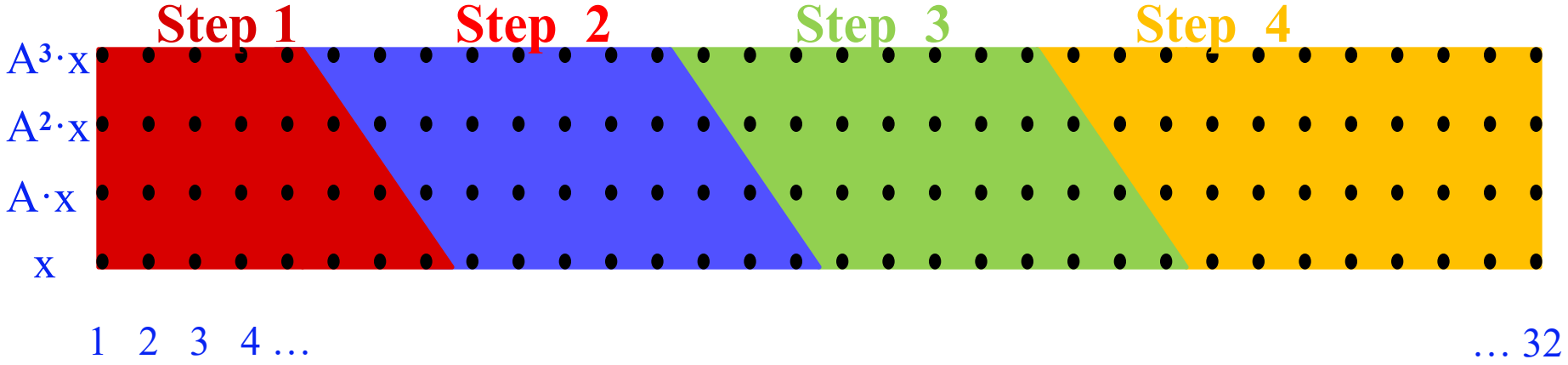
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

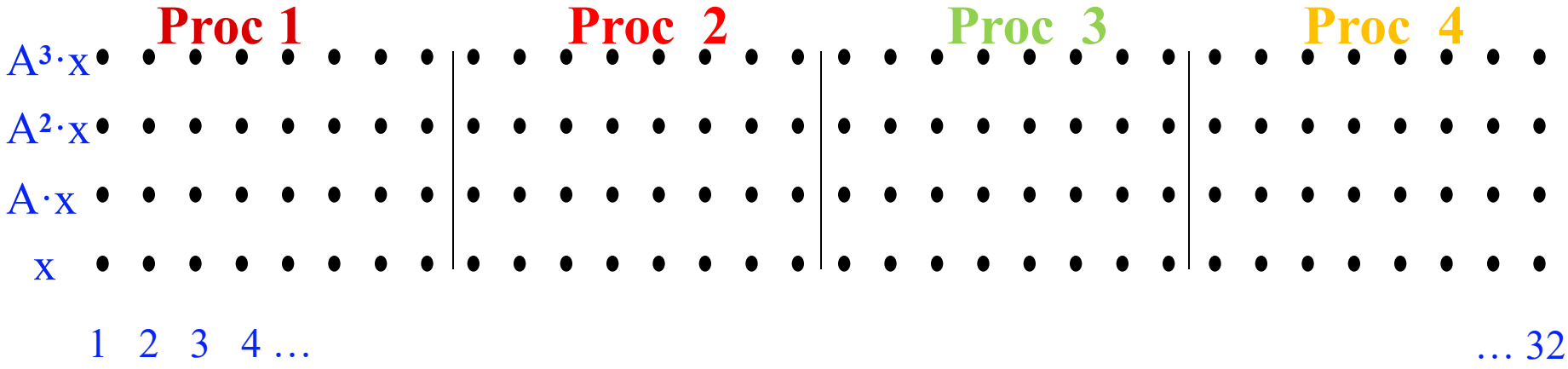
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

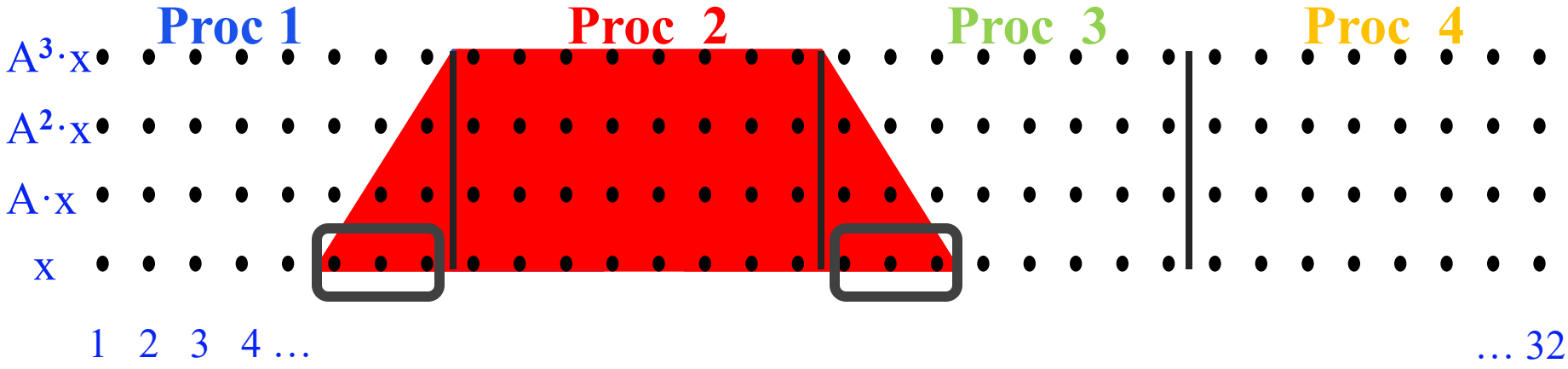
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

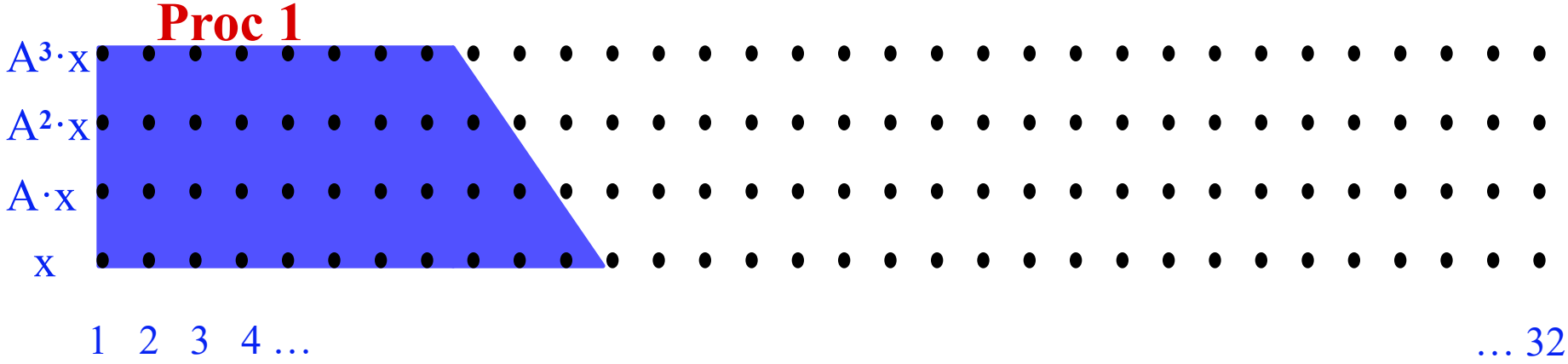
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



- 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]

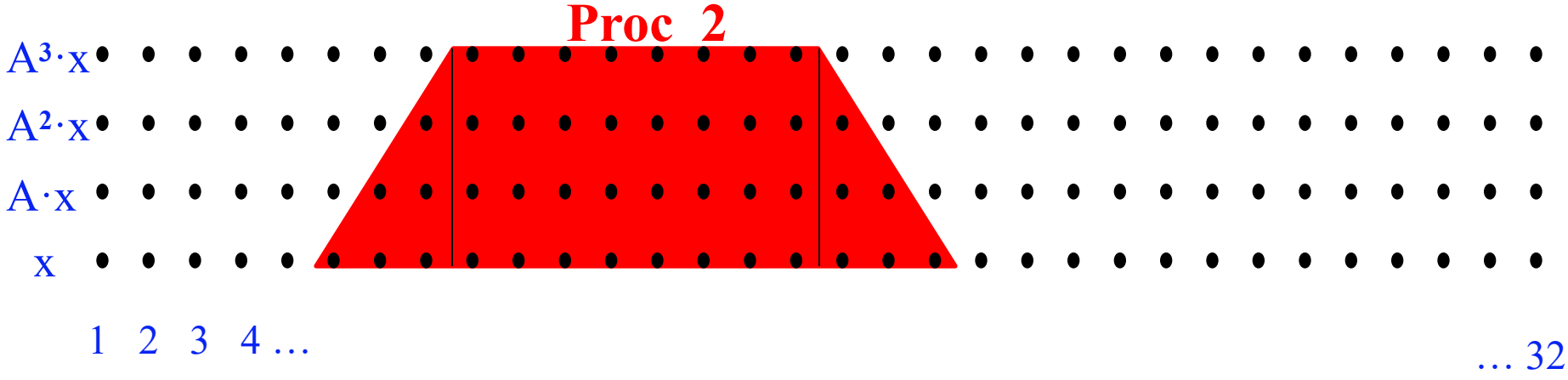
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

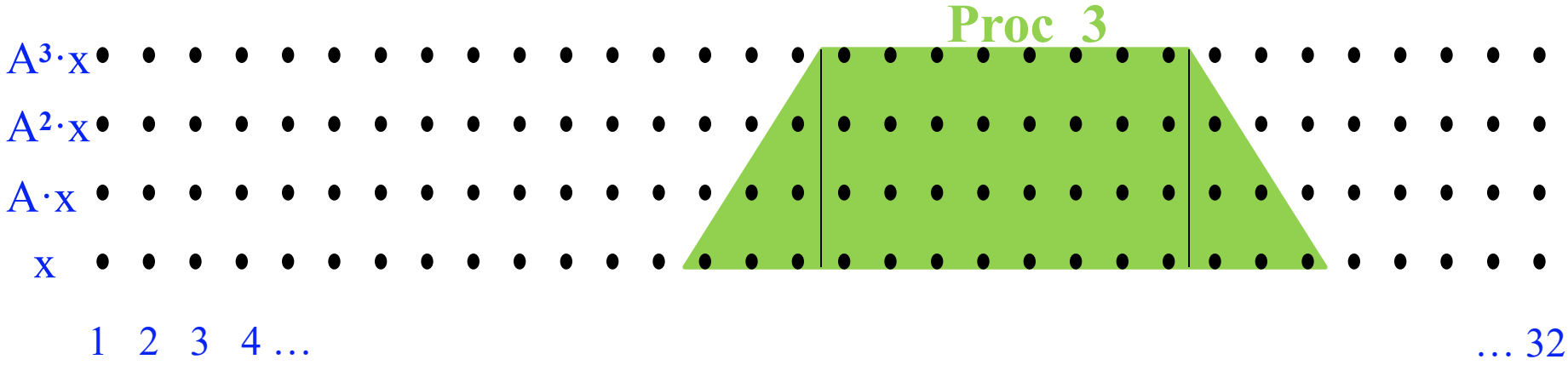
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

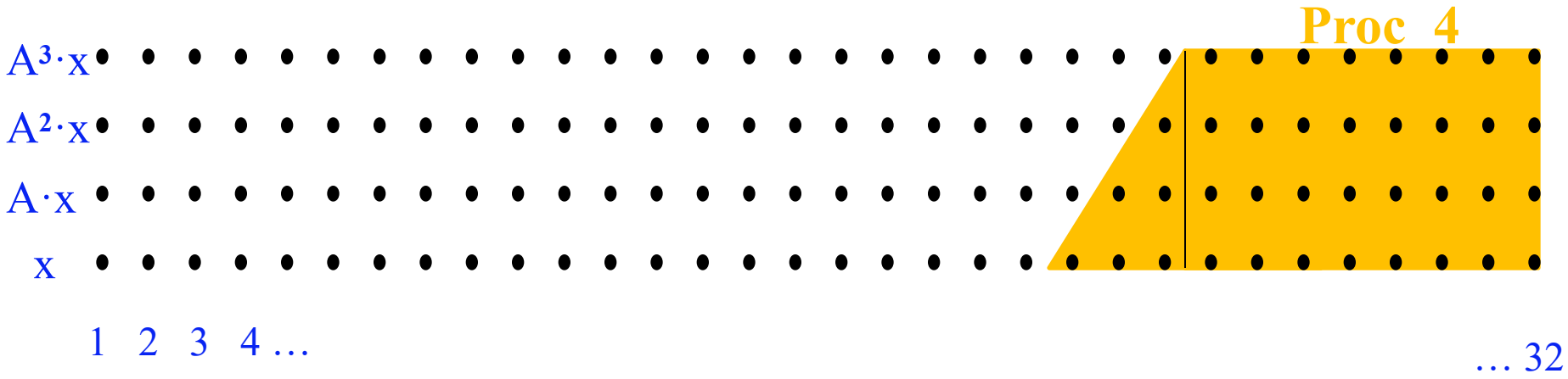
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

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

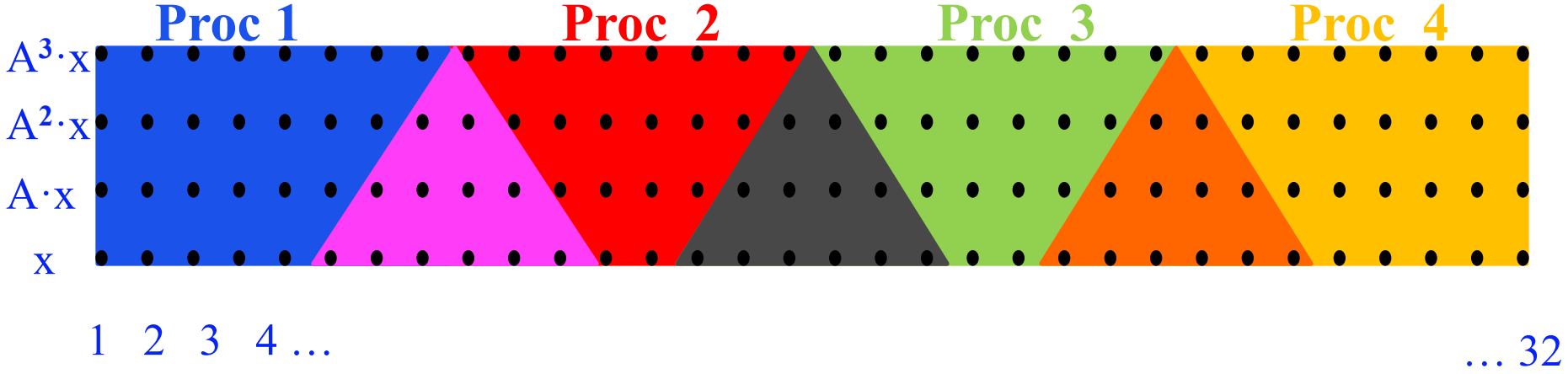
- Replace k iterations of y = A·x with [Ax, A^2x, …, A^kx]



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

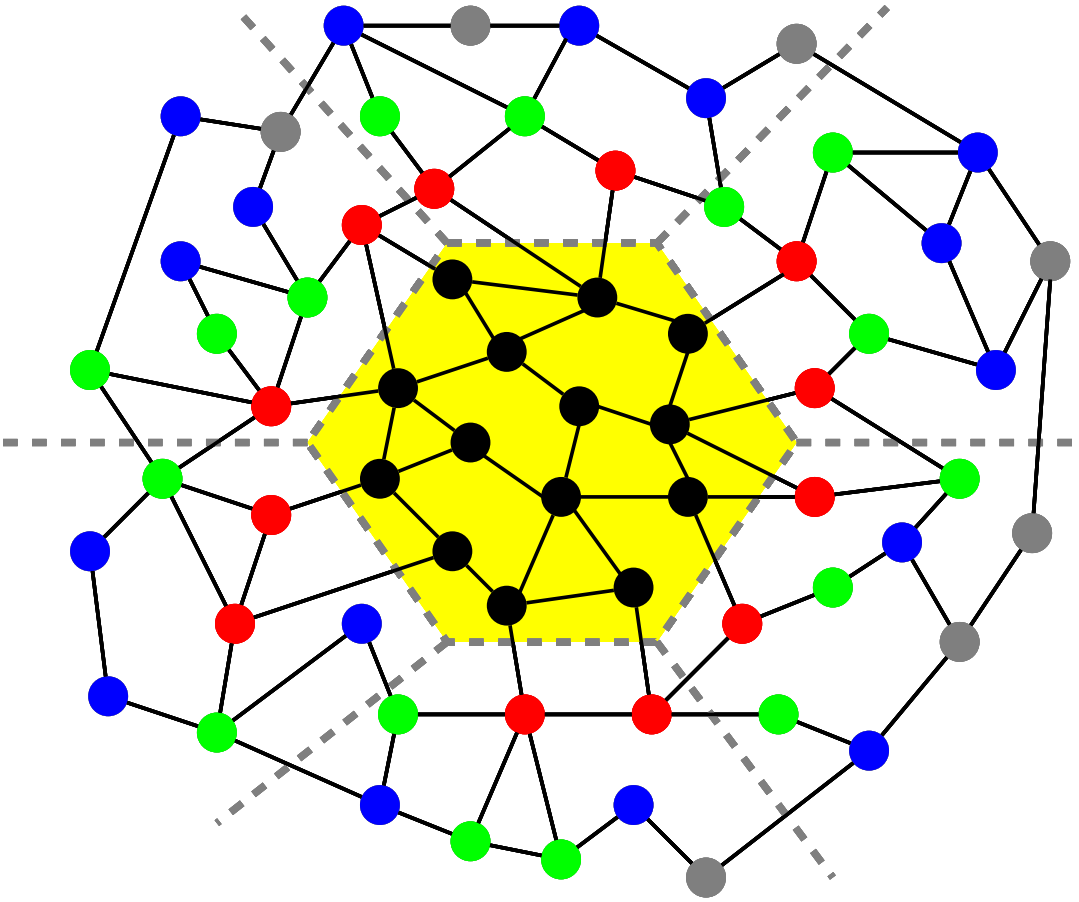
Communication Avoiding Kernels:
The Matrix Powers Kernel : [Ax, A^2x, …, A^kx]

- 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

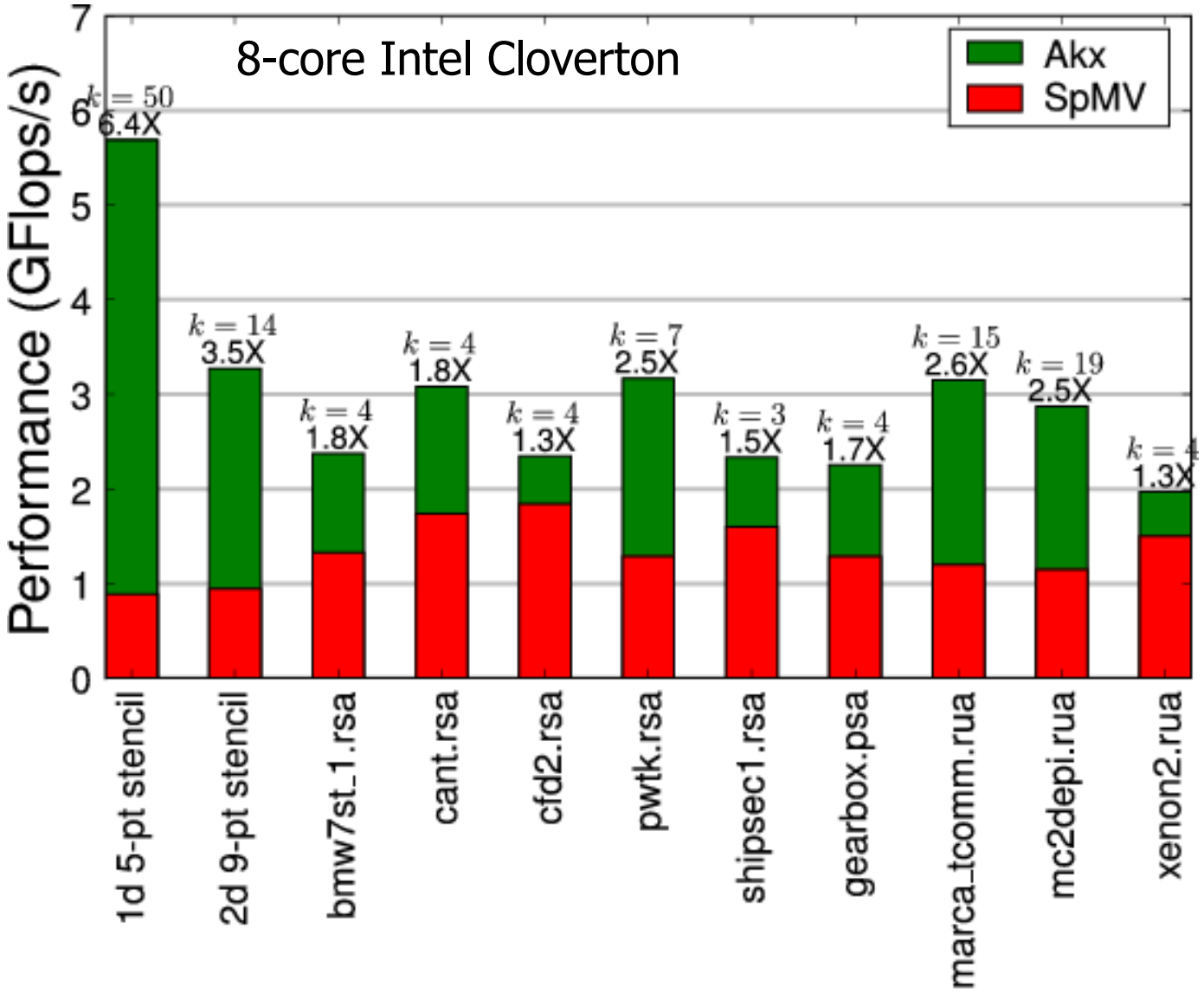


- *Need hypergraph partitioning*
- *For implicit memory management (caches) uses a TSP algorithm for layout*

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

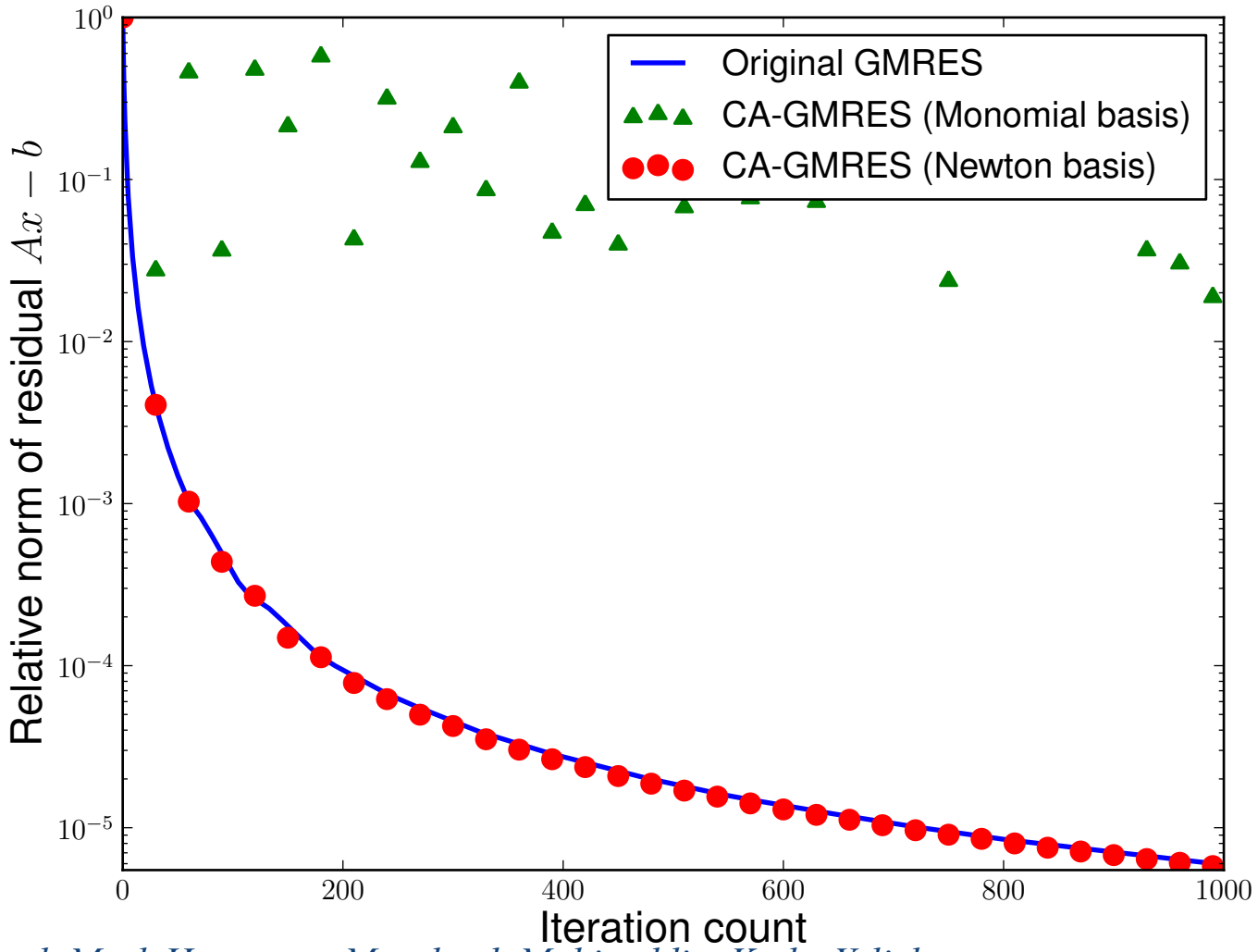
- Measured Multicore (Clovertown) speedups up to 6.4x
- Measured/Modeled sequential OOC speedup up to 3x
- Modeled parallel Petascale speedup up to 6.9x
- Modeled parallel Grid speedup up to 22x

- 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

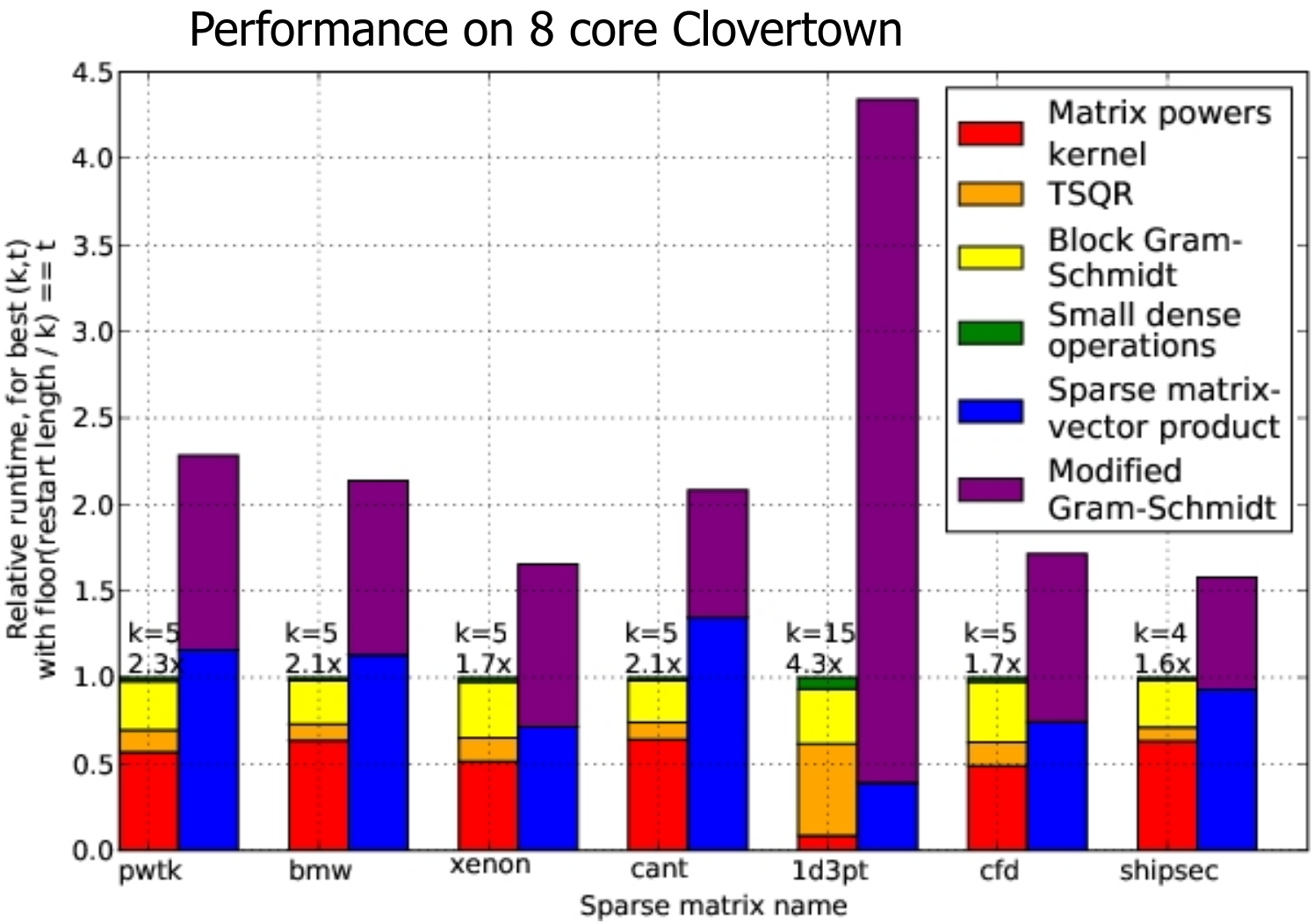
- k-steps of typical iterative solver for sparse Ax=b or Ax=λx
  - Does k SpMVs with starting vector
  - Finds "best" solution among all linear combinations of these k+1 vectors
  - Many such "Krylov Subspace Methods"
    - Conjugate Gradients, GMRES, Lanczos, Arnoldi, …
- Goal: minimize communication 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

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



Jim Demmel, Mark Hoemmen, Marghoob Mohiyuddin, Kathy Yelick

Communication-Avoiding Krylov Method (GMRES)



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

Possible Class Projects

- Experiment with SpMV on modern architectures
  - Which optimizations are most effective on KNL? Haswell?
  - Update pOSKI (team effort)
- Speed up particular matrices of interest
  - Machine learning, "bottom solver" from AMR (done)
  - Matrices from your favorite application
- Explore tuning space of [x,Ax,…,A^kx] kernel
  - Different matrix representations
- Experiment with new frameworks (SPF, Halide)
- Other sparse matrix operations:
  - Triangular solve, matrix-matrix, sparse-dense matrix,…
  - See papers on Bebop pages or talk with us