PARLab Parallel Boot Camp

Computational Patterns
and Autotuning

Jim Demmel

EECS and Mathematics

University of California, Berkeley

Outline

- Productive parallel computing depends on recognizing and exploiting useful patterns
  - Computational (7 Motifs) and Structural
- Simplest case: use “best” existing highly tuned implementation
  - Best: Fastest? Most accurate? Fewest keystrokes?
- Optimizing (some of) the 7 Motifs
  - To minimize time or energy, minimize communication (moving data)
    - Between levels of the memory hierarchy
    - Between processors over a network
  - *Autotuning* to explore large design spaces
    - Too hard (tedious) to write by hand, let machine do it
- SEJITS – how to deliver autotuning to more programmers
- For more details, see
  - CS267: www.cs.berkeley.edu/~demmel/cs267_Spr12
  - 10-hour short course: issnla2010.ba.cnr.it/Courses.htm
  - Papers at bebop.cs.berkeley.edu, parlab.eecs.berkeley.edu

"7 Motifs" of High Performance Computing

- Phil Colella (LBL) identified 7 kernels of which most simulation and data-analysis programs are composed:

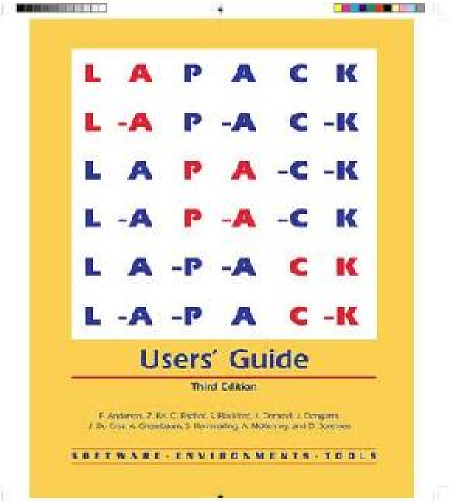
1. Dense Linear Algebra
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where A is a dense matrix
2. Sparse Linear Algebra
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where A is a sparse matrix (mostly zero)
3. Operations on Structured Grids
   - Ex: $A_{new}(i,j) = 4*A(i,j) – A(i-1,j) – A(i+1,j) – A(i,j-1) – A(i,j+1)$
4. Operations on Unstructured Grids
   - Ex: Similar, but list of neighbors varies from entry to entry
5. Spectral Methods
   - Ex: Fast Fourier Transform (FFT)
6. Particle Methods
   - Ex: Compute electrostatic forces on n particles
7. Monte Carlo
   - Ex: Many independent simulations using different inputs

"7 Motifs" of High Performance Computing

- Phil Colella (LBL) identified 7 kernels of which most simulation and data-analysis programs are composed:

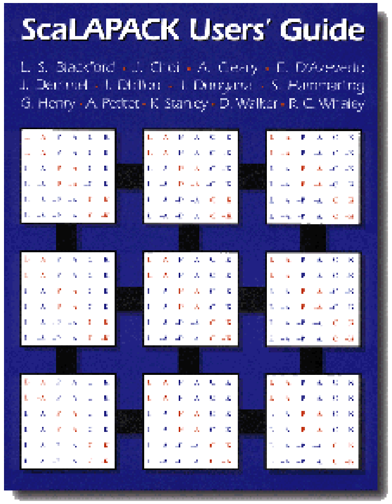
1. **Dense Linear Algebra**
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where A is a dense matrix
2. **Sparse Linear Algebra**
   - Ex: Solve $Ax=b$ or $Ax = \lambda x$ where A is a sparse matrix (mostly zero)
3. **Operations on Structured Grids**
   - Ex: $A_{new}(i,j) = 4*A(i,j) – A(i-1,j) – A(i+1,j) – A(i,j-1) – A(i,j+1)$
4. Operations on Unstructured Grids
   - Ex: Similar, but list of neighbors varies from entry to entry
5. Spectral Methods
   - Ex: Fast Fourier Transform (FFT)
6. Particle Methods
   - Ex: Compute electrostatic forces on n particles
7. Monte Carlo
   - Ex: Many independent simulations using different inputs

Organizing Linear Algebra Motifs - in books and on-line

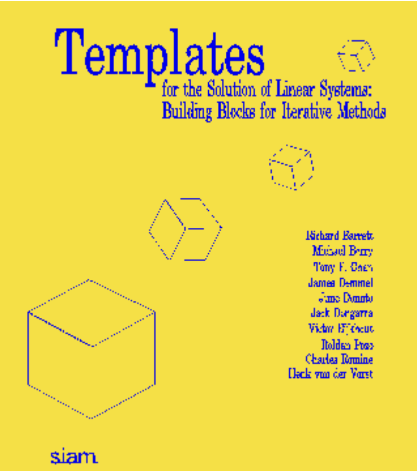


www.netlib.org/lapack

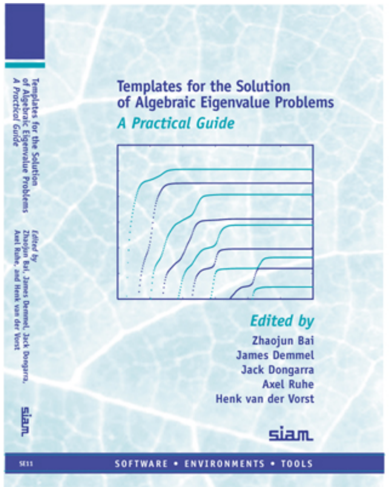
gams.nist.gov



www.netlib.org/scalapack



www.netlib.org/templates



www.cs.utk.edu/~dongarra/etemplates

Why Minimize Communication? (1/2)

- Running time of an algorithm is sum of 3 terms:
  - # flops * time_per_flop
  - # words moved / bandwidth
  - # messages * latency

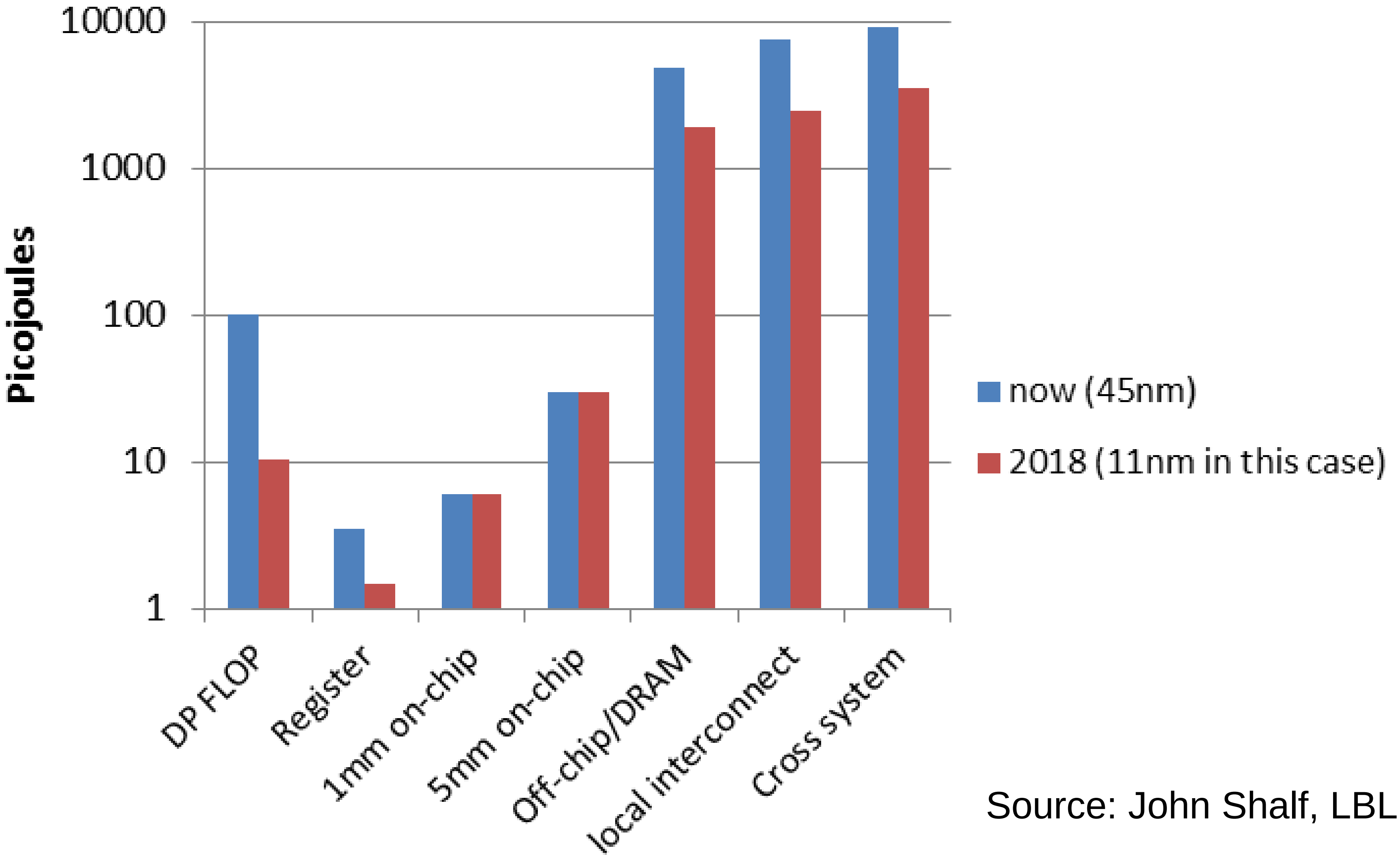
communication

- Time_per_flop << 1/ bandwidth << latency
  - Gaps growing exponentially with time [FOSC]

Annual improvements			
Time_per_flop		Bandwidth	Latency
59%	Network	26%	15%
	DRAM	23%	5%

- Minimize communication to save time

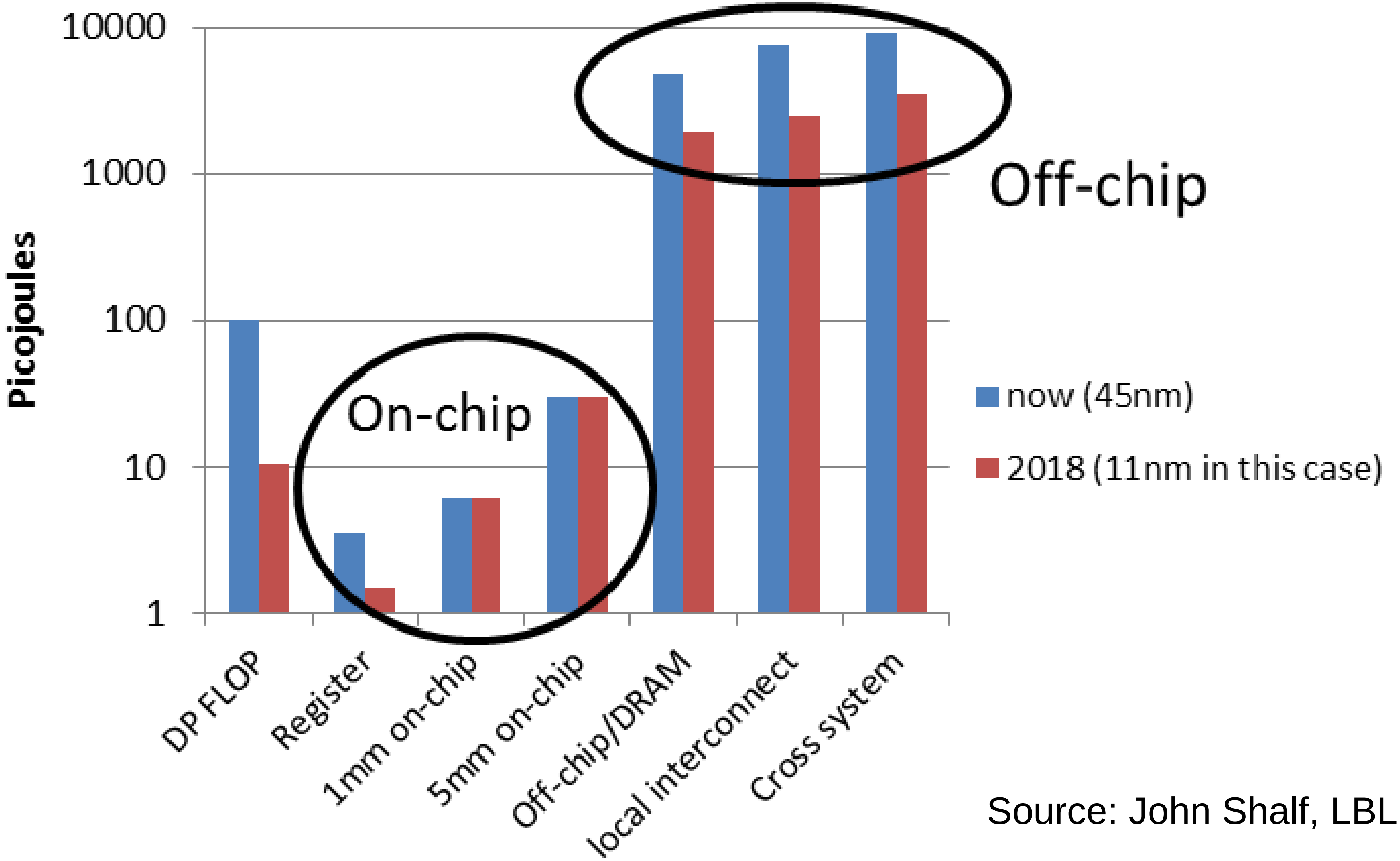
Why Minimize Communication? (2/2)



Source: John Shalf, LBL

Why Minimize Communication? (2/2)

Minimize communication to save energy



Source: John Shalf, LBL

President Obama cites Communication-Avoiding Algorithms in the FY 2012 Department of Energy Budget Request to Congress:

"New Algorithm Improves Performance and Accuracy on Extreme-Scale Computing Systems. **On modern computer architectures, communication between processors takes longer than the performance of a floating point arithmetic operation by a given processor.** ASCR researchers have developed a new method, derived from commonly used linear algebra methods, to **minimize communications between processors and the memory hierarchy, by reformulating the communication patterns specified within the algorithm**. This method has been implemented in the TRILINOS framework, a highly-regarded suite of software, which provides functionality for researchers around the world to solve large scale, complex multi-physics problems."

FY 2010 Congressional Budget, Volume 4, FY2010 Accomplishments, Advanced Scientific Computing Research (ASCR), pages 65-67.

CA-GMRES (Hoemmen, Mohiyuddin, Yelick, JD)
"Tall-Skinny" QR (Grigori, Hoemmen, Langou, JD)

Obstacle to avoiding communication: Low "computational intensity"

- Let f = #arithmetic operations in an algorithm
- Let m = #words of data needed
- Def: q = f/m = computational intensity
- If q small, say q=1, so one op/word, then algorithm can't run faster than memory speed
- But if q large, so many ops/word, then algorithm can (potentially) fetch data, do many ops while in fast memory, only limited by (faster!) speed of arithmetic
- We seek algorithms with high q
  - Still need to be clever to take advantage of high q

DENSE LINEAR ALGEBRA MOTIF

Brief history of (Dense) Linear Algebra software (1/6)

- In the beginning was the do-loop…
  - Libraries like EISPACK (for eigenvalue problems)
- Then the BLAS (1) were invented (1973-1977)
  - Standard library of 15 operations on vectors
    - Ex: $y = \alpha \cdot x + y$ ("AXPY") , dot product, etc
  - Goals
    - Common pattern to ease programming, efficiency, robustness
  - Used in libraries like LINPACK (for linear systems)
    - Source of the name "LINPACK Benchmark" (not the code!)
  - Why BLAS 1 ? 1 loop, do $O(n^1)$ ops on $O(n^1)$ data
  - Computational intensity = $q = 2n/3n = 2/3$ for AXPY
    - Very low!
  - BLAS1, and so LINPACK, limited by memory speed
  - Need something faster …

Brief history of (Dense) Linear Algebra software (2/6)

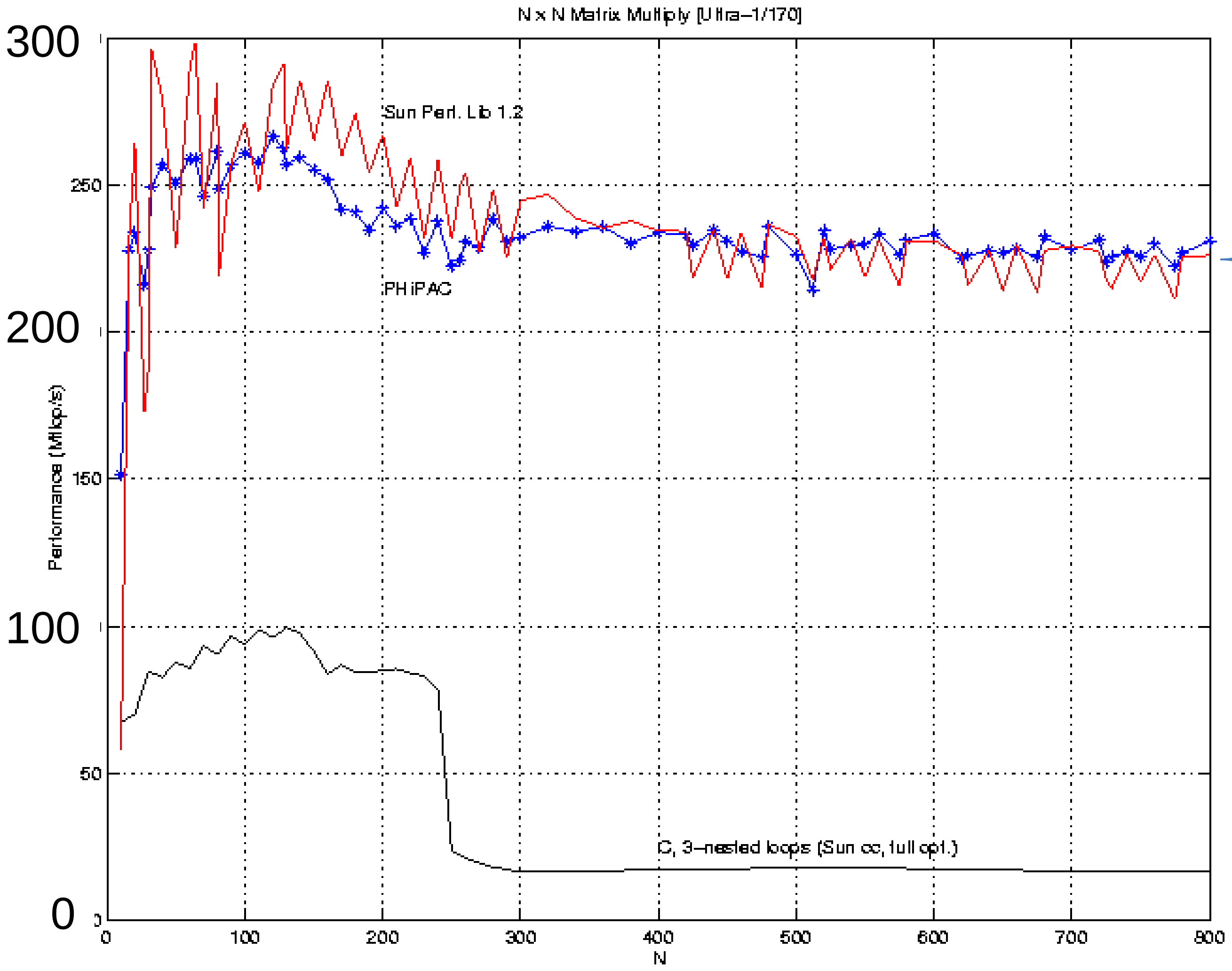
- So the BLAS-2 were invented (1984-1986)
  - Standard library of 25 operations (mostly) on matrix/vector pairs
    - Ex: $y = \alpha\cdot A\cdot x + \beta\cdot y$ ("GEMV"), $A = A + \alpha\cdot x\cdot y^T$ ("GER"), $y = T^{-1}\cdot x$ ("TRSV")
  - Why BLAS 2 ? 2 nested loops, do $O(n^2)$ ops on $O(n^2)$ data
  - But q = computational intensity still just ~ $(2n^2)/(n^2) = 2$
    - Was OK for vector machines, but not for machine with caches, since q still just a small constant

Brief history of (Dense) Linear Algebra software (3/6)

- The next step: BLAS-3 (1987-1988)
  - Standard library of 9 operations (mostly) on matrix/matrix pairs
    - Ex: $C = \alpha \cdot A \cdot B + \beta \cdot C$ ("GEMM"), $C = \alpha \cdot A \cdot A^T + \beta \cdot C$ ("SYRK") , $C = T^{-1} \cdot B$ ("TRSM")
  - Why BLAS 3 ? 3 nested loops, do $O(n^3)$ ops on $O(n^2)$ data
  - So computational intensity $q=(2n^3)/(4n^2) = n/2$ – big at last!
    - Tuning opportunities machines with caches, other mem. hierarchy levels
- How much faster can BLAS 3 go?

Matrix-multiply, optimized several ways

Peak = 330 MFlops.



Optimized Implementations: Vendor (Sun) and Autotuned (PHiPAC)

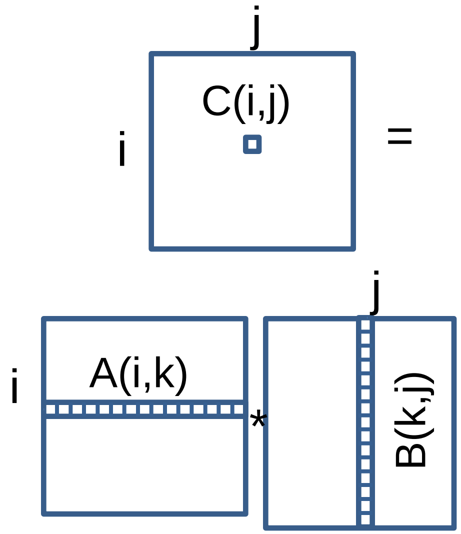
Reference Implementation; Full compiler opt.

Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

Faster Matmul C=A*B by “Blocking”

- Replace usual 3 nested loops …

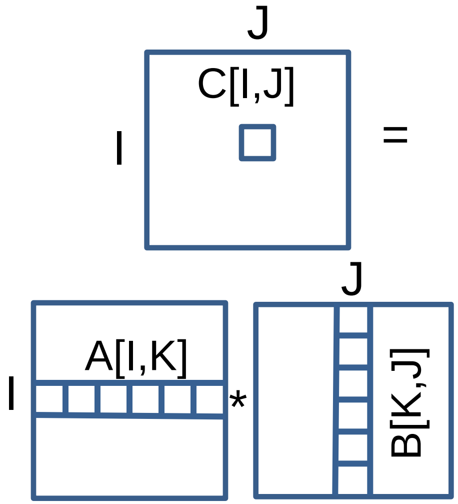
```
for i=1 to n
   for j=1 to n
      for k=1 to n
        C(i,j) =  C(i,j) + A(i,k)*B(k,j)
```



- … by “blocked” version

```
for I=1 to n/b
   for J=1 to n/b
      for K=1 to n/b
        C[I,J] =  C[I,J] + A[I,K]*B[K,J]
```

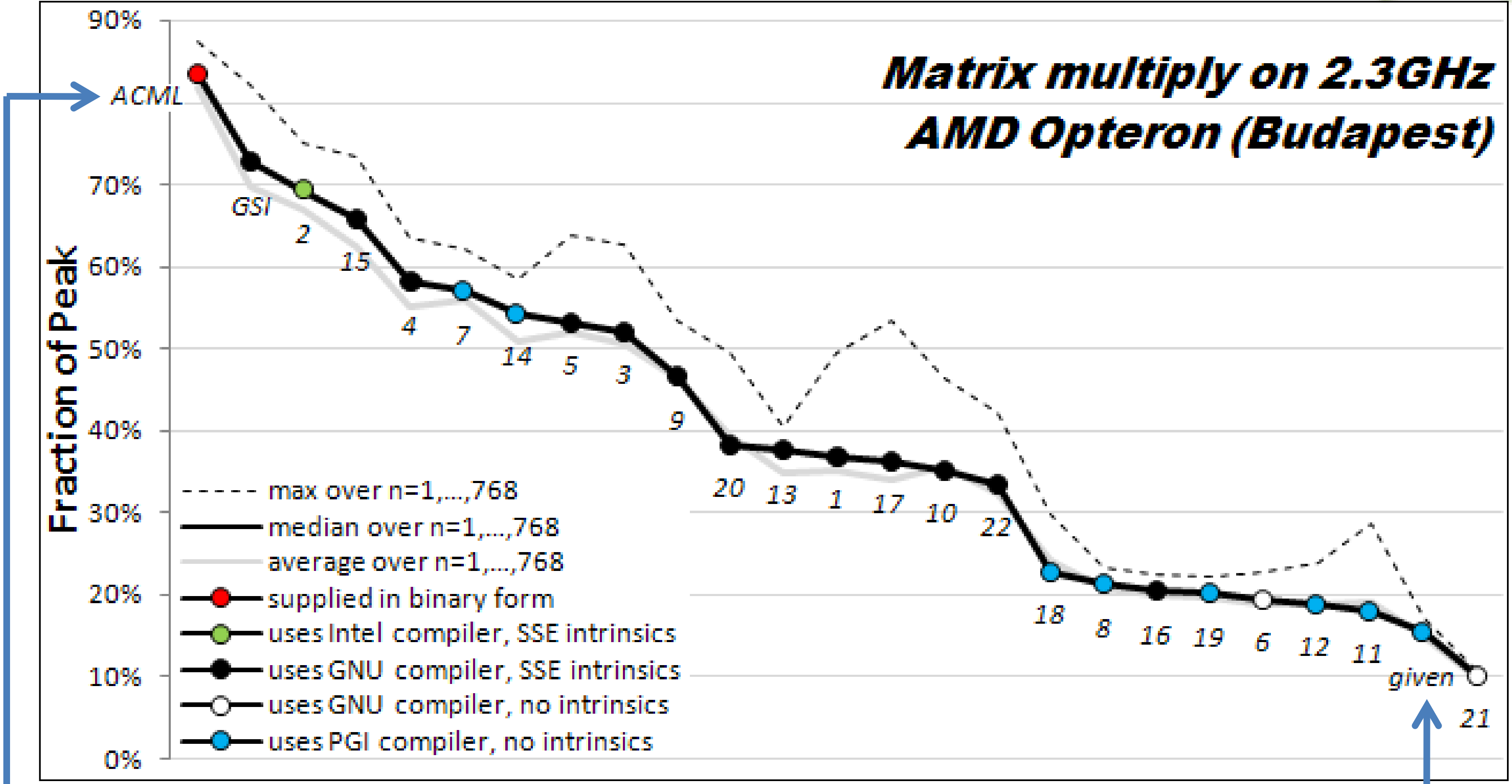
Each C[I,J], A[I,K], B[K,J] is b x b
and all 3 blocks fit in fast memory



Lower bounds on Communication for Matmul

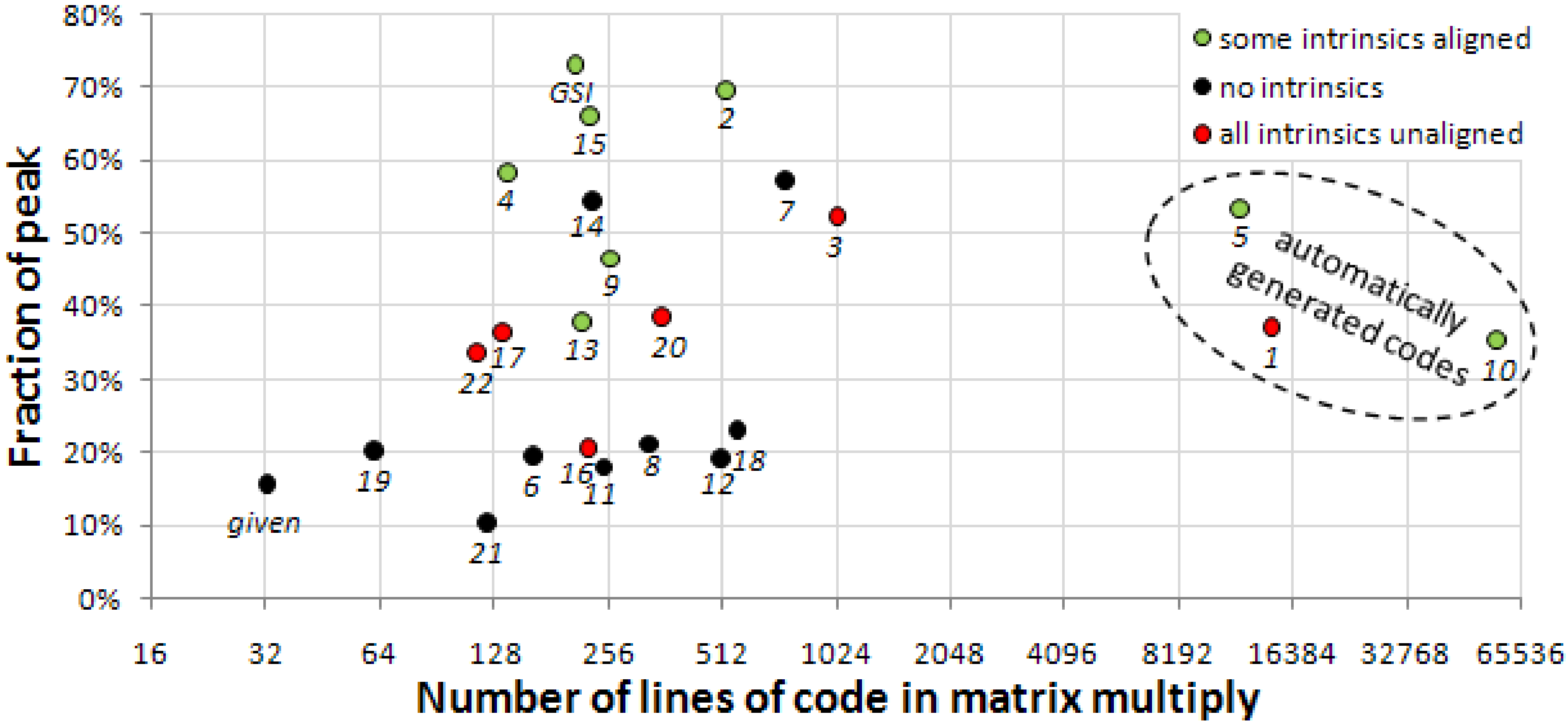
- Assume sequential n^3 algorithm for C=A*B
  - i.e. not Strassen-like
- Assume A, B and C fit in slow memory, but not in fast memory of size M
- Thm: Lower bound on #words_moved to/from slow memory, no matter the order n^3 operations are done, $= \Omega\,(n^3 / M^{1/2})$ [Hong & Kung (1981)]
- Attained by "blocked" algorithm
  - Some other algorithms attain it too
  - Widely implemented in libraries (eg Intel MKL)

How hard is hand-tuning, anyway?

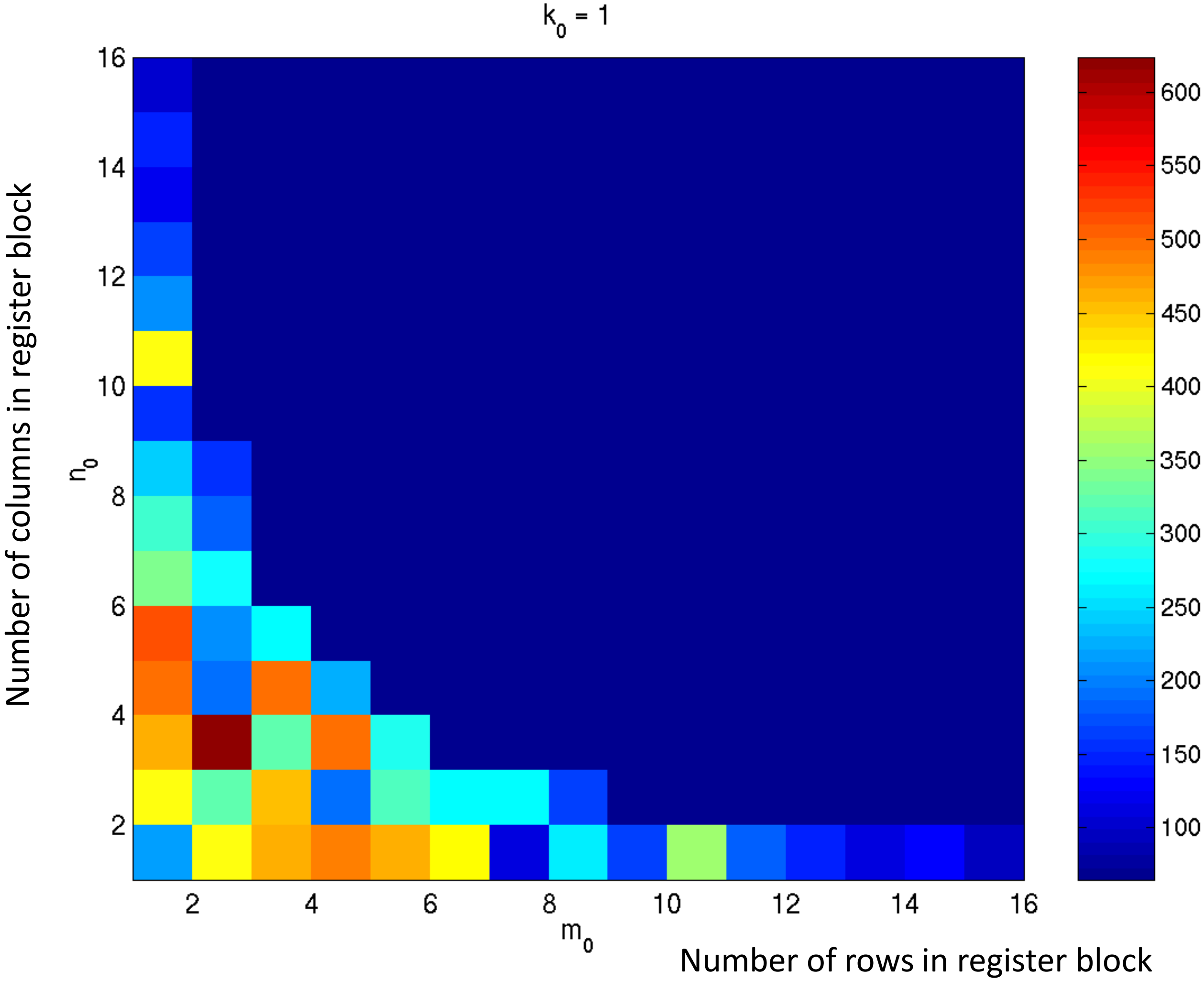


- Results of 22 student teams trying to tune matrix-multiply, in CS267 Spr09
- Students given “blocked” code to start with
  - Still hard to get close to vendor tuned performance (ACML)
- For more discussion, see www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/
- Naïve matmul: just 2% of peak

How hard is hand-tuning, anyway?



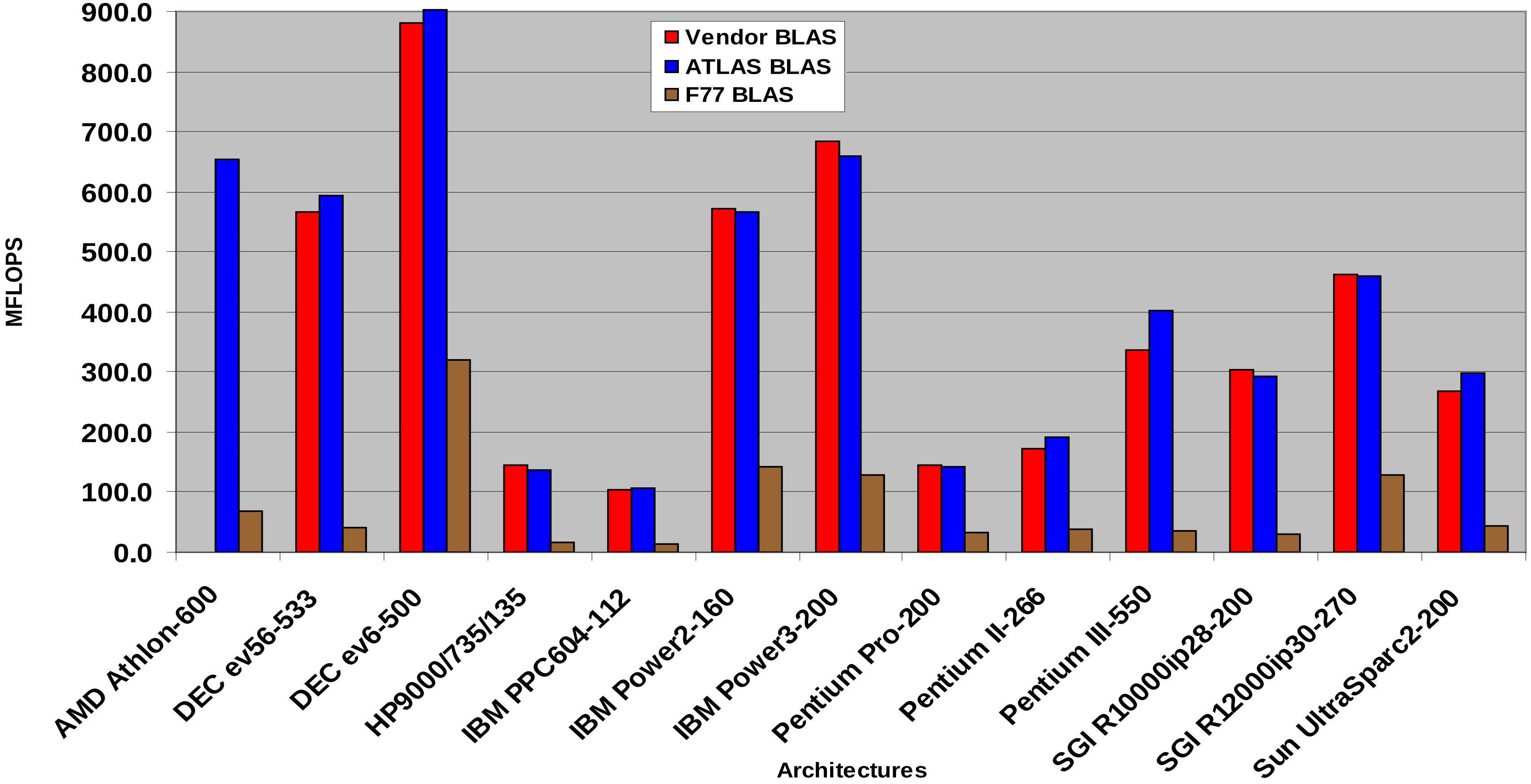
What part of the Matmul Search Space Looks Like



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.
(Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

Autotuning DGEMM with ATLAS (n = 500)

Source: Jack Dongarra



- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.
- ATLAS written by C. Whaley, inspired by PHiPAC, by Asanovic, Bilmes,Chin,D.

Brief history of (Dense) Linear Algebra software (4/6)

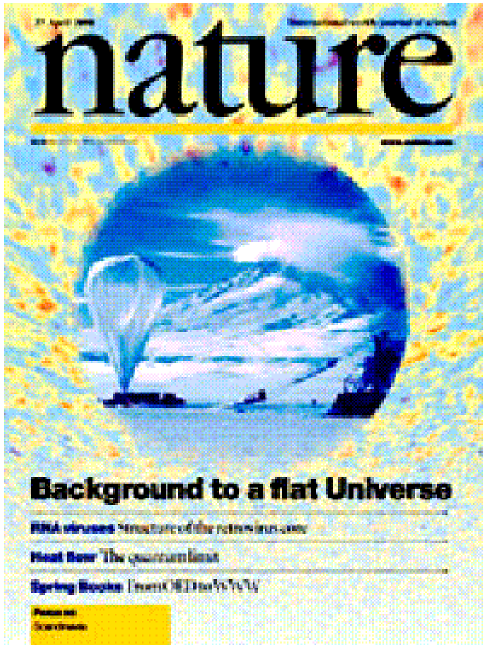
- LAPACK – "Linear Algebra PACKage" - uses BLAS-3 (1989 – now)
  - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of each row to other rows – BLAS-1
    - Need to reorganize GE (and everything else) to use BLAS-3 instead
  - Contents of current LAPACK (summary)
    - Algorithms we can turn into (nearly) 100% BLAS 3 for large n
      - Linear Systems: solve Ax=b for x
      - Least Squares: choose x to minimize $\sqrt{\Sigma_i r_i^2}$ where r=Ax-b
    - Algorithms that are only up to ~50% BLAS 3, rest BLAS 1 & 2
      - "Eigenproblems": Find λ and x where $Ax = \lambda x$
      - Singular Value Decomposition (SVD): $A^TAx=\sigma^2x$
    - Error bounds for everything
    - Lots of variants depending on A's structure (banded, $A=A^T$, etc)
  - Widely used (list later)
  - All at www.netlib.org/lapack

Brief history of (Dense) Linear Algebra software (5/6)

- Is LAPACK parallel?
  - Only if the BLAS are parallel (possible in shared memory)
- ScaLAPACK – "Scalable LAPACK" (1995 – now)
  - For distributed memory – uses MPI
  - More complex data structures, algorithms than LAPACK
    - Only subset of LAPACK's functionality available
    - Work in progress (contributions welcome!)
  - All at www.netlib.org/scalapack

Success Stories for Sca/LAPACK

- Widely used
  - Adopted by Mathworks, Cray, Fujitsu, HP, IBM, IMSL, Intel, NAG, NEC, SGI, …
  - >157M web hits(in 2012, 56M in 2006) @ Netlib (incl. CLAPACK, LAPACK95)
- New science discovered through the solution of dense matrix systems
  - Nature article on the flat universe used ScaLAPACK
  - 1998 Gordon Bell Prize
  - www.nersc.gov/news/reports/newNERSCresults050703.pdf
- Currently funded to improve, update, maintain Sca/LAPACK



Cosmic Microwave Background Analysis, BOOMERanG collaboration, MADCAP code (Apr. 27, 2000).

ScaLAPACK

Lower bound for all "n^3-like" linear algebra

- Let M = "fast" memory size (per processor)

#words_moved (per processor) = Ω(#flops (per processor) / $M^{1/2}$)

- Parallel case: assume either load or memory balanced
  - Holds for
    - Matmul

Lower bound for all "n^3-like" linear algebra

- Let M = "fast" memory size (per processor)

#words_moved (per processor) = Ω(#flops (per processor) / $M^{1/2}$)

#messages_sent $\geq$ #words_moved / largest_message_size

- Parallel case: assume either load or memory balanced

  - Holds for
    - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, …
    - Some whole programs (sequences of these operations, no matter how individual ops are interleaved, eg A^k)
    - Dense and sparse matrices (where #flops << n^3)
    - Sequential and parallel algorithms
    - Some graph-theoretic algorithms (eg Floyd-Warshall)

Lower bound for all "n^3-like" linear algebra

- Let M = "fast" memory size (per processor)

#words_moved (per processor) = Ω(#flops (per processor) / $M^{1/2}$)

#messages_sent (per processor) = Ω(#flops (per processor) / $M^{3/2}$)

- Parallel case: assume either load or memory balanced

SIAM SIAG/LA Best Paper 2012

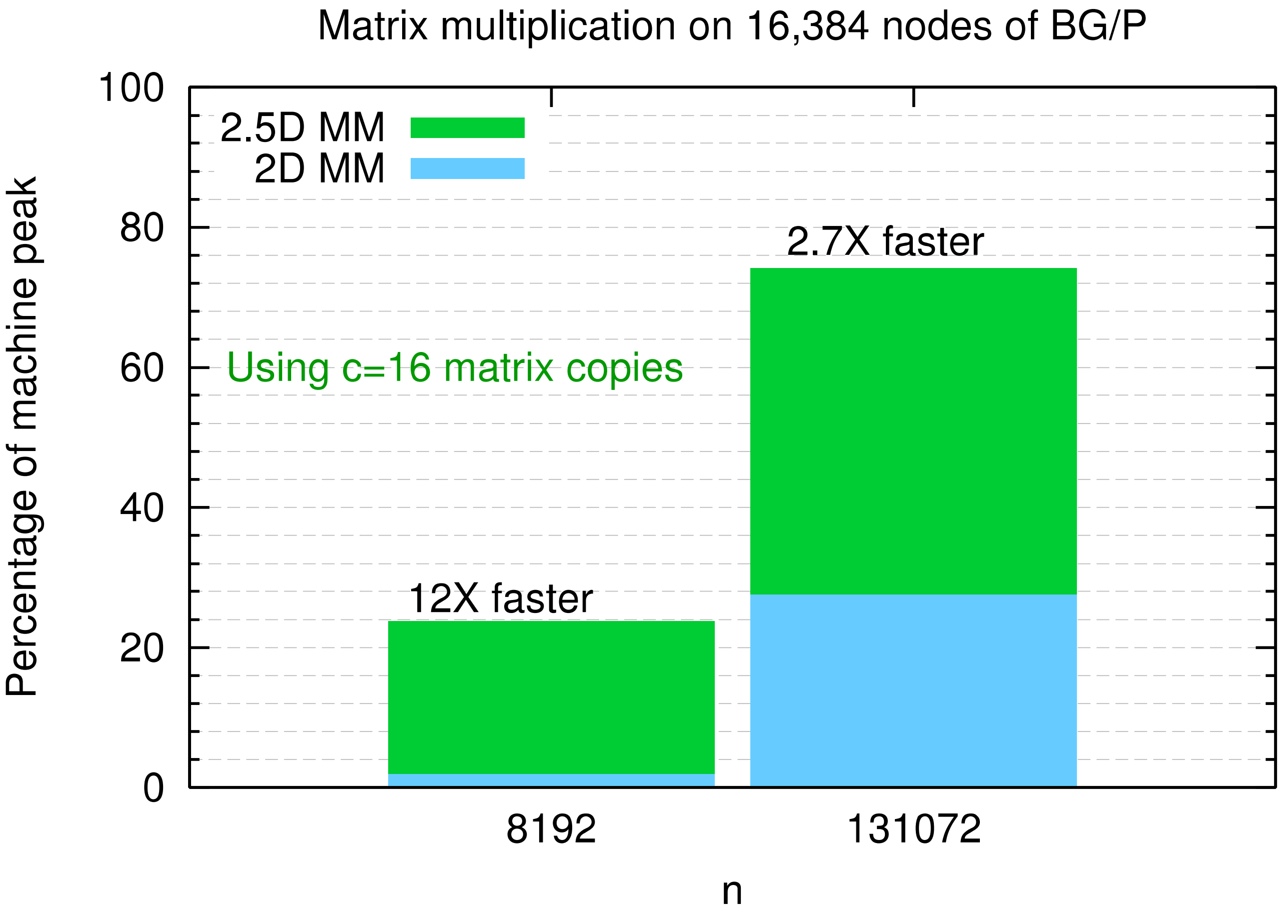
  - Holds for
    - Matmul, BLAS, LU, QR, eig, SVD, tensor contractions, …
    - Some whole programs (sequences of these operations, no matter how individual ops are interleaved, eg A^k)
    - Dense and sparse matrices (where #flops << n^3)
    - Sequential and parallel algorithms
    - Some graph-theoretic algorithms (eg Floyd-Warshall)

Can we attain these lower bounds?

- Do conventional dense algorithms as implemented in LAPACK and ScaLAPACK attain these bounds?
  - Mostly not
- If not, are there other algorithms that do?
  - Yes, for much of dense linear algebra
  - New algorithms, with new numerical properties, new ways to encode answers, new data structures
  - Not just loop transformations (need those too!)
- Only a few sparse algorithms so far
- Lots of work in progress

Example: “2.5D” Matrix multiply

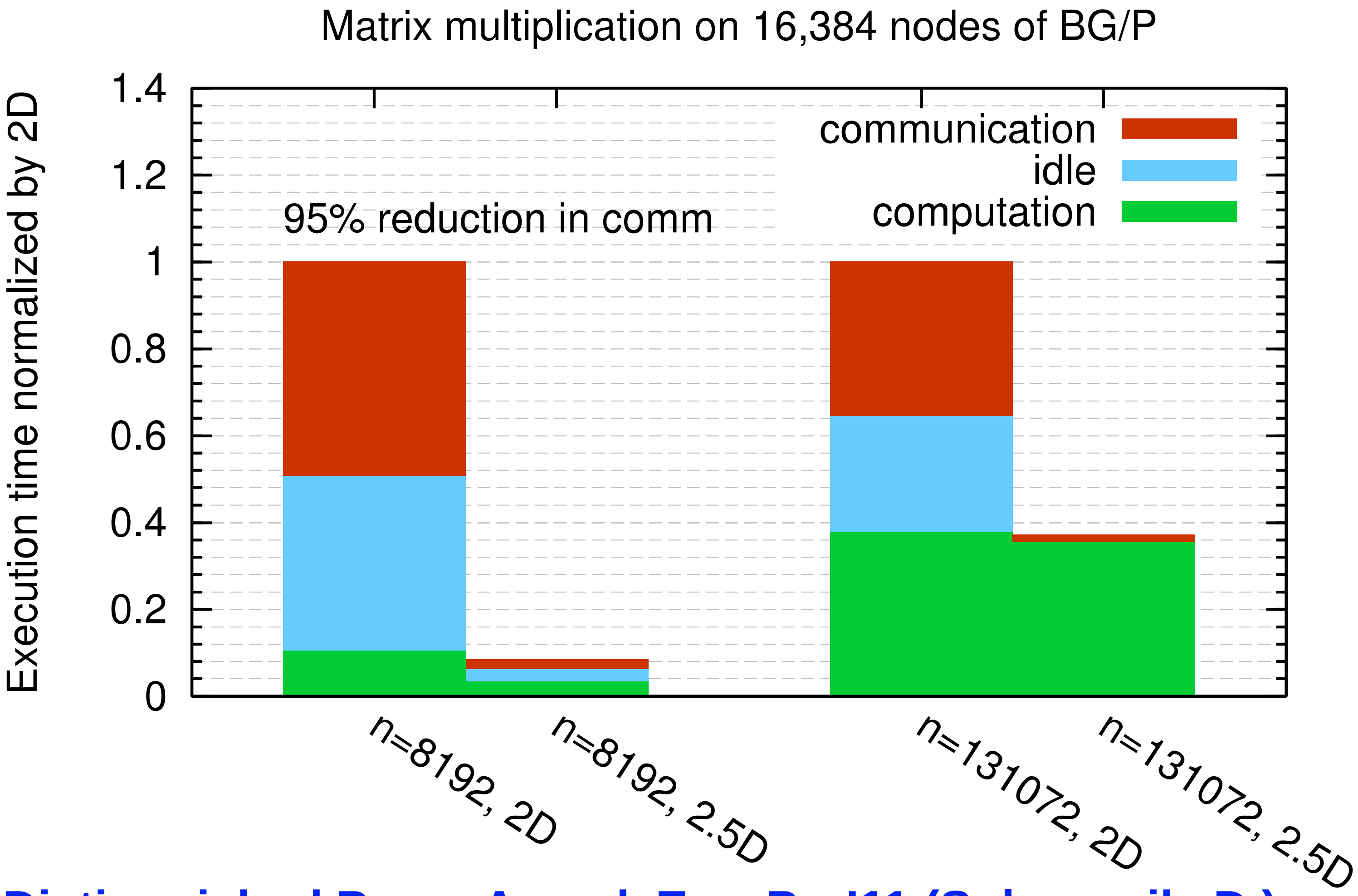
Lower bound decreases as M increases, even beyond minimum needed ($3n^2/p$)



Matrix multiplication on 16,384 nodes of BG/P

2.5D Matrix Multiply Timing Breakdown

c = 16 copies



Distinguished Paper Award, EuroPar’11 (Solomonik, D.)
(SC’11 paper by Solomonik, Bhatele, D.)

TSQR: QR of a Tall, Skinny matrix

$$W = \left(\begin{array}{c} W_0 \\ \hline W_1 \\ \hline W_2 \\ \hline W_3 \end{array}\right)$$

$$\left(\begin{array}{c} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{array}\right) = \left(\begin{array}{c} Q_{01}\ R_{01} \\ \hline Q_{11}\ R_{11} \end{array}\right)$$

$$\left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right) = \left(\begin{array}{c} Q_{02}\ R_{02} \end{array}\right)$$

TSQR: QR of a Tall, Skinny matrix

$$W = \left(\begin{array}{c} W_0 \\ \hline W_1 \\ \hline W_2 \\ \hline W_3 \end{array}\right) = \left(\begin{array}{c} Q_{00}\,R_{00} \\ \hline Q_{10}\,R_{10} \\ \hline Q_{20}\,R_{20} \\ \hline Q_{30}\,R_{30} \end{array}\right) = \left(\begin{array}{cccc} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \cdot \left(\begin{array}{c} R_{00} \\ \hline R_{10} \\ \hline R_{20} \\ \hline R_{30} \end{array}\right)$$

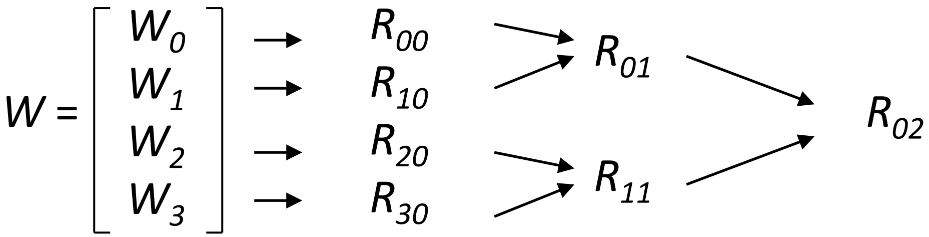
$$\left(\begin{array}{c} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{array}\right) = \left(\begin{array}{c} Q_{01}\,R_{01} \\ \hline Q_{11}\,R_{11} \end{array}\right) = \left(\begin{array}{cc} Q_{01} & \\ \hline & Q_{11} \end{array}\right) \cdot \left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right)$$

$$\left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right) = \left(Q_{02}\,R_{02} \right)$$

Output = $\{\, Q_{00}, Q_{10}, Q_{20}, Q_{30}, Q_{01}, Q_{11}, Q_{02}, R_{02} \,\}$

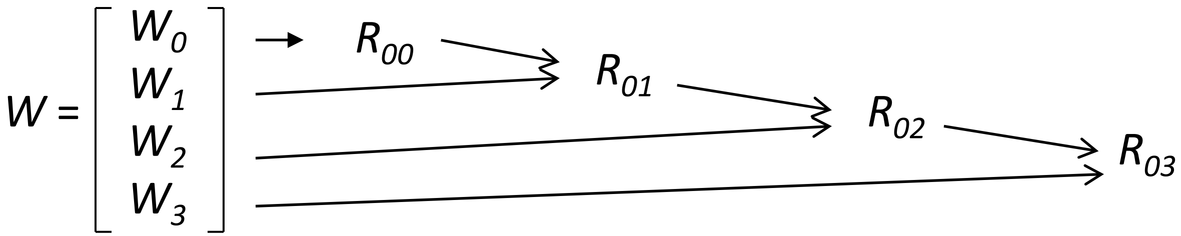
TSQR: An Architecture-Dependent Algorithm

Parallel:



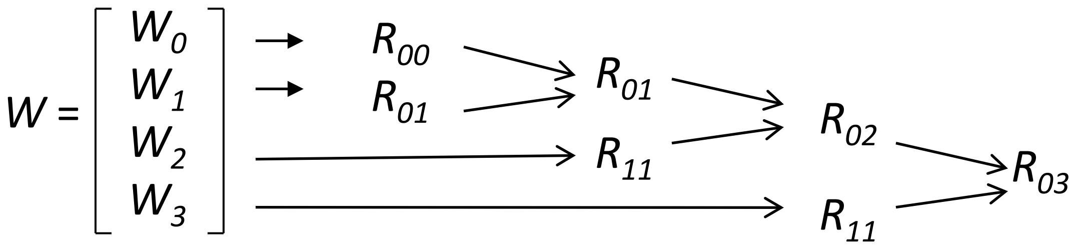
$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}$ R_{00} R_{10} R_{20} R_{30} R_{01} R_{11} R_{02}

Sequential:



$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}$ R_{00} R_{01} R_{02} R_{03}

Dual Core:



$W = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix}$ R_{00} R_{01} R_{01} R_{11} R_{02} R_{11} R_{03}

Multicore / Multisocket / Multirack / Multisite / Out-of-core: ?

Can choose reduction tree dynamically

TSQR Performance Results

- Parallel
  - Intel Clovertown
    - – Up to **8x** speedup (8 core, dual socket, 10M x 10)
  - Pentium III cluster, Dolphin Interconnect, MPICH
    - Up to **6.7x** speedup (16 procs, 100K x 200)
  - BlueGene/L
    - Up to **4x** speedup (32 procs, 1M x 50)
  - Tesla C 2050 / Fermi
    - Up to **13x** (110,592 x 100)
  - Grid – **4x** on 4 cities vs 1 city (Dongarra, Langou et al)
  - Cloud (Gleich, Benson)
- Sequential
  - "Infinite speedup" for out-of-Core on PowerPC laptop
    - As little as 2x slowdown vs (predicted) infinite DRAM
    - LAPACK with virtual memory never finished
- Joint work with Grigori, Hoemmen, Langou, Anderson, Ballard, Keutzer, others

Brief history/*future* of (Dense) Linear Algebra software (6/6)

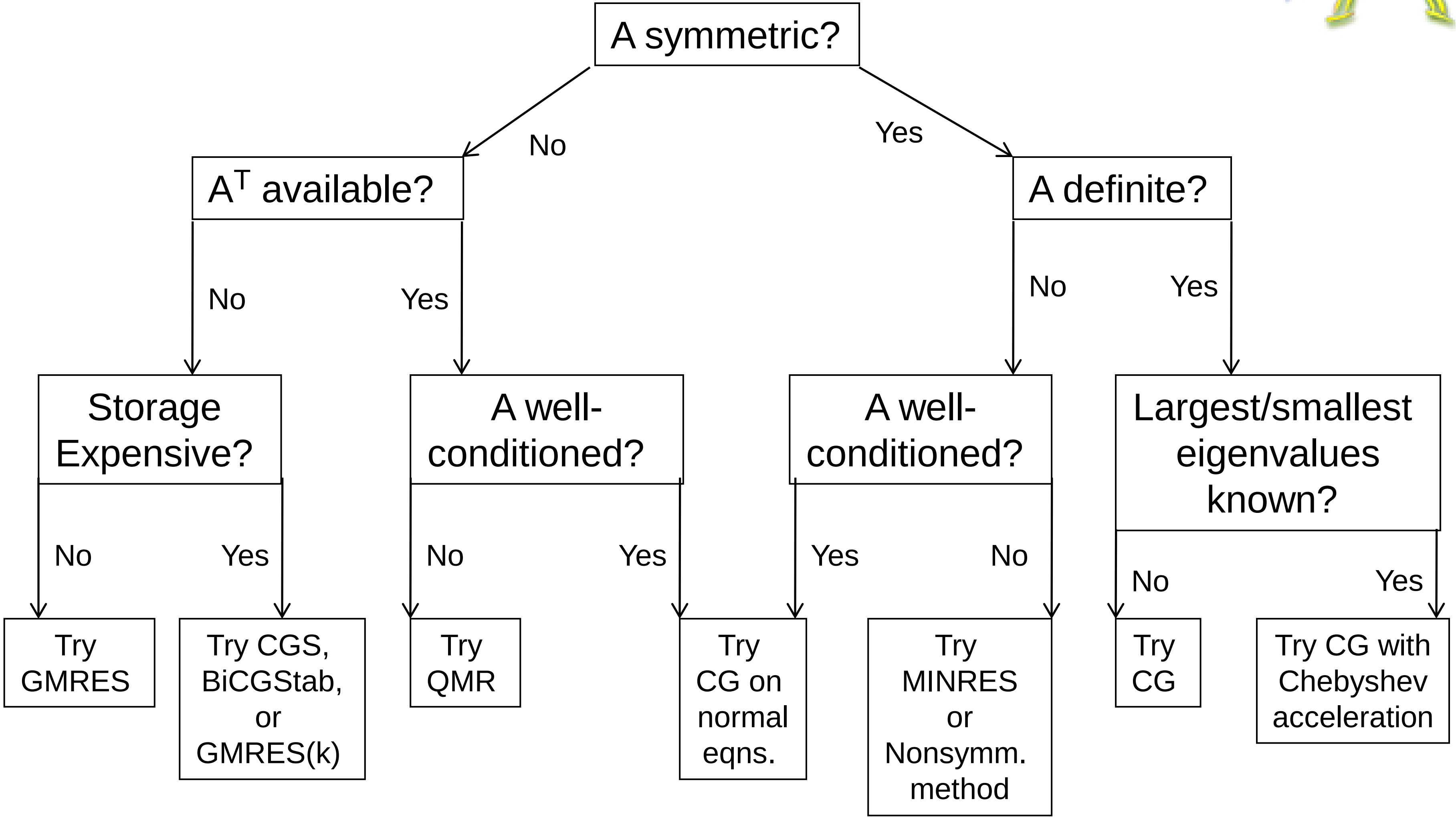
- Communication-Avoiding for everything (open problems…)
  - Extensions to Strassen-like algorithms
- Extensions for multicore
  - PLASMA – Parallel Linear Algebra for Scalable Multicore Architectures
    - Dynamically schedule tasks into which algorithm is decomposed, to minimize synchronization, keep all processors busy
    - Release 2.4.5 at icl.cs.utk.edu/plasma/
- Extensions for heterogeneous architectures, eg CPU + GPU
  - “Benchmarking GPUs to tune Dense Linear Algebra”
    - Best Student Paper Prize at SC08 (Vasily Volkov)
    - Paper, slides and code at www.cs.berkeley.edu/~volkov
  - Lower, matching upper bounds (tech report at bebop.cs.berkeley.edu)
  - MAGMA – Matrix Algebra on GPU and Multicore Architectures
    - Release 1.2.1 at icl.cs.utk.edu/magma/
- How much code generation can we automate?
  - MAGMA , and FLAME (www.cs.utexas.edu/users/flame/)

SPARSE LINEAR ALGEBRA MOTIF

Sparse Matrix Computations

- Similar problems to dense matrices
  - Ax=b, Least squares, Ax = λx, SVD, …
- But different algorithms!
  - Exploit structure: only store, work on nonzeros
  - Direct methods
    - LU, Cholesky for Ax=b, QR for Least squares
    - See crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf for a survey of available serial and parallel sparse solvers
    - See crd.lbl.gov/~xiaoye/SuperLU/index.html for LU codes
  - Iterative methods – for Ax=b, least squares, eig, SVD
    - Use simplest operation: Sparse-Matrix-Vector-Multiply (SpMV)
    - Krylov Subspace Methods: find "best" solution in space spanned by vectors generated by SpMVs

Choosing a Krylov Subspace Method for Ax=b



A symmetric?

No

Yes

A^T available?

No

Yes

A definite?

No

Yes

Storage Expensive?

No

Yes

A well-conditioned?

No

Yes

A well-conditioned?

Yes

No

Largest/smallest eigenvalues known?

No

Yes

Try GMRES

Try CGS, BiCGStab, or GMRES(k)

Try QMR

Try CG on normal eqns.

Try MINRES or Nonsymm. method

Try CG

Try CG with Chebyshev acceleration

- All depend on SpMV
- See www.netlib.org/templates for Ax=b
- See www.cs.ucdavis.edu/~bai/ET/contents.html for Ax=λx and SVD

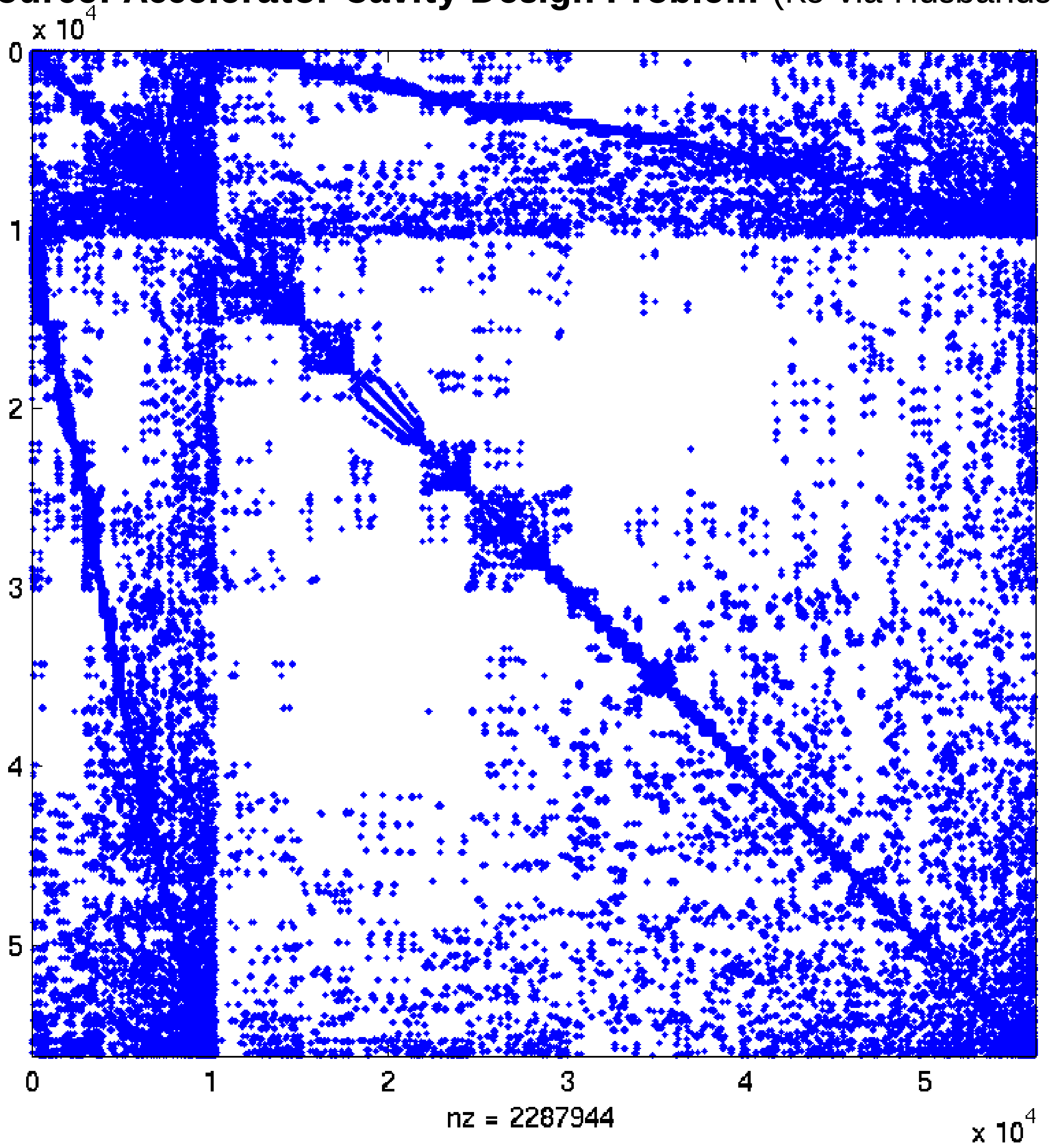
Sparse Outline

- Approaches to Automatic Performance Tuning
- Results for sparse matrix kernels
  - Sparse Matrix Vector Multiplication (SpMV)
  - Sequential and Multicore results
- OSKI = Optimized Sparse Kernel Interface
  - pOSKI = parallel OSKI
- Tuning Entire Sparse Solvers
  - Avoiding Communication
- What is a sparse matrix?

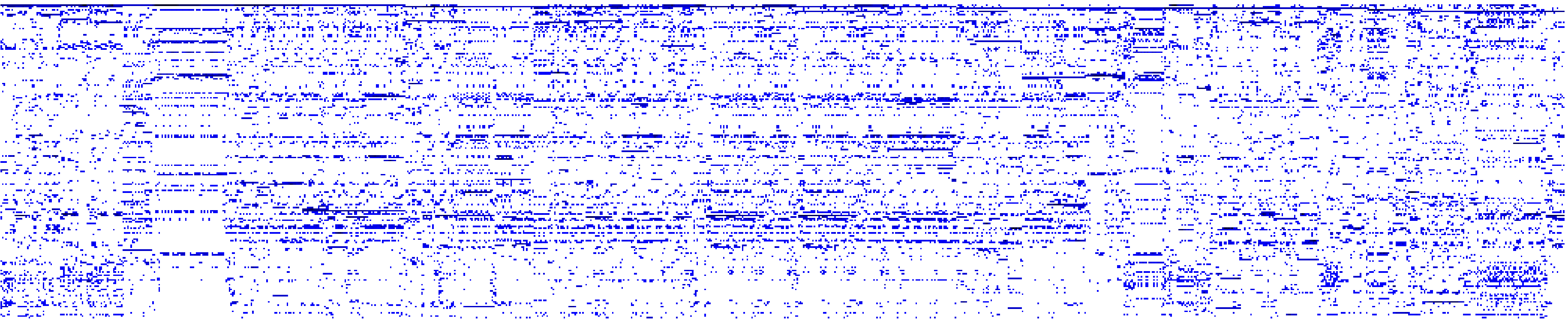
Approaches to Automatic Performance Tuning

- **Goal: Let machine do hard work of writing fast code**
- **Why is tuning dense BLAS "easy"?**
  - Can do the tuning off-line: once per architecture, algorithm
  - Can take as much time as necessary (hours, a week…)
  - At run-time, algorithm choice may depend only on few parameters (matrix dimensions)
- **Can't always do tuning off-line**
  - Algorithm and implementation may strongly depend on data only known at run-time
  - Ex: Sparse matrix nonzero pattern determines both best data structure and implementation of Sparse-matrix-vector-multiplication (SpMV)
  - Part of search for best algorithm must be done (very quickly!) at run-time
- **Tuning FFTs and signal processing**
  - Seems off-line, but maybe not, because of code size
  - www.spiral.net, www.fftw.org

Source: Accelerator Cavity Design Problem (Ko via Husbands)

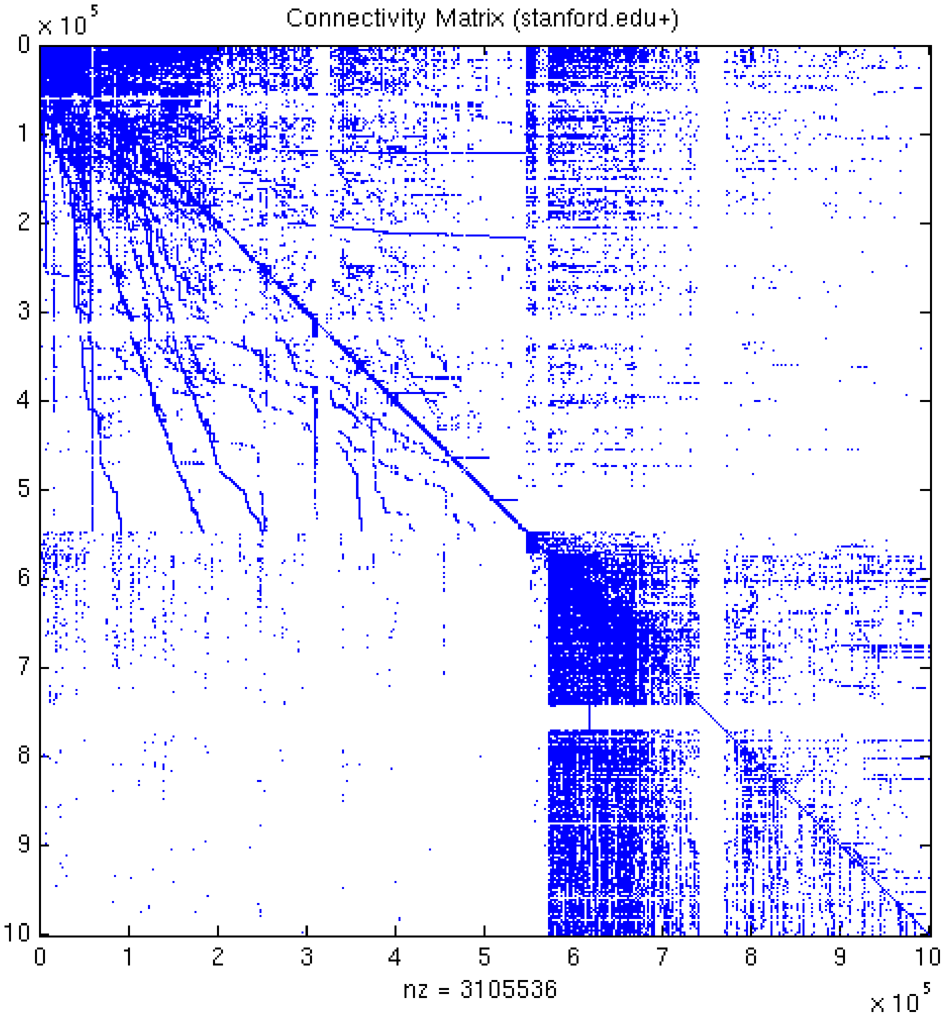


Linear Programming Matrix

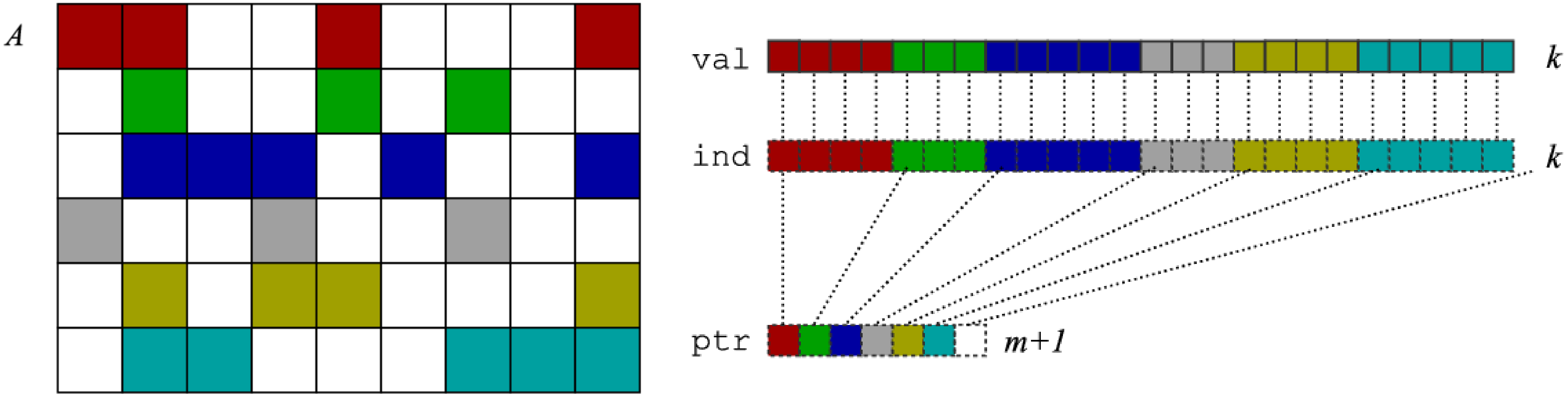


…

A Sparse Matrix You Use Every Day



SpMV with Compressed Sparse Row (CSR) Storage

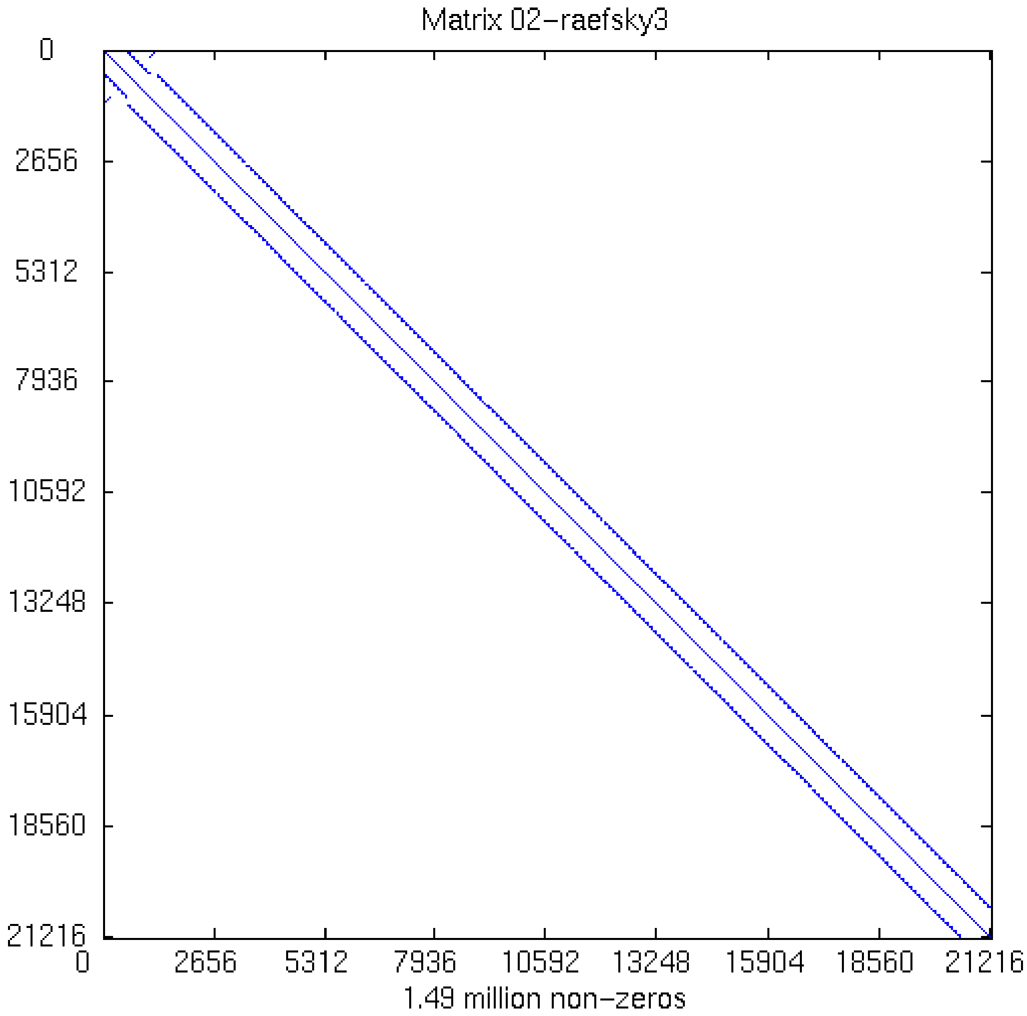


Matrix-vector multiply kernel: y(i) ← y(i) + A(i,j)*x(j)

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

Only 2 flops per
2 mem_refs:
Limited by getting
data from memory

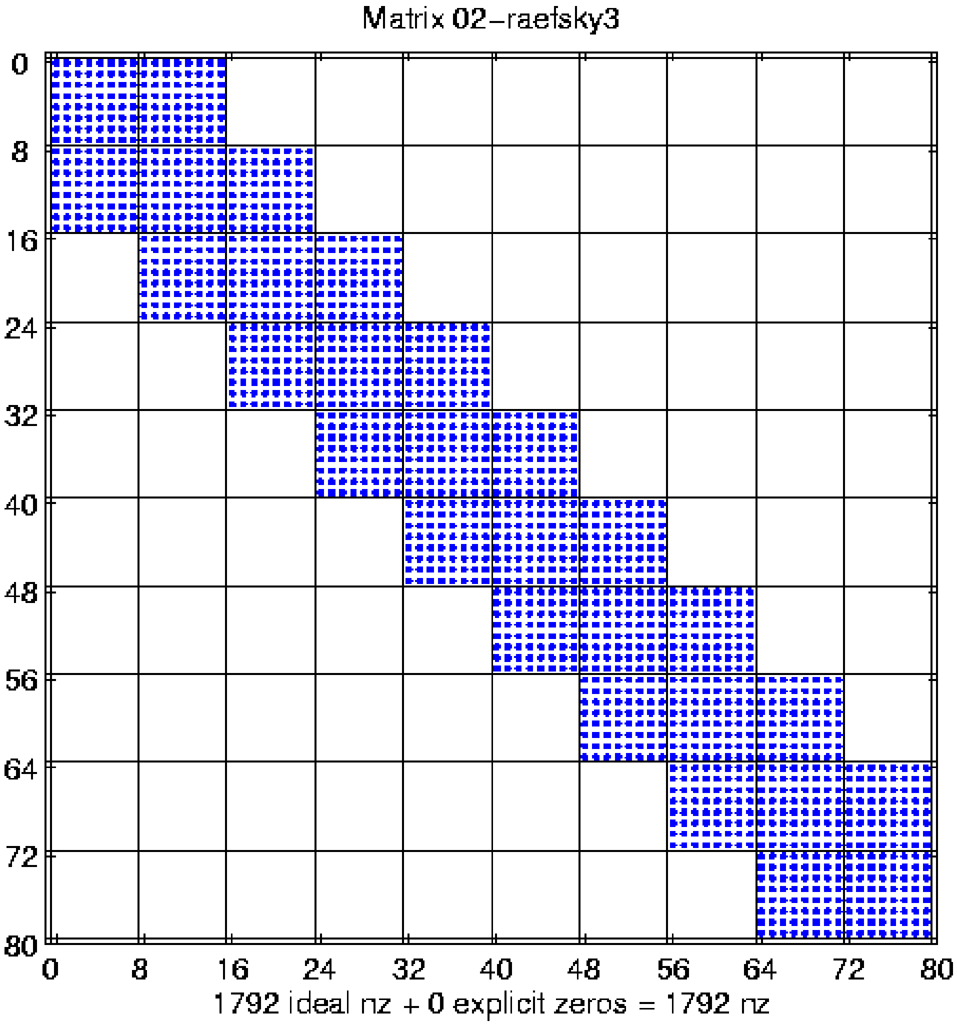
Example: The Difficulty of Tuning



- n = 21200
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem

Example: The Difficulty of Tuning

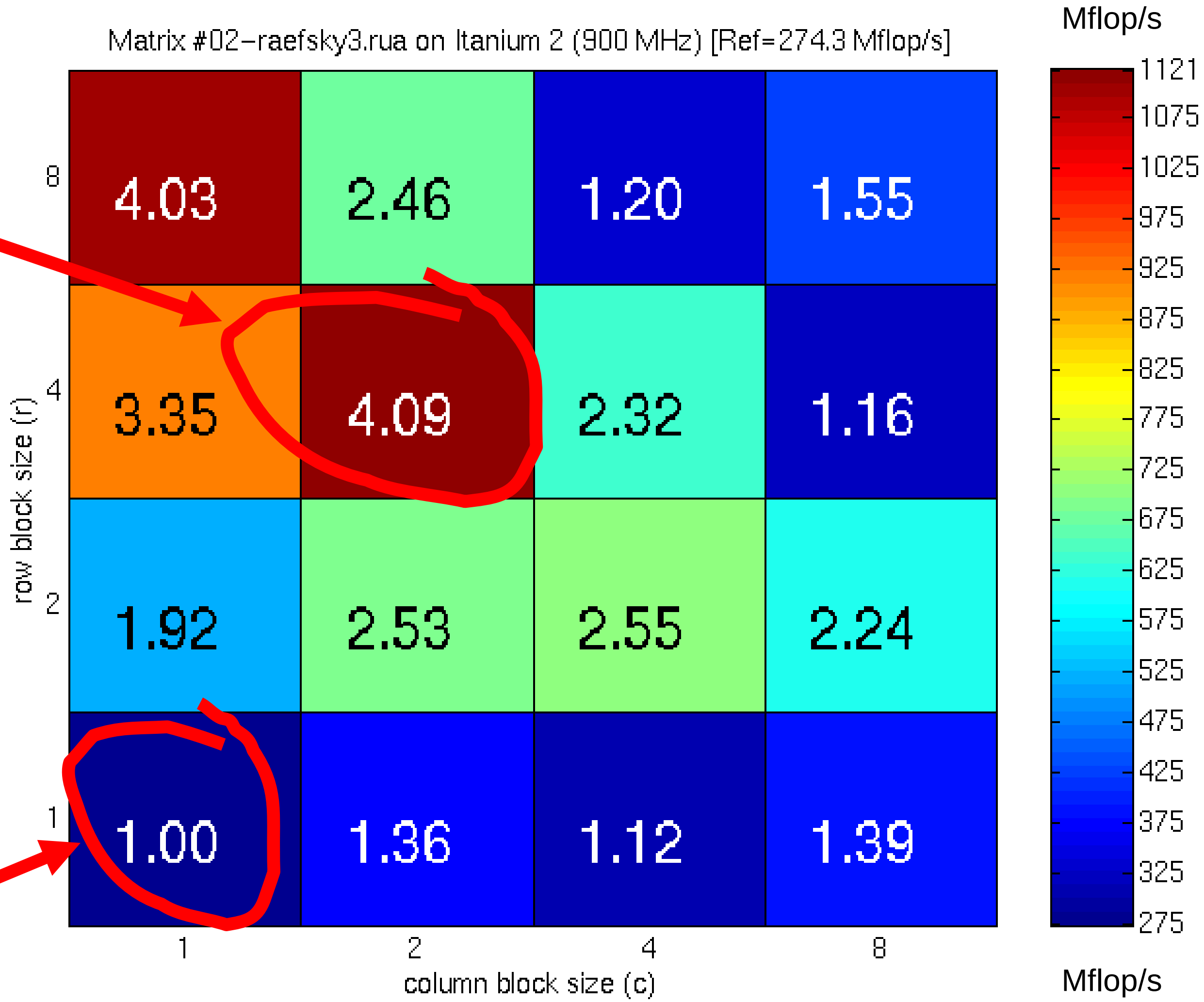


- n = 21200
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem

- **8x8** dense substructure: exploit this to limit #mem_refs

Speedups on Itanium 2: The Need for Search

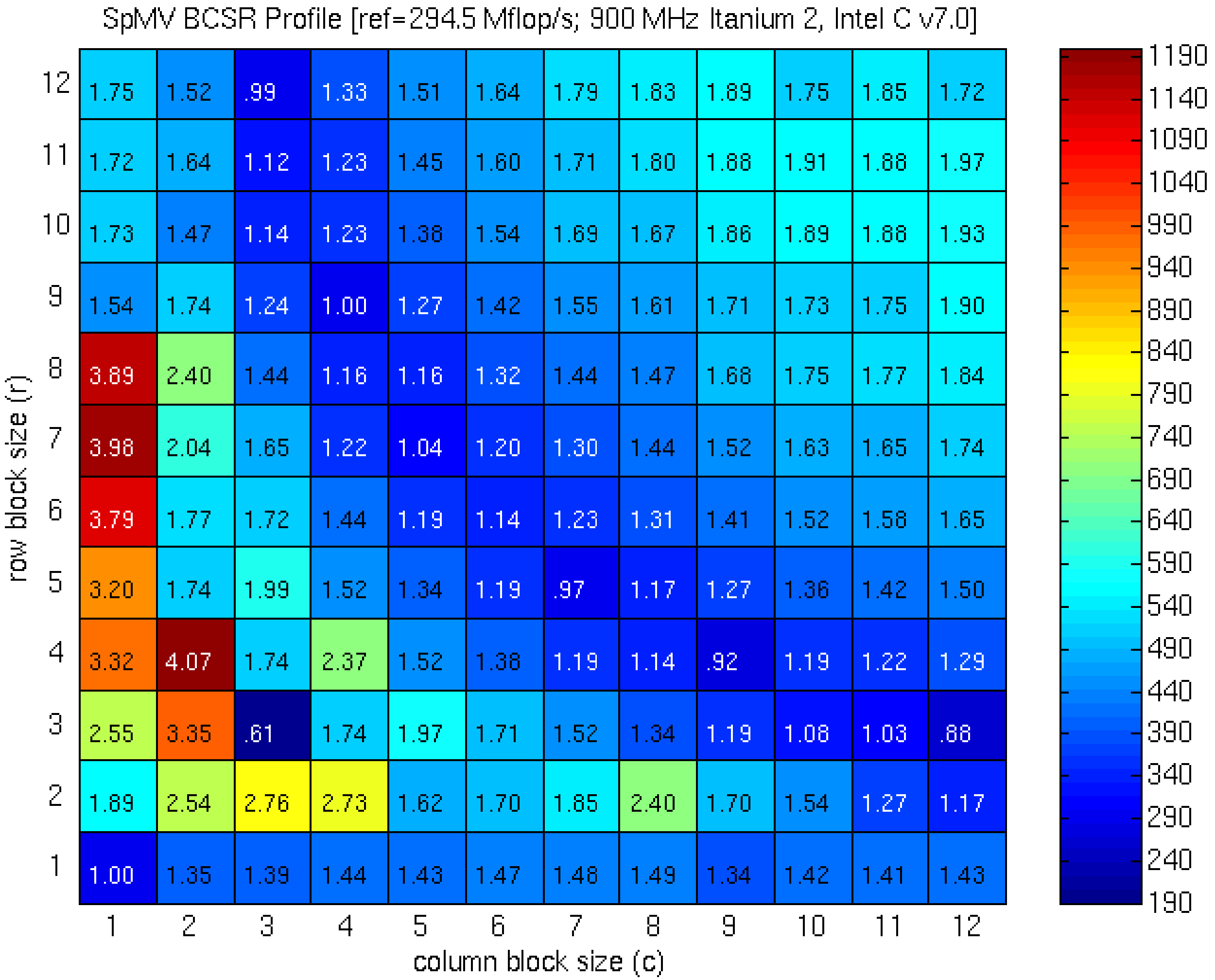


r \ c	1	2	4	8
8	4.03	2.46	1.20	1.55
4	3.35	4.09	2.32	1.16
2	1.92	2.53	2.55	2.24
1	1.00	1.36	1.12	1.39

Best: 4x2

Reference

Register Profile: Itanium 2

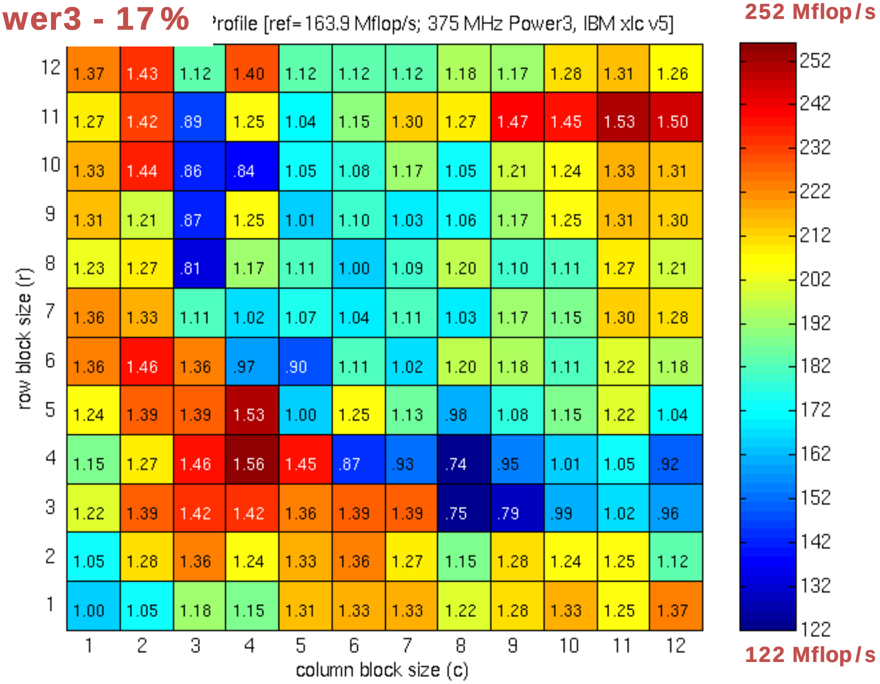


row block size (r)	1	2	3	4	5	6	7	8	9	10	11	12
12	1.75	1.52	.99	1.33	1.51	1.64	1.79	1.83	1.89	1.75	1.85	1.72
11	1.72	1.64	1.12	1.23	1.45	1.60	1.71	1.80	1.88	1.91	1.88	1.97
10	1.73	1.47	1.14	1.23	1.38	1.54	1.69	1.67	1.86	1.89	1.88	1.93
9	1.54	1.74	1.24	1.00	1.27	1.42	1.55	1.61	1.71	1.73	1.75	1.90
8	3.89	2.40	1.44	1.16	1.16	1.32	1.44	1.47	1.68	1.75	1.77	1.84
7	3.98	2.04	1.65	1.22	1.04	1.20	1.30	1.44	1.52	1.63	1.65	1.74
6	3.79	1.77	1.72	1.44	1.19	1.14	1.23	1.31	1.41	1.52	1.58	1.65
5	3.20	1.74	1.99	1.52	1.34	1.19	.97	1.17	1.27	1.36	1.42	1.50
4	3.32	4.07	1.74	2.37	1.52	1.38	1.19	1.14	.92	1.19	1.22	1.29
3	2.55	3.35	.61	1.74	1.97	1.71	1.52	1.34	1.19	1.08	1.03	.88
2	1.89	2.54	2.76	2.73	1.62	1.70	1.85	2.40	1.70	1.54	1.27	1.17
1	1.00	1.35	1.39	1.44	1.43	1.47	1.48	1.49	1.34	1.42	1.41	1.43

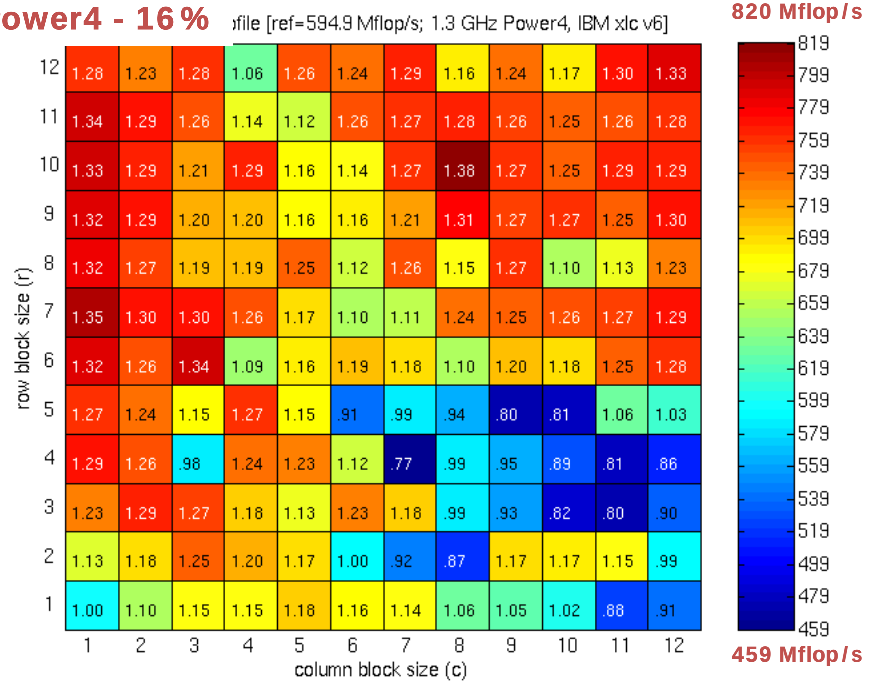
1190 Mflop/s

190 Mflop/s

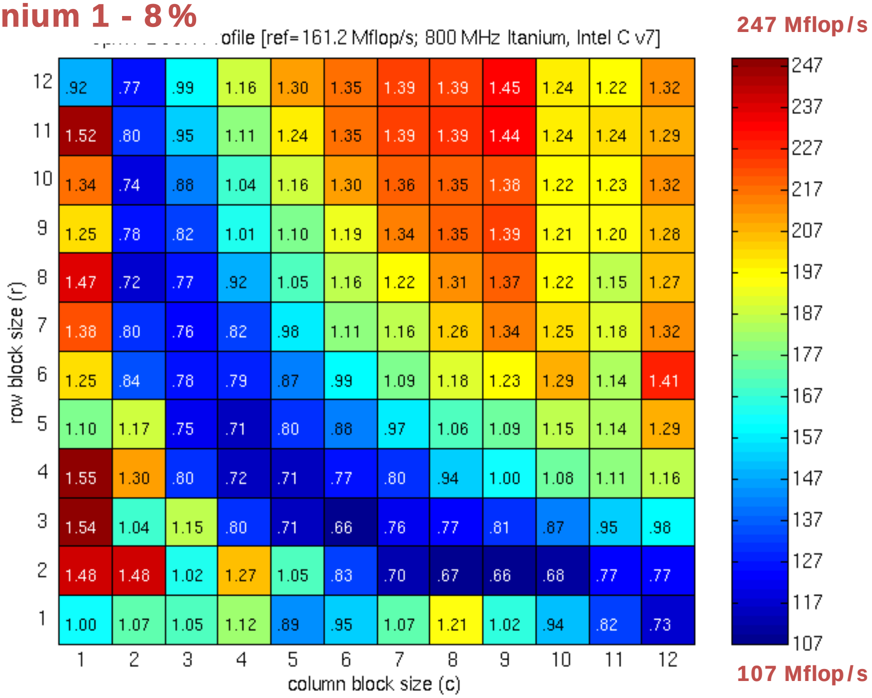
Power3 - 17%



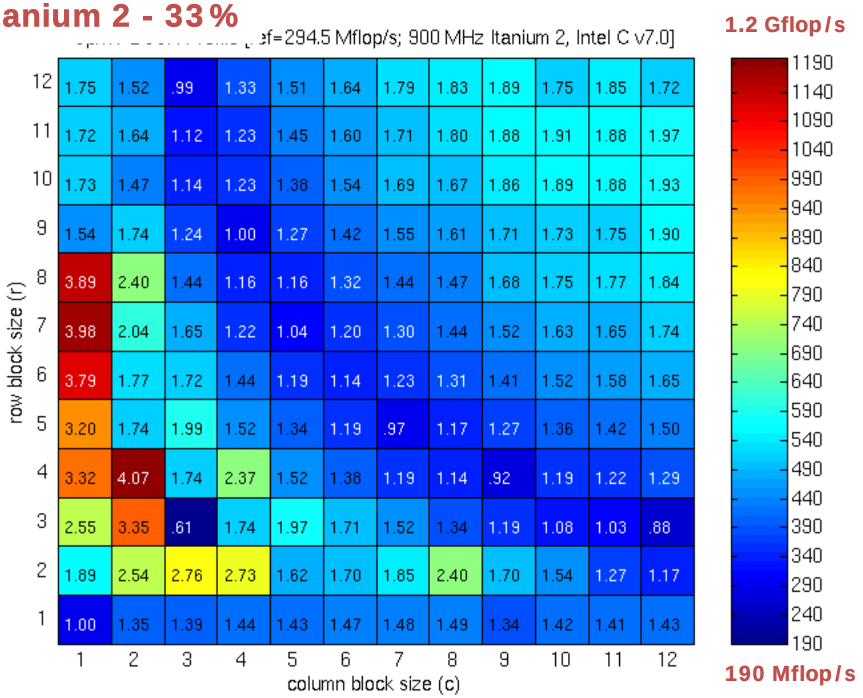
Power4 - 16%



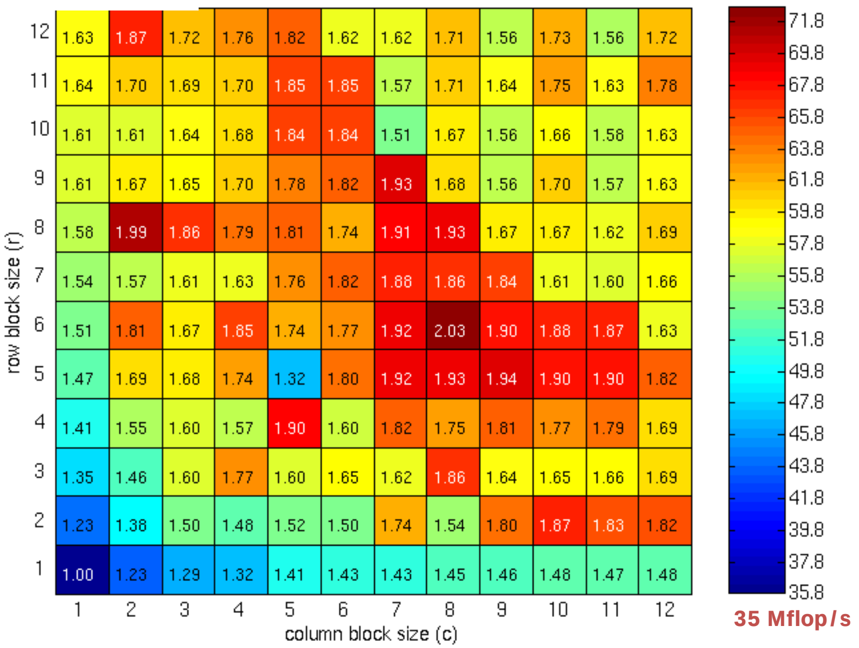
Itanium 1 - 8%



Itanium 2 - 33%



Ultra 2i - 11%

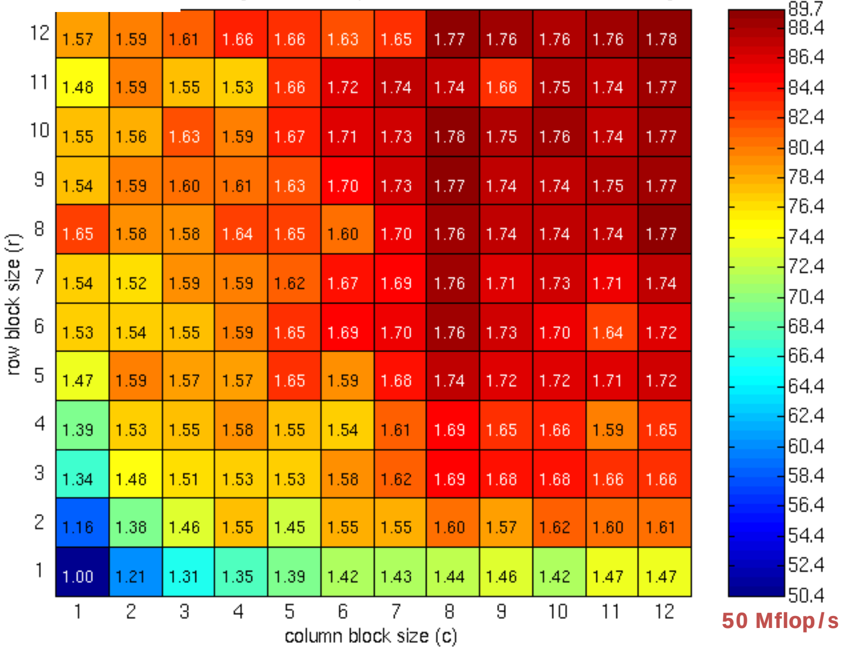

ofile [ref=35.8 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]

72 Mflop/s

35 Mflop/s

row block size (r) \ column block size (c)	1	2	3	4	5	6	7	8	9	10	11	12
12	1.63	1.87	1.72	1.76	1.82	1.62	1.62	1.71	1.56	1.73	1.56	1.72
11	1.64	1.70	1.69	1.70	1.85	1.85	1.57	1.71	1.64	1.75	1.63	1.78
10	1.61	1.61	1.64	1.68	1.84	1.84	1.51	1.67	1.56	1.66	1.58	1.63
9	1.61	1.67	1.65	1.70	1.78	1.82	1.93	1.68	1.56	1.70	1.57	1.63
8	1.58	1.99	1.86	1.79	1.81	1.74	1.91	1.93	1.67	1.67	1.62	1.69
7	1.54	1.57	1.61	1.63	1.76	1.82	1.88	1.86	1.84	1.61	1.60	1.66
6	1.51	1.81	1.67	1.85	1.74	1.77	1.92	2.03	1.90	1.88	1.87	1.63
5	1.47	1.69	1.68	1.74	1.32	1.80	1.92	1.93	1.94	1.90	1.90	1.82
4	1.41	1.55	1.60	1.57	1.90	1.60	1.82	1.75	1.81	1.77	1.79	1.69
3	1.35	1.46	1.60	1.77	1.60	1.65	1.62	1.86	1.64	1.65	1.66	1.69
2	1.23	1.38	1.50	1.48	1.52	1.50	1.74	1.54	1.80	1.87	1.83	1.82
1	1.00	1.23	1.29	1.32	1.41	1.43	1.43	1.45	1.46	1.48	1.47	1.48

Ultra 3 - 5%

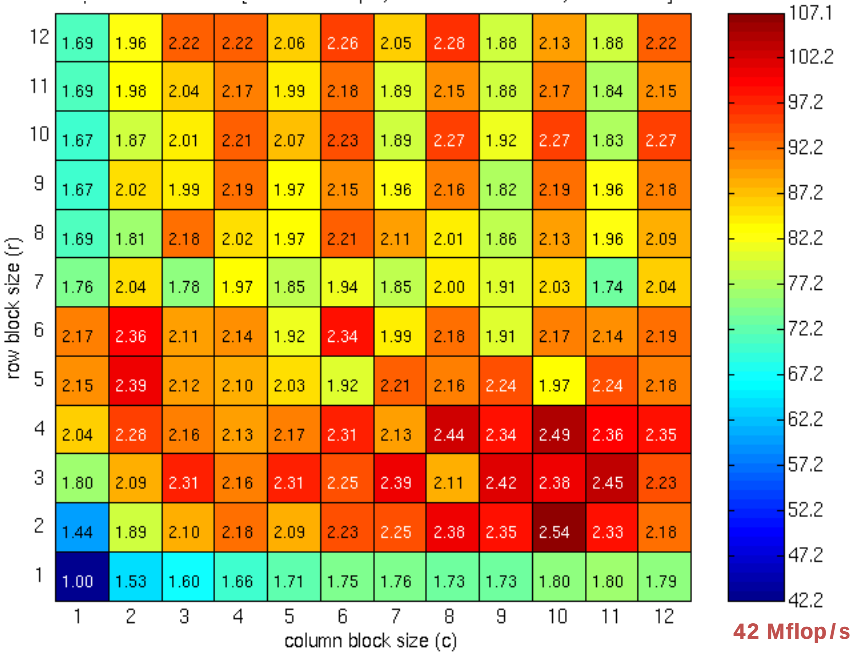

Profile [ref=50.3 Mflop/s; 900 MHz Sun Ultra 3, Sun C v6.0]

90 Mflop/s

50 Mflop/s

row block size (r) \ column block size (c)	1	2	3	4	5	6	7	8	9	10	11	12
12	1.57	1.59	1.61	1.66	1.66	1.63	1.65	1.77	1.76	1.76	1.76	1.78
11	1.48	1.59	1.55	1.53	1.66	1.72	1.74	1.74	1.66	1.75	1.74	1.77
10	1.55	1.56	1.63	1.59	1.67	1.71	1.73	1.78	1.75	1.76	1.74	1.77
9	1.54	1.59	1.60	1.61	1.63	1.70	1.73	1.77	1.74	1.74	1.75	1.77
8	1.65	1.58	1.58	1.64	1.65	1.60	1.70	1.76	1.74	1.74	1.74	1.77
7	1.54	1.52	1.59	1.59	1.62	1.67	1.69	1.76	1.71	1.73	1.71	1.74
6	1.53	1.54	1.55	1.59	1.65	1.69	1.70	1.76	1.73	1.70	1.64	1.72
5	1.47	1.59	1.57	1.57	1.65	1.59	1.68	1.74	1.72	1.72	1.71	1.72
4	1.39	1.53	1.55	1.58	1.55	1.54	1.61	1.69	1.65	1.66	1.59	1.65
3	1.34	1.48	1.51	1.53	1.53	1.58	1.62	1.69	1.68	1.68	1.66	1.66
2	1.16	1.38	1.46	1.55	1.45	1.55	1.55	1.60	1.57	1.62	1.60	1.61
1	1.00	1.21	1.31	1.35	1.39	1.42	1.43	1.44	1.46	1.42	1.47	1.47

Pentium III - 21%

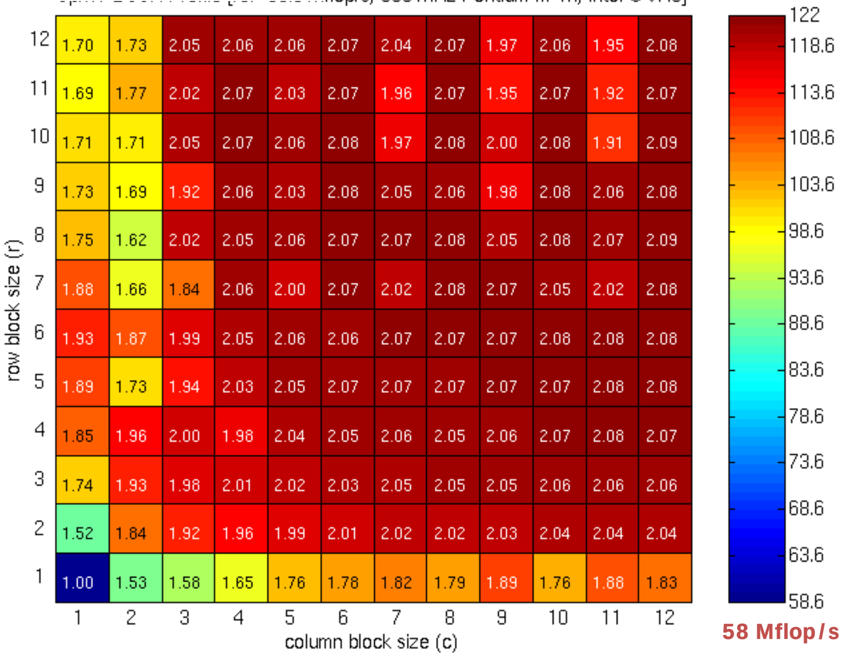

=42.1 Mflop/s; 500 MHz Pentium III, Intel C v7.0]

108 Mflop/s

42 Mflop/s

row block size (r) \ column block size (c)	1	2	3	4	5	6	7	8	9	10	11	12
12	1.69	1.96	2.22	2.22	2.06	2.26	2.05	2.28	1.88	2.13	1.88	2.22
11	1.69	1.98	2.04	2.17	1.99	2.18	1.89	2.15	1.88	2.17	1.84	2.15
10	1.67	1.87	2.01	2.21	2.07	2.23	1.89	2.27	1.92	2.27	1.83	2.27
9	1.67	2.02	1.99	2.19	1.97	2.15	1.96	2.16	1.82	2.19	1.96	2.18
8	1.69	1.81	2.18	2.02	1.97	2.21	2.11	2.01	1.86	2.13	1.96	2.09
7	1.76	2.04	1.78	1.97	1.85	1.94	1.85	2.00	1.91	2.03	1.74	2.04
6	2.17	2.36	2.11	2.14	1.92	2.34	1.99	2.18	1.91	2.17	2.14	2.19
5	2.15	2.39	2.12	2.10	2.03	1.92	2.21	2.16	2.24	1.97	2.24	2.18
4	2.04	2.28	2.16	2.13	2.17	2.31	2.13	2.44	2.34	2.49	2.36	2.35
3	1.80	2.09	2.31	2.16	2.31	2.25	2.39	2.11	2.42	2.38	2.45	2.23
2	1.44	1.89	2.10	2.18	2.09	2.23	2.25	2.38	2.35	2.54	2.33	2.18
1	1.00	1.53	1.60	1.66	1.71	1.75	1.76	1.73	1.73	1.80	1.80	1.79

Pentium III-M - 15%

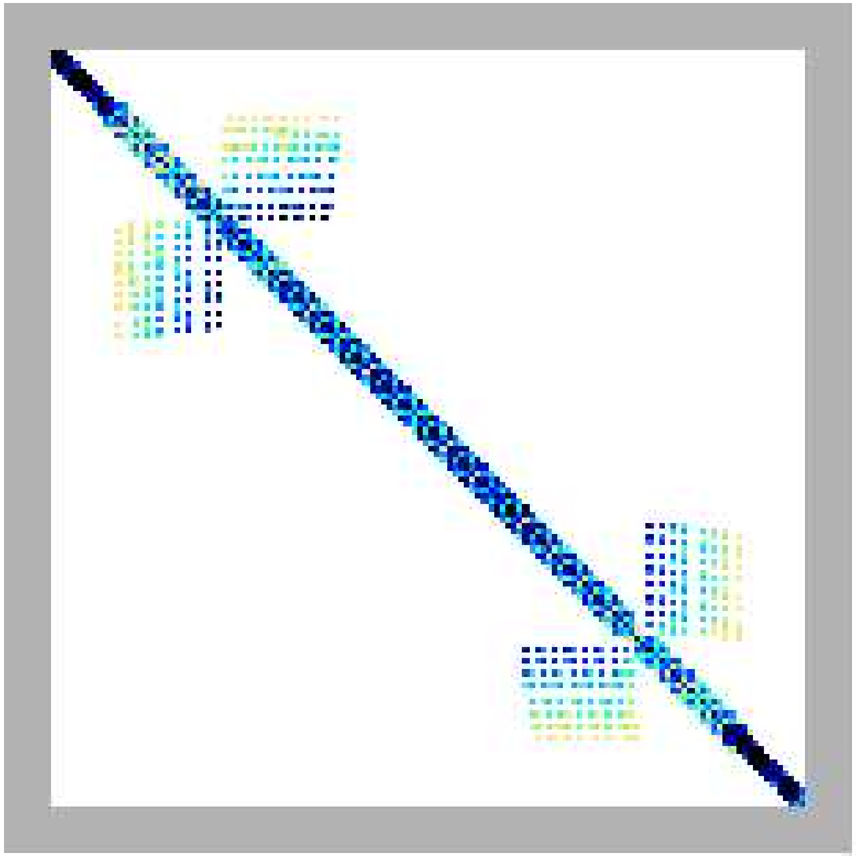

..flop/s; 800 MHz Pentium III–M, Intel C v7.0]

122 Mflop/s

58 Mflop/s

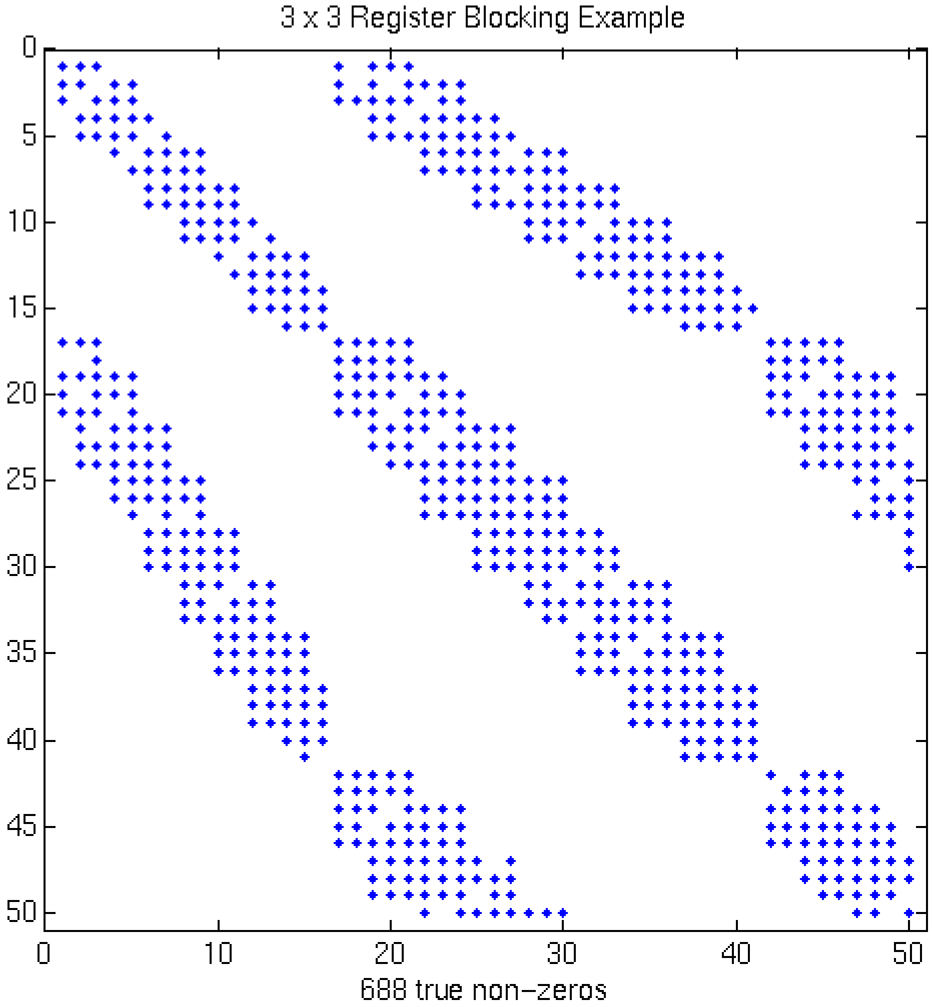
row block size (r) \ column block size (c)	1	2	3	4	5	6	7	8	9	10	11	12
12	1.70	1.73	2.05	2.06	2.06	2.07	2.04	2.07	1.97	2.06	1.95	2.08
11	1.69	1.77	2.02	2.07	2.03	2.07	1.96	2.07	1.95	2.07	1.92	2.07
10	1.71	1.71	2.05	2.07	2.06	2.08	1.97	2.08	2.00	2.08	1.91	2.09
9	1.73	1.69	1.92	2.06	2.03	2.08	2.05	2.06	1.98	2.08	2.06	2.08
8	1.75	1.62	2.02	2.05	2.06	2.07	2.07	2.08	2.05	2.08	2.07	2.09
7	1.88	1.66	1.84	2.06	2.00	2.07	2.02	2.08	2.07	2.05	2.02	2.08
6	1.93	1.87	1.99	2.05	2.06	2.06	2.07	2.07	2.07	2.08	2.08	2.08
5	1.89	1.73	1.94	2.03	2.05	2.07	2.07	2.07	2.07	2.07	2.08	2.08
4	1.85	1.96	2.00	1.98	2.04	2.05	2.06	2.05	2.06	2.07	2.08	2.07
3	1.74	1.93	1.98	2.01	2.02	2.03	2.05	2.05	2.05	2.06	2.06	2.06
2	1.52	1.84	1.92	1.96	1.99	2.01	2.02	2.02	2.03	2.04	2.04	2.04
1	1.00	1.53	1.58	1.65	1.76	1.78	1.82	1.79	1.89	1.76	1.88	1.83

Another example of tuning challenges



- More complicated non-zero structure in general
- N = 16614
- NNZ = 1.1M

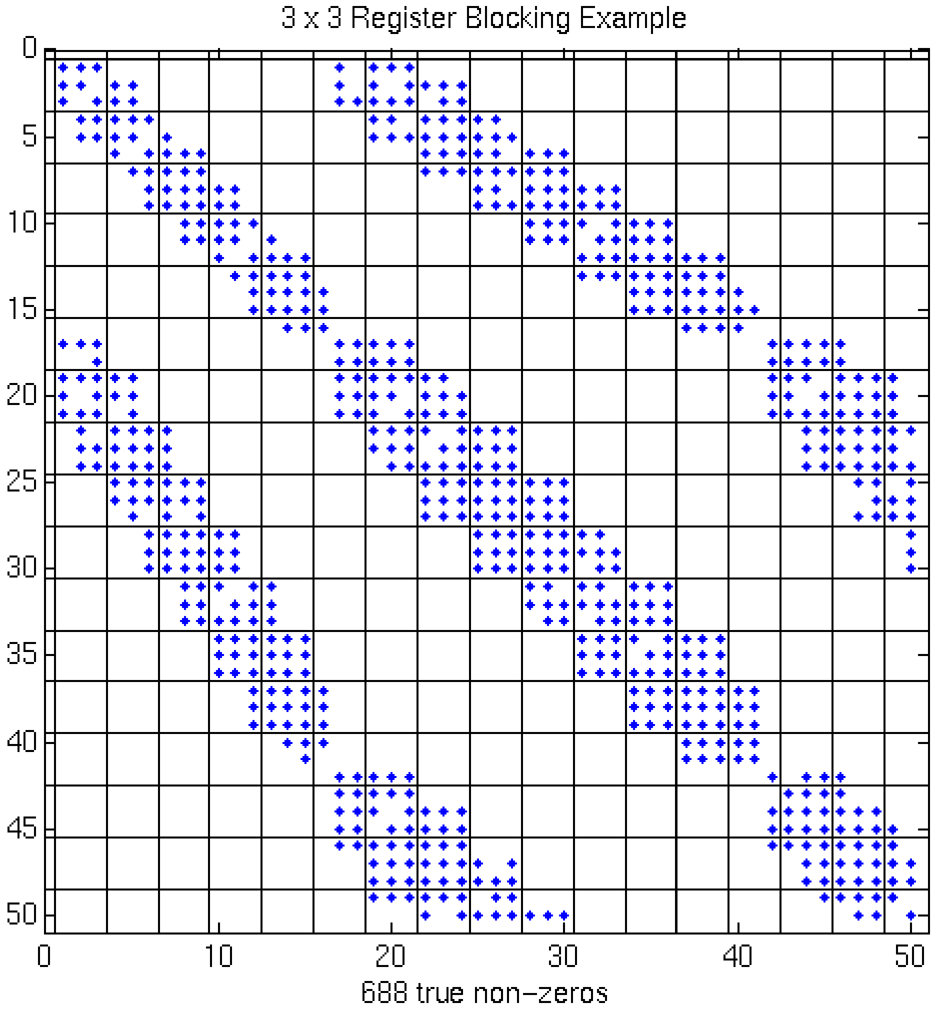
Zoom in to top corner



- More complicated non-zero structure in general

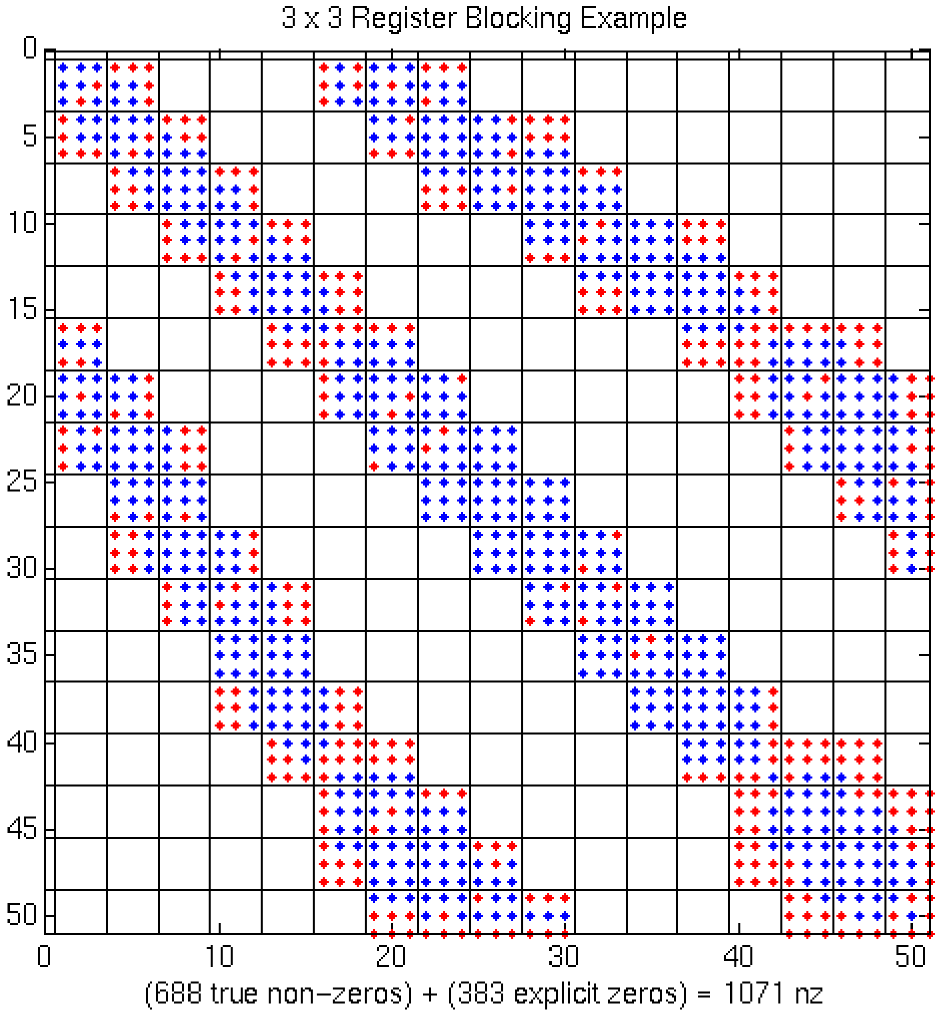
- N = 16614
- NNZ = 1.1M

3x3 blocks look natural, but…



- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
- But would lead to lots of "fill-in"

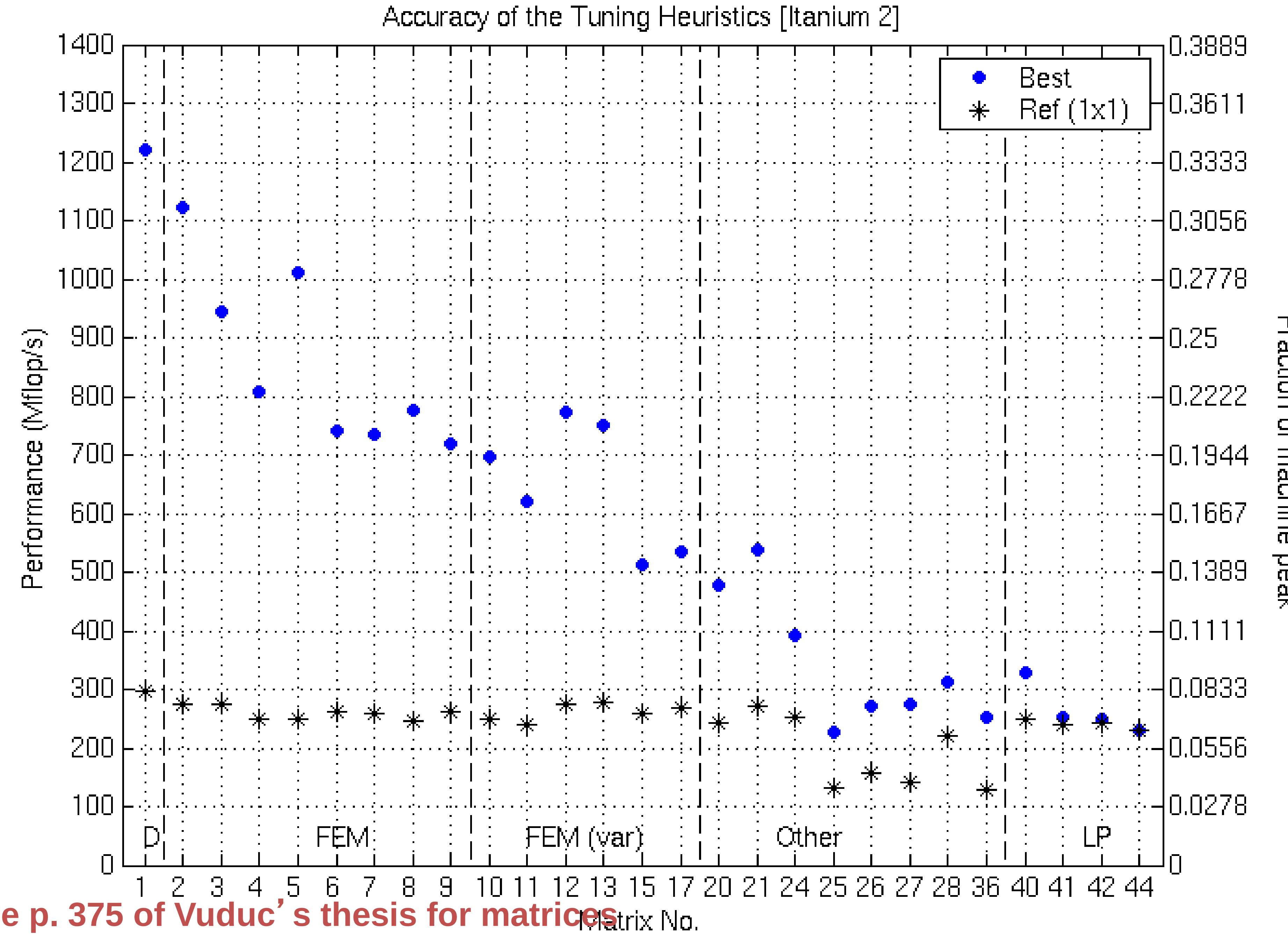
Extra Work Can Improve Efficiency!



- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
  - Fill-in explicit zeros
  - Unroll 3x3 block multiplies
  - “Fill ratio” = 1.5

Selecting Register Block Size r x c

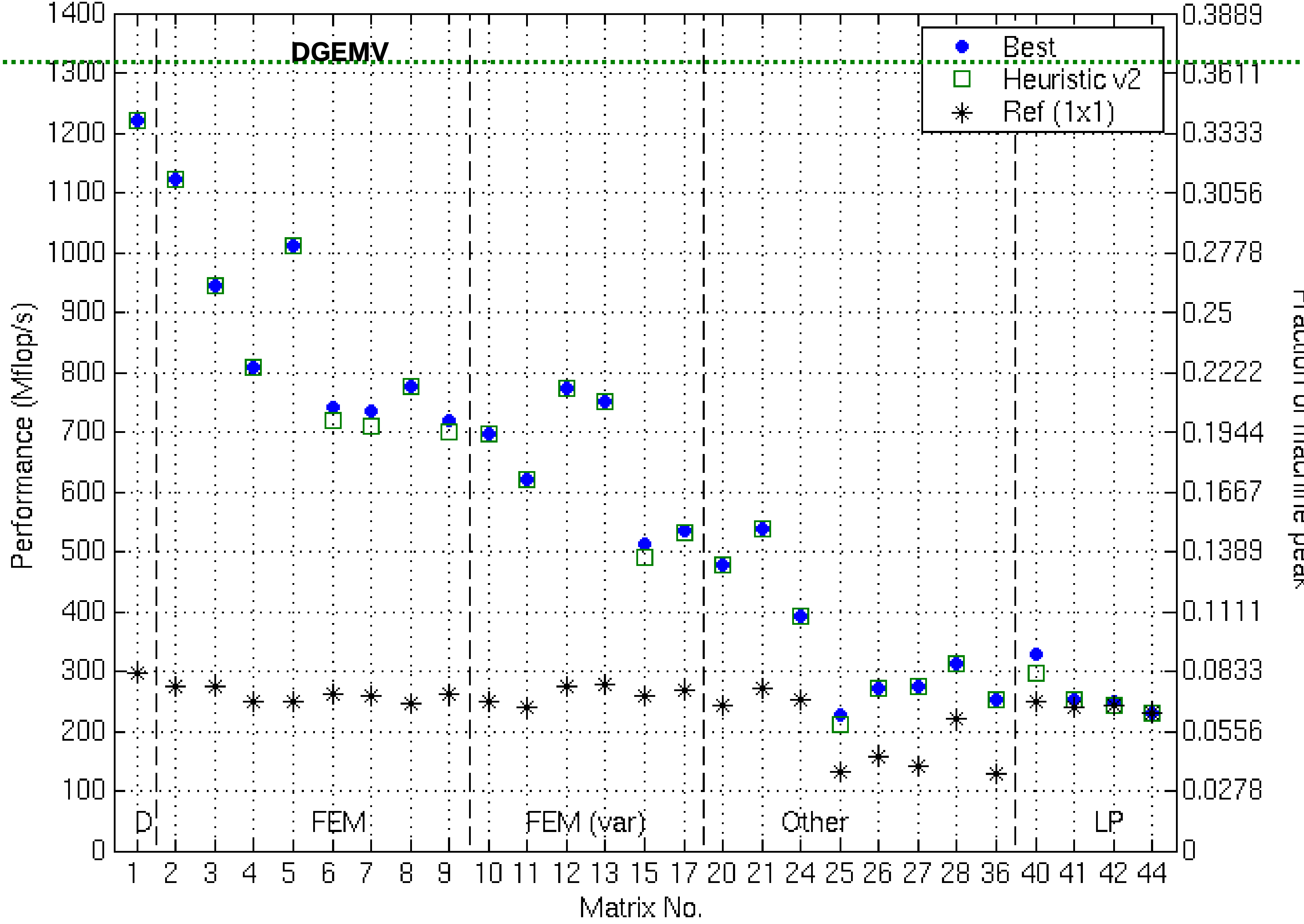
- **Off-line benchmark**
  - Precompute **Mflops(r,c)** using dense A for each r x c
  - Once per machine/architecture
- **Run-time "search"**
  - Sample *A* to estimate **Fill(r,c)** for each r x c
  - Control cost = O(*s*·nnz) by controlling fraction *s* ∈ [0,1] sampled
  - Control *s* automatically by computing statistical confidence intervals, by monitoring variance
- **Run-time heuristic model**
  - Choose r, c to minimize **time ~ Fill(r,c) / Mflops(r,c)**
- **Cost of tuning**
  - Lower bound: convert matrix in 5 to 40 unblocked SpMVs
  - Heuristic: 1 to 11 SpMVs
- **Tuning only useful when we do many SpMVs**
  - Common case, eg in sparse solvers

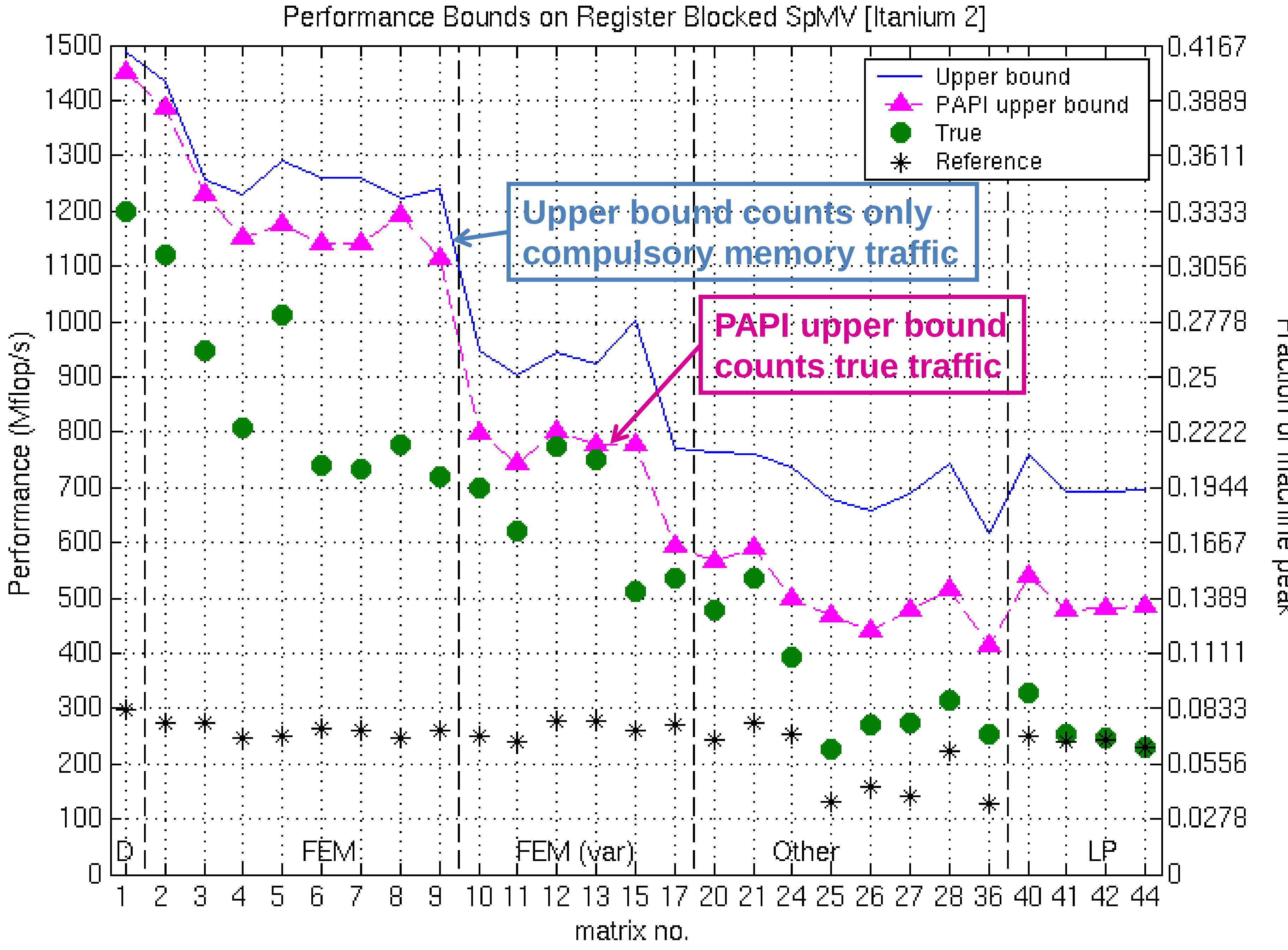


See p. 375 of Vuduc’s thesis for matrices

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

Accuracy of the Tuning Heuristics [Itanium 2]





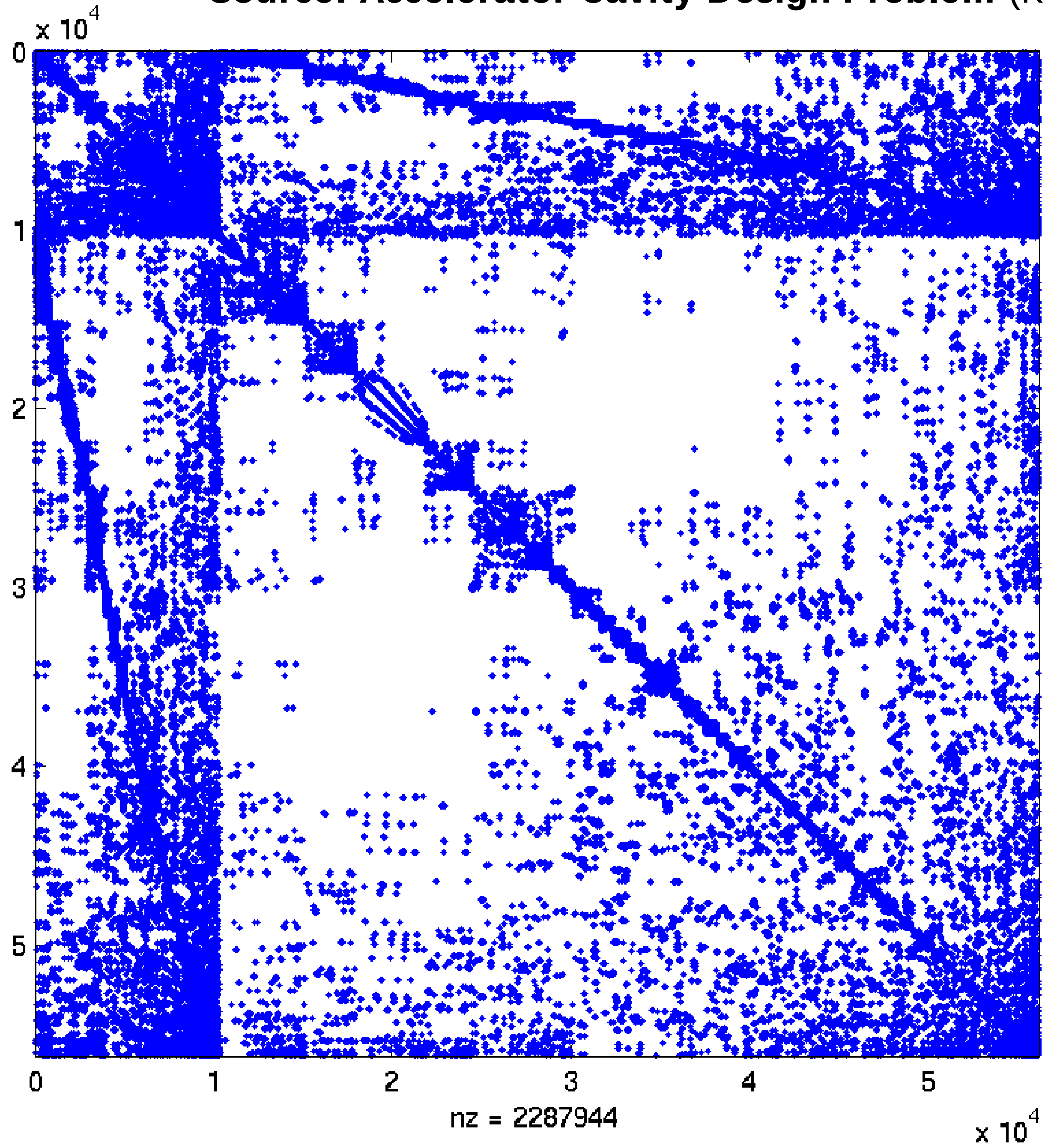
Upper bound counts only compulsory memory traffic

PAPI upper bound counts true traffic

Summary of Other Sequential Performance Optimizations

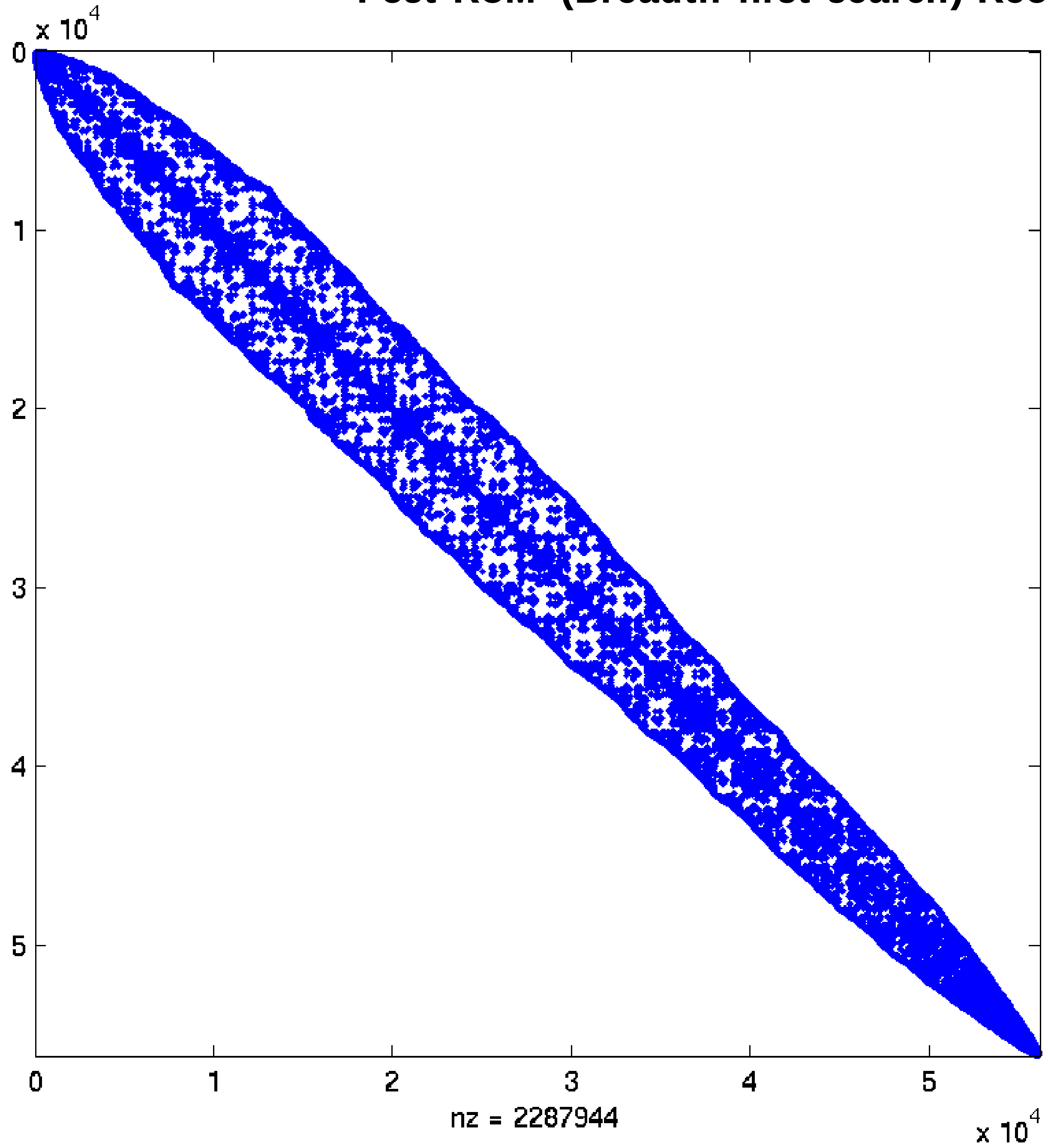
- Optimizations for SpMV
  - **Register blocking (RB)**: up to **4x** over CSR
  - **Variable block splitting**: **2.1x** over CSR, **1.8x** over RB
  - **Diagonals**: **2x** over CSR
  - **Reordering** to create dense structure + **splitting**: **2x** over CSR
  - **Symmetry**: **2.8x** over CSR, 2.6x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

Source: Accelerator Cavity Design Problem (Ko via Husbands)



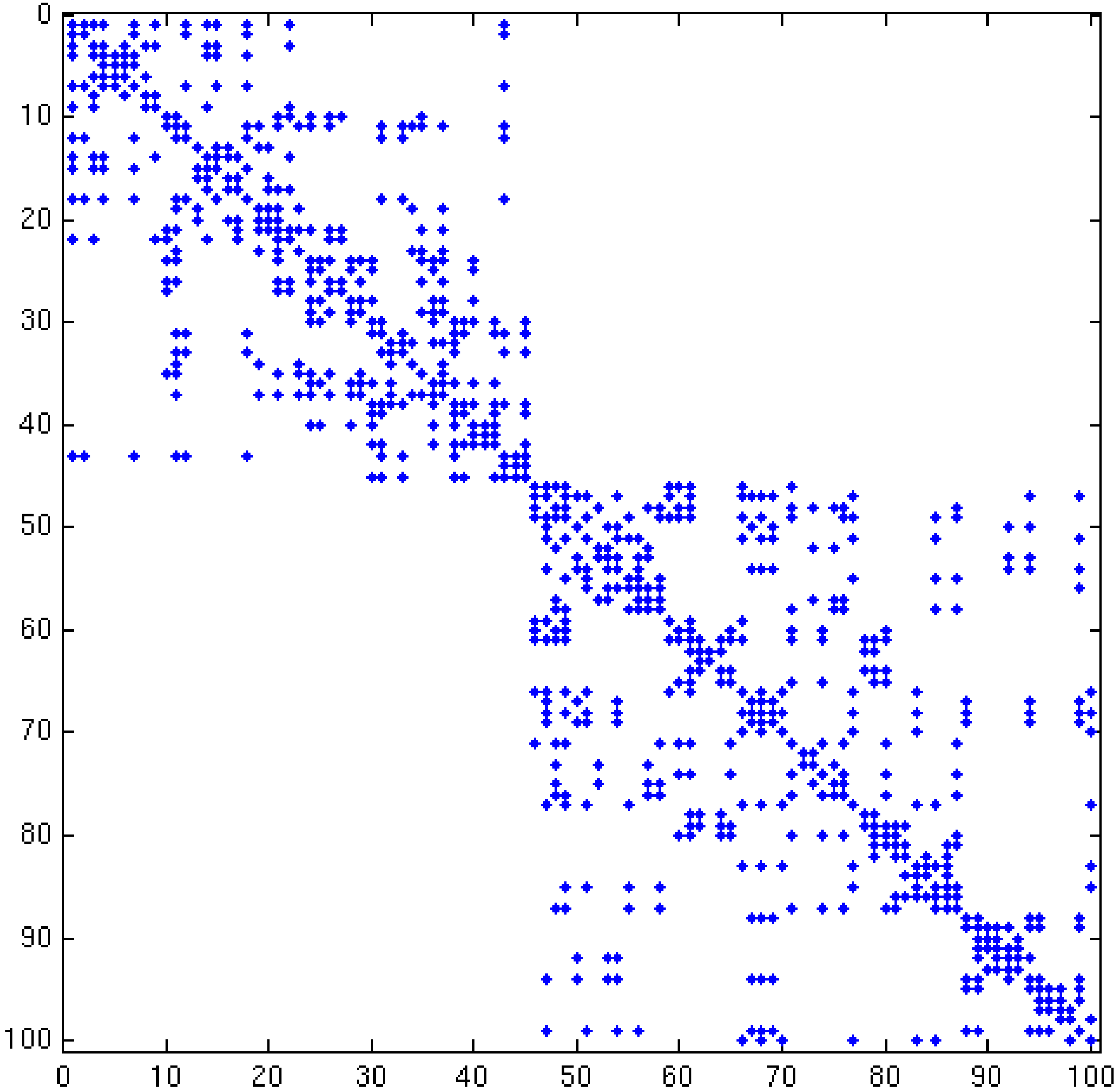
Can we reorder the rows and columns to create dense blocks, to accelerate SpMV?

Post-RCM (Breadth-first-search) Reordering



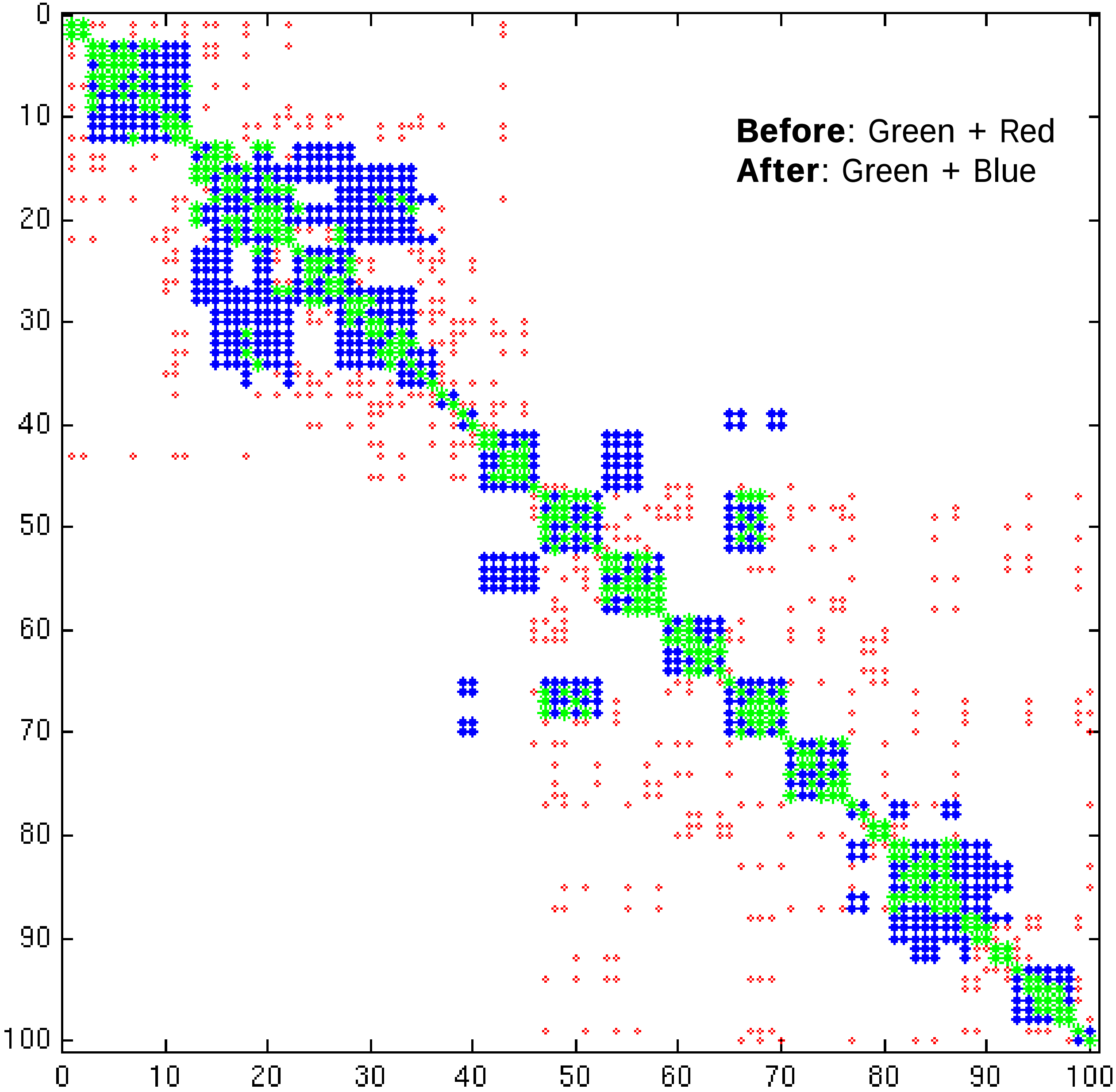
Moving nonzeros nearer the diagonal should create dense block, but let’s zoom in and see…

100x100 Submatrix Along Diagonal



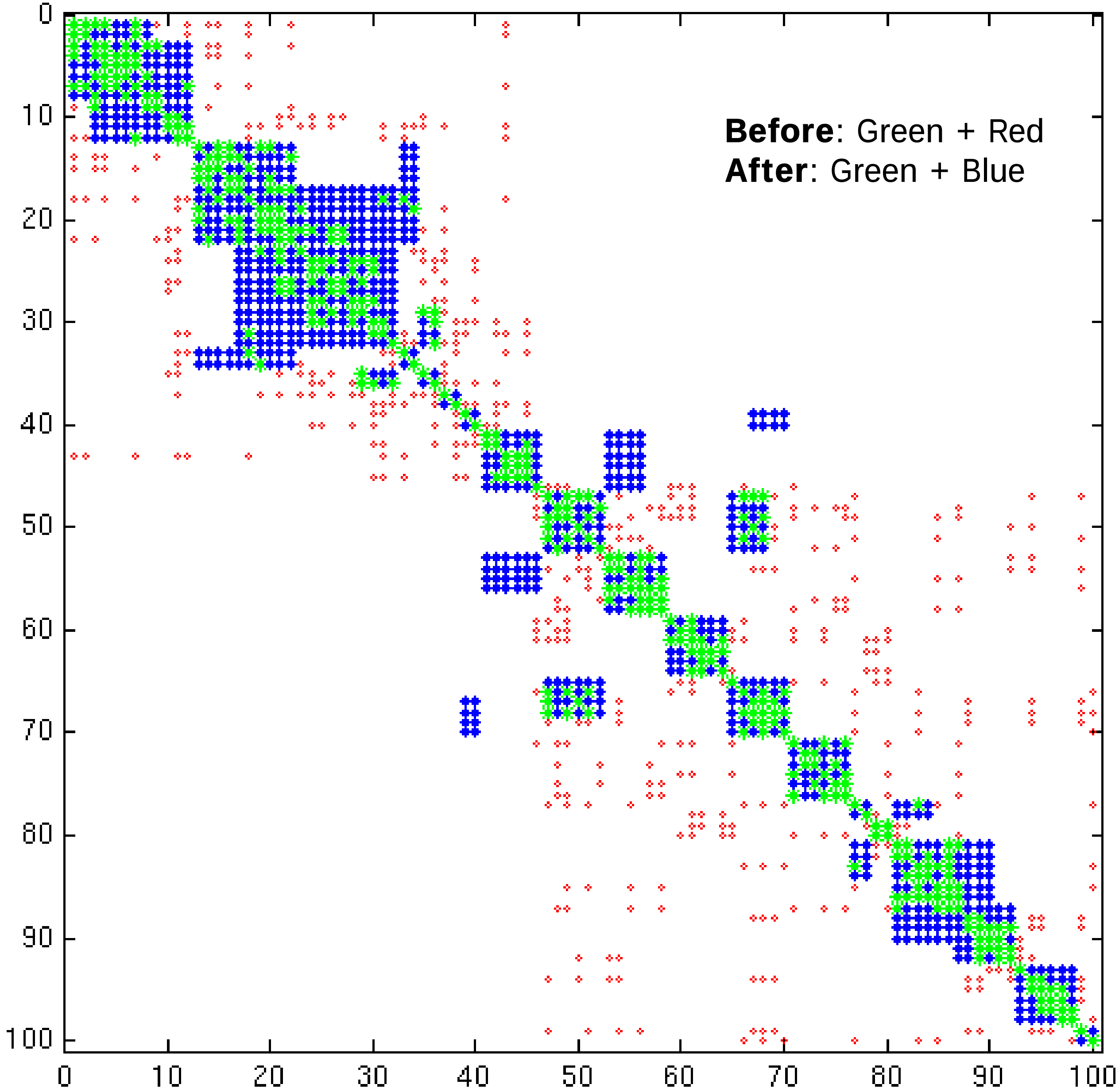
Here is the top 100x100 submatrix before RCM

“Microscopic” Effect of RCM Reordering



Here is the top 100x100 submatrix after RCM: red entries move to the blue locations.
More dense blocks, but could be better, so let’s try reordering again, using TSP (Travelling Saleman Problem)

"Microscopic" Effect of Combined RCM+TSP Reordering



Before: Green + Red
After: Green + Blue

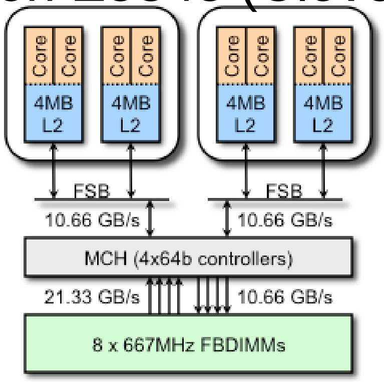
Here is the top 100x100 submatrix after RCM and TSP: red entries move to the blue locations. Lots of dense blocks, as desired!

Speedups (using symmetry too):

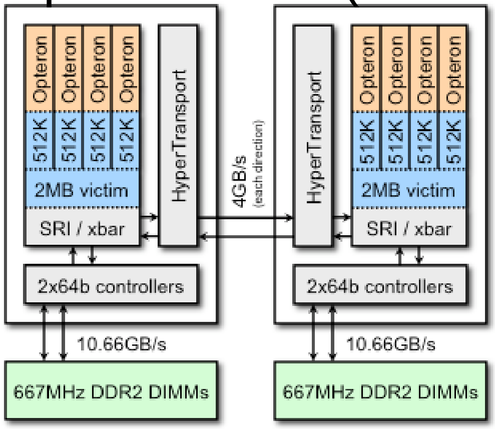
Itanium 2:	1.7x
Pentium 4:	2.1x
Power 4:	2.1x
Ultra 3:	3.3x

Multicore SMPs Used for Tuning SpMV

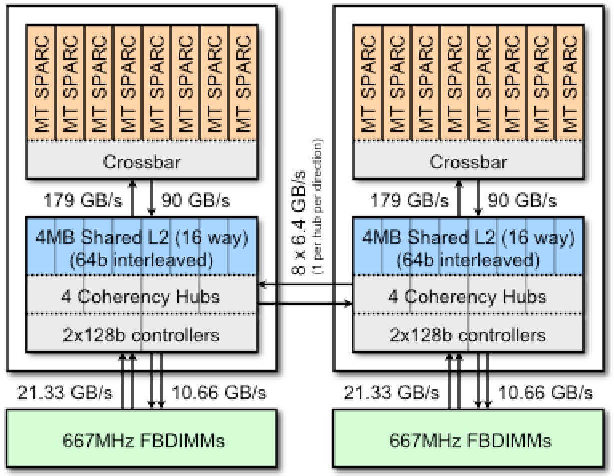
Intel Xeon E5345 (Clovertown)



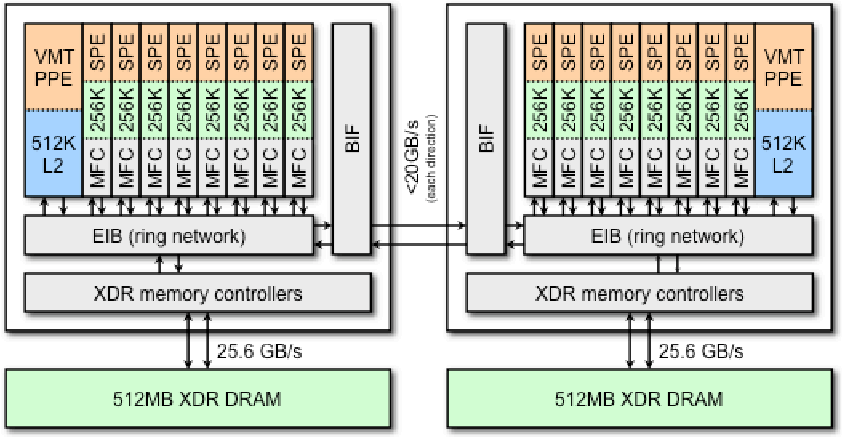
AMD Opteron 2356 (Barcelona)



Sun T2+ T5140 (Victoria Falls)



IBM QS20 Cell Blade



Multicore SMPs Used for Tuning SpMV

Intel Xeon E5345 (Clovertown)

- Cache based
- 8 Threads
- 75 GFlops
- 21/10 GB/s R/W BW

AMD Opteron 2356 (Barcelona)

- Cache based
- 8 Threads
- NUMA
- 74 GFlops
- 21 GB/s R/W BW

Sun T2+ T5140 (Victoria Falls)

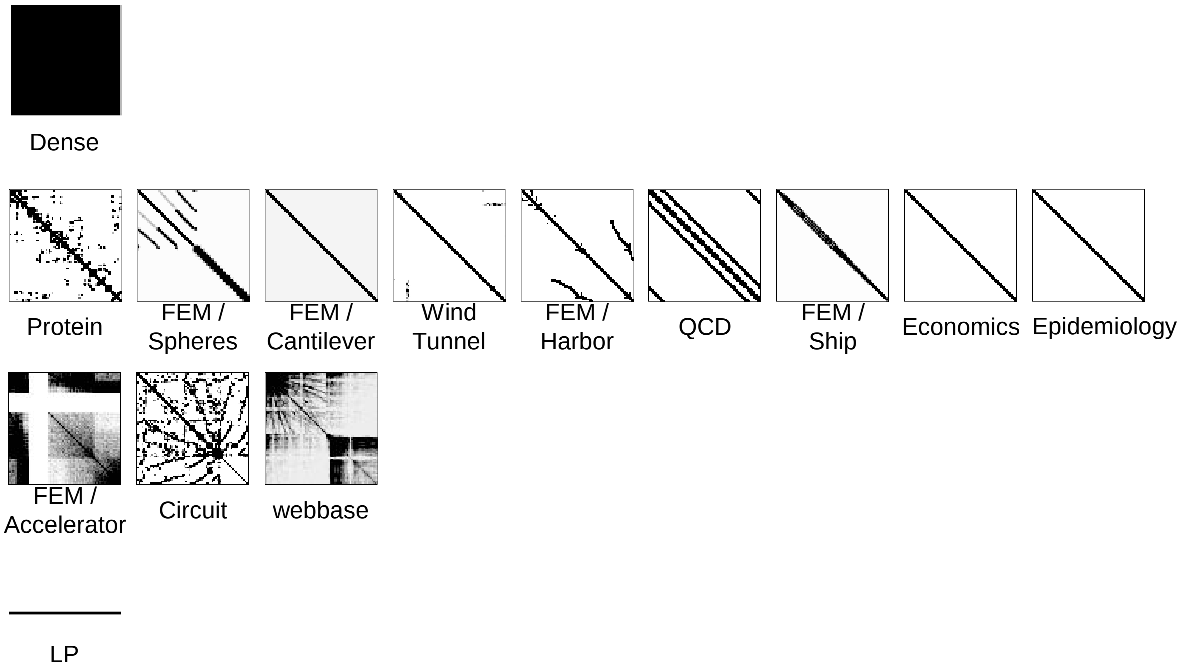
- Cache based
- 128 Threads (CMT)
- NUMA
- 19 GFlops
- 42/21 GB/s R/W BW

IBM QS20 Cell Blade

- Local-Store based
- 16 Threads
- NUMA
- 29 Gflops (SPEs only)
- 51 GB/s R/W BW

Set of 14 test matrices

- All bigger than the caches of our SMPs



2K x 2K Dense matrix stored in sparse format

Dense

Well Structured (sorted by nonzeros/row)

Protein, FEM / Spheres, FEM / Cantilever, Wind Tunnel, FEM / Harbor, QCD, FEM / Ship, Economics, Epidemiology

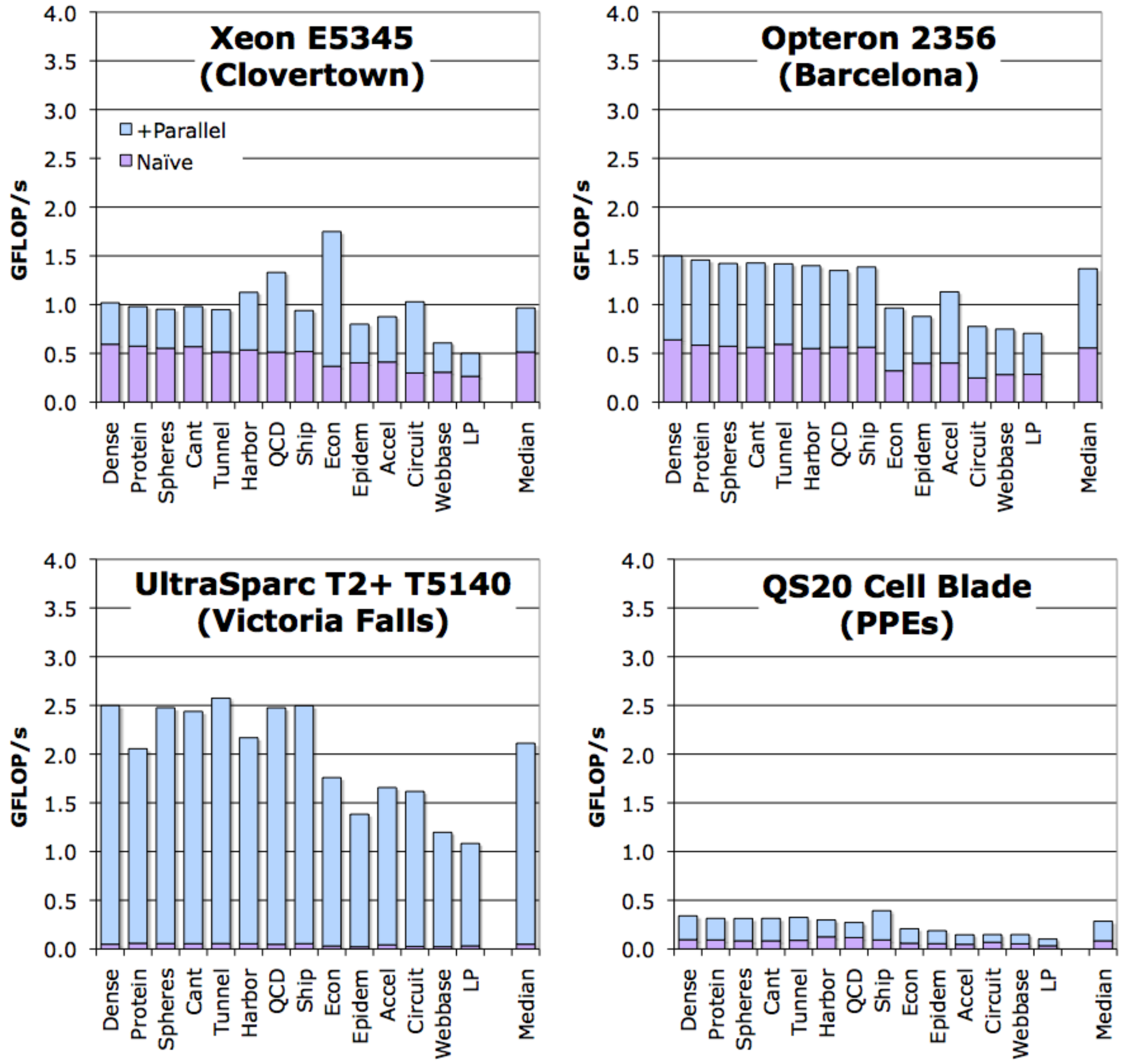
Poorly Structured hodgepodge

FEM / Accelerator, Circuit, webbase

Extreme Aspect Ratio (linear programming)

LP

SpMV Performance: Naive parallelization



- Out-of-the box SpMV performance on a suite of 14 matrices
- **Scalability isn't great:**

Compare to # threads

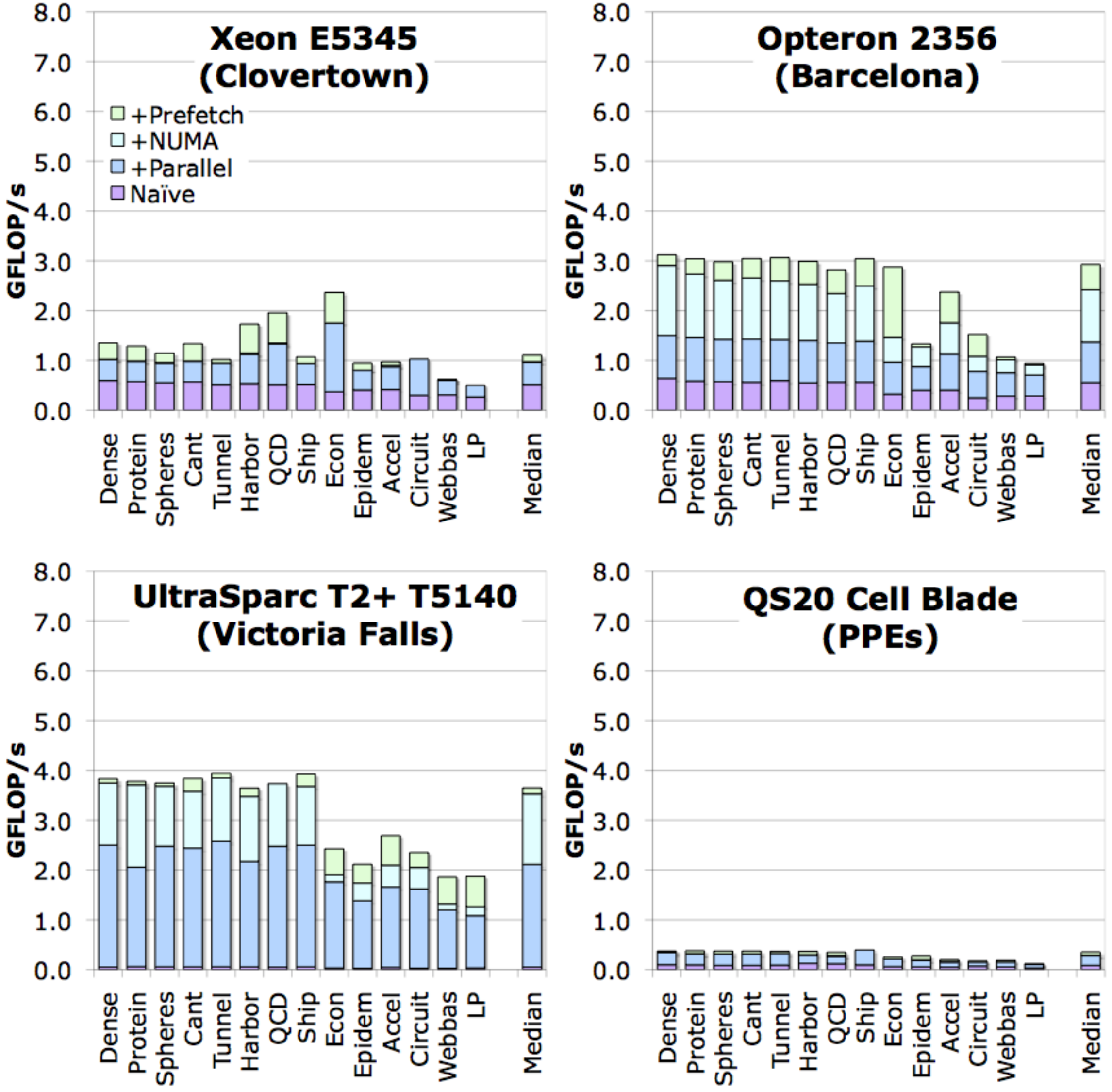
8 8

128 16

🟦 Naïve Pthreads

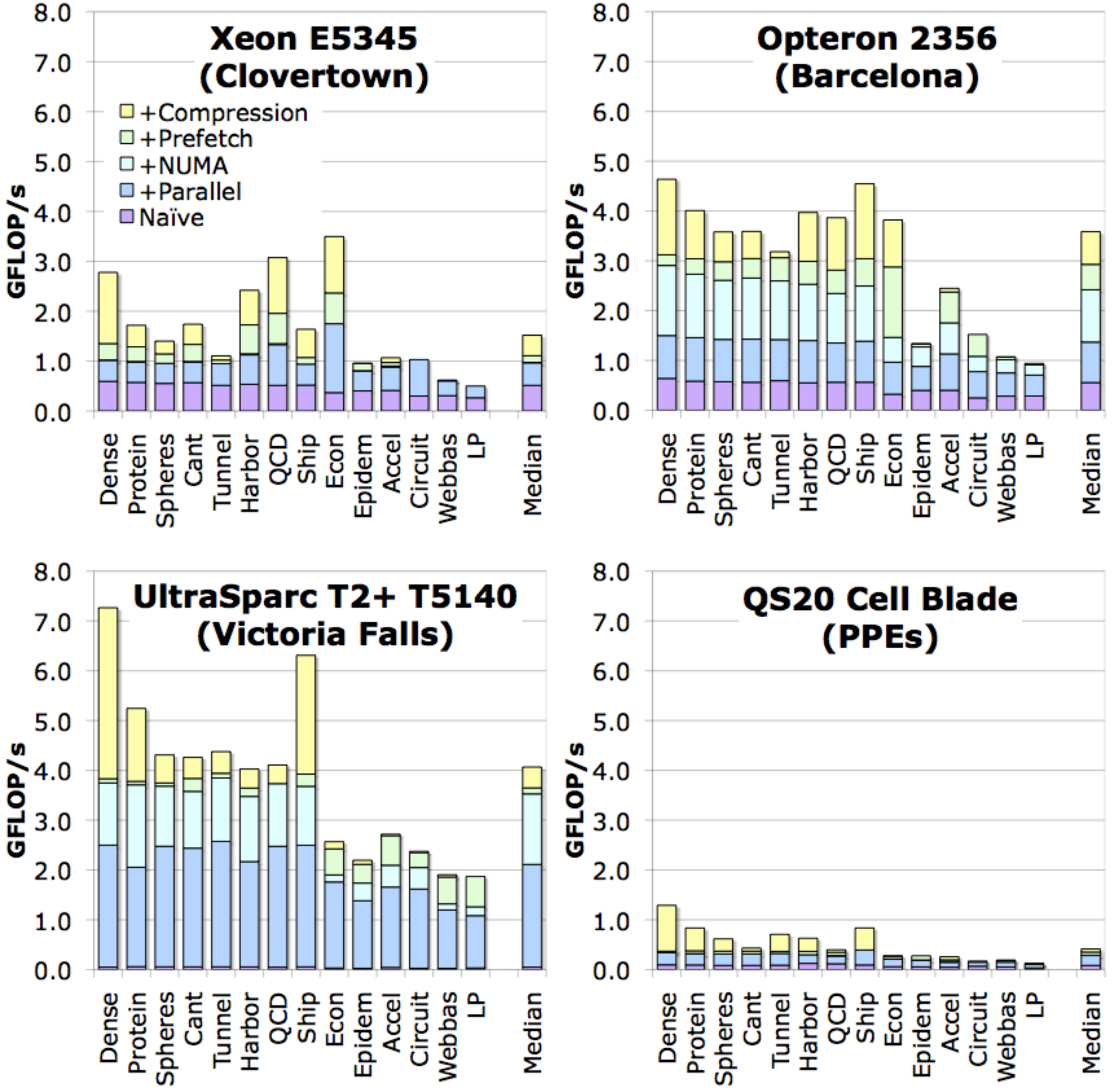
🟪 Naïve

SpMV Performance: NUMA and Software Prefetching



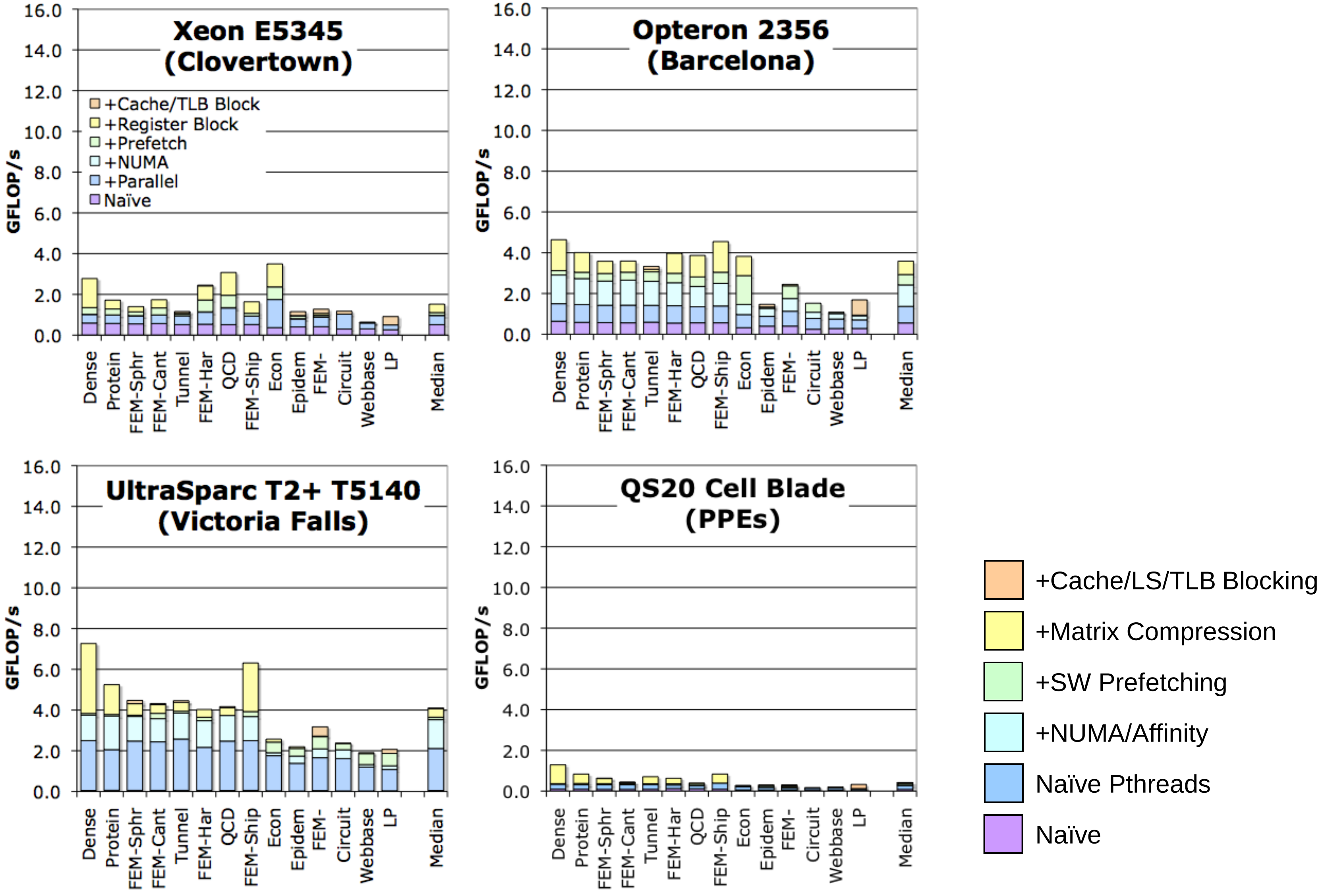
- NUMA-aware allocation is essential on NUMA SMPs.
- Explicit software prefetching can boost bandwidth and change cache replacement policies
- used **exhaustive** search

SpMV Performance: "Matrix Compression"

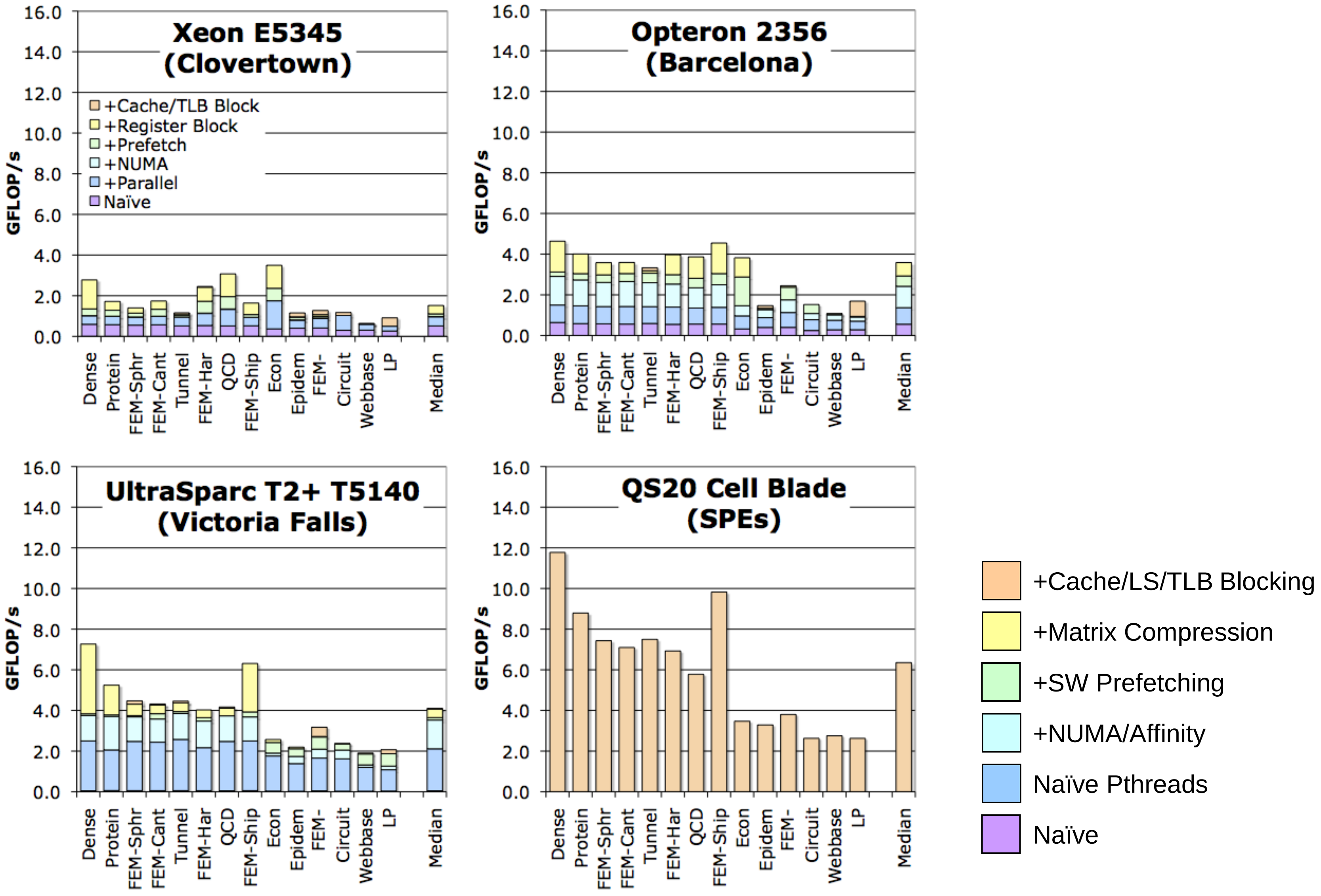


- Compression includes
  - register blocking
  - other formats
  - smaller indices
- Use **heuristic** rather than search

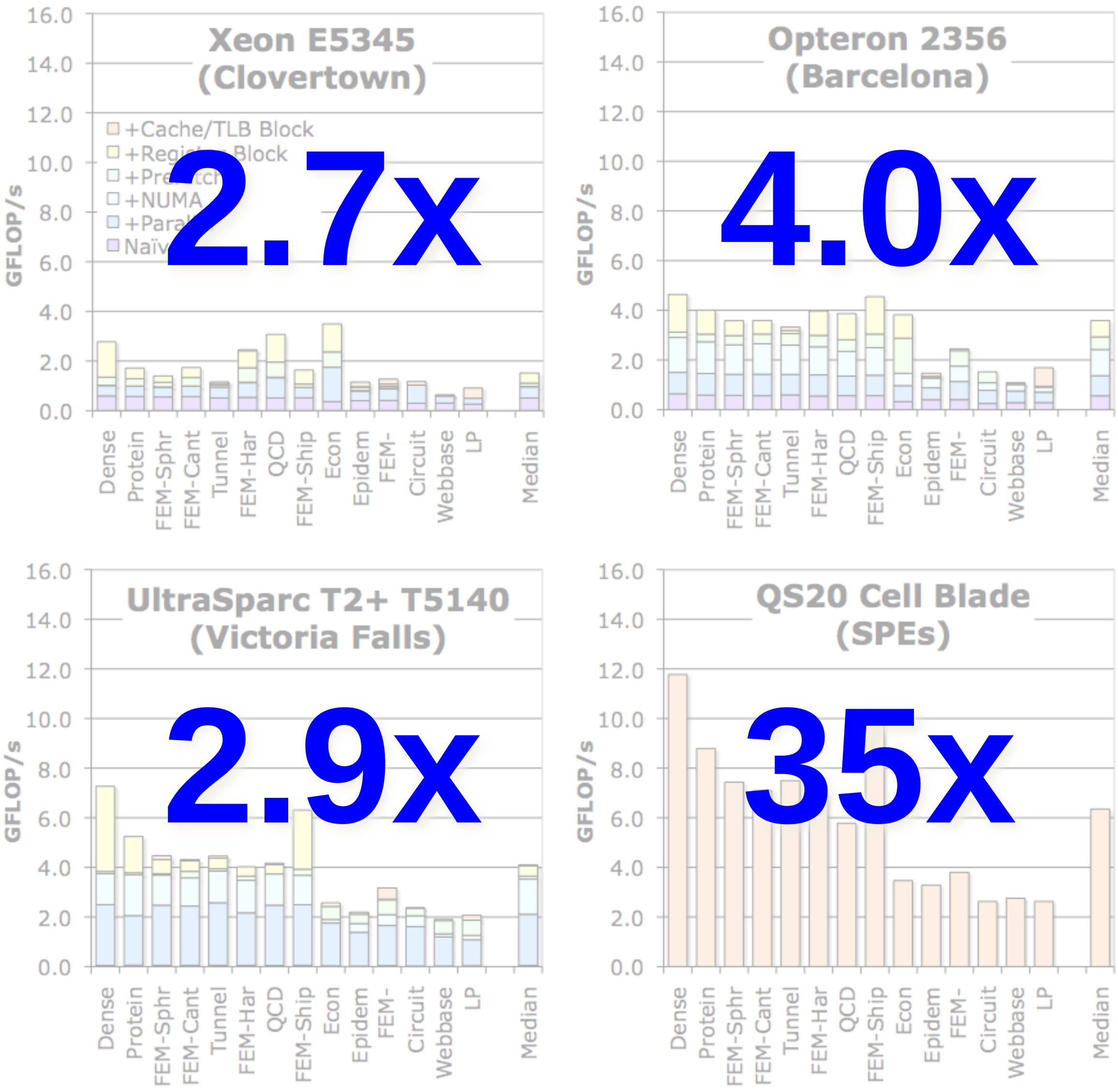
SpMV Performance: cache and TLB blocking



SpMV Performance: Architecture specific optimizations



SpMV Performance: max speedup



2.7x

4.0x

2.9x

35x

- 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

Optimized Sparse Kernel Interface - pOSKI

- Provides sparse kernels automatically tuned for matrix & machine
  - BLAS-style functionality: SpMV, Ax & A^Ty
  - Hides complexity of run-time tuning
- Faster than previous implementations
  - Up to 7.8x over reference serial implementation on Sandy Bridge E
  - Up to 4.5x over OSKI on Sandy Bridge E
  - Up to 2.1x over MKL on Nehalem
- bebop.cs.berkeley.edu/poski
- Ongoing work by the Berkeley Benchmarking and Optimization (BeBop) group

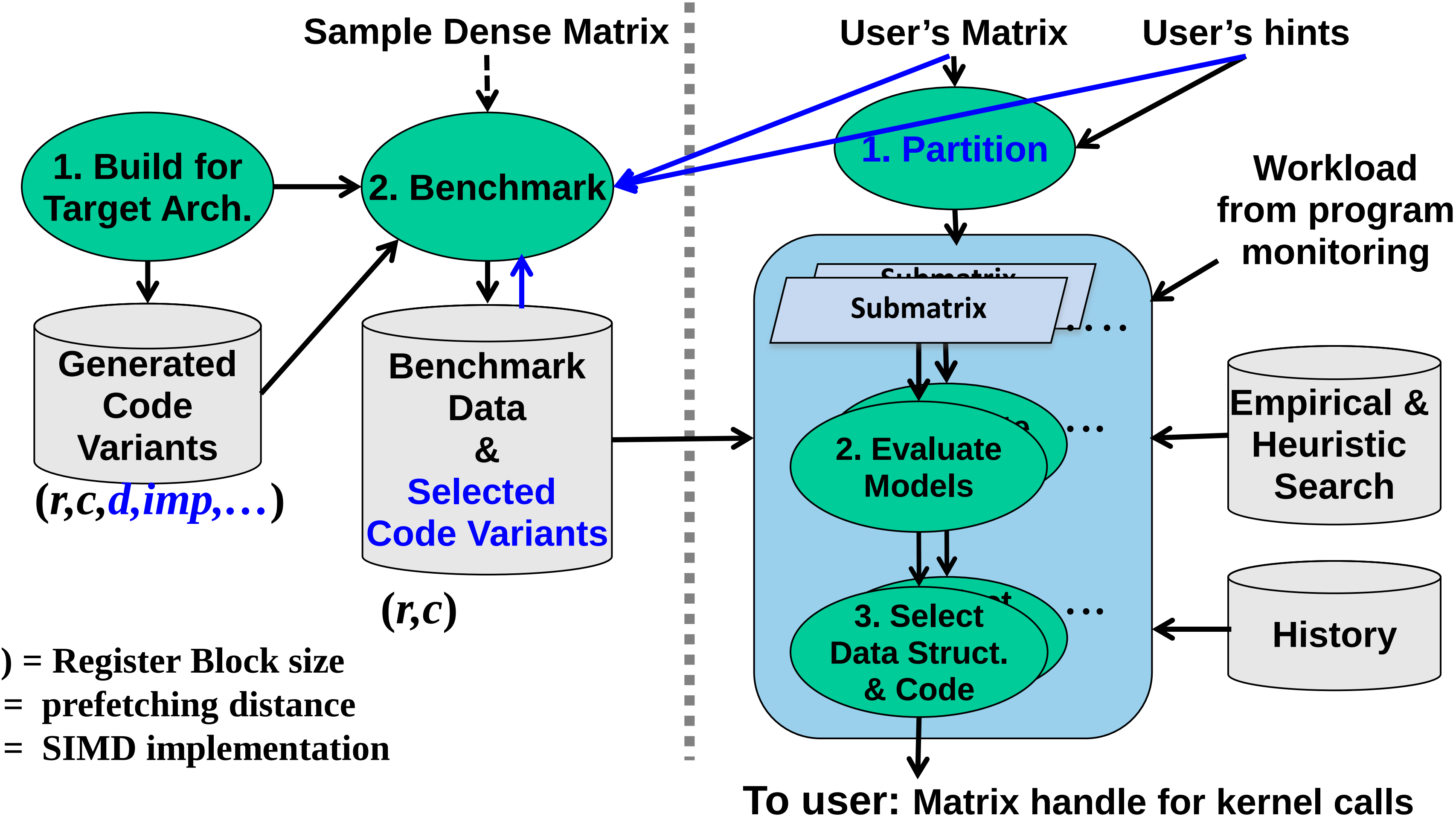
Optimizations in pOSKI, so far

- Fully automatic heuristics for
  - Sparse matrix-vector multiply (Ax, A^Tx)
    - 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
- Other kernels just in OSKI so far
  - A^TAx, A^kx
  - $A^{-1}x$: Sparse triangular solver (SpTS)
- Other optimizations under development
  - Cache-level blocking, Reordering (RCM, TSP), variable block structure, index compressing, Symmetric storage, etc.

How pOSKI Tunes (Overview)

Library Install-Time (offline)

Application Run-Time



Sample Dense Matrix

1. Build for Target Arch.

2. Benchmark

Generated Code Variants

(*r*,*c*,*d*,*imp*,…)

Benchmark Data & Selected Code Variants

(*r*,*c*)

User's Matrix

User's hints

1. Partition

Workload from program monitoring

Submatrix

…

2. Evaluate Models

…

3. Select Data Struct. & Code

…

Empirical & Heuristic Search

History

To user: Matrix handle for kernel calls

(*r*,*c*) = Register Block size

(*d*) = prefetching distance

(*d*) = SIMD implementation

How pOSKI Tunes (Overview)

- At library build/install-time
  - Generate code variants
    - Code generator (Phyton) generates code variants for various implementations
  - Collect benchmark data
    - Measures and records speed of possible sparse data structure and code variants on target architecture
  - Select best code variants & benchmark data
    - prefetching distance, SIMD implementation
  - Installation process uses standard, portable GNU AutoTools
- At run-time
  - Library "tunes" using heuristic models
    - Models analyze user's matrix & benchmark data to choose optimized data structure and code
    - User may re-collect benchmark data with user's sparse matrix (under development)
  - Non-trivial tuning cost: up to ~40 mat-vecs
    - Library limits the time it spends tuning based on estimated workload
      - provided by user or inferred by library
    - User may reduce cost by saving tuning results for application on future runs with same or similar matrix (under development)

How to call pOSKI: Basic Usage

- May gradually migrate existing apps
  - Step 1: “Wrap” existing data structures
  - Step 2: Make BLAS-like kernel calls

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
```

```
/* Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
   my_matmult( ptr, ind, val, α, x, β, y );
```

How to call pOSKI: Basic Usage

- May gradually migrate existing apps
  - Step 1: "Wrap" existing data structures
  - Step 2: Make BLAS-like kernel calls

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
/* Step 1: Create a default pOSKI thread object */
poski_threadarg_t *poski_thread = poski_InitThread();
/* Step 2: Create pOSKI wrappers around this data */
poski_mat_t A_tunable = poski_CreateMatCSR(ptr, ind, val, nrows, ncols,
   nnz, SHARE_INPUTMAT, poski_thread, NULL, …);
poski_vec_t x_view = poski_CreateVec(x, ncols, UNIT_STRIDE, NULL);
poski_vec_t y_view = poski_CreateVec(y, nrows, UNIT_STRIDE, NULL);

/* Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
   my_matmult( ptr, ind, val, α, x, β, y );
```

How to call pOSKI: Basic Usage

- May gradually migrate existing apps
  - Step 1: “Wrap” existing data structures
  - Step 2: Make BLAS-like kernel calls

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
/* Step 1: Create a default pOSKI thread object */
poski_threadarg_t *poski_thread = poski_InitThread();
/* Step 2: Create pOSKI wrappers around this data */
poski_mat_t A_tunable = poski_CreateMatCSR(ptr, ind, val, nrows, ncols,
   nnz, SHARE_INPUTMAT, poski_thread, NULL, …);
poski_vec_t x_view = poski_CreateVec(x, ncols, UNIT_STRIDE, NULL);
poski_vec_t y_view = poski_CreateVec(y, nrows, UNIT_STRIDE, NULL);

/* Step 3: Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
   poski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
```

How to call pOSKI: Tune with Explicit Hints

- User calls “tune” routine (optional)
  - May provide explicit tuning hints

```
poski_mat_t A_tunable = poski_CreateMatCSR( … );
   /* … */
/* Tell pOSKI we will call SpMV 500 times (workload hint) */
poski_TuneHint_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view,500);
/* Tell pOSKI we think the matrix has 8x8 blocks (structural hint) */
poski_TuneHint_Structure(A_tunable, HINT_SINGLE_BLOCKSIZE, 8, 8);

/* Ask pOSKI to tune */
poski_TuneMat(A_tunable);

for( i = 0; i < 500; i++ )
   poski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
```

How to call pOSKI: Implicit Tuning

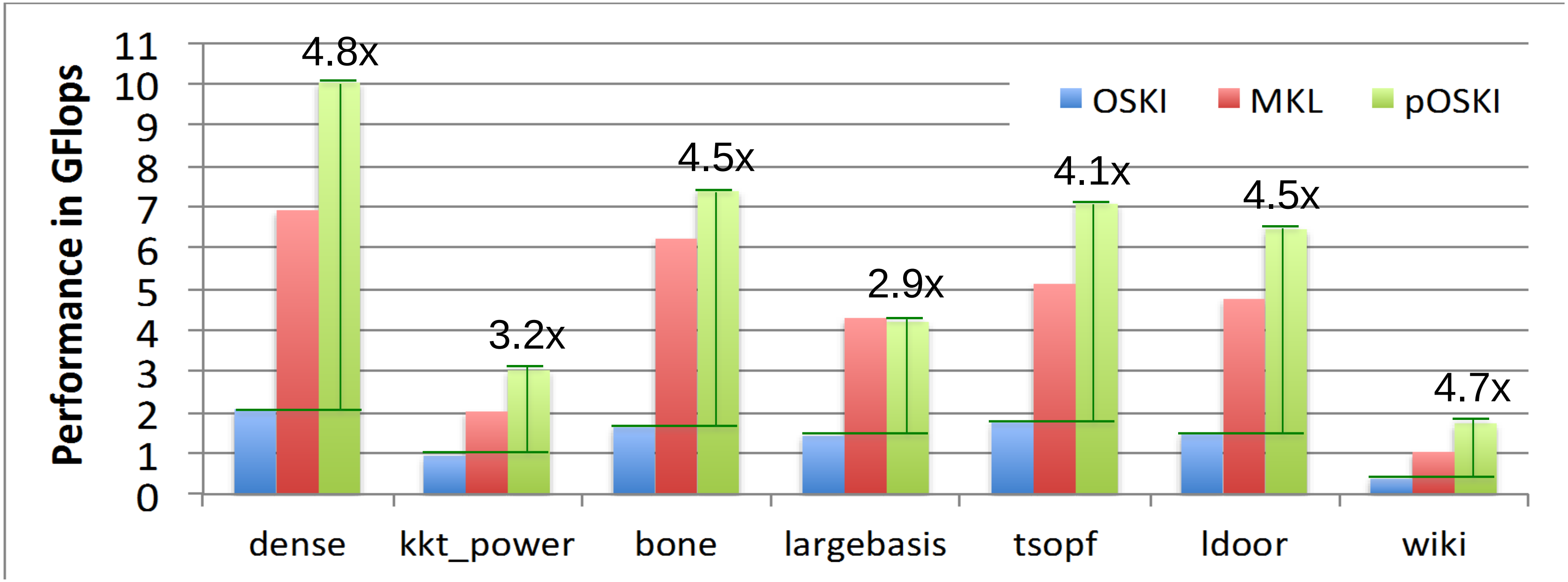
- Ask library to infer workload (optional)
  - Library profiles all kernel calls
  - May periodically re-tune

```
poski_mat_t A_tunable = poski_CreateMatCSR( … );
/* … */

for( i = 0; i < 500; i++ ) {
    poski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
    poski_TuneMat(A_tunable); /* Ask pOSKI to tune */
}
```

Performance on Intel Sandy Bridge E

- Jaketown: i7-3960X @ 3.3 GHz
- #Cores: 6 (2 threads per core), L3:15MB
- pOSKI SpMV (Ax) with double precision float-point
- MKL Sparse BLAS Level 2: *mkl_dcsrmv()*



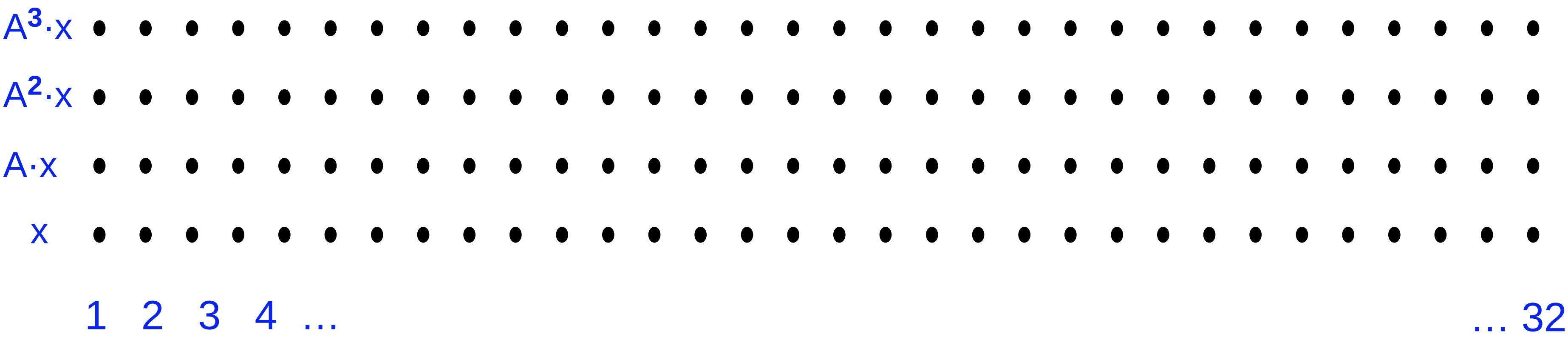
Avoiding Communication in Sparse Linear Algebra

- Computational intensity of one SpMV ≤ 2, limits performance
- k-steps of typical iterative solver for Ax=b or Ax=λx
  - Does k SpMVs with starting vector (eg with b, if solving Ax=b)
  - 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, numerical stability issues

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]

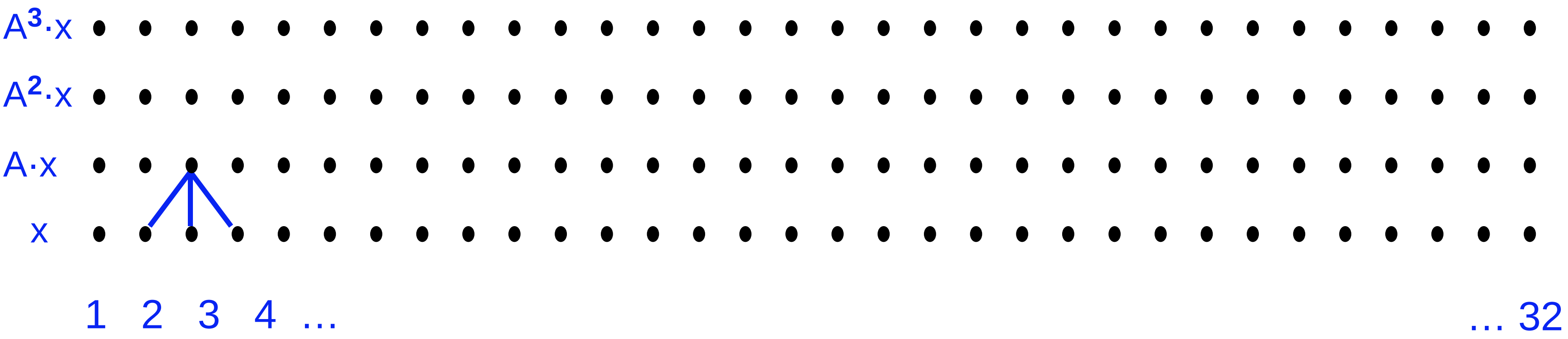


- Example: A tridiagonal, n=32, k=3
- Works for any "well-partitioned" A

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]

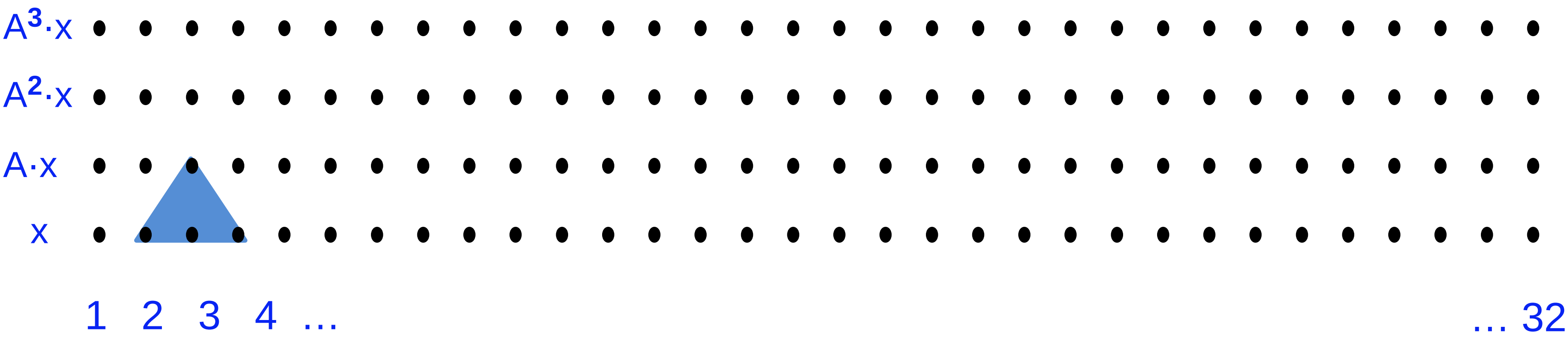


- Example: A tridiagonal, n=32, k=3

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]



A^3·x

A^2·x

A·x

x

1 2 3 4 …

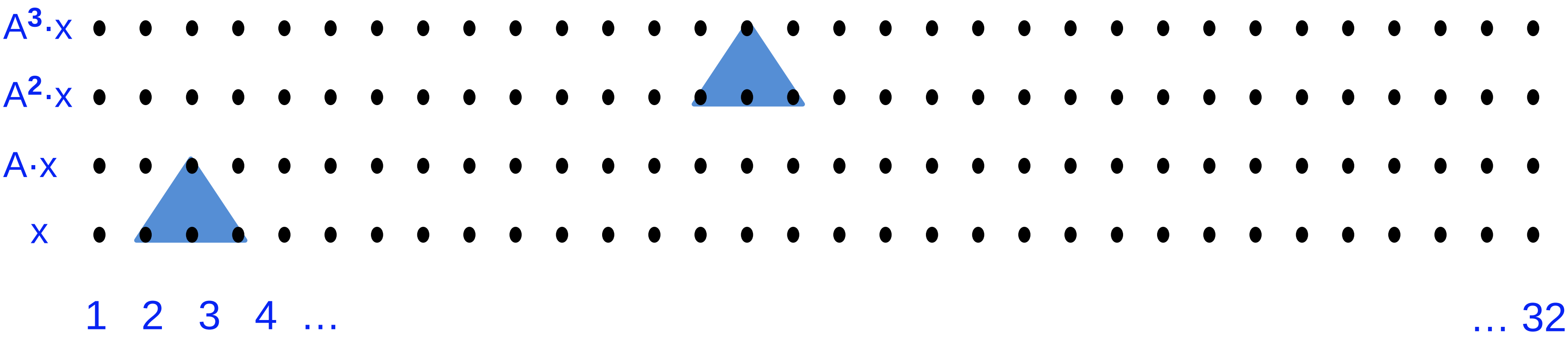
… 32

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

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$

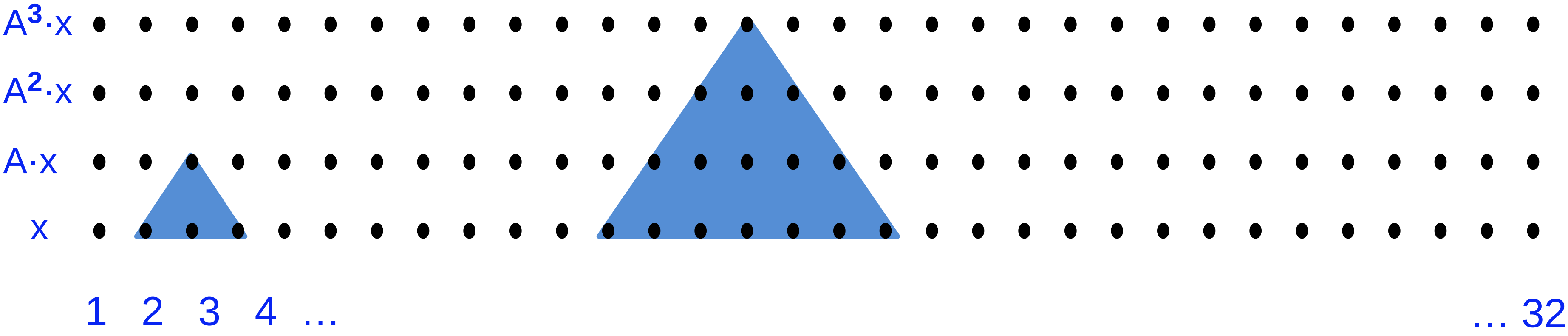


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$

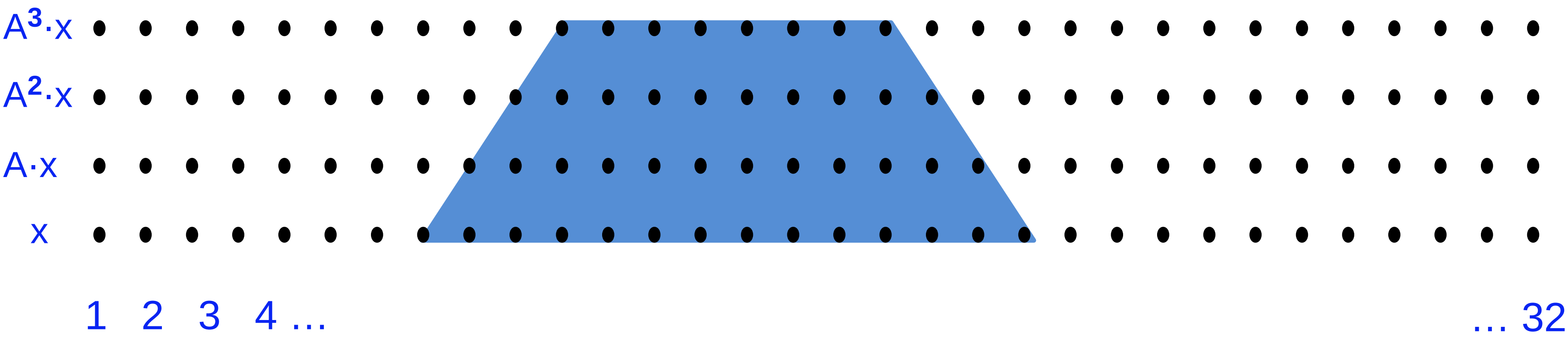


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

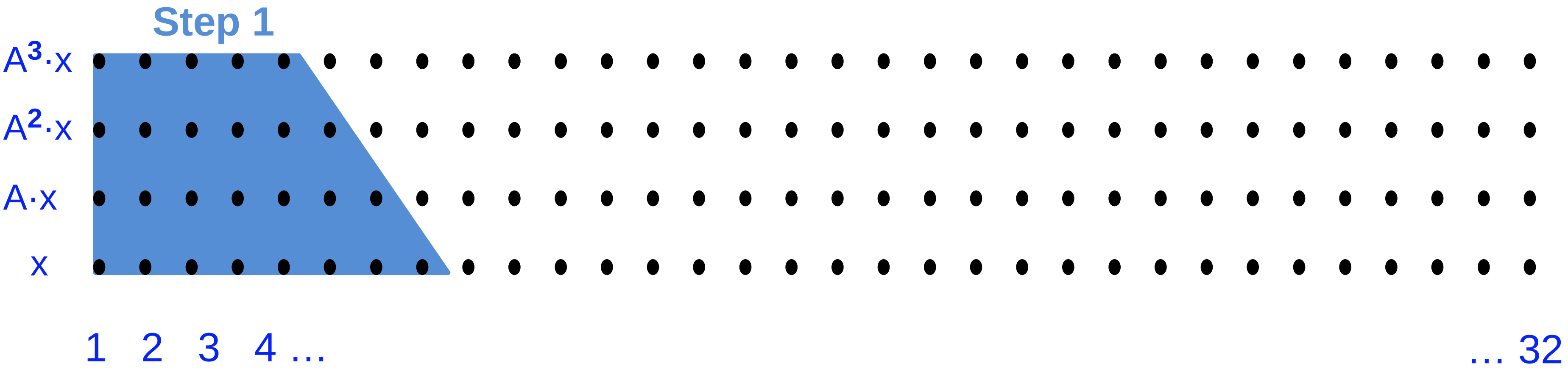
- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$



- 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

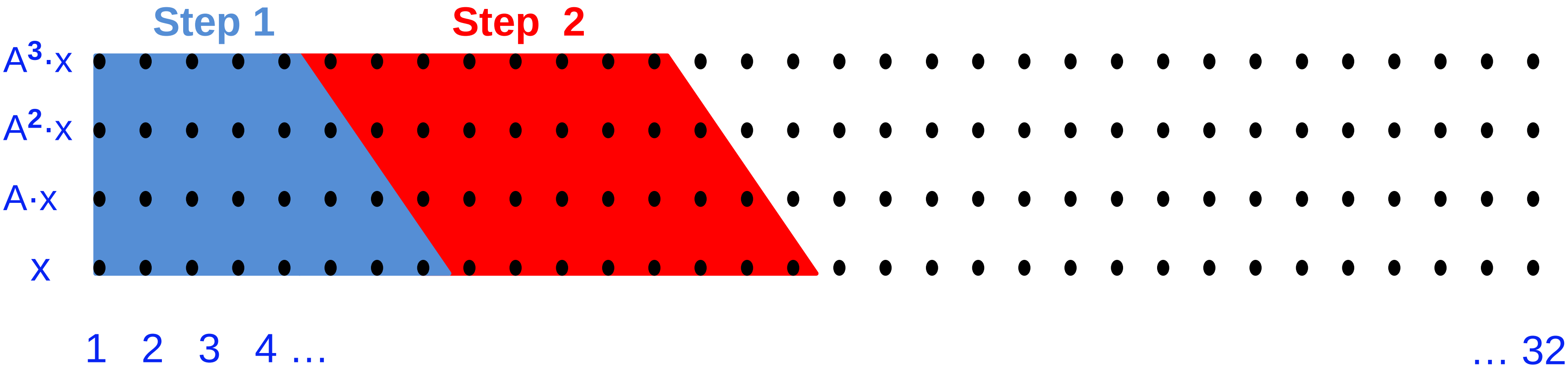


- Example: A tridiagonal, n=32, k=3

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

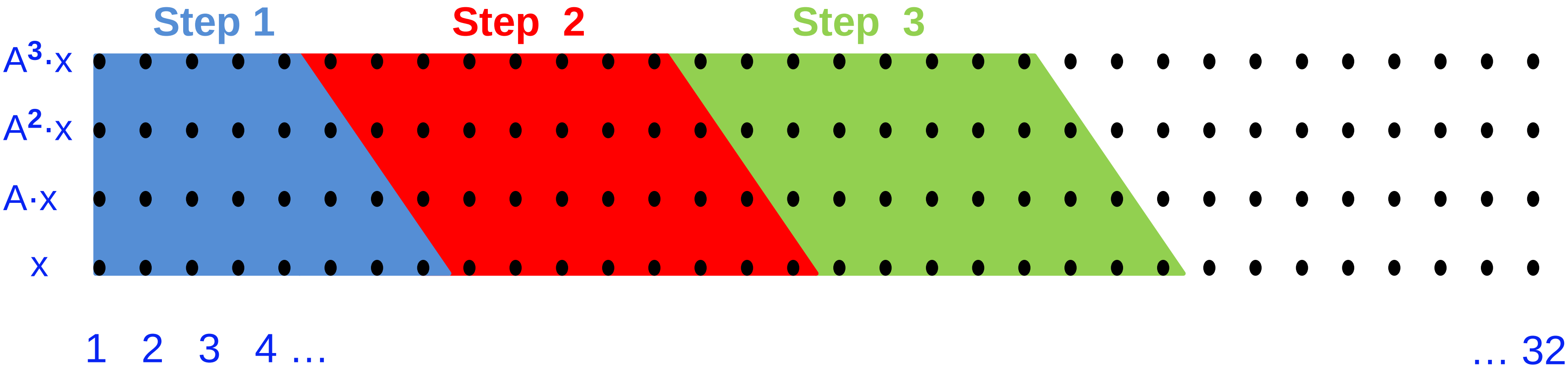


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : [Ax, A^{2}x, …, A^{k}x]

- Replace k iterations of y = A·x with [Ax, A^{2}x, …, A^{k}x]
- Sequential Algorithm

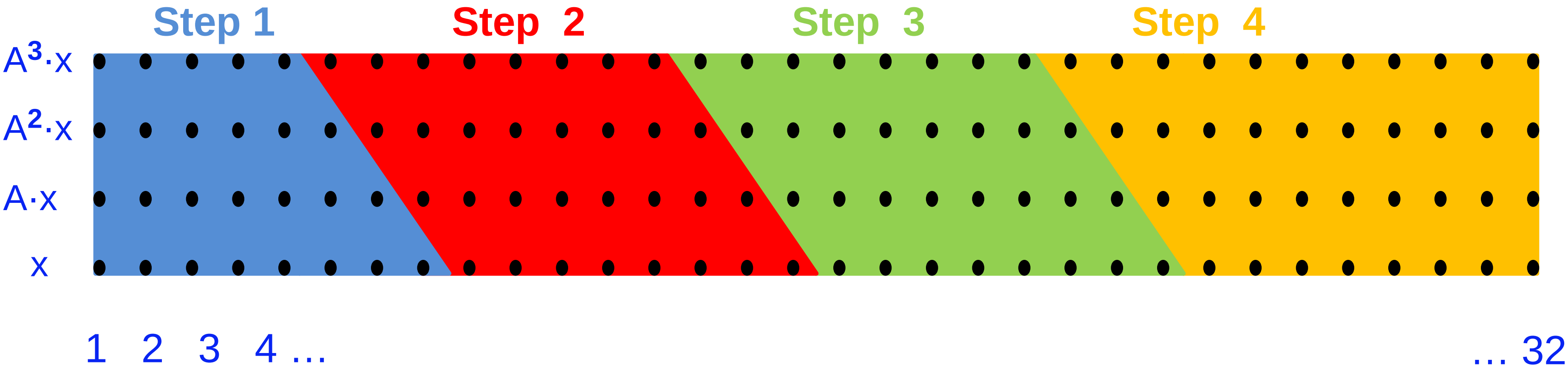


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Sequential Algorithm

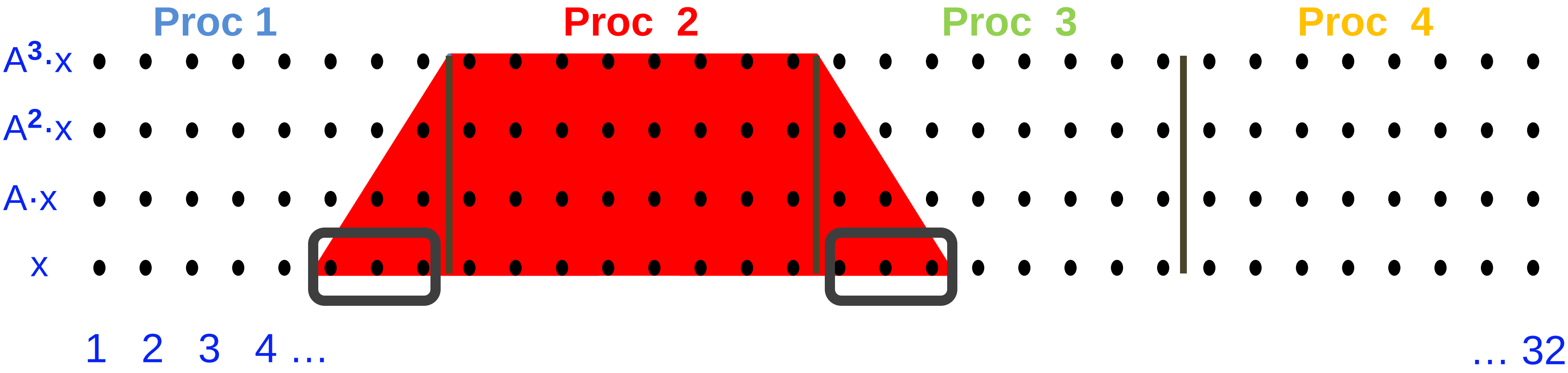


- Example: A tridiagonal, n=32, k=3

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

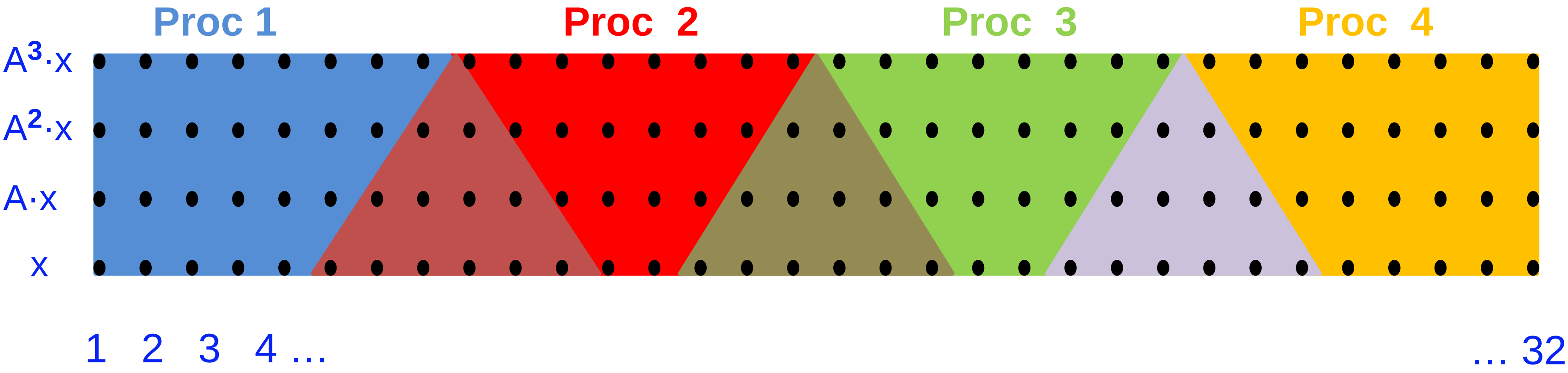


- Example: A tridiagonal, n=32, k=3
- Each processor communicates once with neighbors

Communication Avoiding Kernels:

The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Parallel Algorithm



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

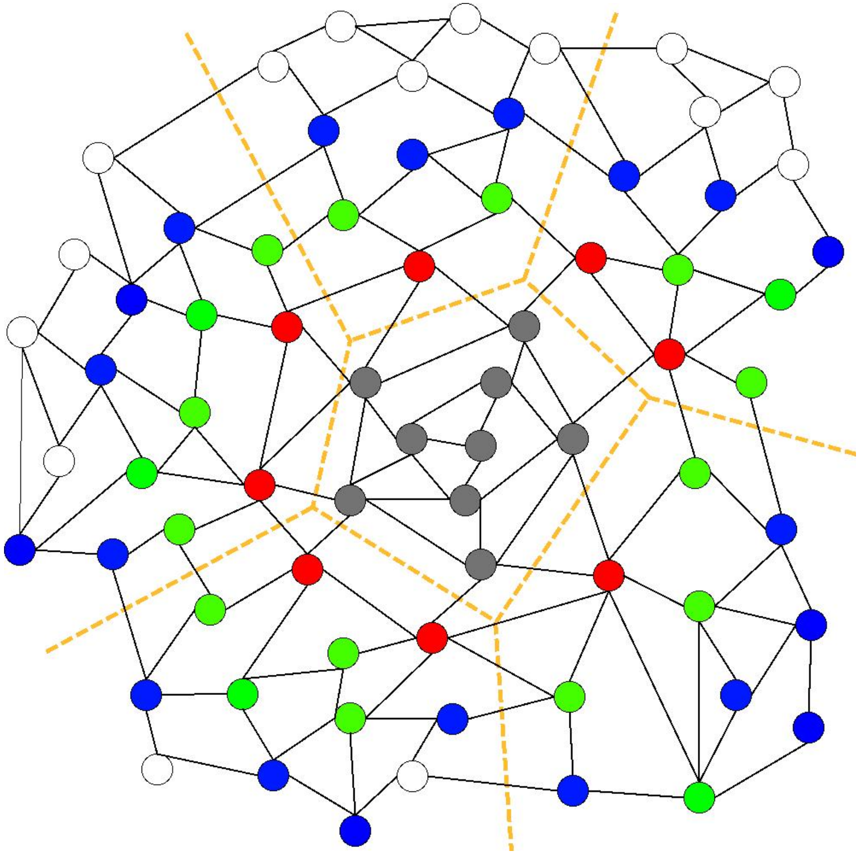
Communication Avoiding Kernels:

The Matrix Powers Kernel : [Ax, A^{2}x, …, A^{k}x]

Same idea works for general sparse matrices

Partitioning by rows ➔
 Graph partitioning

Processing left to right ➔
 Traveling Salesman Problem



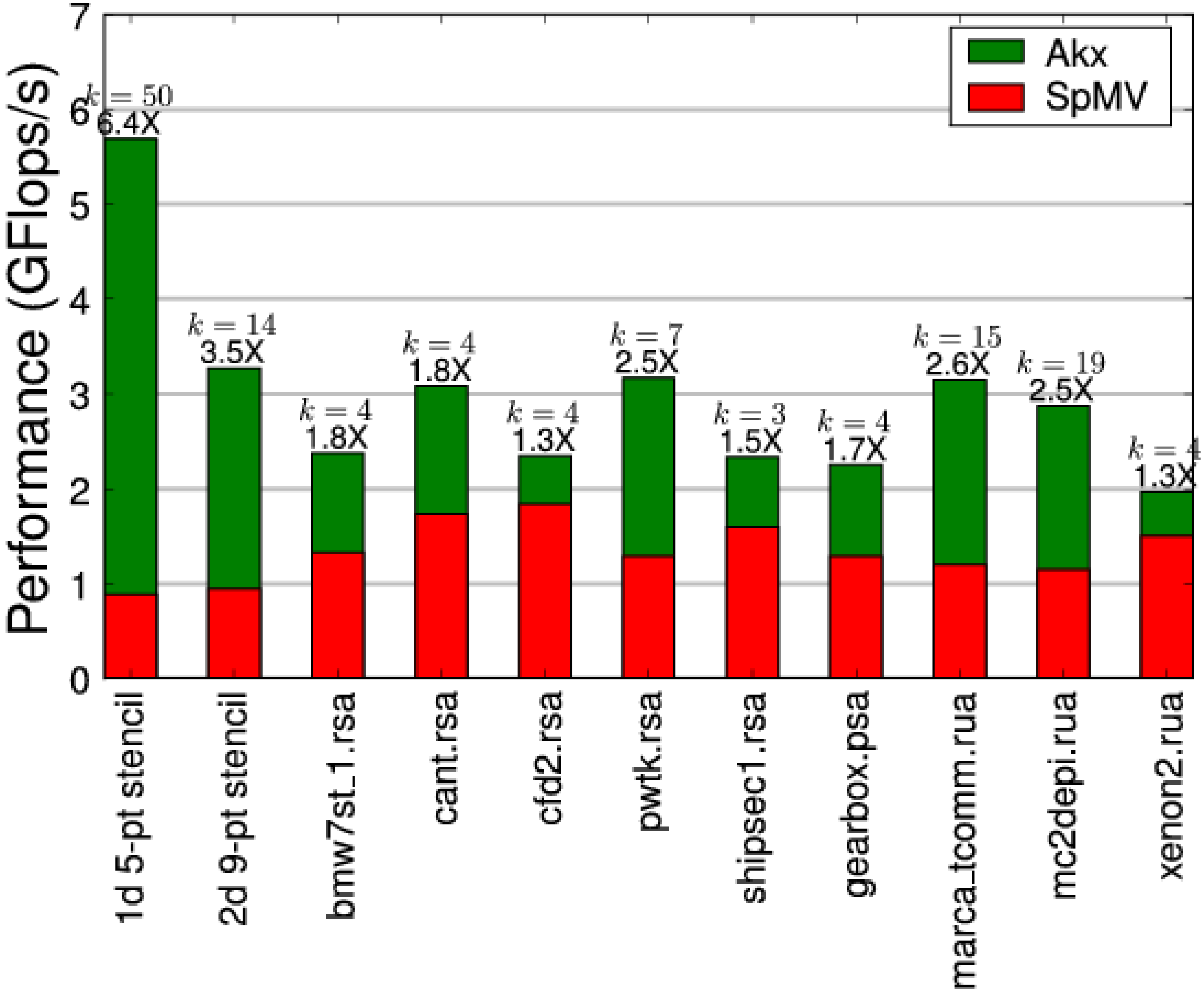
What about multicore?

- Two kinds of communication to minimize
  - Between processors on the chip
  - Between on-chip cache and off-chip DRAM
- Use hybrid of both techniques described so far
  - Use parallel optimization so each core can work independently
  - Use sequential optimization to minimize off-chip DRAM traffic of each core

Speedups on Intel Clovertown (8 core)

Test matrices include stencils and practical matrices

See SC09 paper on bebop.cs.berkeley.edu for details



Minimizing Communication of GMRES

Classical GMRES for Ax=b

```
for i=1 to k
   w = A * v(i-1)
   MGS(w, v(0),…,v(i-1))
       … Modified Gram-Schmidt
       …  to make w orthogonal
   update v(i), H
       … H = matrix of coeffs
       …    from MGS
endfor
solve LSQ problem with H for x
```

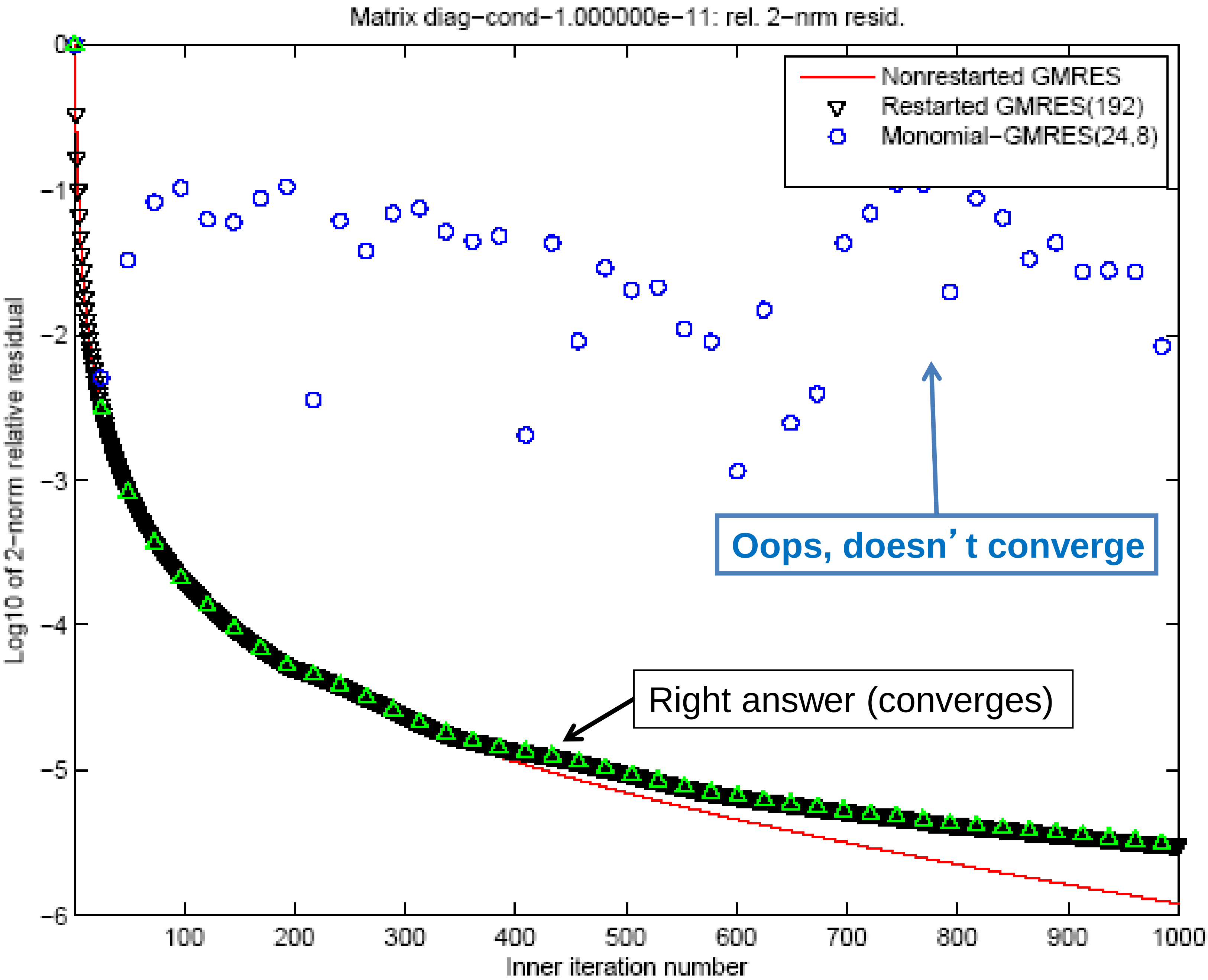
Communication cost =
k copies of A, vectors from
slow to fast memory

Communication-Avoiding GMRES, ver. 1

```
W = [ v, Av, $A^2v$, … , $A^kv$ ]
[Q,R] = TSQR(W)
     …  "Tall Skinny QR"
     … new optimal QR discussed before
Build H from R
solve LSQ problem with H for x
```

Communication cost =
O(1) copy of A, vectors from
slow to fast memory

Let's confirm that we still get the right answer …



Oops, doesn’t converge

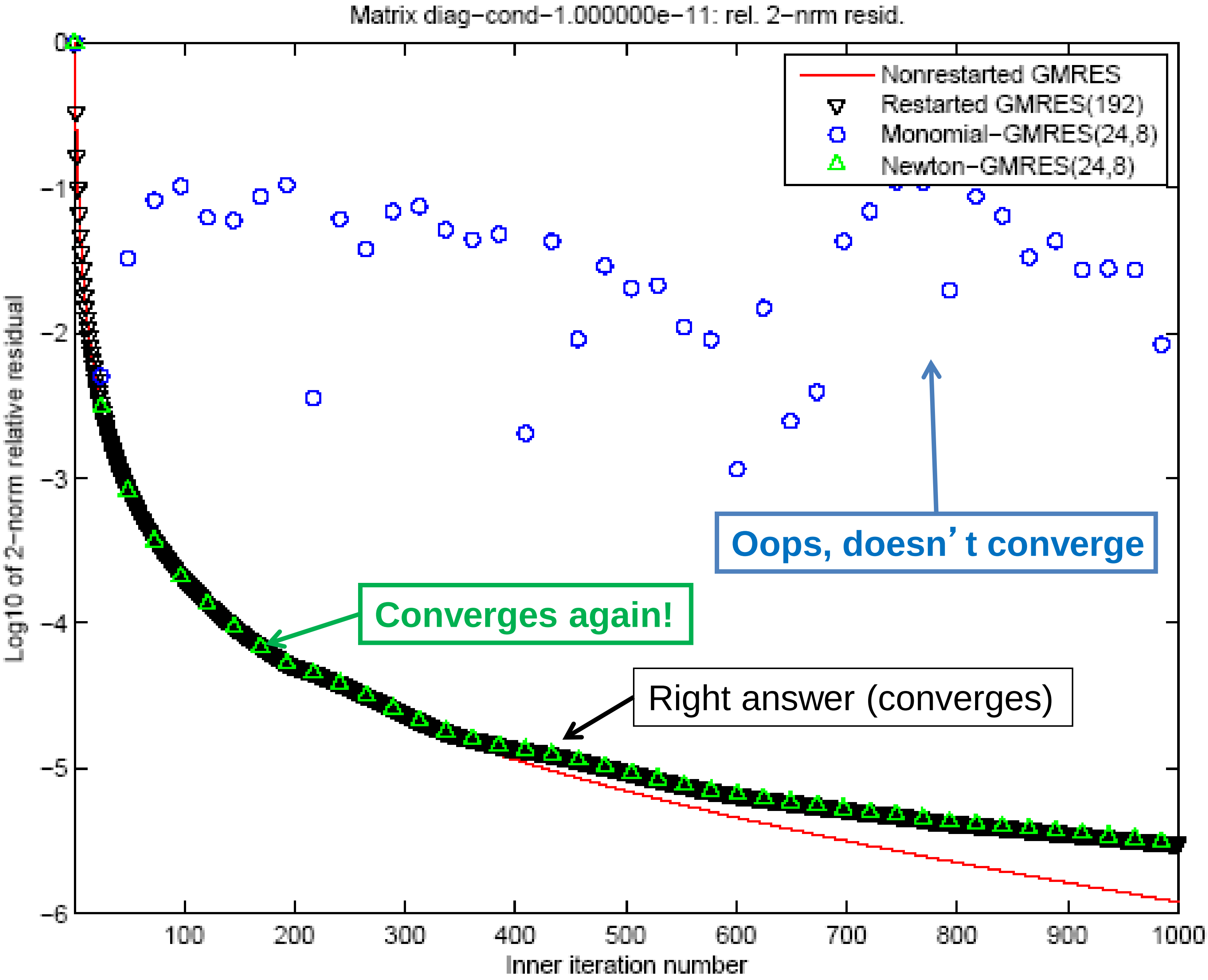
Right answer (converges)

Minimizing Communication of GMRES (and getting the right answer)

Communication-Avoiding GMRES, ver. 2

```
W = [ v, $p_1(A)v$, $p_2(A)v$, … , $p_k(A)v$ ]
    … where $p_i(A)v$ is a degree-i polynomial in A multiplied by v
    … polynomials chosen to keep vectors independent
[Q,R] = TSQR(W)
    … "Tall Skinny QR"
    … new optimal QR discussed before
Build H from R
    … slightly different R from before
solve LSQ problem with H for x
```

Communication cost still optimal:
O(1) copy of A, vectors from
slow to fast memory



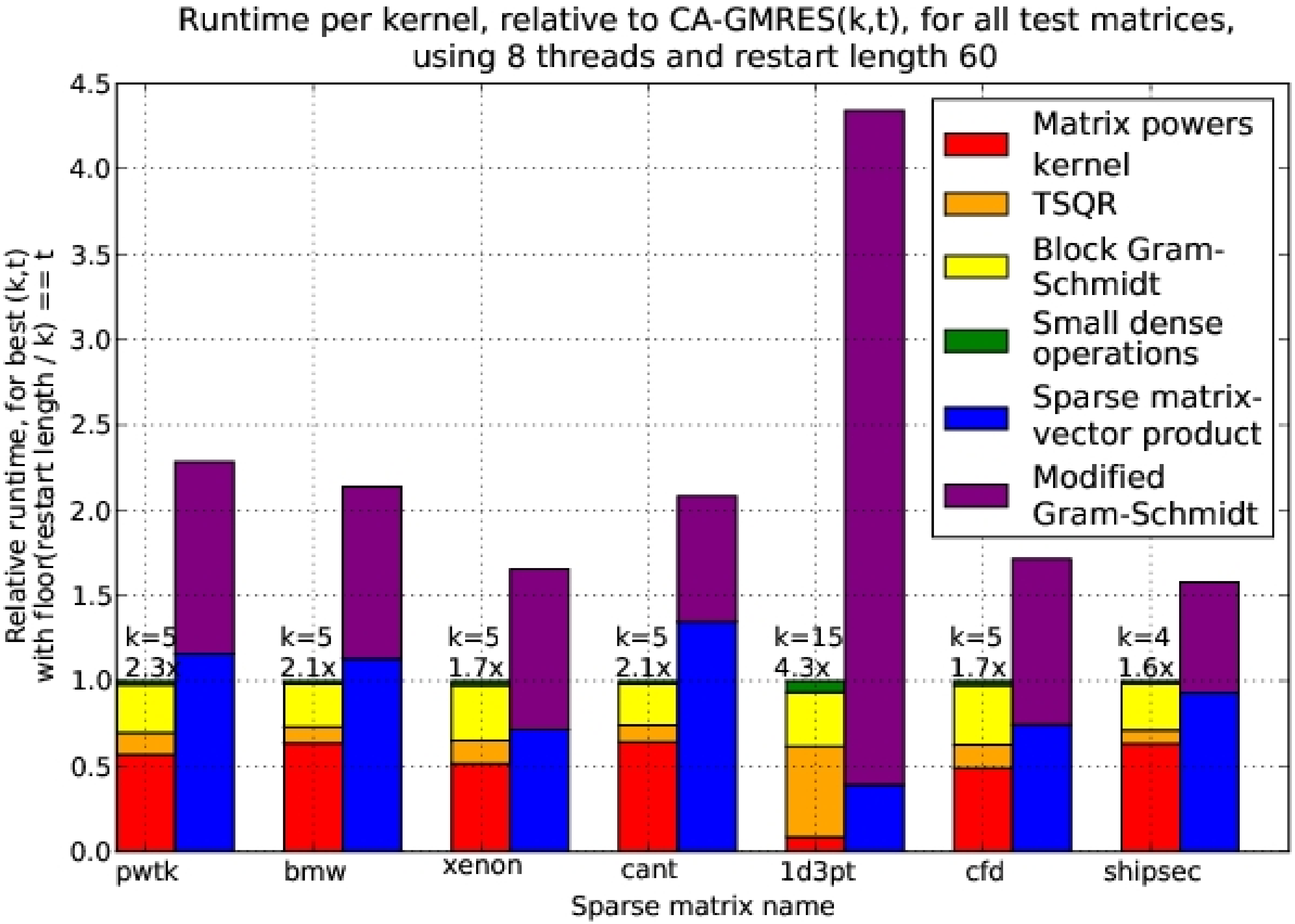
Oops, doesn’t converge

Converges again!

Right answer (converges)

Speed ups on 8-core Clovertown

CA-GMRES = Communication-Avoiding GMRES

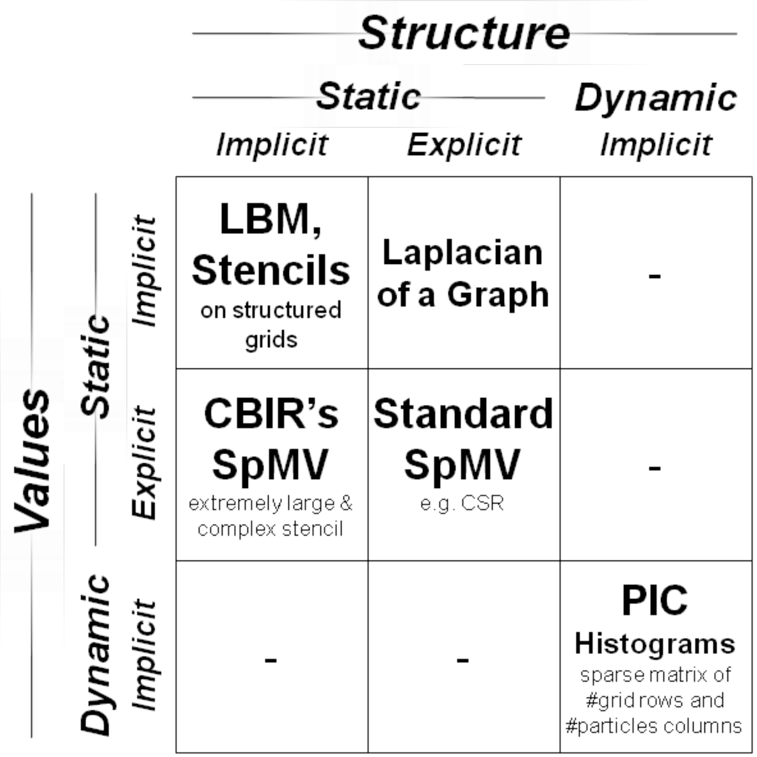


Paper by Mohiyuddin, Hoemmen, D. in Supercomputing09

Summary of what is known, open

- GMRES
  - Can independently choose k to optimize speed, restart length r to optimize convergence
  - **Need to "co-tune" Akx kernel and TSQR**
  - Know how to use more stable polynomial bases
  - Proven speedups
- Can similarly reorganize other Krylov methods
  - Arnoldi and Lanczos, for Ax = λx and for Ax = λMx
  - Conjugate Gradients (CG), for Ax = b
  - Biconjugate Gradients (BiCG), CG Squared (CGS), BiCGStab for Ax=b
  - Other Krylov methods?
- Preconditioning – how to handle MAx = Mb

What is a sparse matrix?



			Structure		
			Static		Dynamic
			Implicit	Explicit	Implicit
Values	Static	Implicit	LBM, Stencils on structured grids	Laplacian of a Graph	-
		Explicit	CBIR's SpMV extremely large & complex stencil	Standard SpMV e.g. CSR	-
	Dynamic	Implicit	-	-	PIC Histograms sparse matrix of #grid rows and #particles columns

- How much infrastructure (for code creation, tuning or interfaces) can we reuse for all these cases?

Sparse Conclusions

- Fast code must minimize communication
  - Especially for sparse matrix computations because communication dominates
- Generating fast code for a single SpMV
  - Design space of possible algorithms must be searched at run-time, when sparse matrix available
  - Design space should be searched automatically
- Biggest speedups from minimizing communication in an entire sparse solver
  - Many more opportunities to minimize communication in multiple SpMVs than in one
  - Requires transforming entire algorithm
  - Lots of open problems
- For more information, see bebop.cs.berkeley.edu

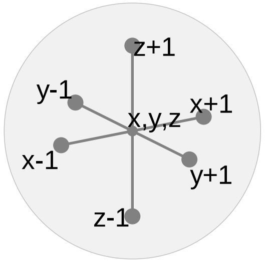
STRUCTURED GRID MOTIF

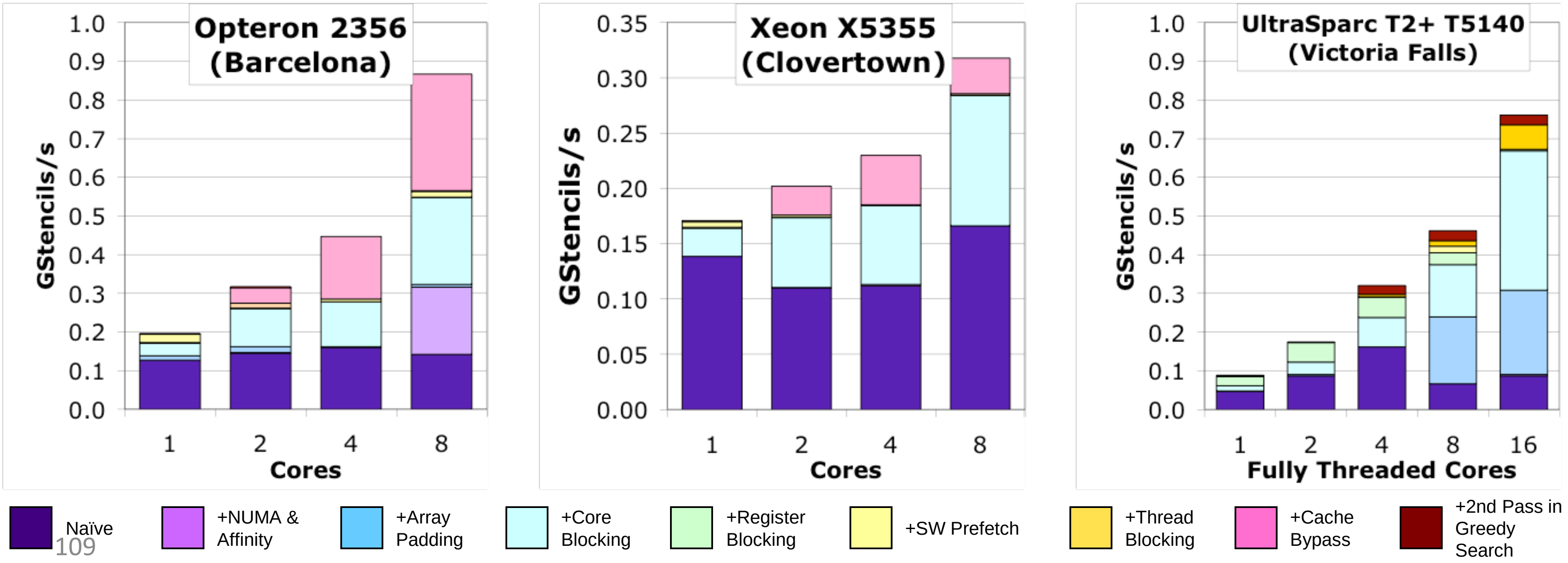
Source: Sam Williams

Structured Grids

Finite Difference Operators

- Applying the finite difference method to PDEs on structured grids produces **stencil operators** that must be applied to all points in the discretized grid.
- Consider the 7-point Laplacian Operator
- Challenged by bandwidth, temporal reuse, efficient SIMD, etc… but trivial to (correctly) parallelize
- **most optimizations can be independently implemented,** **(but not performance independent)**
- core (cache) blocking and cache bypass were clearly integral to performance

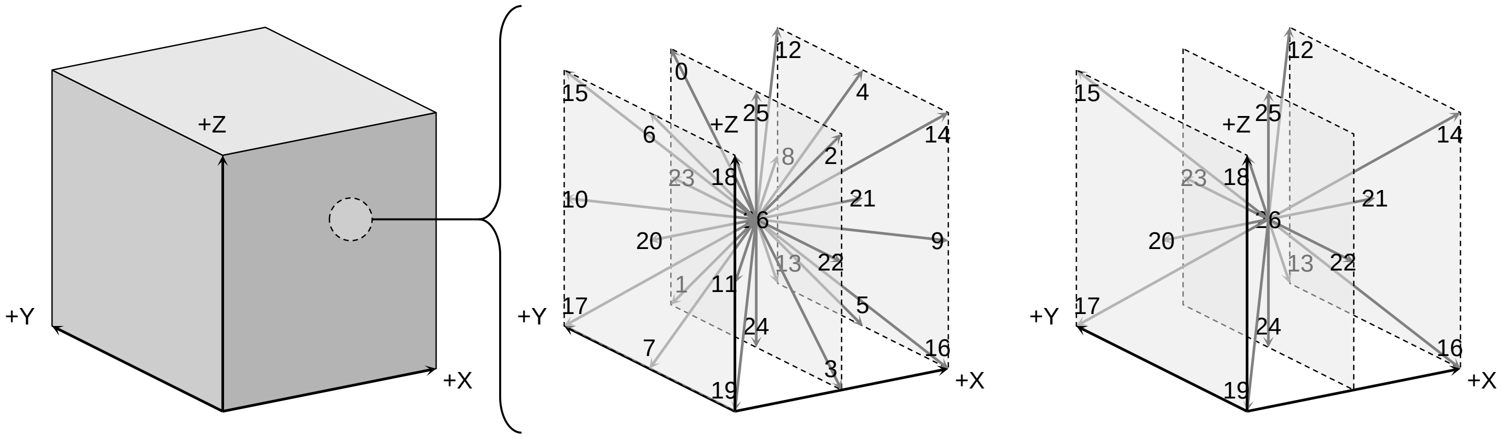




Structured Grids

Lattice Boltzmann Methods

- LBMHD simulates charged plasmas in a magnetic field (MHD) via Lattice Boltzmann Method (LBM) applied to CFD and Maxwell’s equations.
- To monitor density, momentum, and magnetic field, it requires maintaining two “velocity” distributions
  - 27 (scalar) element velocity distribution for momentum
  - 15 (Cartesian) element velocity distribution for magnetic field
  - = 632 bytes / grid point / time step
- Jacobi-like time evolution requires ~1300 flops and ~1200 bytes of memory traffic



macroscopic variables

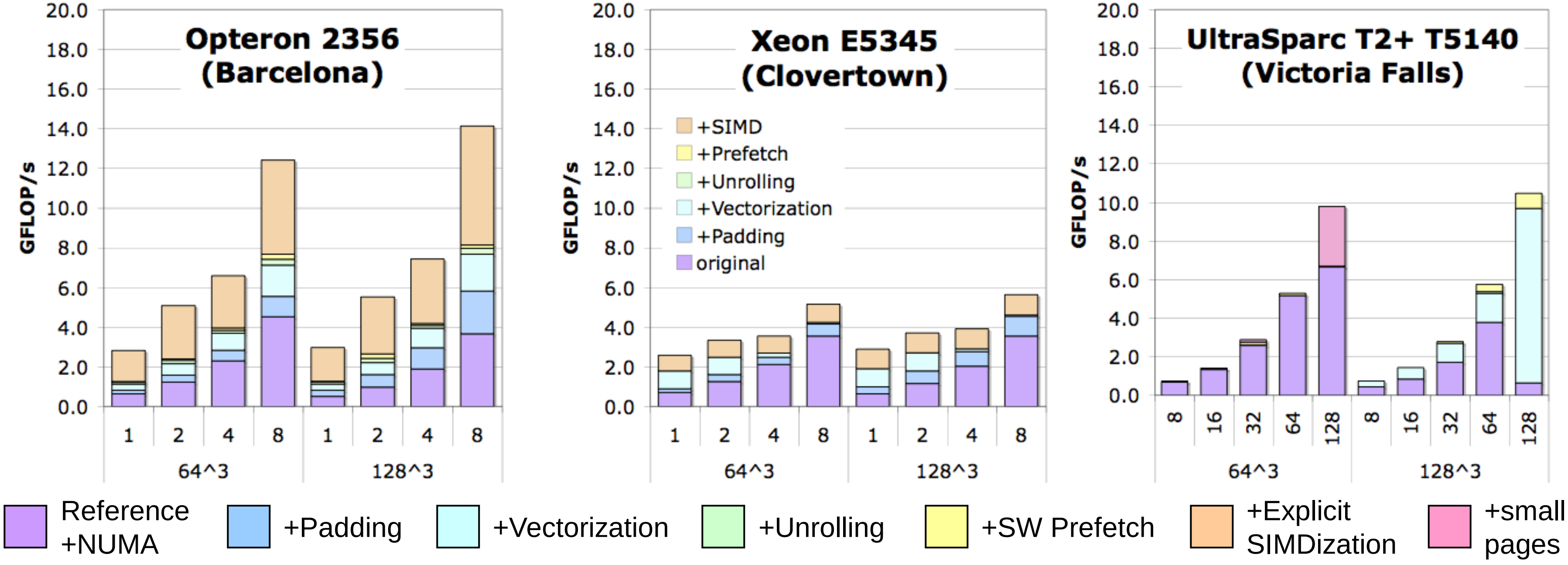
momentum distribution

magnetic distribution

Structured Grids

Lattice Boltzmann Methods

- Challenged by:
  - The higher flop:byte ratio of ~1.0 is still bandwidth-limiting
  - TLB locality (touch 150 pages per lattice update)
  - cache associativity (150 disjoint lines)
  - efficient SIMDization
- easy to (correctly) parallelize
- **explicit SIMDization & SW prefetch are dependent on unrolling**
- Ultimately, 2 of 3 machines are bandwidth-limited

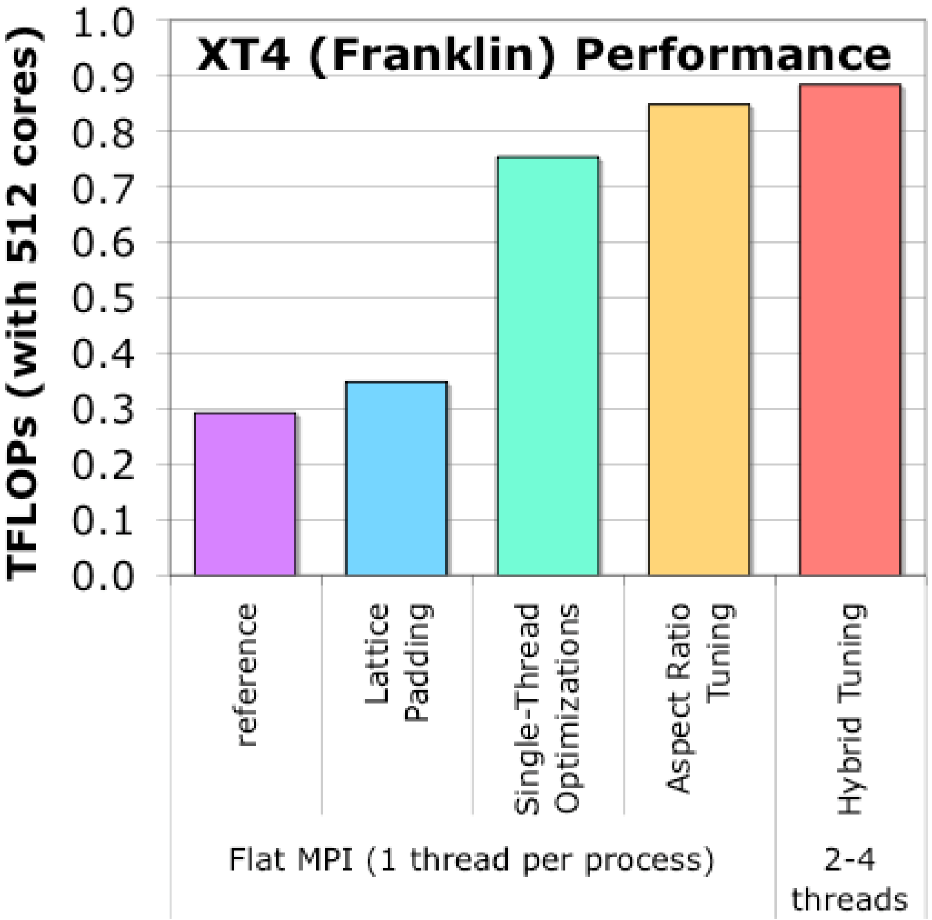


*collision() only

Structured Grids

Lattice Boltzmann Methods

- **Distributed Memory & Hybrid**
- MPI, MPI+pthreads, MPI+OpenMP (SPMD, SPMD^2, SPMD+Fork/Join)
- Observe that for this large problem, **auto-tuning flat MPI delivered significant boosts (2.5x)**
- Extending auto-tuning to include the domain decomposition and balance between threads and processes **provided an extra 17%**
- 2 processes with 2 threads was best (true for Pthreads and OpenMP)



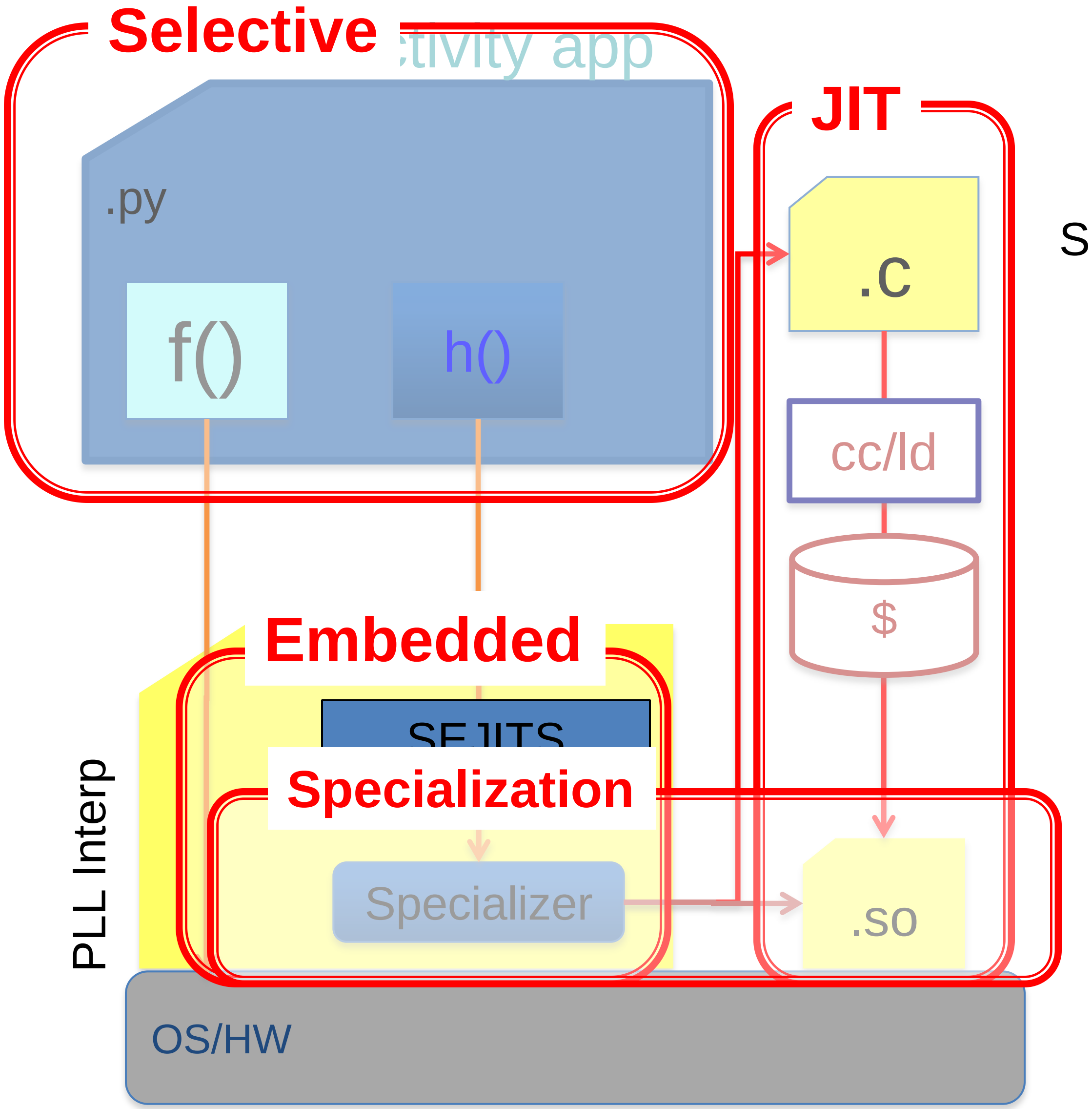
DELIVERING AUTOTUNING WITH SEJITS

Source: Shoaib Kamil

What is SEJITS?

- Goal: Let non-expert programmers quickly write their algorithms in an easy-to-use language, but still get high performance
  - First example: Python
- By using common “patterns” to write algorithms, and hints about tuning opportunities, enable system to autotune
- SEJITS = Selective Embedded Just-in-time Specialization

Delivering Autotuning via SEJITS



Several examples exist now:

- Structured Grids/Stencils
- CA-Conjugate Gradient
- Tuned SpMV over other semirings

Summary

- “Design spaces” for algorithms and implementations are large and growing
- Finding the best algorithm/implementation by hand is hard and getting harder
- Ideally, we would have a database of “techniques” that would grow over time, and be searched automatically whenever a new input and/or machine comes along
- Lots of work to do…