PARLab Parallel Boot Camp

Parallel Computing Motifs

(aka Computational Patterns)

Jim Demmel

EECS and Mathematics

University of California, Berkeley

Outline

- Productive parallel computing depends on recognizing and exploiting known useful patterns
  - Design, computational, and mathematical
- Recall the 7 (then 13) Motifs
- Optimizing (some of) the 7 motifs
  - To minimize time, minimize communication, not necessarily flops
    - Time = #flops * time_per_flop + #words_sent /BW + #messages * latency
    - Time_per_flop << 1/BW << latency, growing apart exponentially
  - Best parallel algorithm not always from parallelizing the best sequential algorithm
    - Want #messages << #word_sent << #flops
  - Autotuning to explore large design spaces
- See CS267 for details
  - www.cs.berkeley.edu/~demmel/cs267_Spr09

"7 Dwarfs" 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 = λx where A is a dense matrix
2. Sparse Linear Algebra
   - Ex: Solve Ax=b or Ax = λx 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

6 additional motifs of parallel computing

- By examining a variety of applications, ParLab identified 6 more common underlying computational patterns:

1. Combinational Logic
   - Ex: Circuits(!), encryption
2. Finite State Machine
   - Ex: Compiling, discrete event simulation
3. Graph Traversal
   - Ex: Natural language processing, sorting
4. Dynamic Programming
   - Ex: Data base query optimization, machine learning
5. Backtrack and Branch & Bound
   - Ex: Constraint satisfactions, games (like chess)
6. Graphical Models
   - Ex: Hidden Markov Models, Bayesian Networks

- For more examples and analysis, see "Berkeley View" report:

  www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

Programming Pattern Language 1.0 Keutzer& Mattson

Applications

Choose your high level structure – what is the structure of my application? Guided expansion

Pipe-and-filter
Agent and Repository
Process Control
Event based, implicit invocation

Model-view controller
Iterator
Map reduce
Layered systems
Arbitrary Static Task Graph

Choose your high level architecture - Guided decomposition

Task Decomposition ↔ Data Decomposition

Group Tasks Order groups data sharing data access

Identify the key computational patterns – what are my key computations? Guided instantiation

Graph Algorithms
Dynamic Programming
Dense Linear Algebra
Sparse Linear Algebra
Unstructured Grids
Structured Grids

Graphical models
Finite state machines
Backtrack Branch and Bound
N-Body methods
Circuits
Spectral Methods

Productivity Layer

Refine the structure - what concurrent approach do I use? Guided re-organization

Event Based
Divide and Conquer

Data Parallelism
Geometric Decomposition

Pipeline
Discrete Event

Task Parallelism
Graph algorithms

Digital Circuits

Efficiency Layer

Utilize Supporting Structures – how do I implement my concurrency? Guided mapping

Fork/Join
CSP

Distributed Array Shared-Data

Shared Queue
Shared Hash Table

Master/worker
Loop Parallelism
BSP

Implementation methods – what are the building blocks of parallel programming? Guided implementation

Thread Creation/destruction
Process/Creation/destruction

Message passing
Collective communication

Speculation
Transactional memory

Barriers
Mutex

Semaphores

What you (might) want to know about a motif

- How to use it
  - What problems does it solve
  - How to choose solution approach, if more than one
- How to find the best software available now
  - Best: fastest? most accurate? fewest keystrokes?
- How are the best implementations built?
  - Algorithms
  - Autotuning
- Open problems, current work, thesis problems…

Algorithms for 2D (3D) Poisson Equation ($N = n^2$ (n^3) vars)

Algorithm	Serial	PRAM	Memory	#Procs
• Dense LU	N^3	N	N^2	N^2
• Band LU	N^2 ($N^{7/3}$)	N	$N^{3/2}$ ($N^{5/3}$)	N ($N^{4/3}$)
• Jacobi	N^2 ($N^{5/3}$)	N ($N^{2/3}$)	N	N
• Explicit Inv.	N^2	log N	N^2	N^2
• Conj.Gradients	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$ (1/3) *log N	N	N
• Red/Black SOR	$N^{3/2}$ ($N^{4/3}$)	$N^{1/2}$ ($N^{1/3}$)	N	N
• Sparse LU	$N^{3/2}$ (N^2)	$N^{1/2}$	N*log N ($N^{4/3}$)	N
• FFT	N*log N	log N	N	N
• Multigrid	N	$\log^2 N$	N	N
• Lower bound	N	log N	N	

PRAM is an idealized parallel model with ∞ procs, zero cost communication

Reference: James Demmel, Applied Numerical Linear Algebra, SIAM, 1997.

For more details, Ma221 offered this semester!

Need Parallelism and Locality for Performance

- Moving data most expensive operation
  - Between cache and DRAM, or processors on a network…
- Goal: algorithms that minimize data movement, as well as parallelize
- Recall simple cost model
  - Time = #flops · γ + #words_moved · β + #messages · α
  - γ = time_per_flop, β = 1/bandwidth , α = latency
  - α » β » γ, gaps growing exponentially over time
- If #words_moved ≈ # flops in an algorithm, it will be limited by bandwidth
  - Seek algorithms where #flops » #words_moved » #messages
  - Notation: q ≡ *computational intensity* = #flops/#words_moved

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 (mostly) on vectors
    - Ex: $y = \alpha \cdot x + y$ ("AXPY") , dot product, $(\Sigma_i x_i^2)^{1/2}$, $x = \alpha \cdot x$, etc
  - Goals
    - Common pattern to ease programming, readability
    - Robustness, via careful coding (avoiding over/underflow)
    - Portability + efficiency via machine specific implementations
  - Used in libraries like LINPACK (for linear systems)
    - Source of the name "LINPACK Benchmark" (not the code!)
  - Why BLAS 1 ? They do $O(n^1)$ ops on $O(n^1)$ data
    - Ex: Computational intensity = q = 2/3 for AXPY
    - Limited by memory speed

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

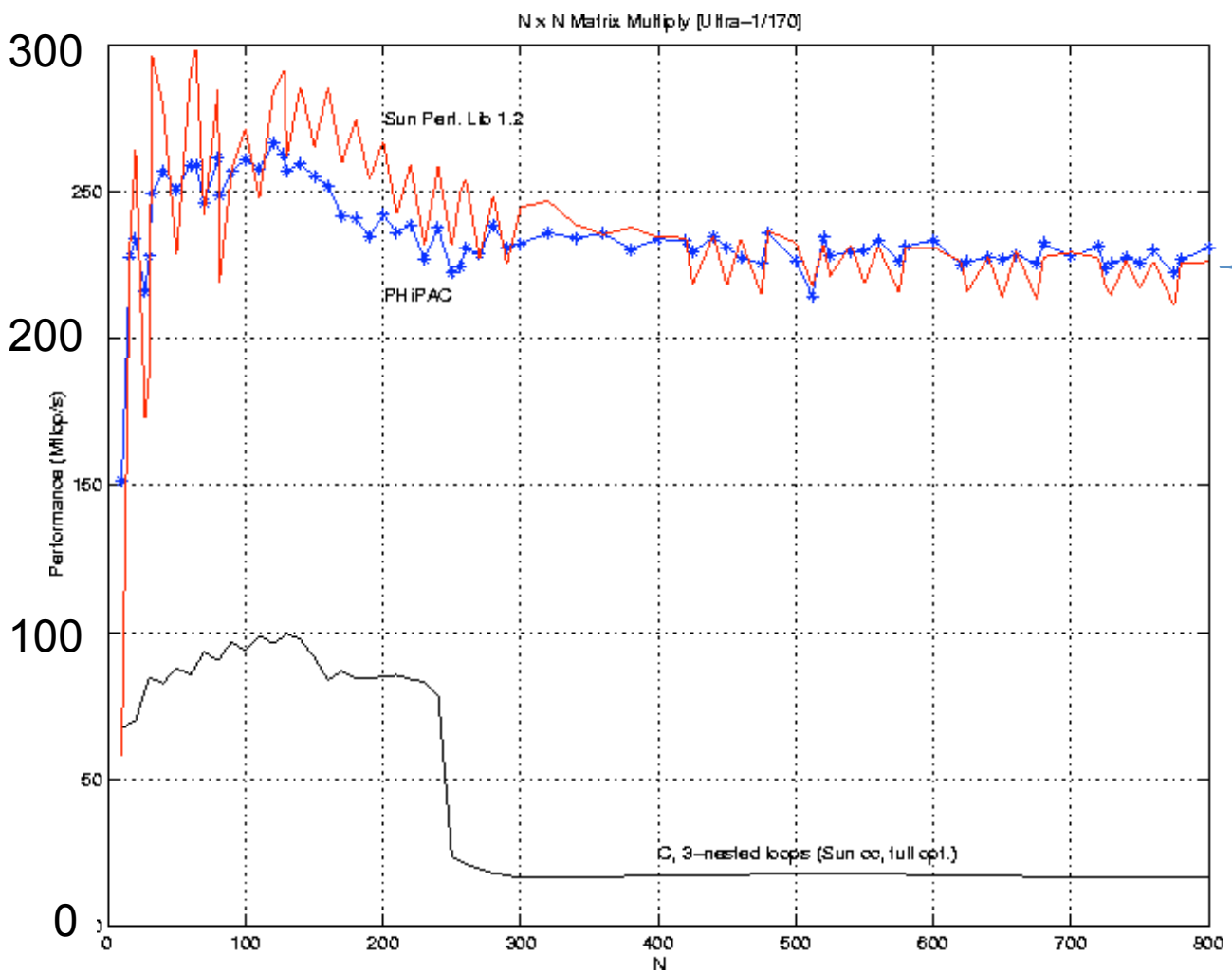
- So the BLAS-1 weren't fast enough
  - Computational intensity too low, so data movement was bottleneck
- So the BLAS-2 were invented (1984-1986)
  - Standard library of 25 operations (mostly) on matrix/vector pairs
    - Ex: $y = \alpha \cdot A \cdot x + \beta \cdot y$ ("GEMV"), $A = A + \alpha \cdot x \cdot y^T$ ("GER"), $y = T^{-1} \cdot x$ ("TRSV")
  - Why BLAS 2 ? They do $O(n^2)$ ops on $O(n^2)$ data
  - So q=computational intensity still just $\sim(2n^2)/(n^2) = 2$
    - 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 ? They 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 BLAS1/2/3 code so far?
  - See www.netlib.org/blas;
  - Reference implementation only, not optimized
    - Ex: Just usual 3 nested loops for GEMM
  - Source: 142 routines, 31K LOC, Testing: 28K LOC
  - How fast is the reference implementation?

Matrix-multiply, optimized several ways

Peak = 330 MFlops.



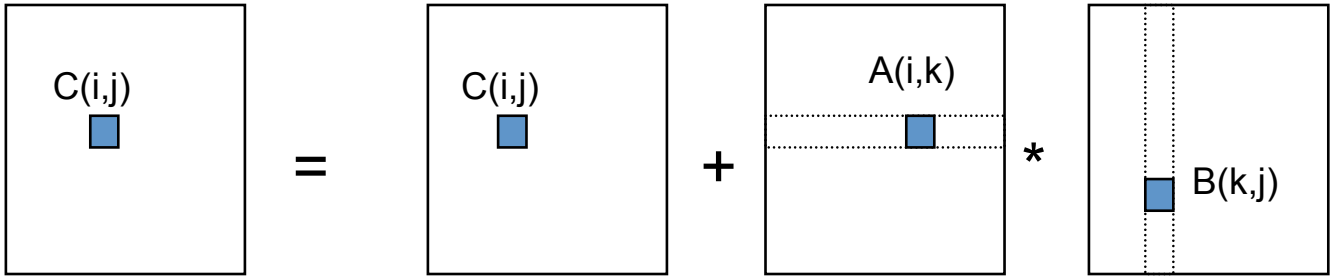
Optimized Implementations: Vendor (Sun) and Autotuned (PHiPAC)

Reference Implementation; Full compiler opt.

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

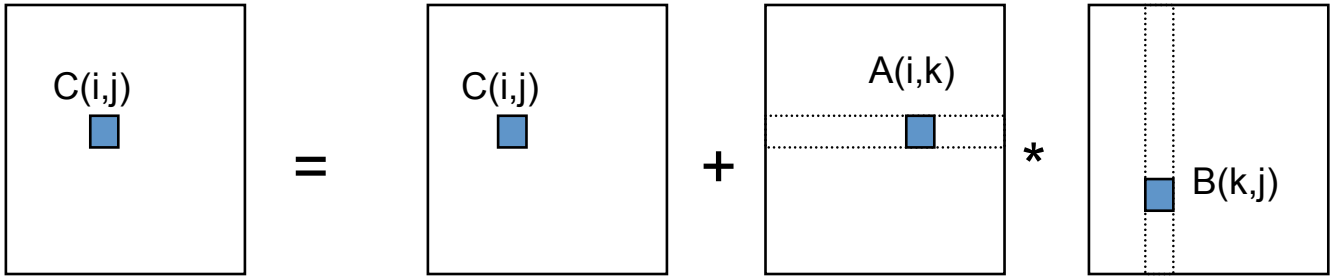
Naive n x n Matrix Multiply C=C+A*B with Fast and Slow Memories

```
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)
```



Naive n x n Matrix Multiply C=C+A*B with Fast and Slow Memories

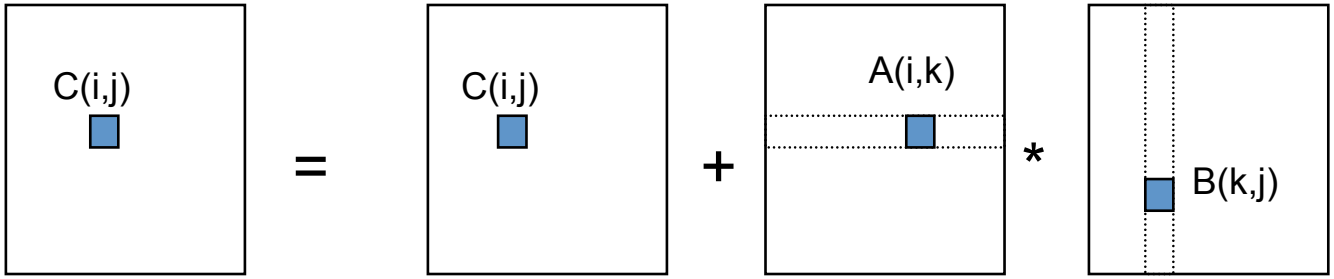
```
for i = 1 to n
    for j = 1 to n
        {read C(i,j) from slow to fast memory: n^2 words moved}
        for k = 1 to n
            {read A(i,k) from slow to fast memory : n^3 words moved}
            {read B(k,j) from slow to fast memory : n^3 words moved}
            C(i,j) = C(i,j) + A(i,k) * B(k,j)  ...  update C(i,j) in fast memory
        {write C(i,j) from fast to slow memory : n^2 words moved}
... 2n^2 + 2n^3 words moved altogether, so computational intensity just ≈ 1
```



Blocked n x n Matrix Multiply C=C+A*B with Fast and Slow Memories

Each n x n matrix consists of b x b blocks, numbered, eg, C(1,1) to C(n/b,n/b)

```
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)  ...  b x b matmul, all in fast memory
```

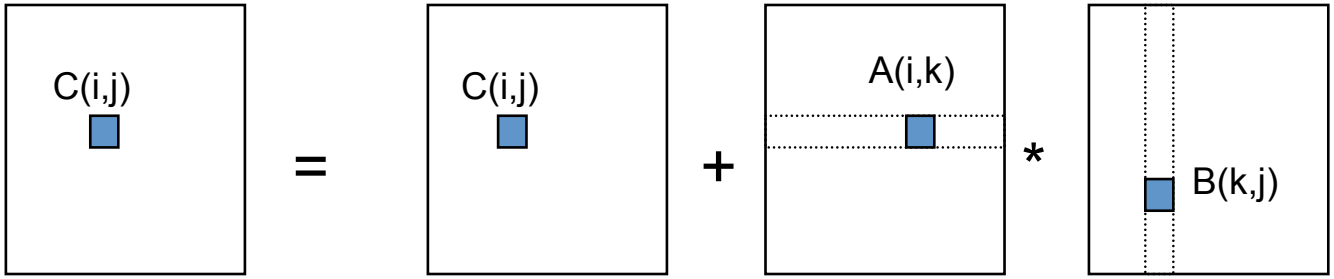


Blocked n x n Matrix Multiply C=C+A*B with Fast and Slow Memories

Each n x n matrix consists of b x b blocks, numbered, eg, C(1,1) to C(n/b,n/b)

```
for i = 1 to n/b
    for j = 1 to n/b
        {read  b x b block C(i,j) from slow to fast memory: n^2 words moved}
        for k = 1 to n/b
            {read  b x b block A(i,k) from slow to fast memory: n^3/b words moved}
            {read  b x b block B(k,j) from slow to fast memory: n^3/b words moved}
            C(i,j) = C(i,j) + A(i,k) * B(k,j)  ...  b x b matmul, all in fast memory
        {write  b x b block C(i,j) from fast to slow memory: n^2 words moved}
```

... $2n^2 + 2n^3/b$ words moved altogether, so computational intensity grows to $\approx b$



Blocked n x n Matrix Multiply C=C+A*B with Fast and Slow Memories

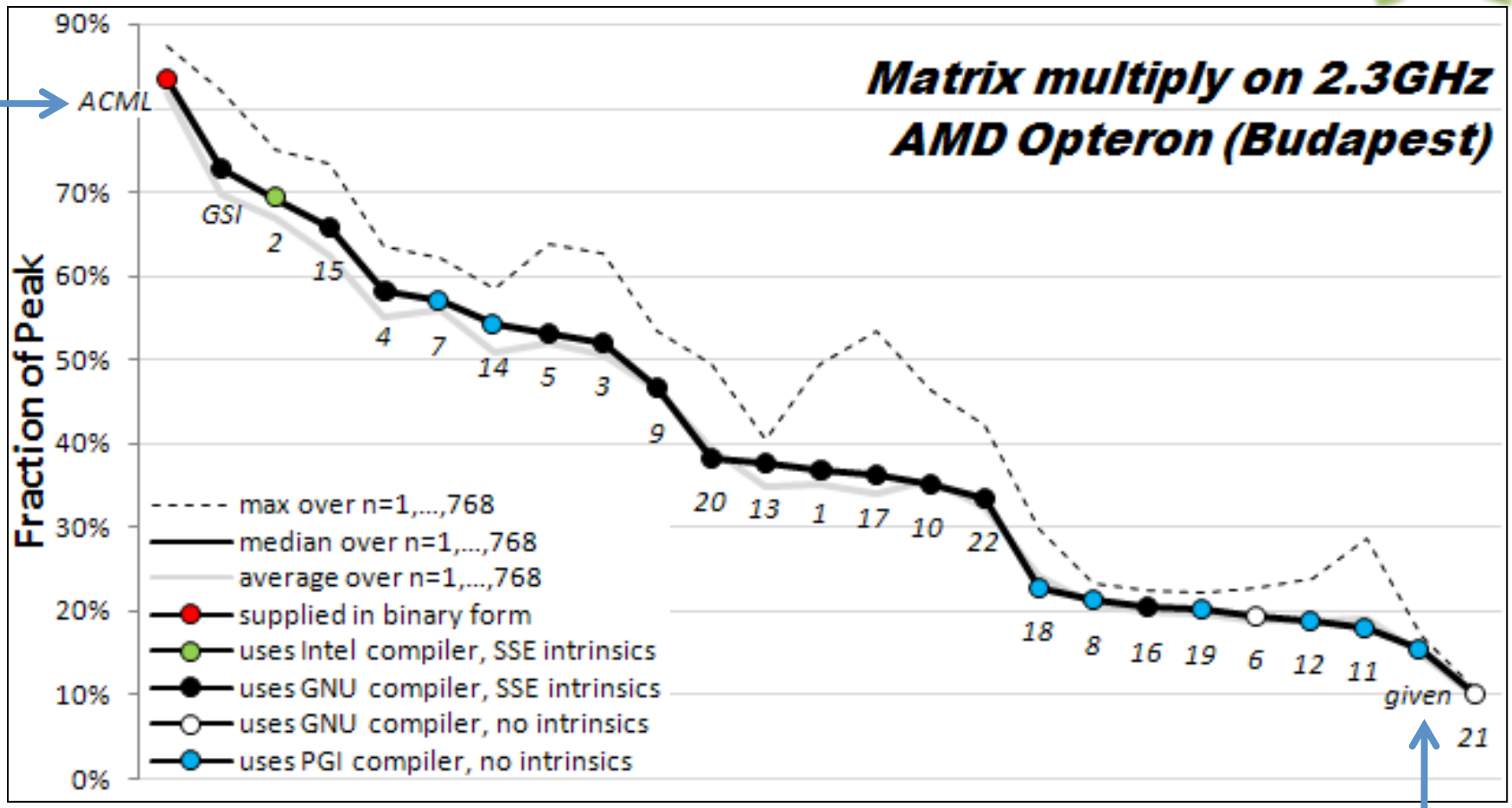
Each n x n matrix consists of b x b blocks, numbered, eg, C(1,1) to C(n/b,n/b)

```
for i = 1 to n/b
    for j = 1 to n/b
        {read  b x b block C(i,j) from slow to fast memory: n^2 words moved}
        for k = 1 to n/b
            {read  b x b block A(i,k) from slow to fast memory: n^3/b words moved}
            {read  b x b block B(k,j) from slow to fast memory: n^3/b words moved}
            C(i,j) = C(i,j) + A(i,k) * B(k,j)  ...  b x b matmul, all in fast memory
        {write  b x b block C(i,j) from fast to slow memory: n^2 words moved}
```

... $2n^2 + 2n^3/b$ words moved altogether, so computational intensity grows to $\approx b$

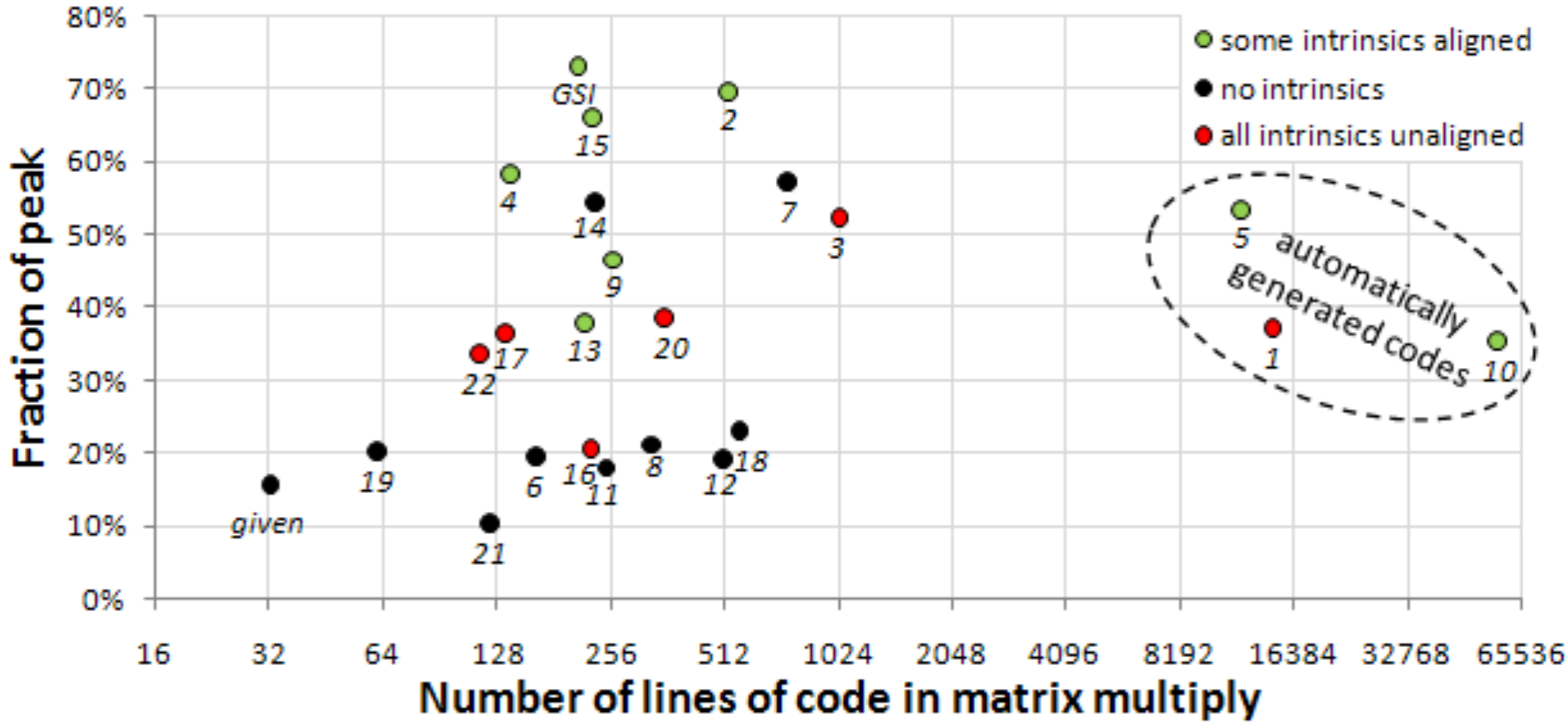
- The bigger the block size b, the fewer words moved; how big can b be?
- We are assuming 3 bxb blocks fit in fast memory
- So if fast memory has size M, we need $3b^2 \le M$, or #words moved $\ge 2 \cdot 3^{1/2} \cdot n^3 / M^{1/2}$
- Thm (Hong & Kung, 1981) : moving $\Omega(n^3 / M^{1/2})$ words is a lower bound
  - True only for classical matmul with n^3 multiplies, not Strassen etc.

How hard is hand-tuning, anyway?

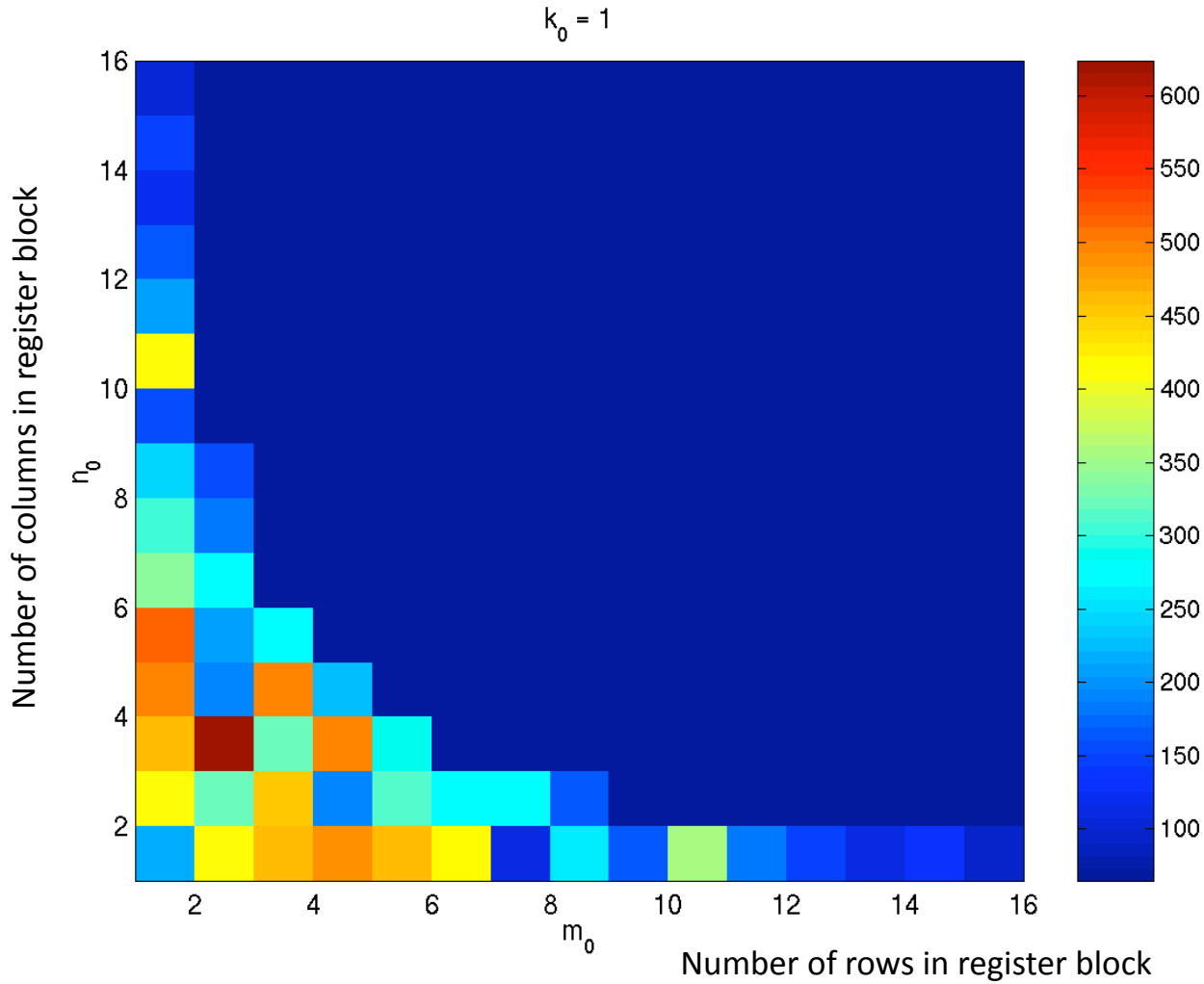


- Results of 22 student teams trying to tune matrix-multiply, in CS267 Spr09
- Students given "blocked" code to start with
  - Still hard to get close to vendor tuned performance (ACML)
- For more discussion, see www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/

How hard is hand-tuning, anyway?



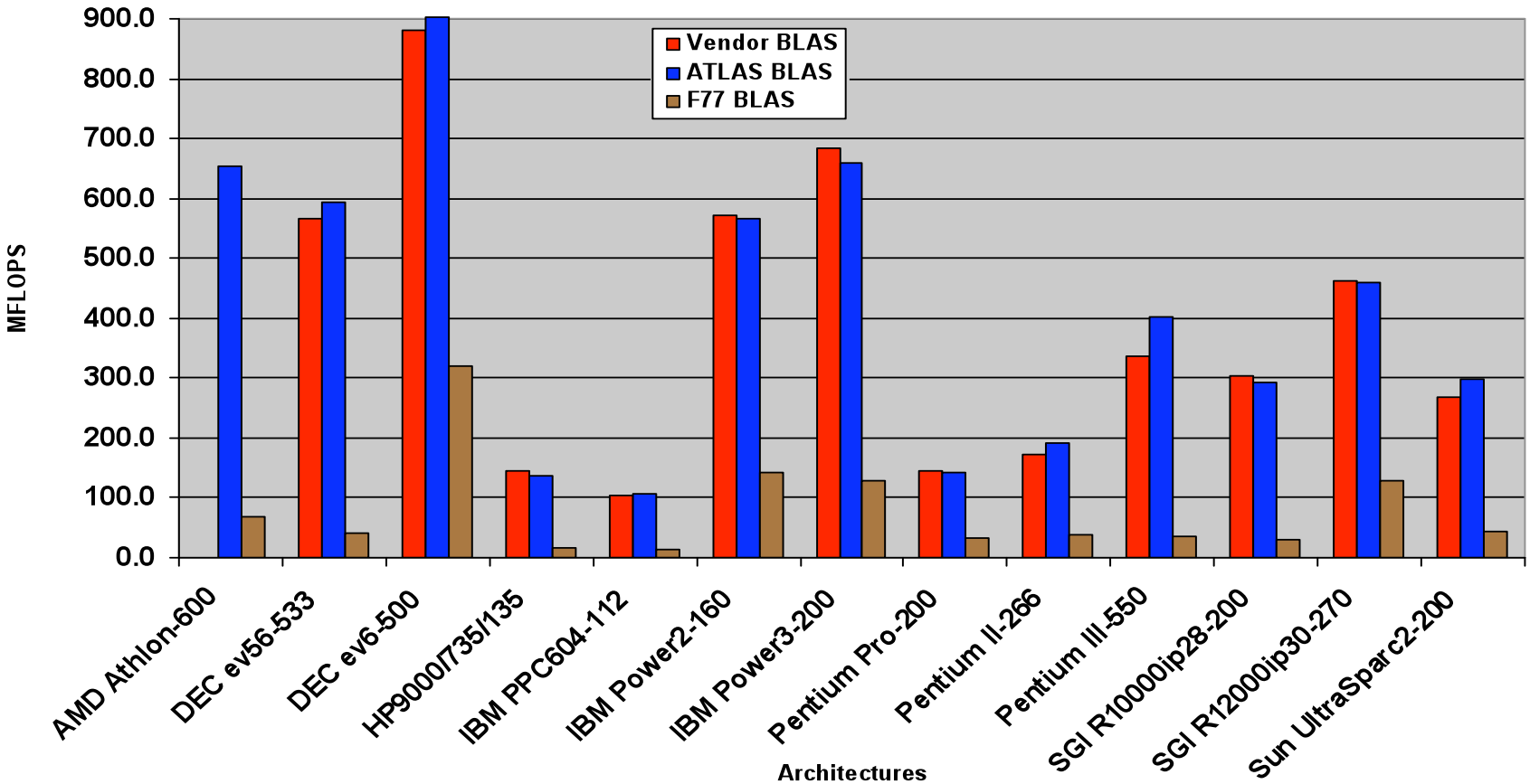
What part of the Matmul Search Space Looks Like



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)

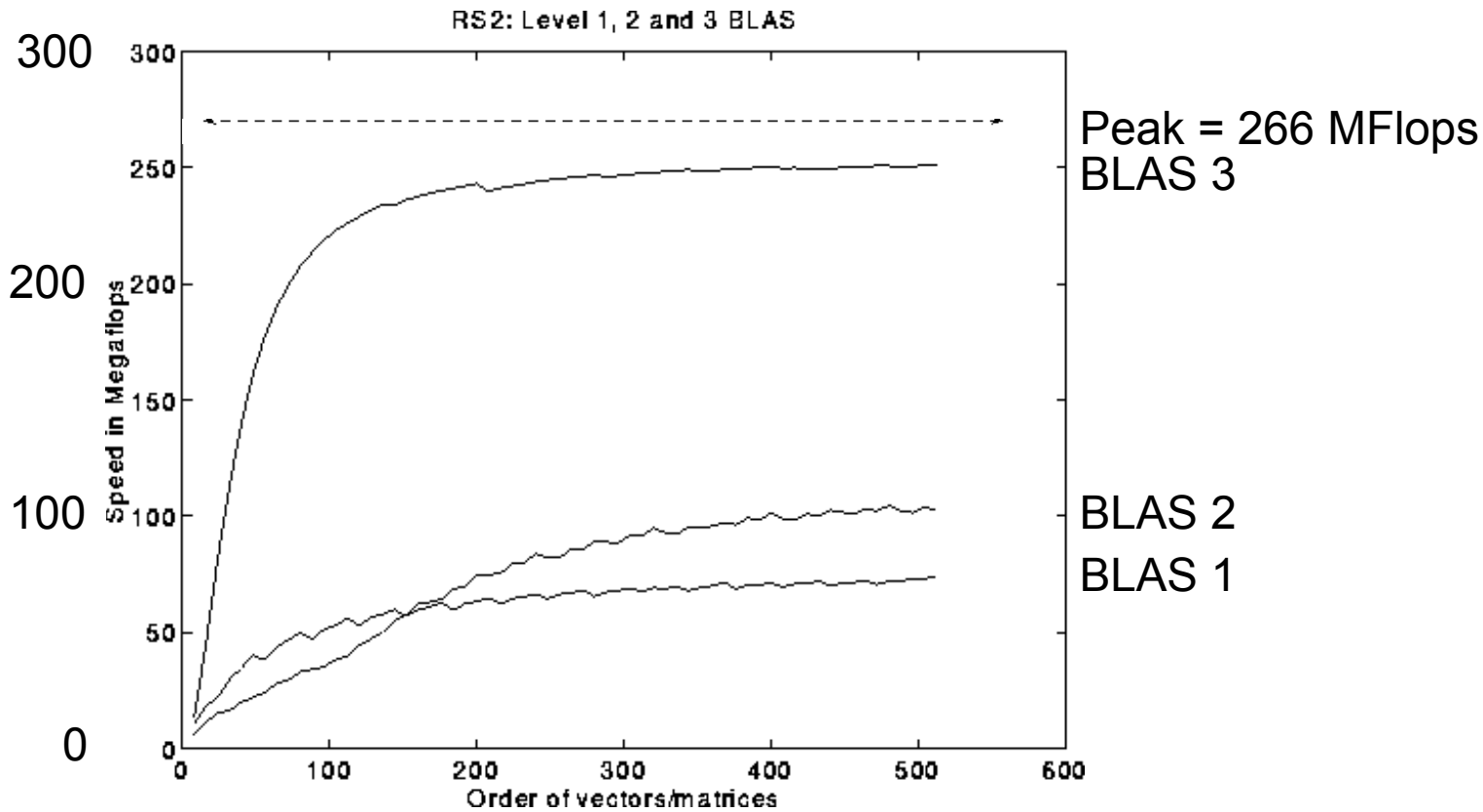
Autotuning DGEMM with ATLAS (n = 500)

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.

BLAS 1/2/3 speeds on IBM RS6000/590



BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

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

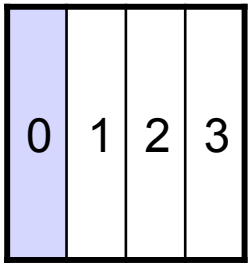
- LAPACK – "Linear Algebra PACKage" - uses BLAS-3 (1989 – now)
  - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of each row to other rows – BLAS-1
    - Need to reorganize GE (and everything else) to use BLAS-3 instead
  - Contents of LAPACK (summary)
    - Algorithms we can 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)
  - How much code? (Release 3.2, Nov 2008) (www.netlib.org/lapack)
    - Source: 1582 routines, 490K LOC, Testing: 352K LOC
  - But can we move data less, and go faster?

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

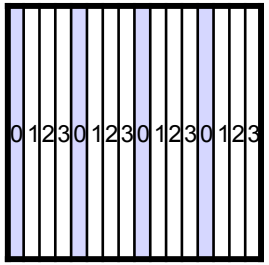
- Is LAPACK parallel?
  - Only if the BLAS are parallel (possible in shared memory)
- ScaLAPACK – "Scalable LAPACK" (1995 – now)
  - For distributed memory – uses MPI
  - More complex data structures, algorithms than LAPACK
    - Only subset of LAPACK's functionality available
    - Work in progress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

Different Parallel Data Layouts for Matrices

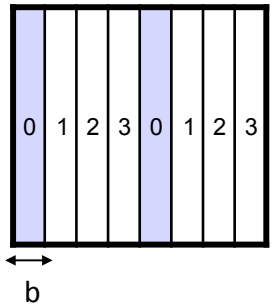
Bad load balance on many submatrices



1) 1D Column Blocked Layout

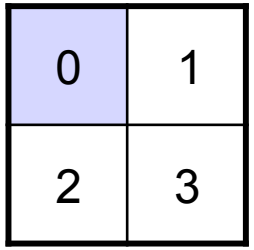


2) 1D Column Cyclic Layout

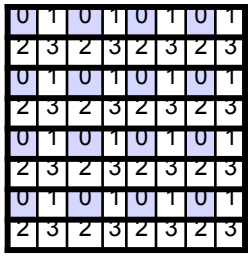


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



5) 2D Row and Column Blocked Layout



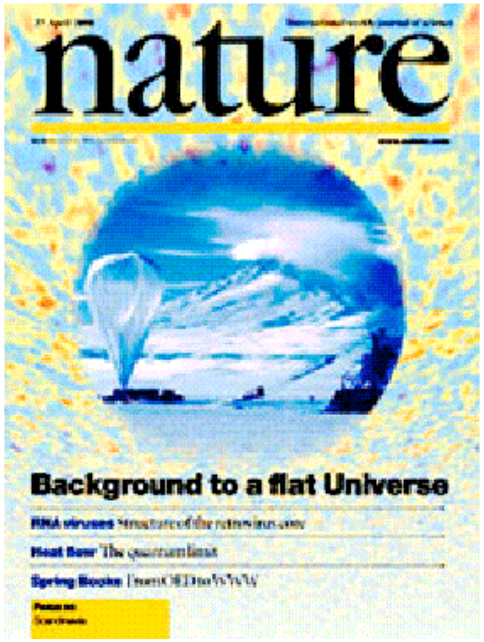
Generalizes others

Best load balance on submatrices

6) 2D Row and Column Block Cyclic Layout

Success Stories for Sca/LAPACK

- Widely used
  - Adopted by Mathworks, Cray, Fujitsu, HP, IBM, IMSL, Intel, NAG, NEC, SGI, …
  - >100M web hits(in 2009, 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



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

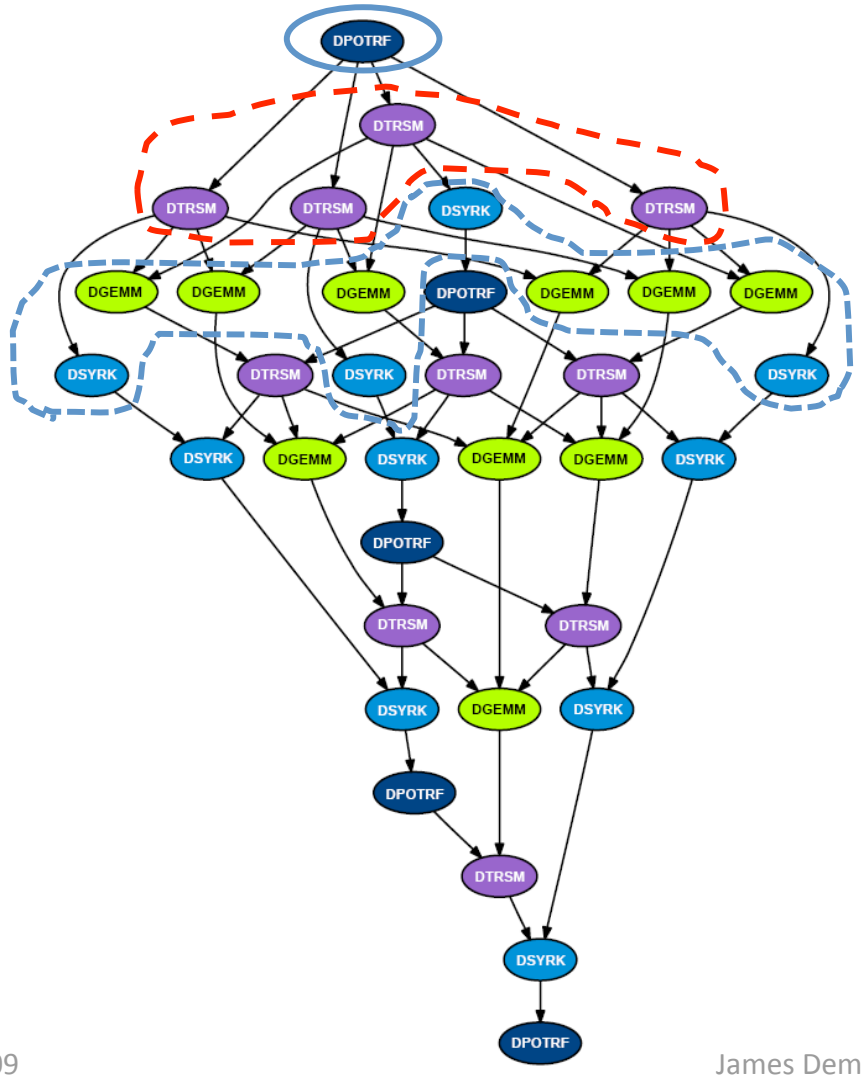
ScaLAPACK

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

- Extensions for multicore
  - LAPACK can use parallel BLAS, ScaLAPACK can use shared-memory MPI, but could be faster…
  - PLASMA – Parallel Linear Algebra for Scalable Multicore Architectures
    - Release 2.0.0 at icl.cs.utk.edu/plasma/
- Extensions for GPUs
  - "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
  - MAGMA – Matrix Algebra on GPU and Multicore Architectures
    - Release 0.1 at icl.cs.utk.edu/magma/
- Communication-minimizing linear algebra
  - Are there lower bounds on data movement for all linear algebra?
  - Are there algorithms that achieve them? Are they in libraries?
- How much code generation can we automate?
  - MAGMA , and FLAME (www.cs.utexas.edu/users/flame/)

PLASMA: Expressing Parallelism with a DAG

- DAG = Directed Acyclic Graph, of tasks
- Tasks are multiplying submatrices, etc.
- Sample DAG for Cholesky with 5 blocks per row/column



- Can process DAG in any order respecting dependencies
- What is the best schedule?
  - Static vs dynamic?
  - Programmer builds DAG vs compiler or run-time system?
  - Build and schedule whole DAG (size = $O((n/b)^3)$) or just "front"
  - Use locality hints?

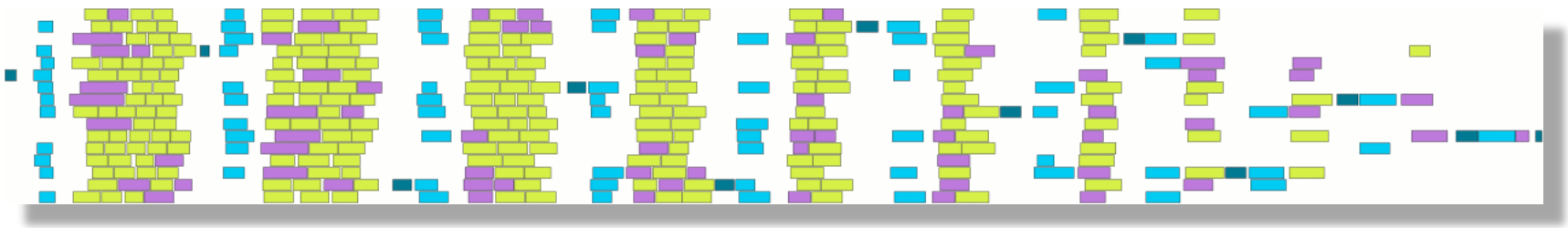
DAG courtesy of Jakub Kurzak, UTK

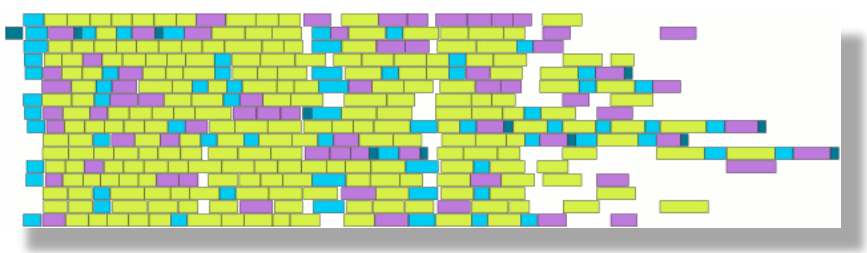
Measured Results for Tiled Cholesky

- Measured on Intel Tigerton 2.4 GHz

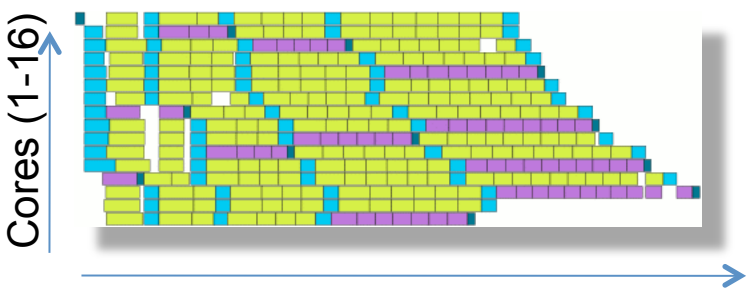
Slide courtesy of Jakub Kurzak, UTK

Cilk (www.cilk.com): programmer defined spawn and sync





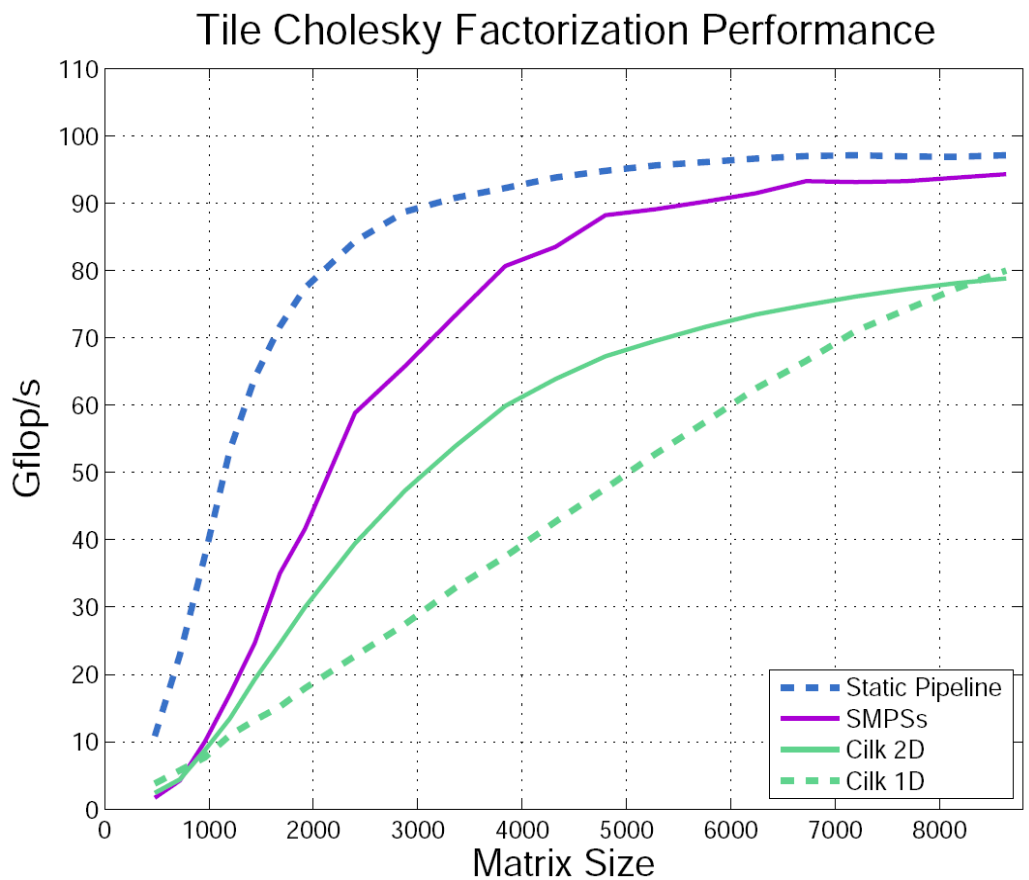
SMPSs (www.bsc.es) : compiler-based with annotations of argument dependencies



PLASMA: static schedule supplied by programmer

More Measured Results for Tiled Cholesky

Slide courtesy of Jakub Kurzak, UTK



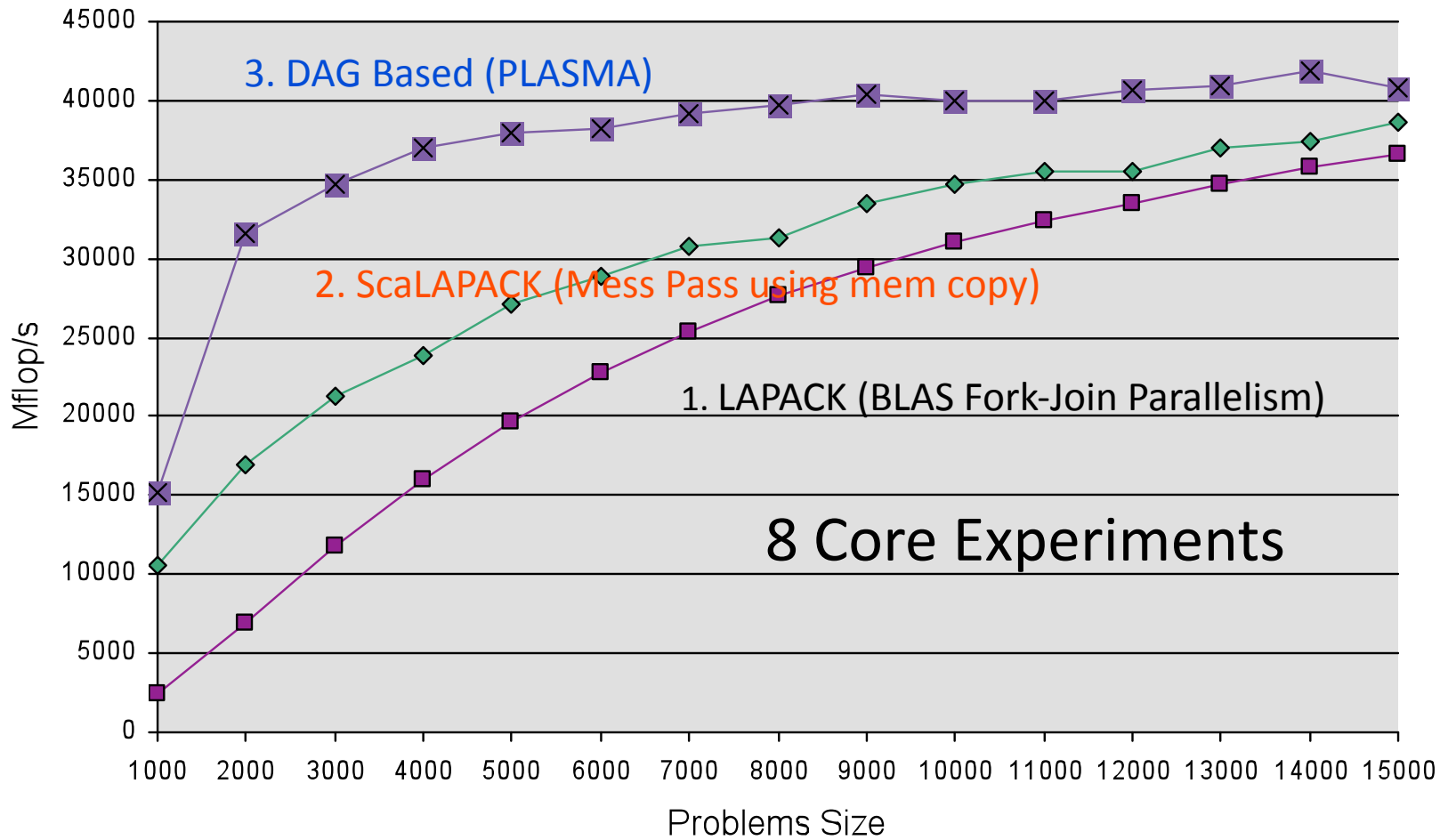
- PLASMA (static pipeline) – best
- SMPSs – somewhat worse
- Cilk 2D – inferior
- Cilk 1D – still worse

quad-socket, quad-core (16 cores total) Intel Tigerton 2.4 GHz

Intel's Clovertown Quad Core

3 Implementations of LU factorization
Quad core w/2 sockets per board, w/ 8 Threads

Source: Jack Dongarra



Scheduling on Multicore – Next Steps

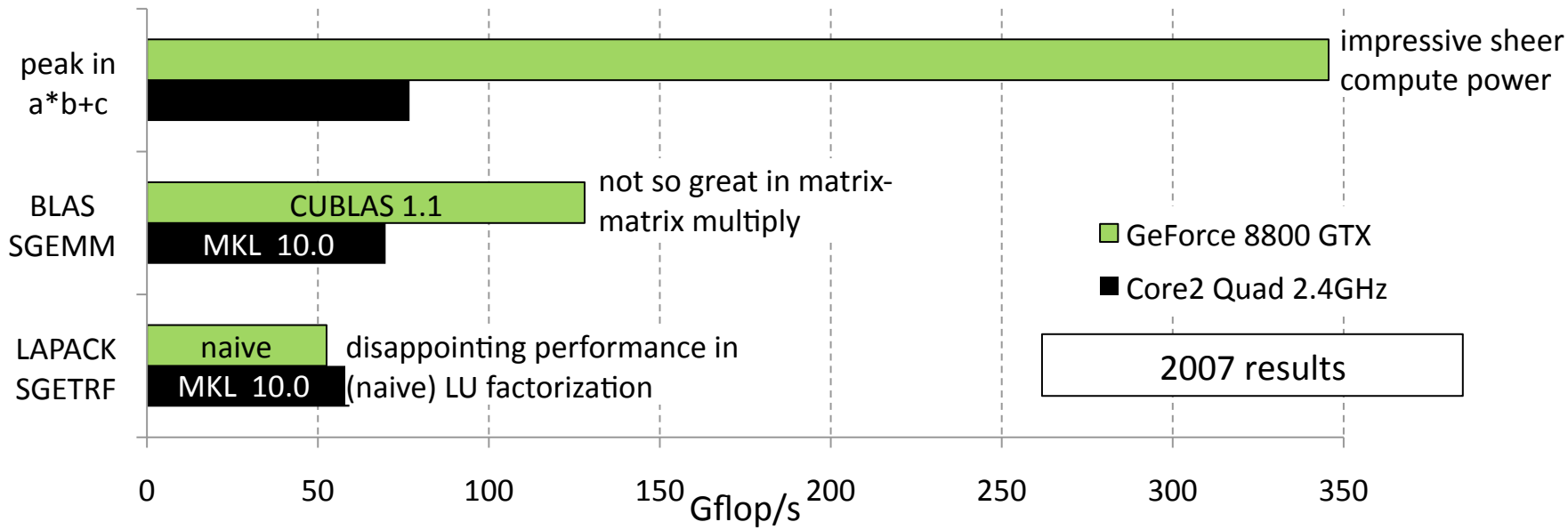
- PLASMA 2.0.0 released
  - Just Cholesky, QR, LU, using static scheduling
  - LU does not do partial pivoting – Stability?
  - icl.cs.utk.edu/plasma/
- Future of PLASMA
  - Add dynamic scheduling, similar to SMPSs
    - DAGs for eigenproblems are too complicated to do by hand
  - Depend on user annotations and API, not compiler
  - Still assume homogeneity of available cores
    - Heterogeneous case: MAGMA

Dense Linear Algebra on GPUs

- Source: Vasily Volkov's SC08 paper
- New challenges
  - More complicated memory hierarchy
  - Not like "L1 inside L2 inside …",
    - Need to choose which memory to use carefully
    - Need to move data explicitly
  - GPU does some operations much faster than CPU, but not all
  - CPU and GPU like different data layouts

Motivation

- NVIDIA released CUBLAS 1.0 in 2007: BLAS for GPUs
- Allows easy port of LAPACK to GPU (consider single precision only)



- Goal: understand bottlenecks in the dense linear algebra kernels
  - Requires detailed understanding of the GPU architecture
  - Result 1: New coding recommendations for high performance on GPUs
  - Result 2: New , fast variants of LU, QR, Cholesky, other routines

(Some new) NVIDIA coding recommendations

- Minimize communication with CPU memory
- Keep as much data in registers as possible
  - Largest, fastest on-GPU memory
  - Vector-only operations
- Use as little shared memory as possible
  - Smaller, slower than registers; use for communication, sharing only
  - Speed limit: 66% of peak with one shared mem argument
- Use vector length VL=64, not max VL = 512
  - Strip mine longer vectors into shorter ones
  - Avoids wasting memory to replicate scalars

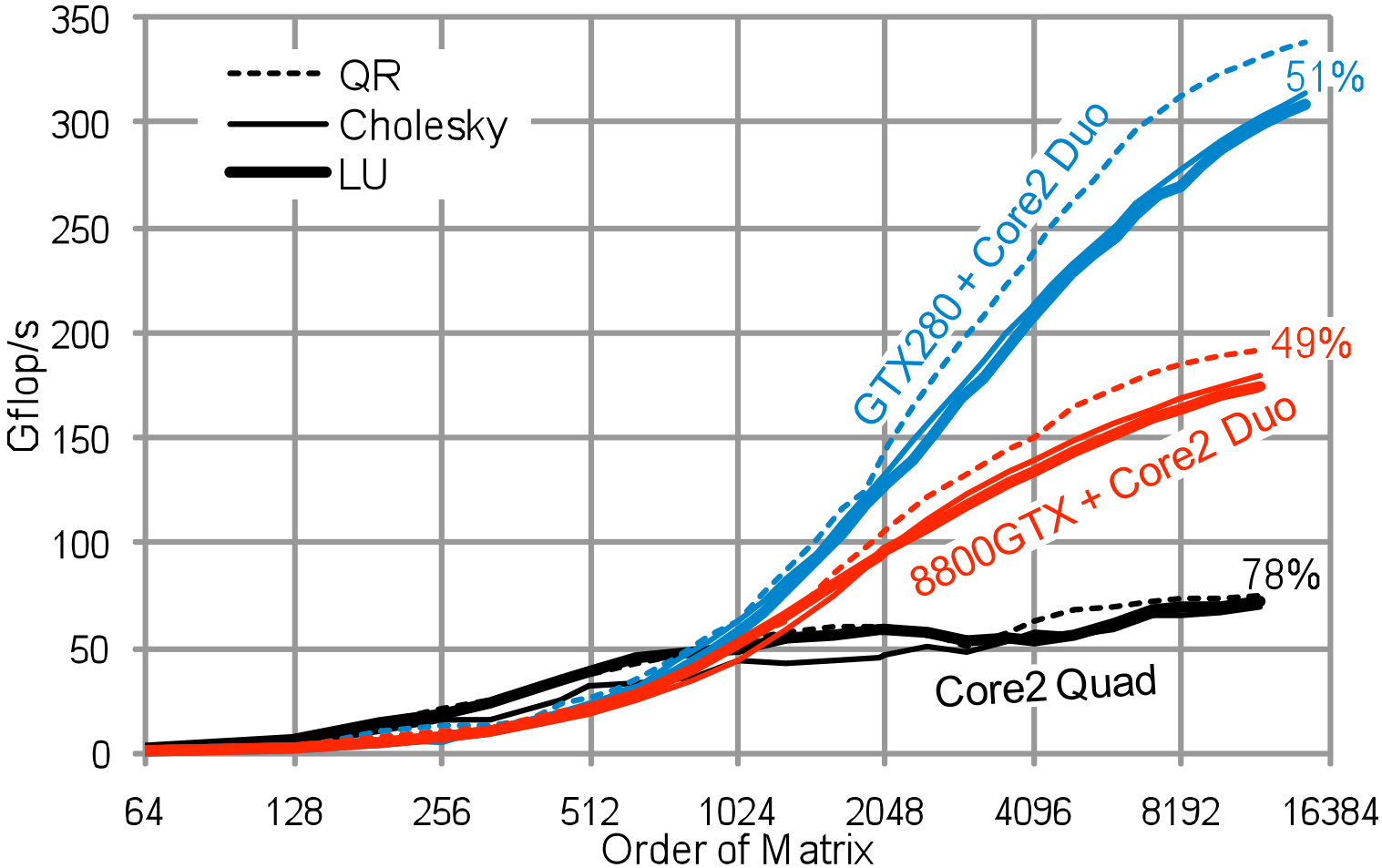
- Final matmul code similar to Cray X1 or IBM 3090 vector codes

Optimizing Matrix Factorizations on GPUs

- Use GPU to compute matrix-matrix multiplies only
  - Do everything else, like factorizing panels, on the CPU
- Use look-ahead to overlap computations on CPU and GPU
- Batch Pivoting
- Use row-major layout on GPU, column-major on CPU
  - Requires extra (but fast) matrix transpose for each CPU-GPU transfer
- Substitute triangular solves of $LX=B$ by TRSM with multiply by L^{-1}
  - At worst squares pivot growth factor in error bound (assume small anyway)
  - Can check $||L^{-1}||$, use TRSM if too large
- Use two-level and variable size blocking as finer tuning
  - Thicker blocks impose lower bandwidth requirements in SGEMM
  - Variable size blocking improves CPU/GPU load balance

- Use column-cyclic layout when computing using two GPUs
  - Requires no data exchange between GPUs in pivoting
  - Cyclic layout is used on GPUs only so does not affect panel factorization

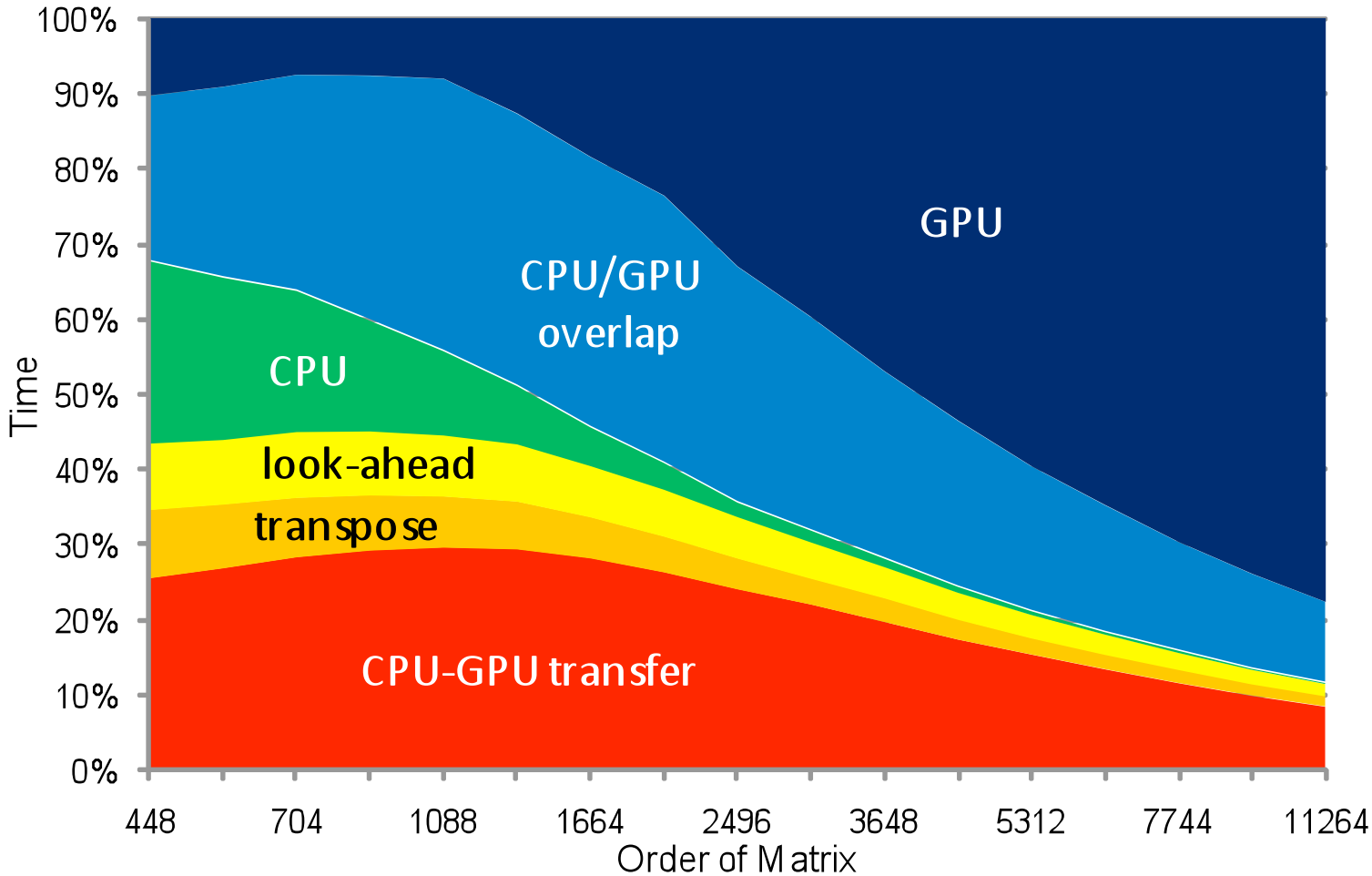
Performance Results



Our solution runs at ~50% of the system's peak (shown on the right)
It is bound by SGEMM that runs at 60% of the GPU-only peak

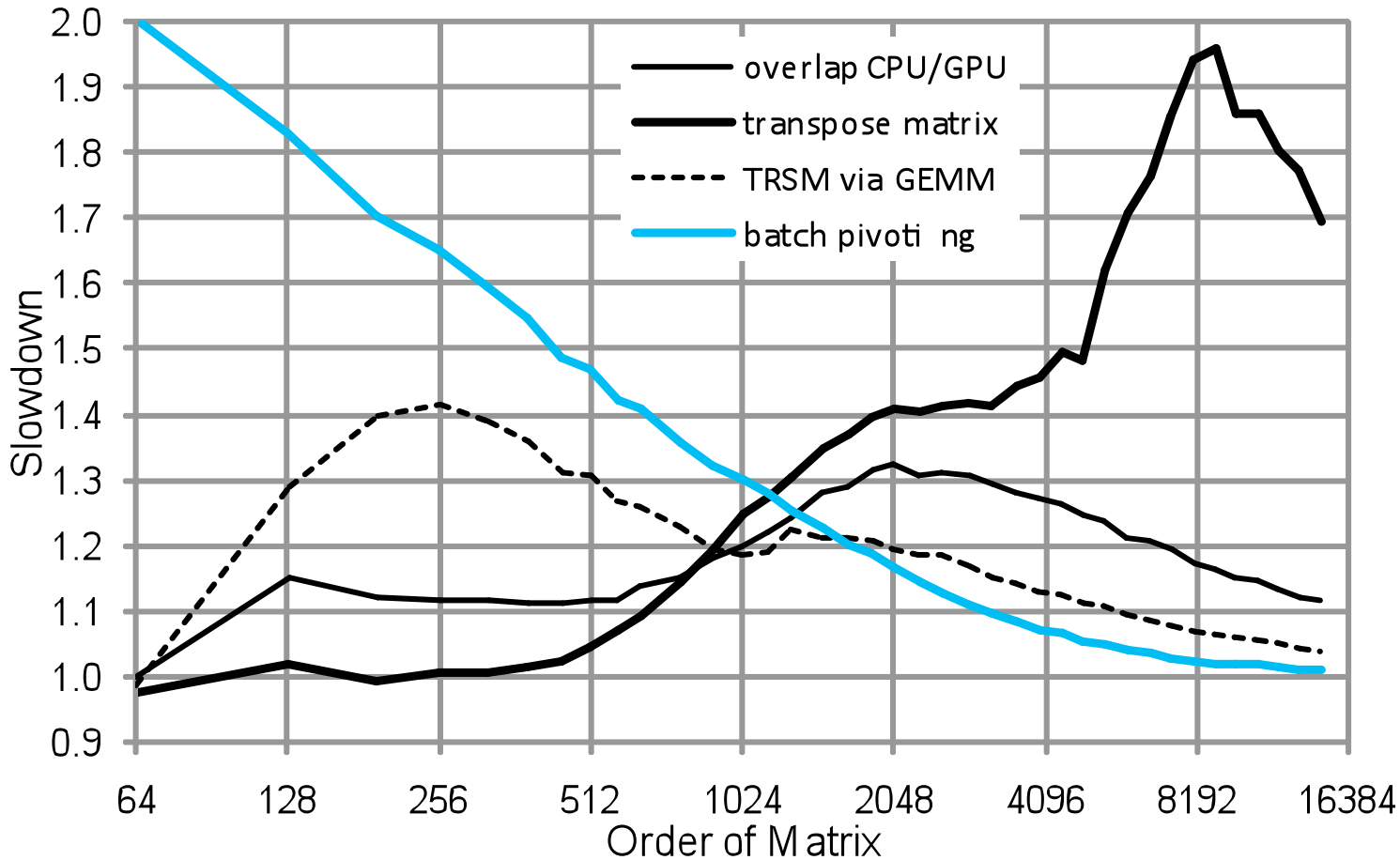
Where does the time go?

- Time breakdown for LU on 8800 GTX

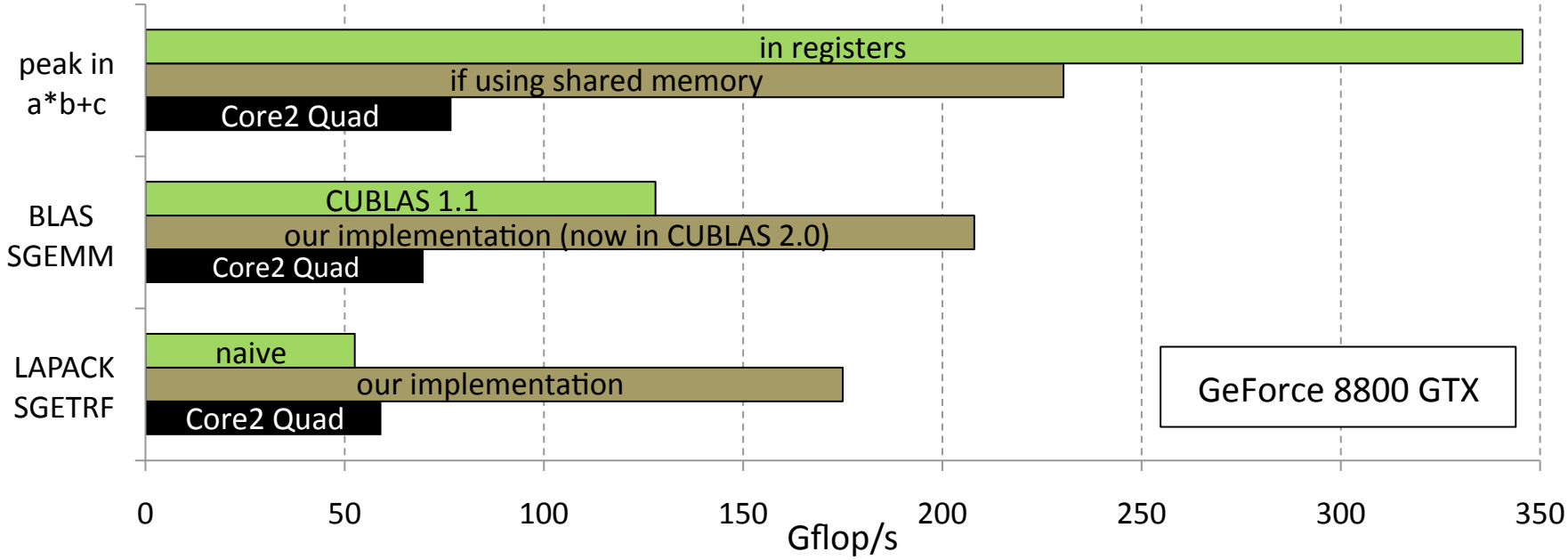


Importance of various optimizations on GPU

- Slowdown when omitting one of the optimizations on GTX 280



Results for matmul, LU on NVIDIA



- What we've achieved:
  - Identified realistic peak speed of GPU architecture
  - Achieved a large fraction of this peak in matrix multiply
  - Achieved a large fraction of the matrix multiply rate in dense factorizations

Communication-Minimizing Linear Algebra

- Lower bounds on data movement for all linear algebra?
- Recall Thm (Hong,Kung,1981): Any sequential matmul doing n^3 multiplies must move $\Omega(n^3/M^{1/2})$ words between slow memory and fast memory of size M (achieved by blocked matmul)
- Parallel Case: Thm (Irony,Tiskin,Toledo,2004): Any parallel matmul on p processors with $O(n^2/p)$ memory/processor must move $\Omega(n^2/p^{1/2})$ words between processors (achieved by Cannon's Alg.)

Communication-Minimizing Linear Algebra

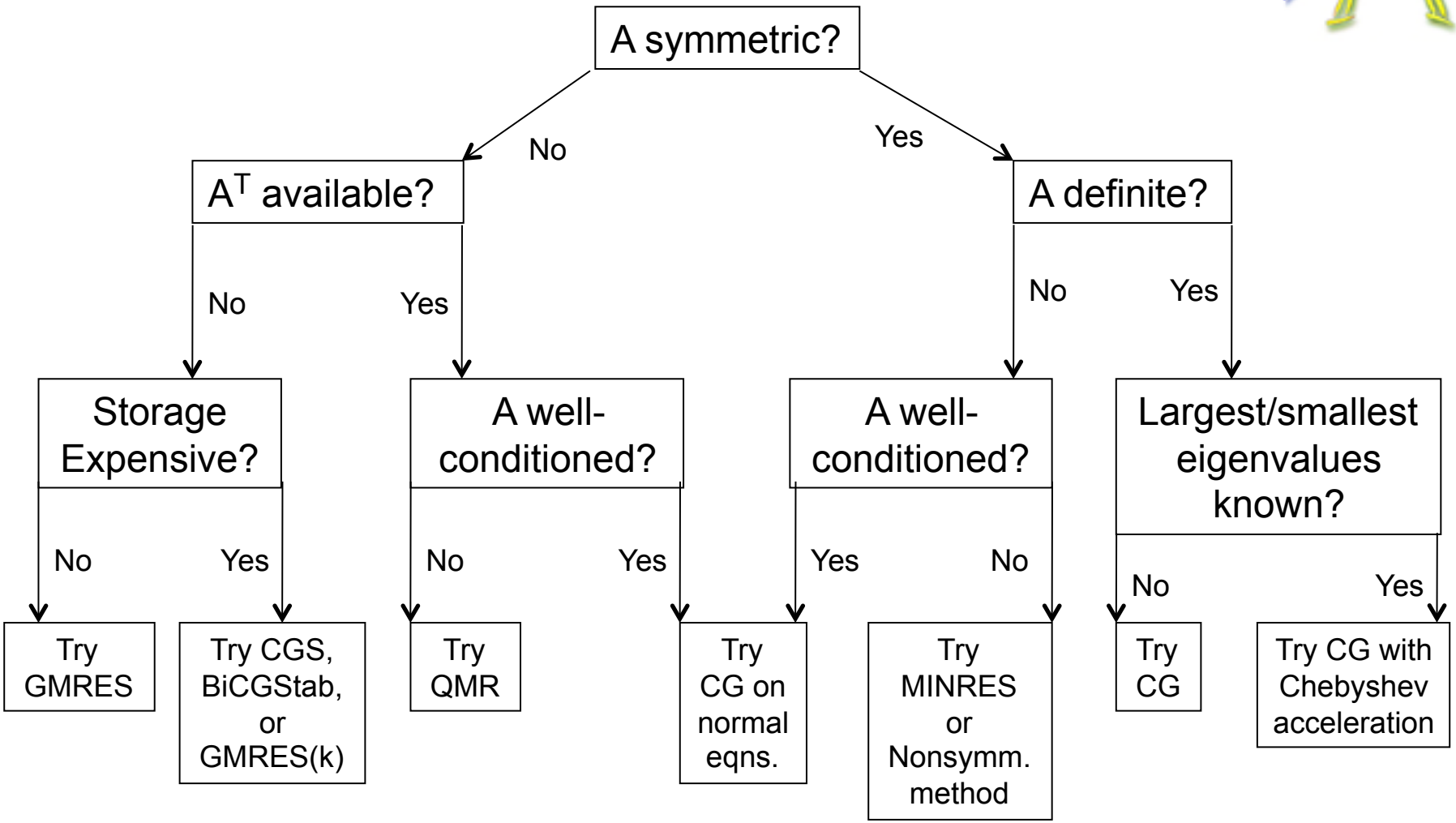
- Can we attain lower bounds $\Omega(F/M^{1/2})$ on #words moved, and $\Omega(F/M^{3/2})$ on #messages?
- Not by algorithms in Sca/LAPACK
  - Except for Cholesky, with right block size
- But new (dense) algorithms do attain them!
  - Dense LU: need to abandon classical partial pivoting
    - Measured speedups up to 5.5x on Cray XT4 (1M x 150)
    - Predicted speedups up to 81x on IBM BG/Q
  - Dense QR: need to represent Q factor in new way
    - Measured speedups to to 8x on Intel Clovertown (1M x 10)
  - Eig and SVD: work in progress
  - Sparse linear algebra:
    - some positive results in progress, lots of open questions
- See www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-62.html for lower bounds, and references therein for algorithms

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/index.html for LU codes
    - See crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf for a survey of available serial and parallel sparse solvers
  - 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



- 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

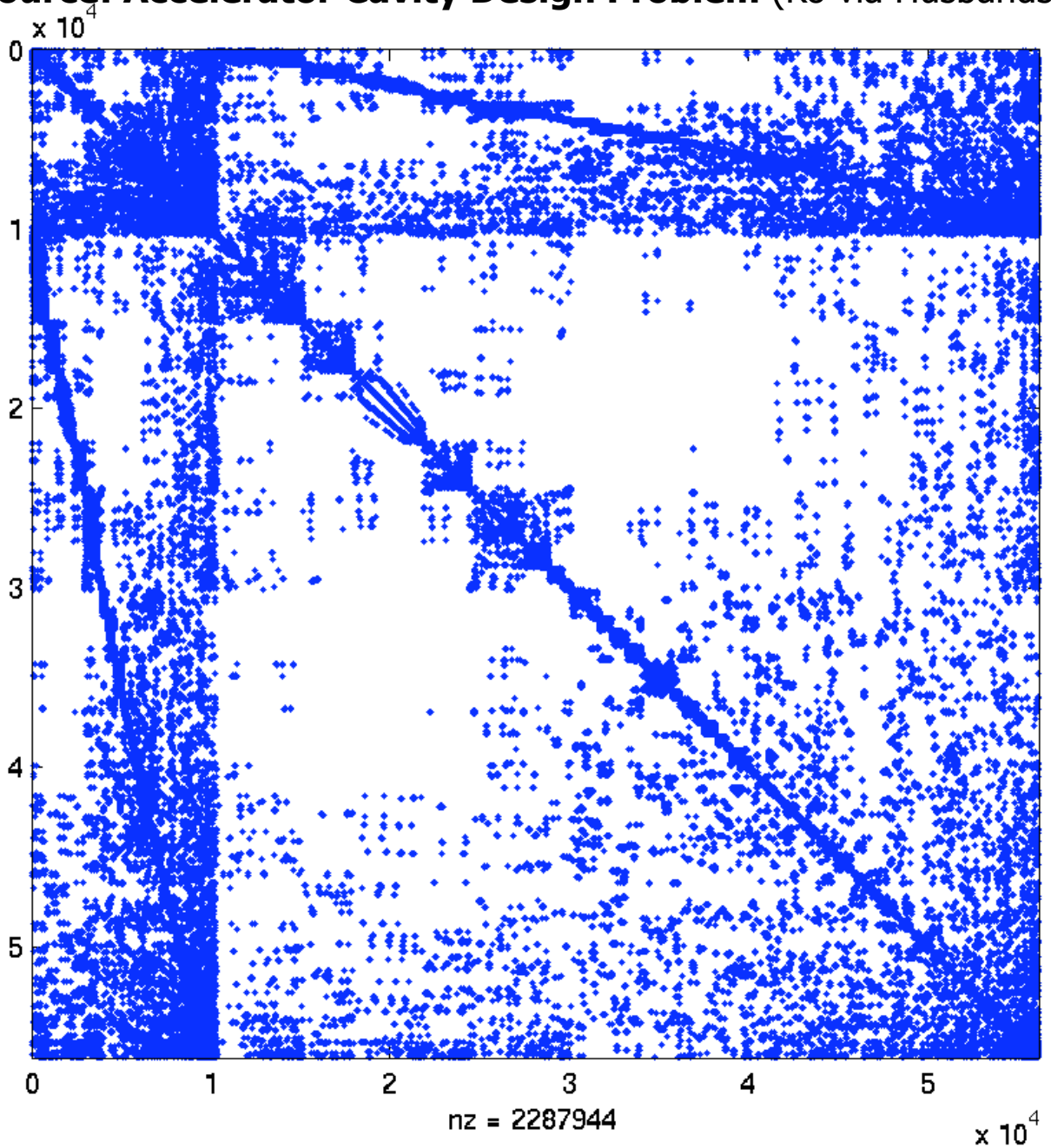
Sparse Outline

- Approaches to Automatic Performance Tuning
- Results for sparse matrix kernels
  - Sparse Matrix Vector Multiplication (SpMV)
  - Sequential and Multicore results
- OSKI = Optimized Sparse Kernel Interface
- Tuning Entire Sparse Solvers

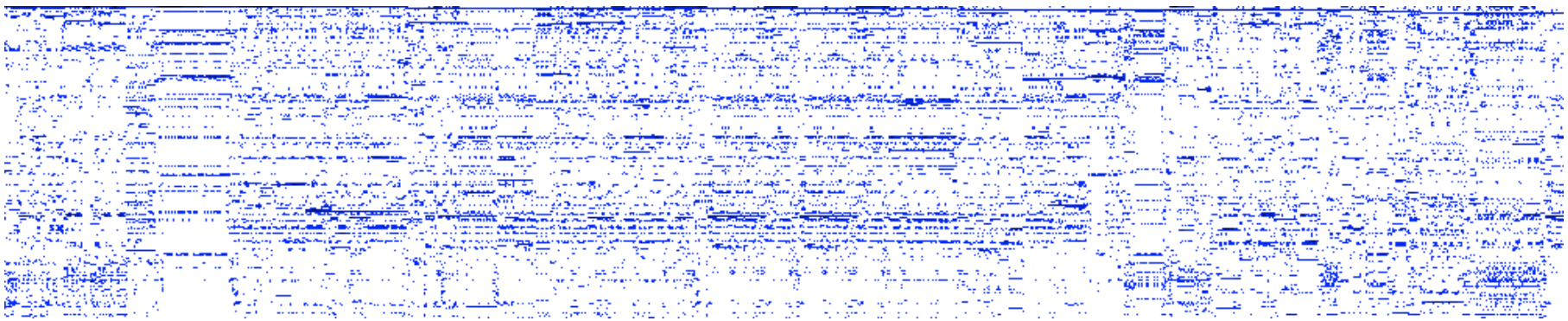
Approaches to Automatic Performance Tuning

- **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**
  - **www.spiral.net, www.fftw.org**

Source: Accelerator Cavity Design Problem (Ko via Husbands)

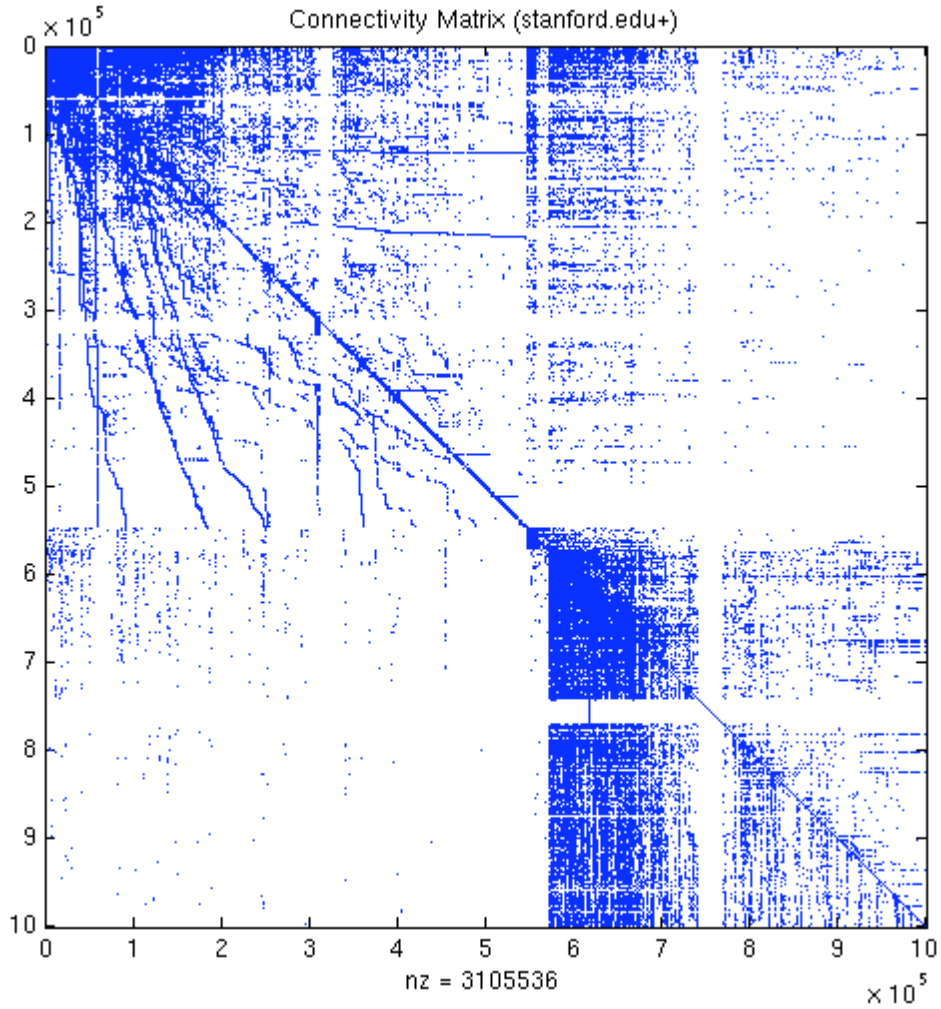


Linear Programming Matrix

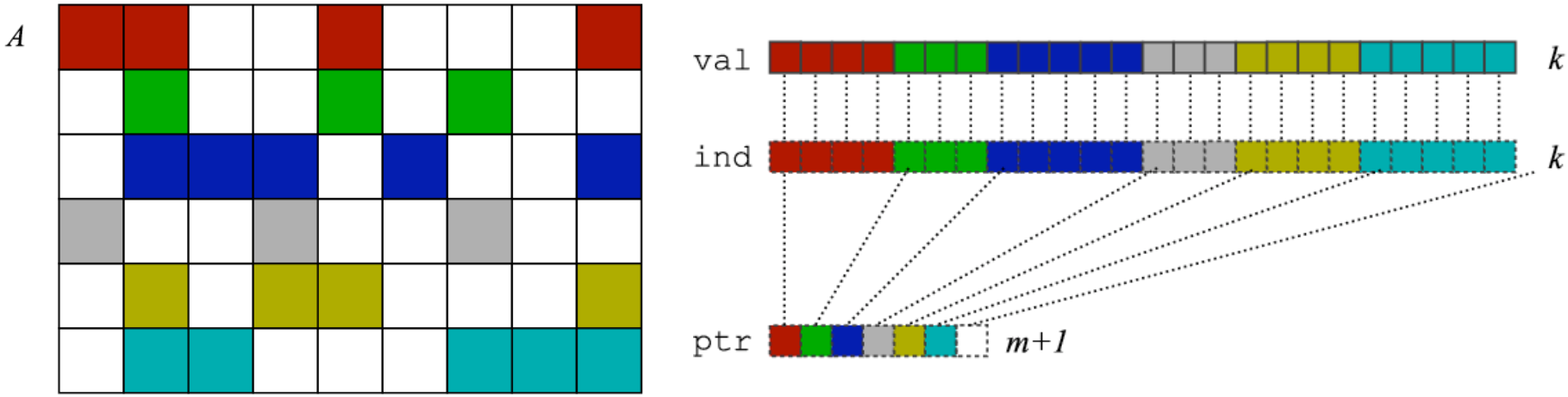


…

A Sparse Matrix You Use Every Day



SpMV with Compressed Sparse Row (CSR) Storage

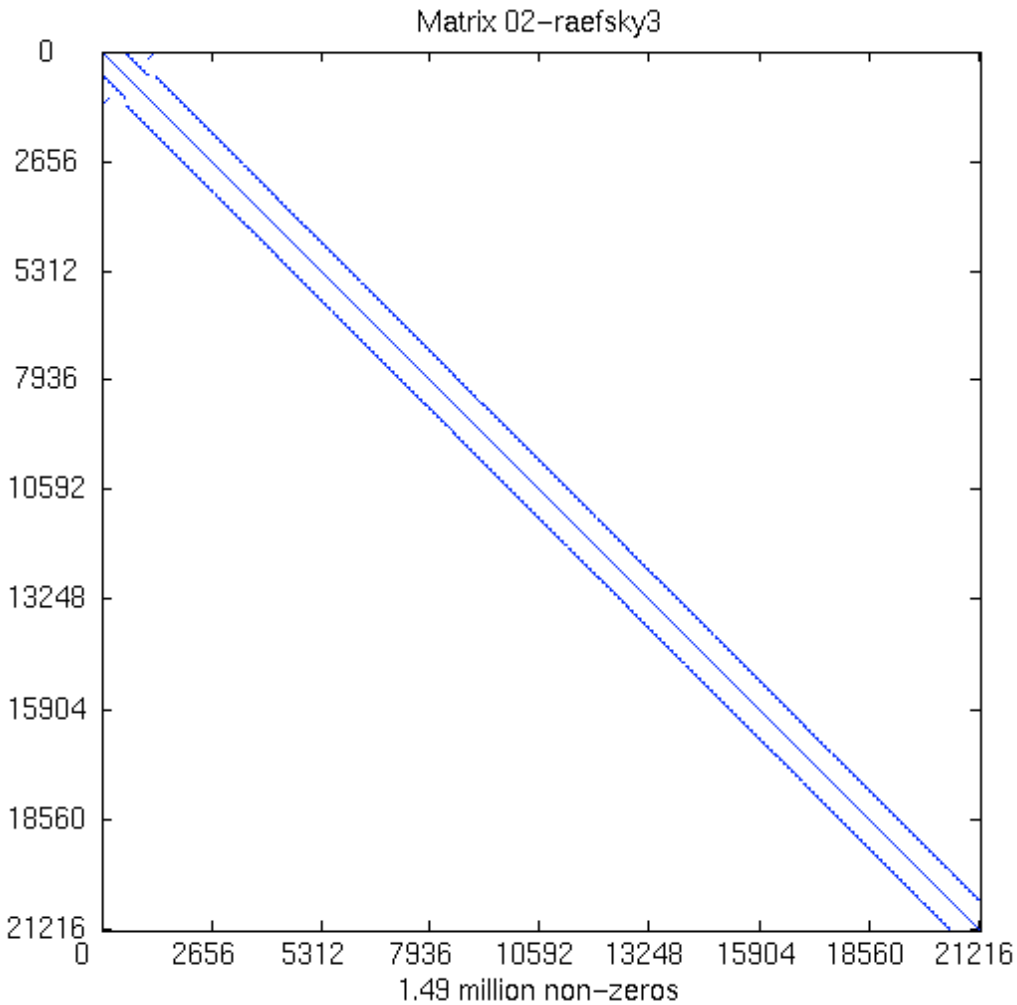


Matrix-vector multiply kernel: y(i) ← y(i) + A(i,j)*x(j)

```
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

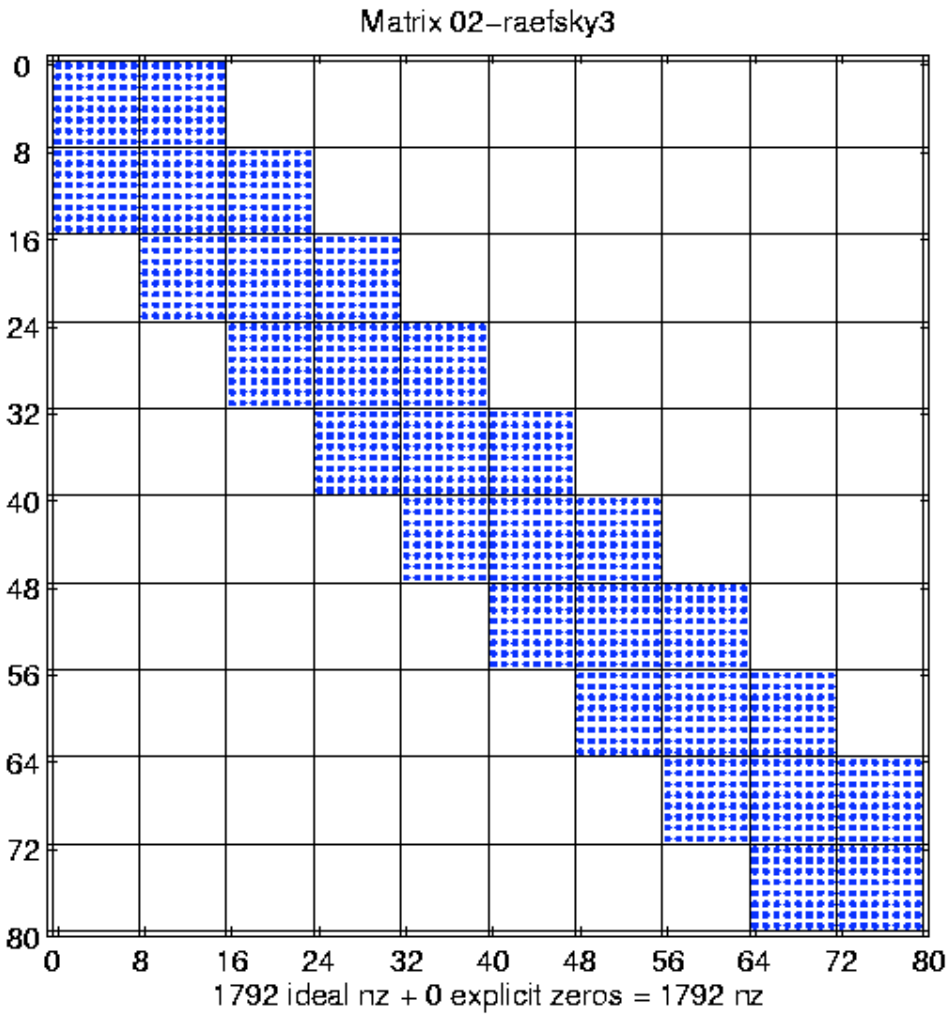
Example: The Difficulty of Tuning



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

- Source: NASA structural analysis problem

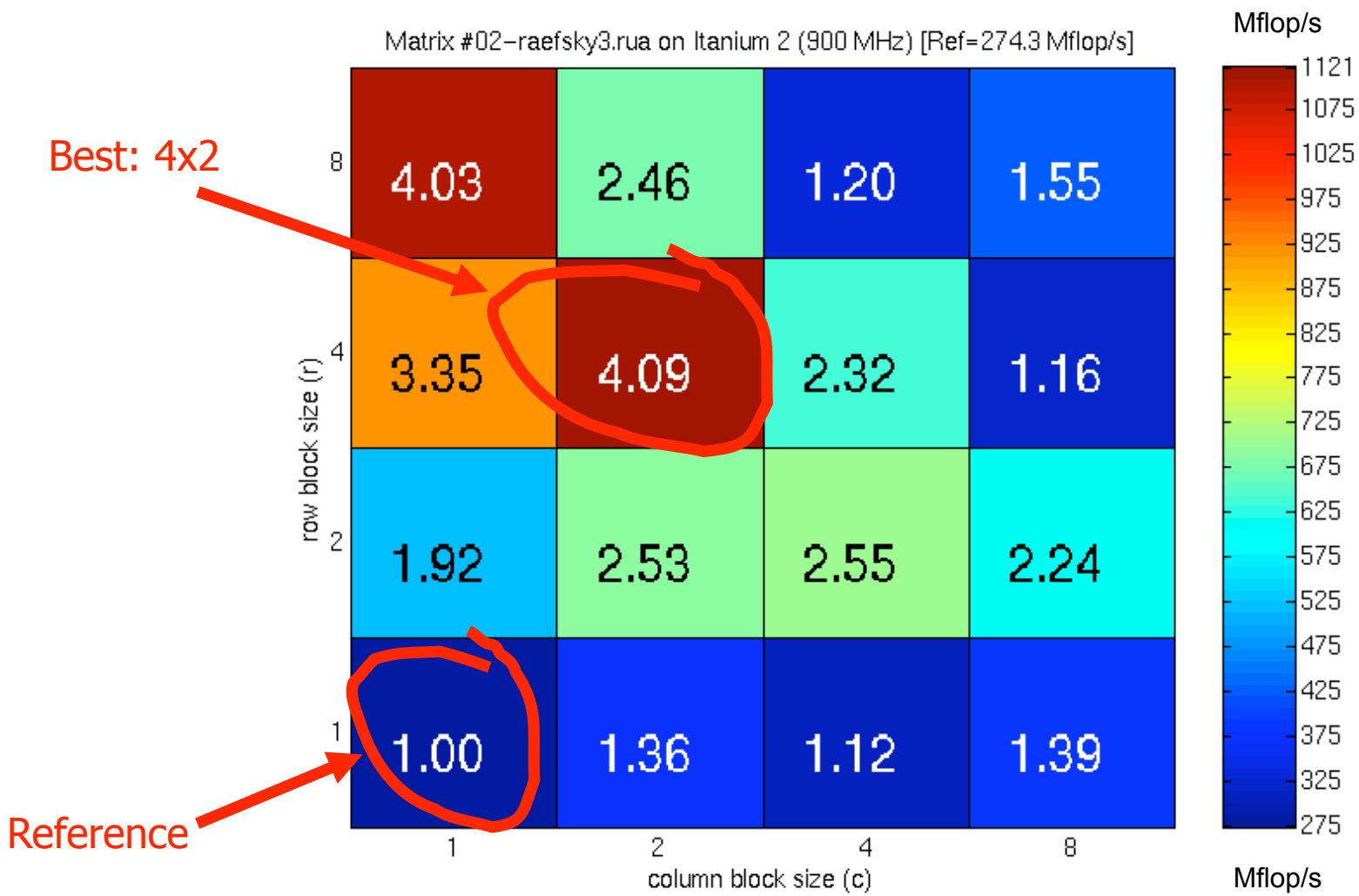
Example: The Difficulty of Tuning



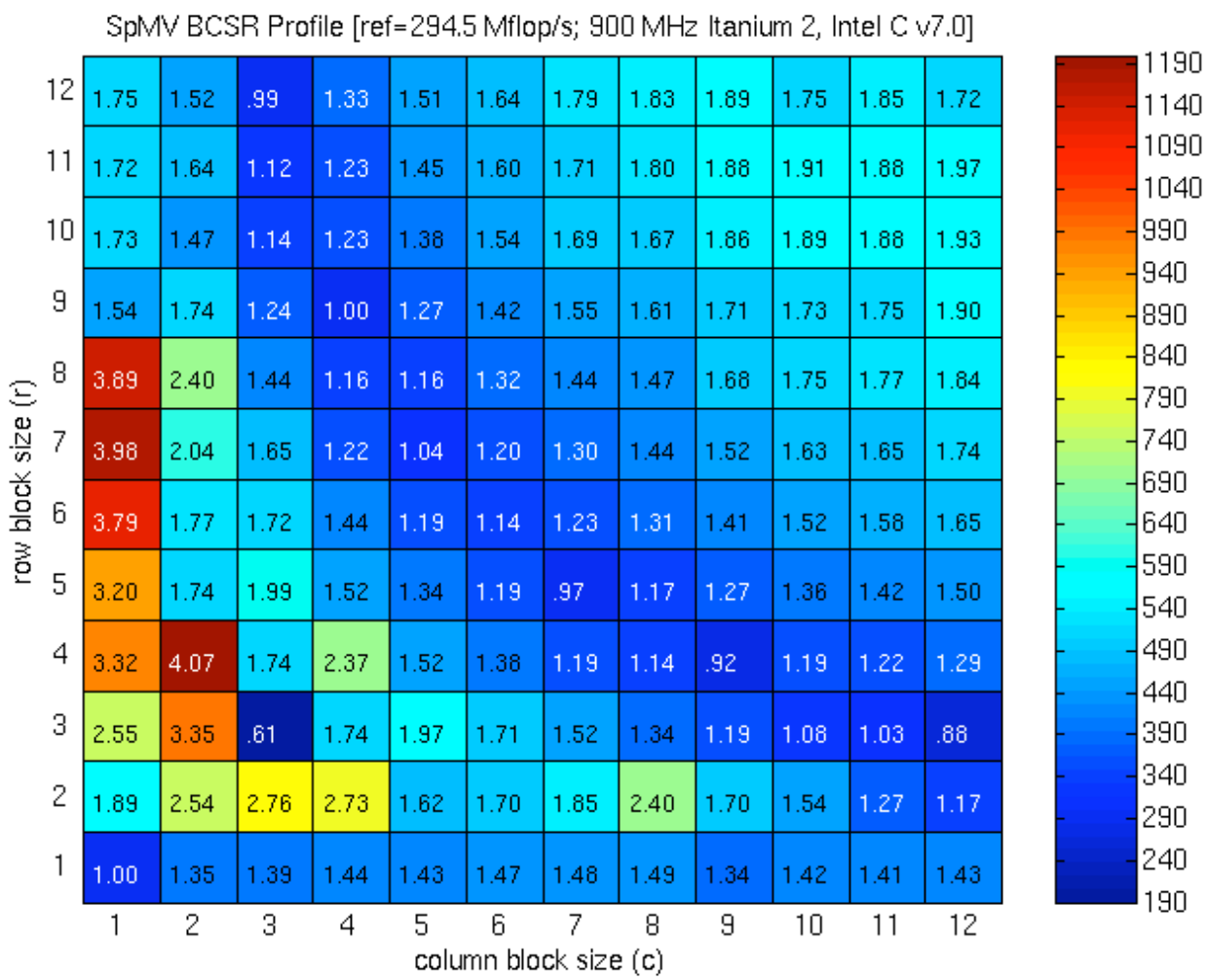
- n = 21200
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem
- **8x8** dense substructure: exploit this to limit #mem_refs

Speedups on Itanium 2: The Need for Search



Register Profile: Itanium 2



1190 Mflop/s

190 Mflop/s

Power3 - 17%

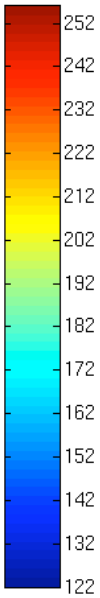
rofile [ref=163.9 Mflop/s; 375 MHz Power3, IBM xlc v5]

252 Mflop/s

r \ c	1	2	3	4	5	6	7	8	9	10	11	12
12	1.37	1.43	1.12	1.40	1.12	1.12	1.12	1.18	1.17	1.28	1.31	1.26
11	1.27	1.42	.89	1.25	1.04	1.15	1.30	1.27	1.47	1.45	1.53	1.50
10	1.33	1.44	.86	.84	1.05	1.08	1.17	1.05	1.21	1.24	1.33	1.31
9	1.31	1.21	.87	1.25	1.01	1.10	1.03	1.06	1.17	1.25	1.31	1.30
8	1.23	1.27	.81	1.17	1.11	1.00	1.09	1.20	1.10	1.11	1.27	1.21
7	1.36	1.33	1.11	1.02	1.07	1.04	1.11	1.03	1.17	1.15	1.30	1.28
6	1.36	1.46	1.36	.97	.90	1.11	1.02	1.20	1.18	1.11	1.22	1.18
5	1.24	1.39	1.39	1.53	1.00	1.25	1.13	.98	1.08	1.15	1.22	1.04
4	1.15	1.27	1.46	1.56	1.45	.87	.93	.74	.95	1.01	1.05	.92
3	1.22	1.39	1.42	1.42	1.36	1.39	1.39	.75	.79	.99	1.02	.96
2	1.05	1.28	1.36	1.24	1.33	1.36	1.27	1.15	1.28	1.24	1.25	1.12
1	1.00	1.05	1.18	1.15	1.31	1.33	1.33	1.22	1.28	1.33	1.25	1.37

row block size (r)

column block size (c)



122 Mflop/s

Power4 - 16%

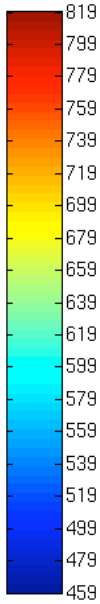
ofile [ref=594.9 Mflop/s; 1.3 GHz Power4, IBM xlc v6]

820 Mflop/s

r \ c	1	2	3	4	5	6	7	8	9	10	11	12
12	1.28	1.23	1.28	1.06	1.26	1.24	1.29	1.16	1.24	1.17	1.30	1.33
11	1.34	1.29	1.26	1.14	1.12	1.26	1.27	1.28	1.26	1.25	1.26	1.28
10	1.33	1.29	1.21	1.29	1.16	1.14	1.27	1.38	1.27	1.25	1.29	1.29
9	1.32	1.29	1.20	1.20	1.16	1.16	1.21	1.31	1.27	1.27	1.25	1.30
8	1.32	1.27	1.19	1.19	1.25	1.12	1.26	1.15	1.27	1.10	1.13	1.23
7	1.35	1.30	1.30	1.26	1.17	1.10	1.11	1.24	1.25	1.26	1.27	1.29
6	1.32	1.26	1.34	1.09	1.16	1.19	1.18	1.10	1.20	1.18	1.25	1.28
5	1.27	1.24	1.15	1.27	1.15	.91	.99	.94	.80	.81	1.06	1.03
4	1.29	1.26	.98	1.24	1.23	1.12	.77	.99	.95	.89	.81	.86
3	1.23	1.29	1.27	1.18	1.13	1.23	1.18	.99	.93	.82	.80	.90
2	1.13	1.18	1.25	1.20	1.17	1.00	.92	.87	1.17	1.17	1.15	.99
1	1.00	1.10	1.15	1.15	1.18	1.16	1.14	1.06	1.05	1.02	.88	.91

row block size (r)

column block size (c)



459 Mflop/s

Itanium 1 - 8%

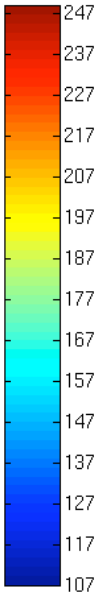
ofile [ref=161.2 Mflop/s; 800 MHz Itanium, Intel C v7]

247 Mflop/s

r \ c	1	2	3	4	5	6	7	8	9	10	11	12
12	.92	.77	.99	1.16	1.30	1.35	1.39	1.39	1.45	1.24	1.22	1.32
11	1.52	.80	.95	1.11	1.24	1.35	1.39	1.39	1.44	1.24	1.24	1.29
10	1.34	.74	.88	1.04	1.16	1.30	1.36	1.35	1.38	1.22	1.23	1.32
9	1.25	.78	.82	1.01	1.10	1.19	1.34	1.35	1.39	1.21	1.20	1.28
8	1.47	.72	.77	.92	1.05	1.16	1.22	1.31	1.37	1.22	1.15	1.27
7	1.38	.80	.76	.82	.98	1.11	1.16	1.26	1.34	1.25	1.18	1.32
6	1.25	.84	.78	.79	.87	.99	1.09	1.18	1.23	1.29	1.14	1.41
5	1.10	1.17	.75	.71	.80	.88	.97	1.06	1.09	1.15	1.14	1.29
4	1.55	1.30	.80	.72	.71	.77	.80	.94	1.00	1.08	1.11	1.16
3	1.54	1.04	1.15	.80	.71	.66	.76	.77	.81	.87	.95	.98
2	1.48	1.48	1.02	1.27	1.05	.83	.70	.67	.66	.68	.77	.77
1	1.00	1.07	1.05	1.12	.89	.95	1.07	1.21	1.02	.94	.82	.73

row block size (r)

column block size (c)



107 Mflop/s

Itanium 2 - 33%

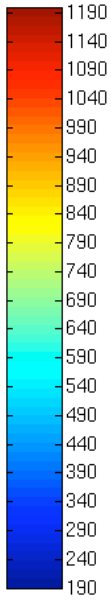
ef=294.5 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]

1.2 Gflop/s

r \ c	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

row block size (r)

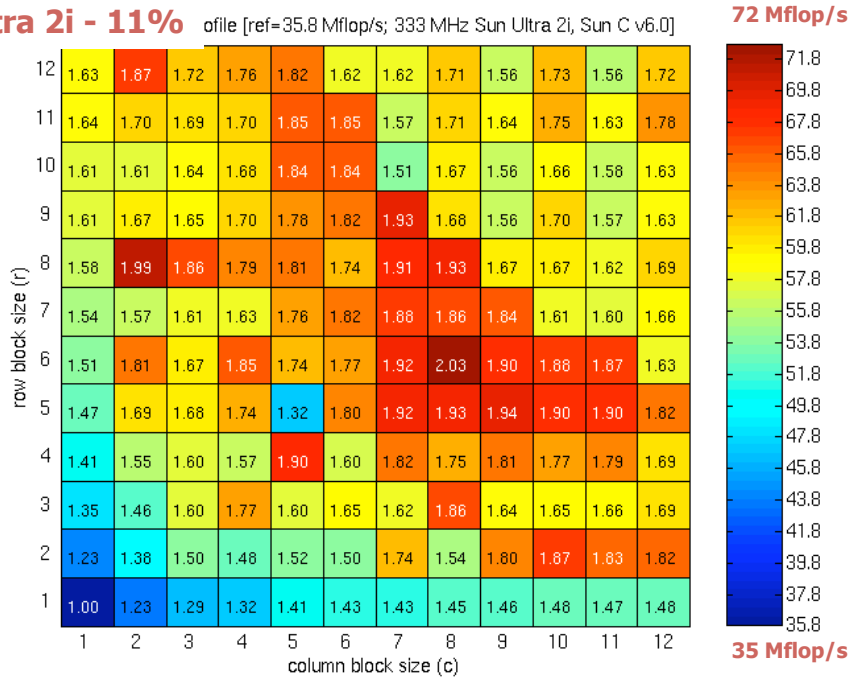
column block size (c)



190 Mflop/s

Ultra 2i - 11%

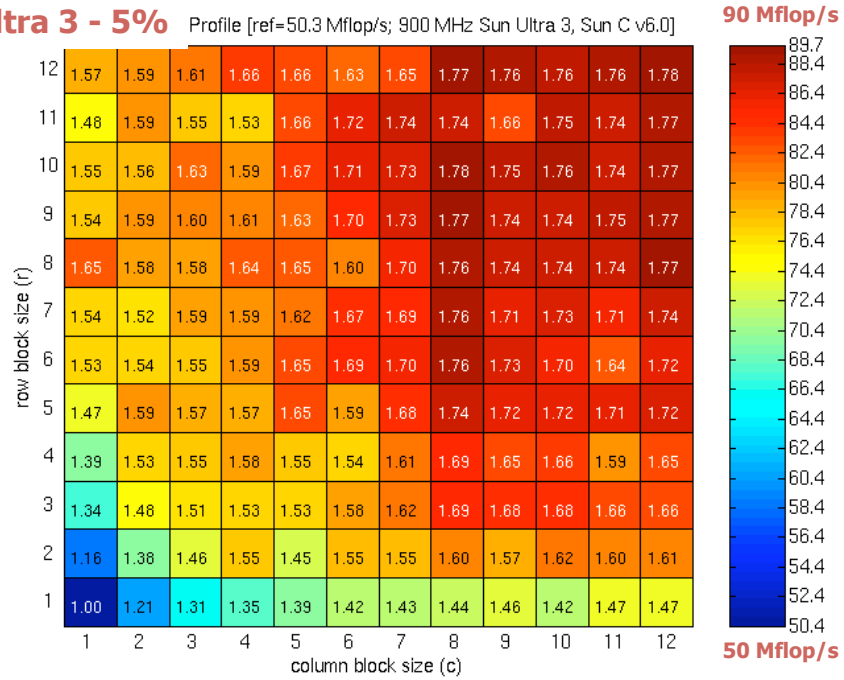
ofile [ref=35.8 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]



r \ 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%

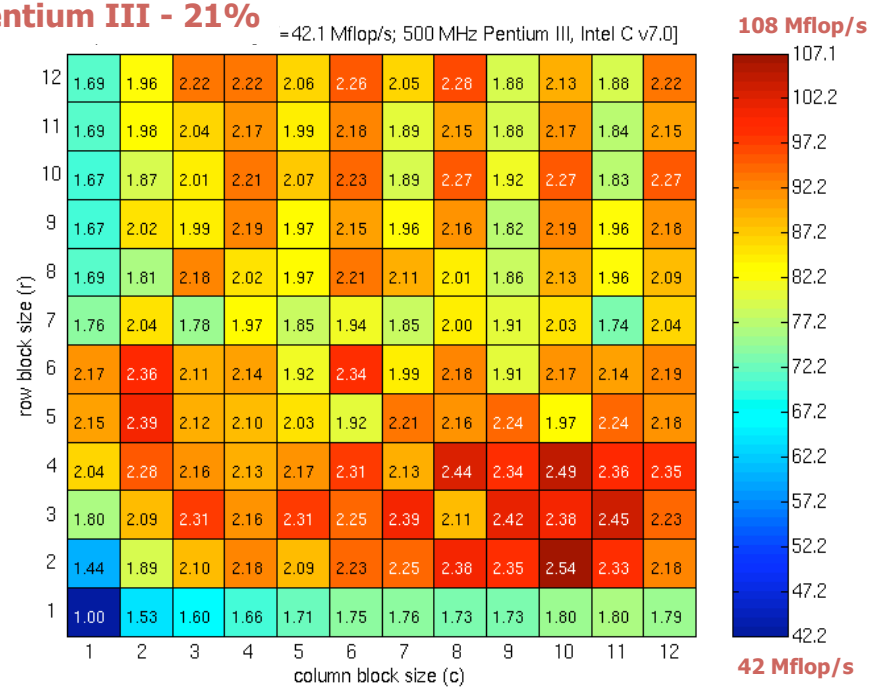
Profile [ref=50.3 Mflop/s; 900 MHz Sun Ultra 3, Sun C v6.0]



r \ 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	1.66	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%

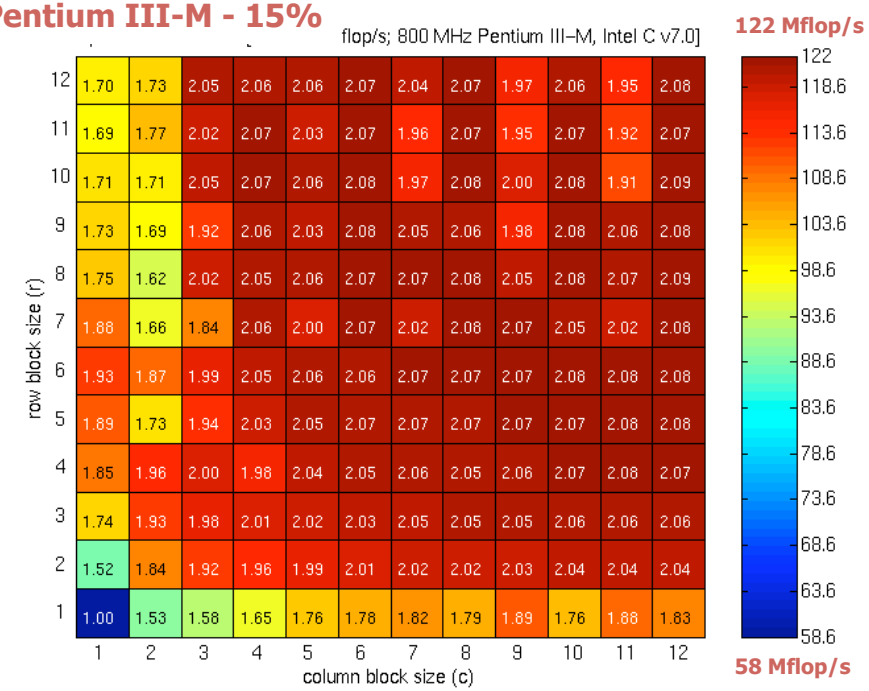
=42.1 Mflop/s; 500 MHz Pentium III, Intel C v7.0]



r \ 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

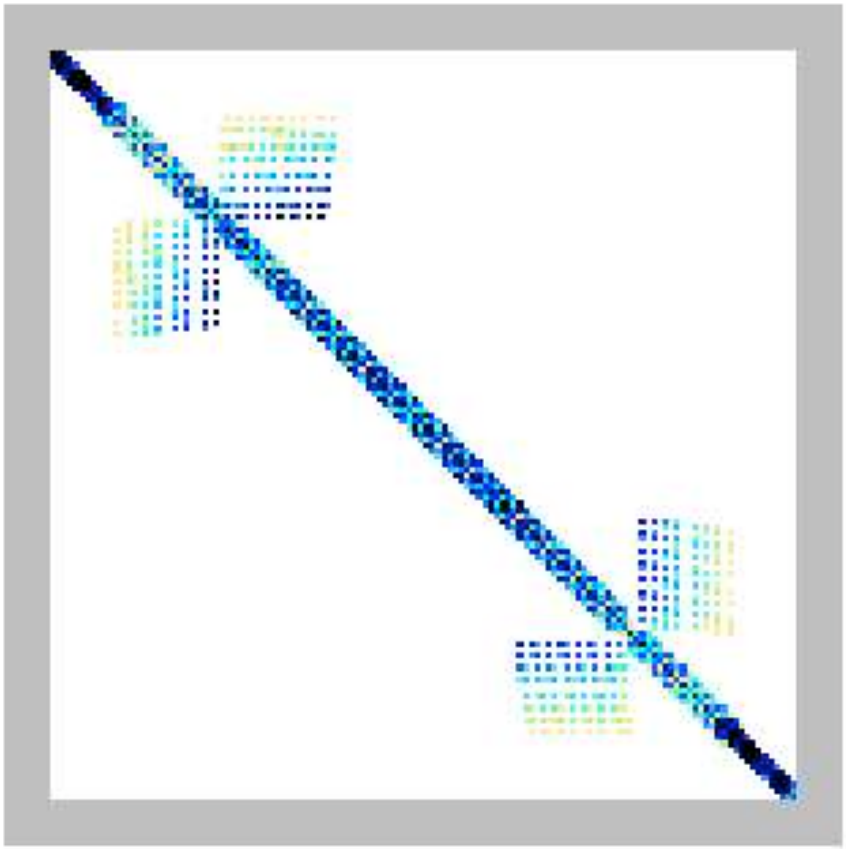
Pentium III-M - 15%

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



r \ 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

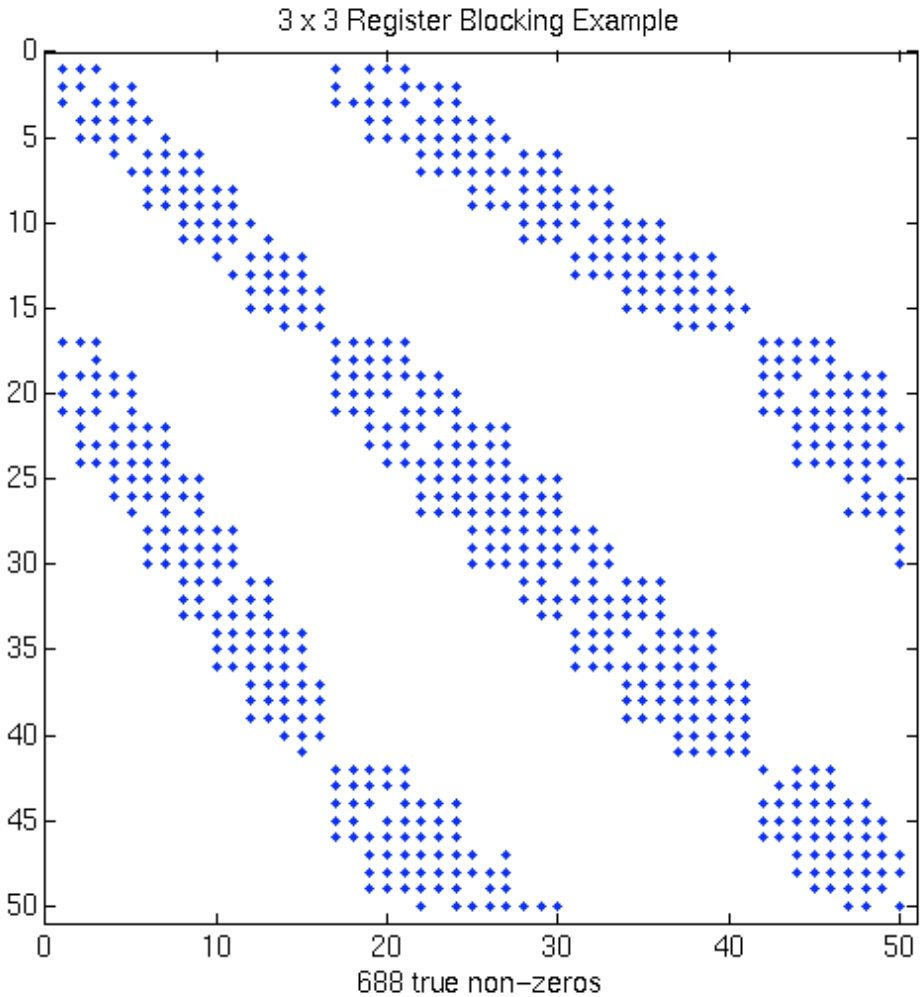
Another example of tuning challenges



- More complicated non-zero structure in general

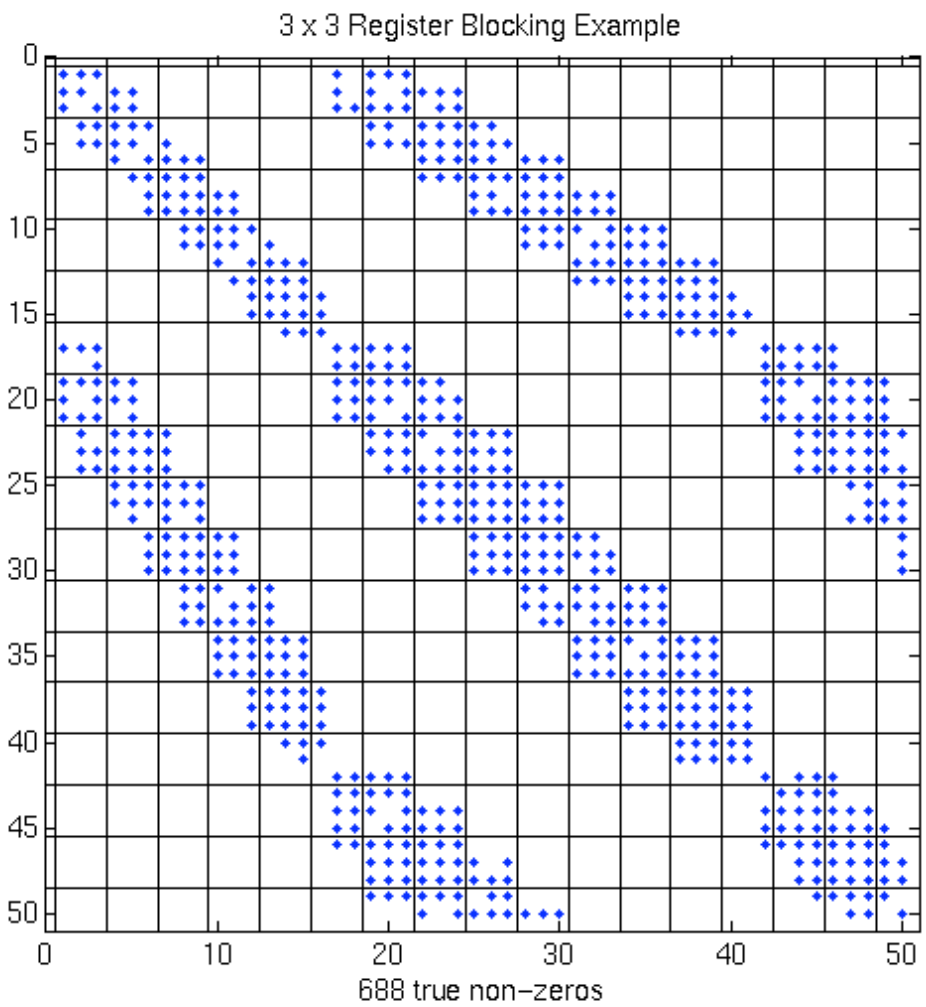
- N = 16614
- NNZ = 1.1M

Zoom in to top corner



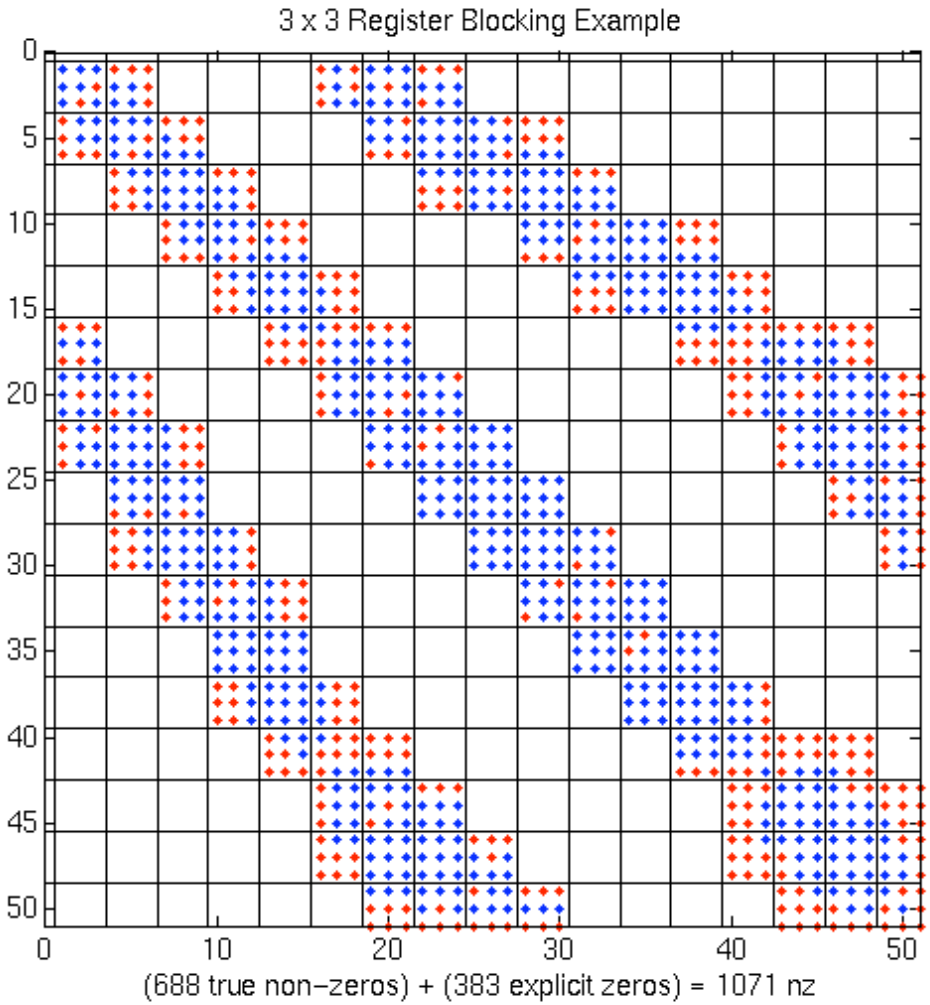
- More complicated non-zero structure in general
- N = 16614
- NNZ = 1.1M

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"

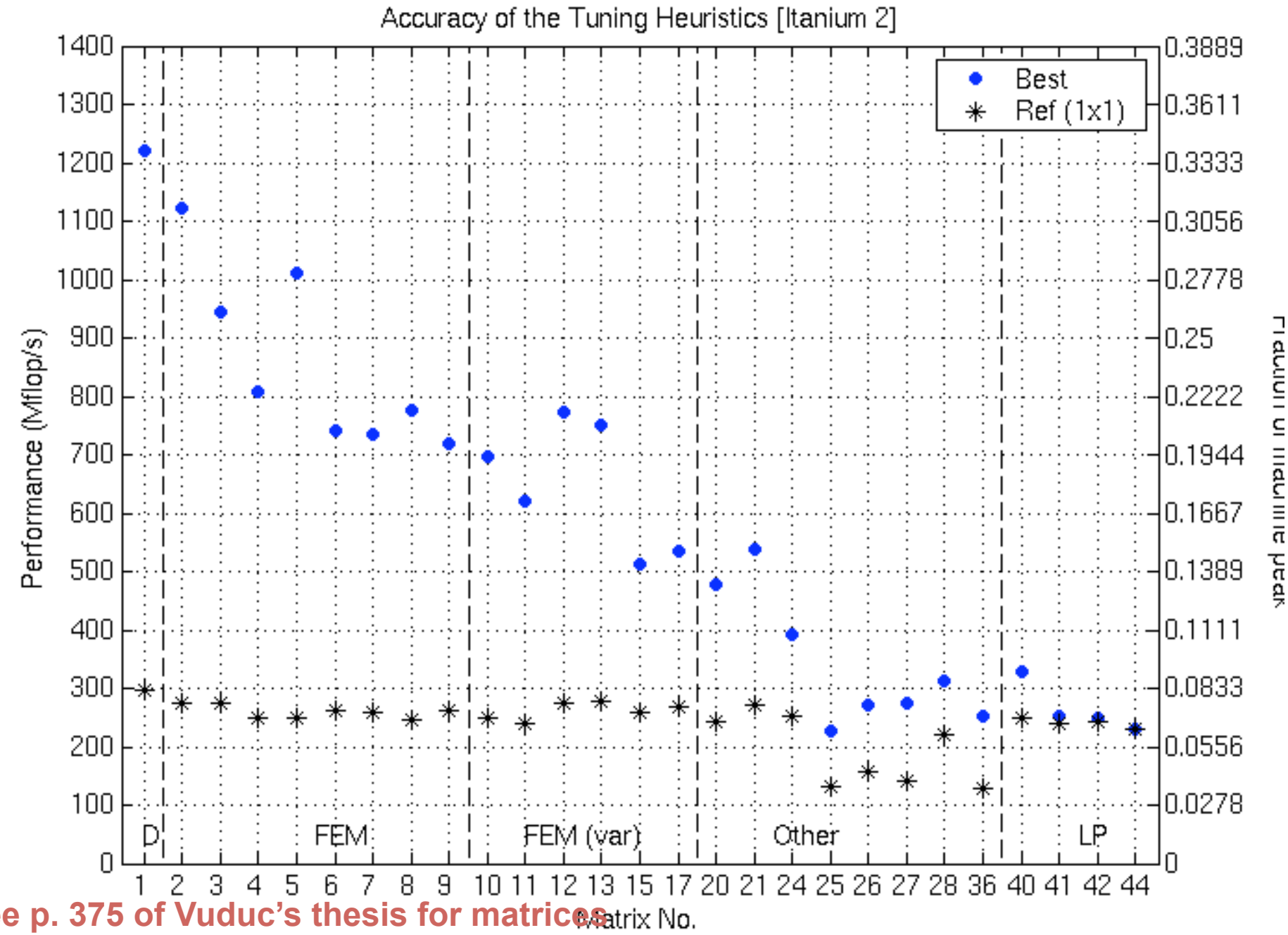
Extra Work Can Improve Efficiency!



- 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

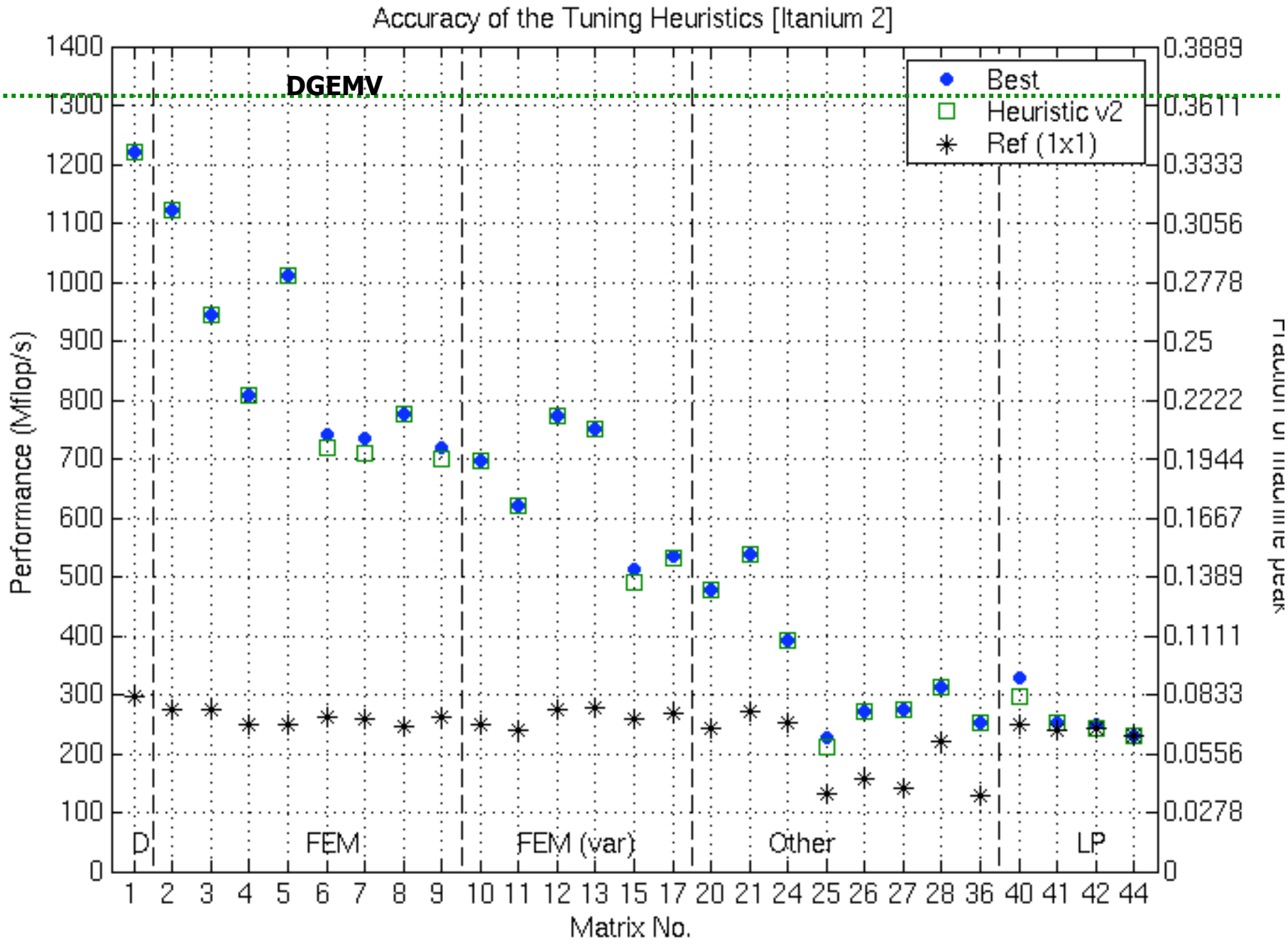
Selecting Register Block Size r x c

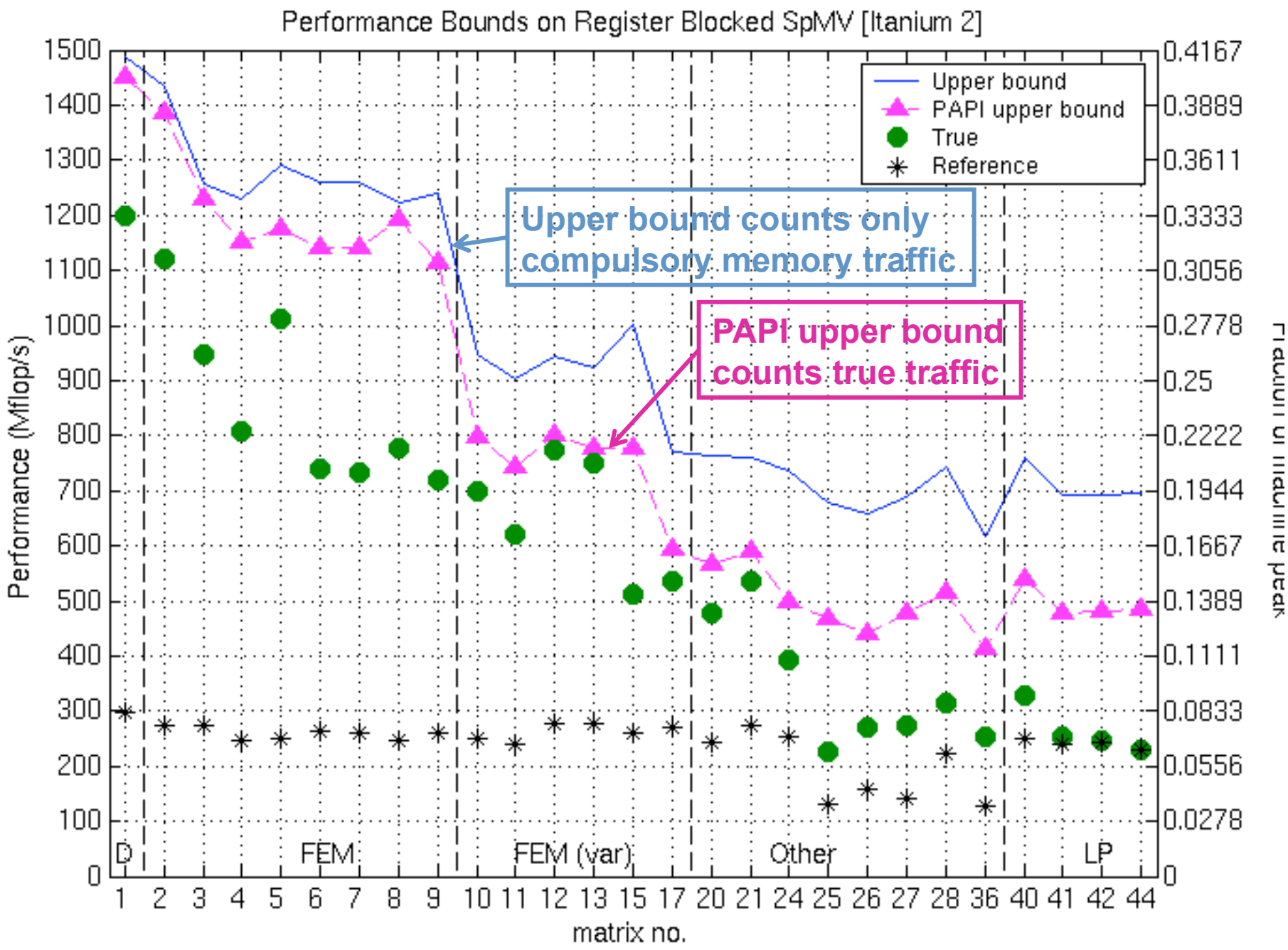
- **Off-line benchmark**
  - Precompute **Mflops(r,c)** using dense A for each r x c
  - Once per machine/architecture
- **Run-time "search"**
  - Sample *A* to estimate **Fill(r,c)** for each r x c
  - Control cost = O(s·nnz) by controlling fraction $s \in [0,1]$ sampled
  - Control *s* automatically 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")

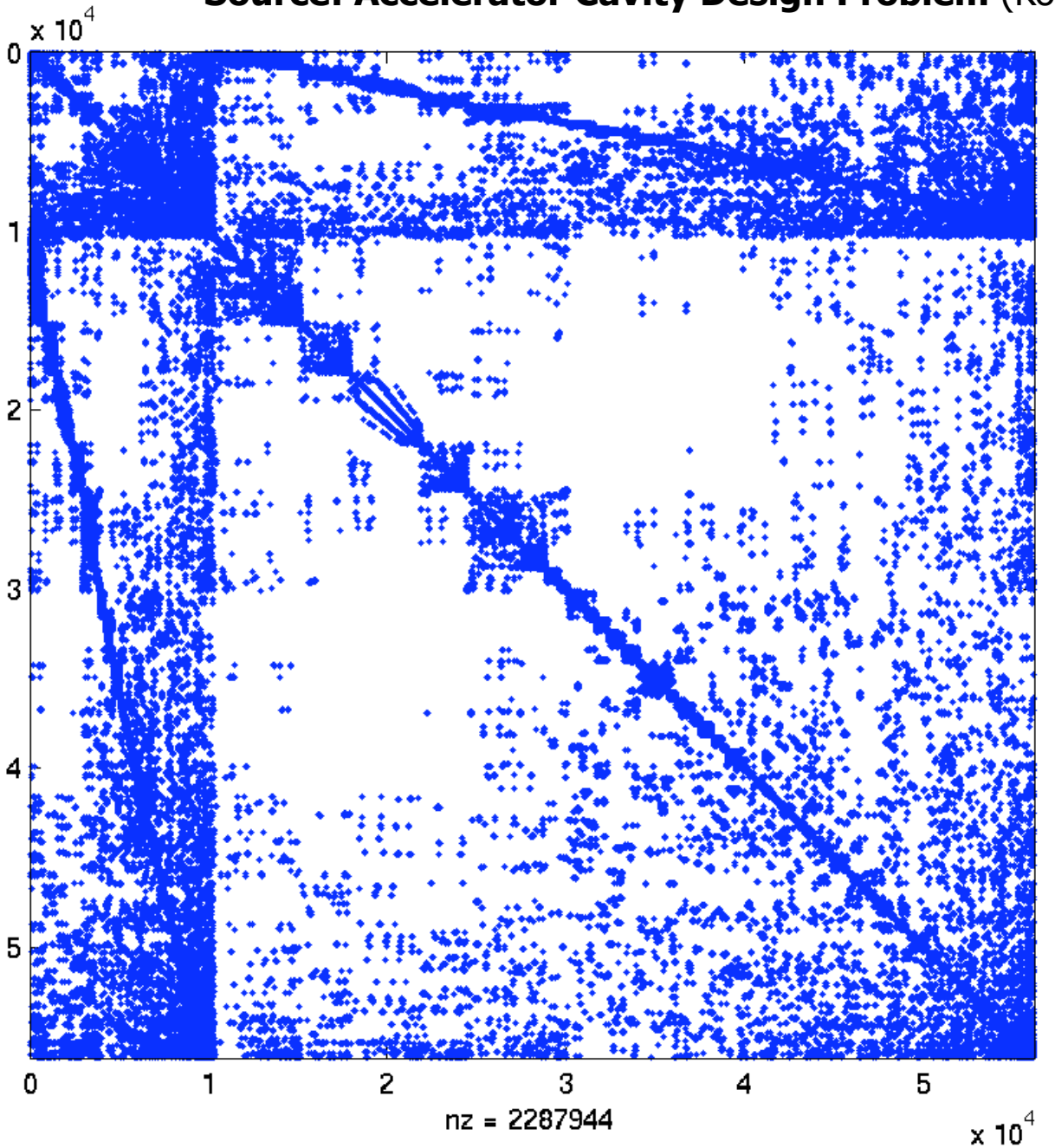

Accuracy of the Tuning Heuristics [Itanium 2]
DGEMV
Best
Heuristic v2
Ref (1x1)
Performance (Mflop/s)
Fraction of machine peak
Matrix No.
D
FEM
FEM (var)
Other
LP
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
0.3889
0.3611
0.3333
0.3056
0.2778
0.25
0.2222
0.1944
0.1667
0.1389
0.1111
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


Performance Bounds on Register Blocked SpMV [Itanium 2]
Upper bound
PAPI upper bound
True
Reference
Upper bound counts only compulsory memory traffic
PAPI upper bound counts true traffic
Performance (Mflop/s)
Fraction of machine peak
matrix no.
D
FEM
FEM (var)
Other
LP
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
0.4167
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

Summary of Other Performance Optimizations

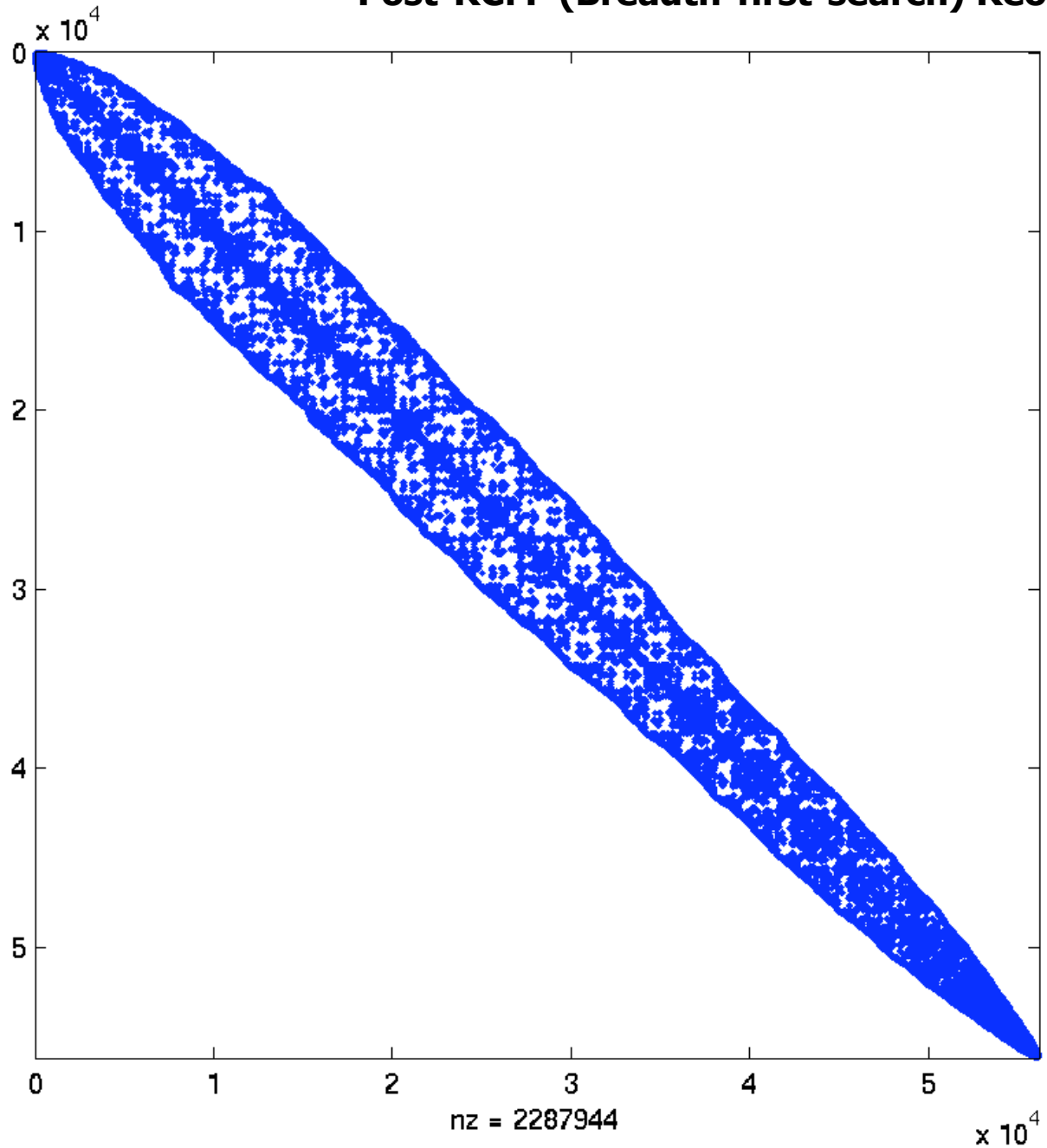
- Optimizations for SpMV
  - **Register blocking (RB)**: up to **4x** over CSR
  - **Variable block splitting**: **2.1x** over CSR, **1.8x** over RB
  - **Diagonals**: **2x** over CSR
  - **Reordering** to create dense structure + **splitting**: **2x** over CSR
  - **Symmetry**: **2.8x** over CSR, 2.6x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

Source: Accelerator Cavity Design Problem (Ko via Husbands)



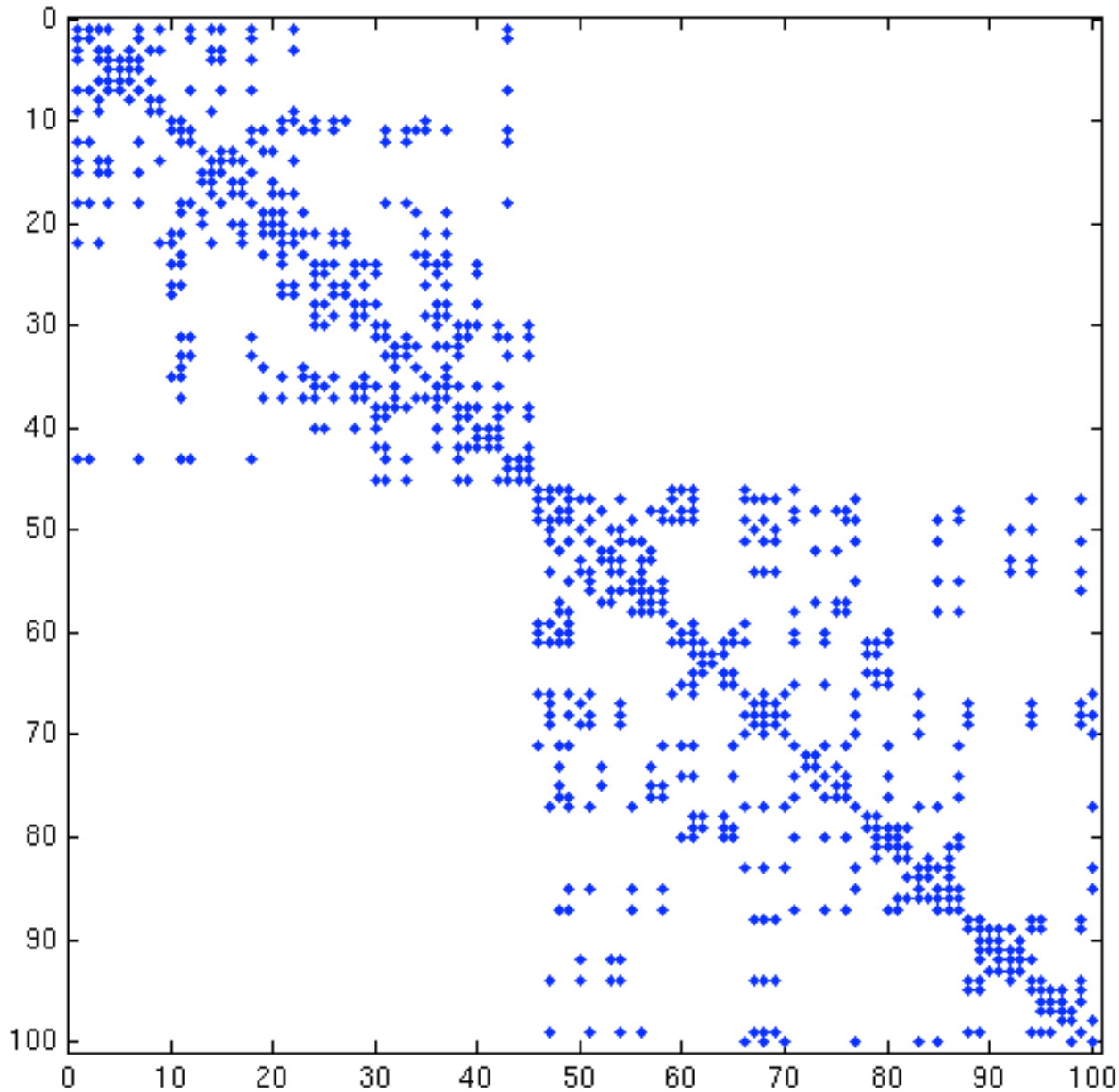
Can we reorder the rows and columns to create dense blocks, to accelerate SpMV?

Post-RCM (Breadth-first-search) Reordering



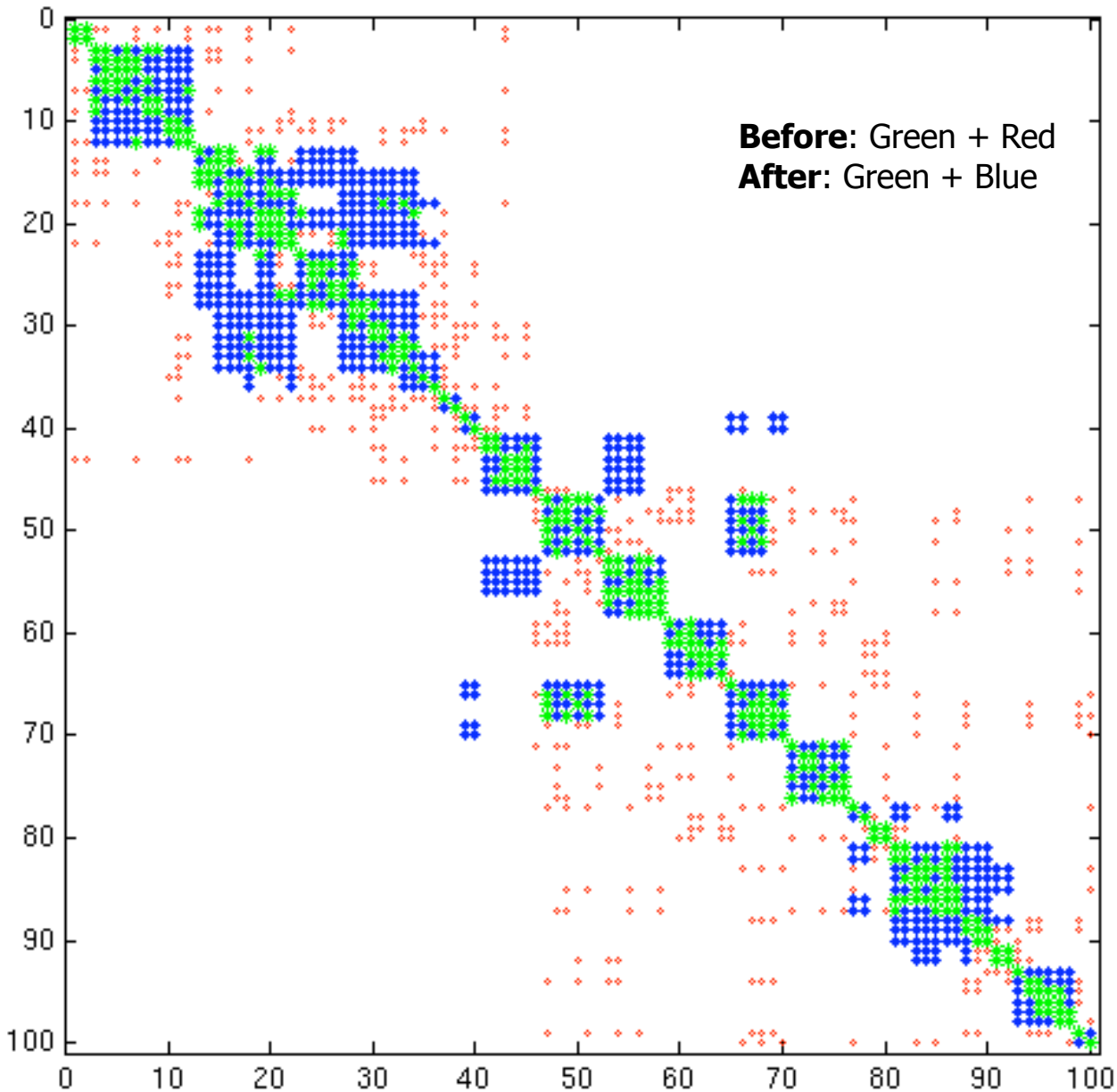
Moving nonzeros nearer the diagonal should create dense block, but let's zoom in and see…

100x100 Submatrix Along Diagonal



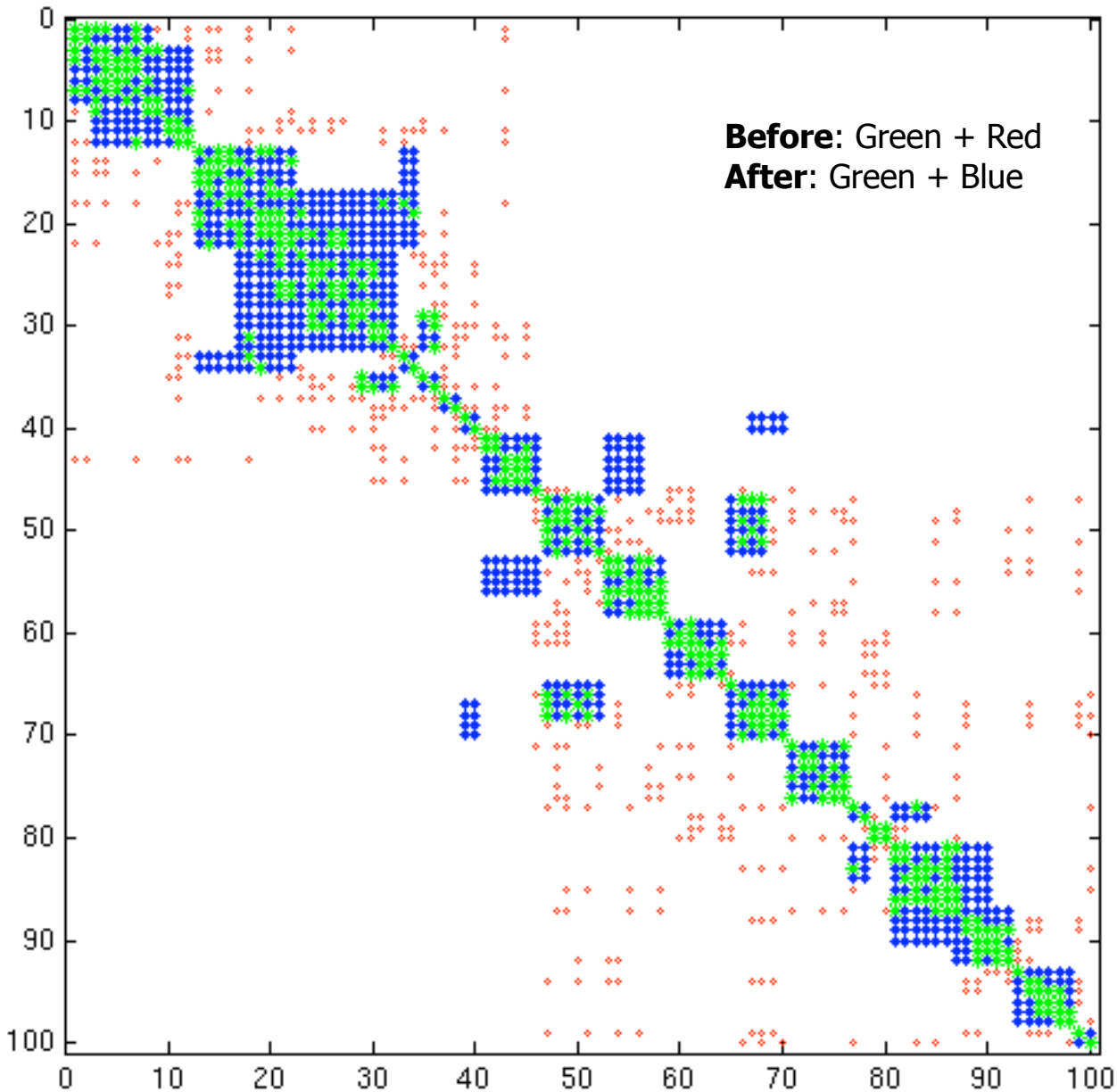
Here is the top 100x100 submatrix before RCM

"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



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

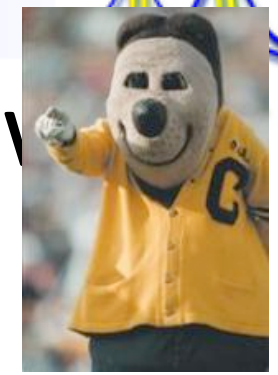
Speedups (using symmetry too):

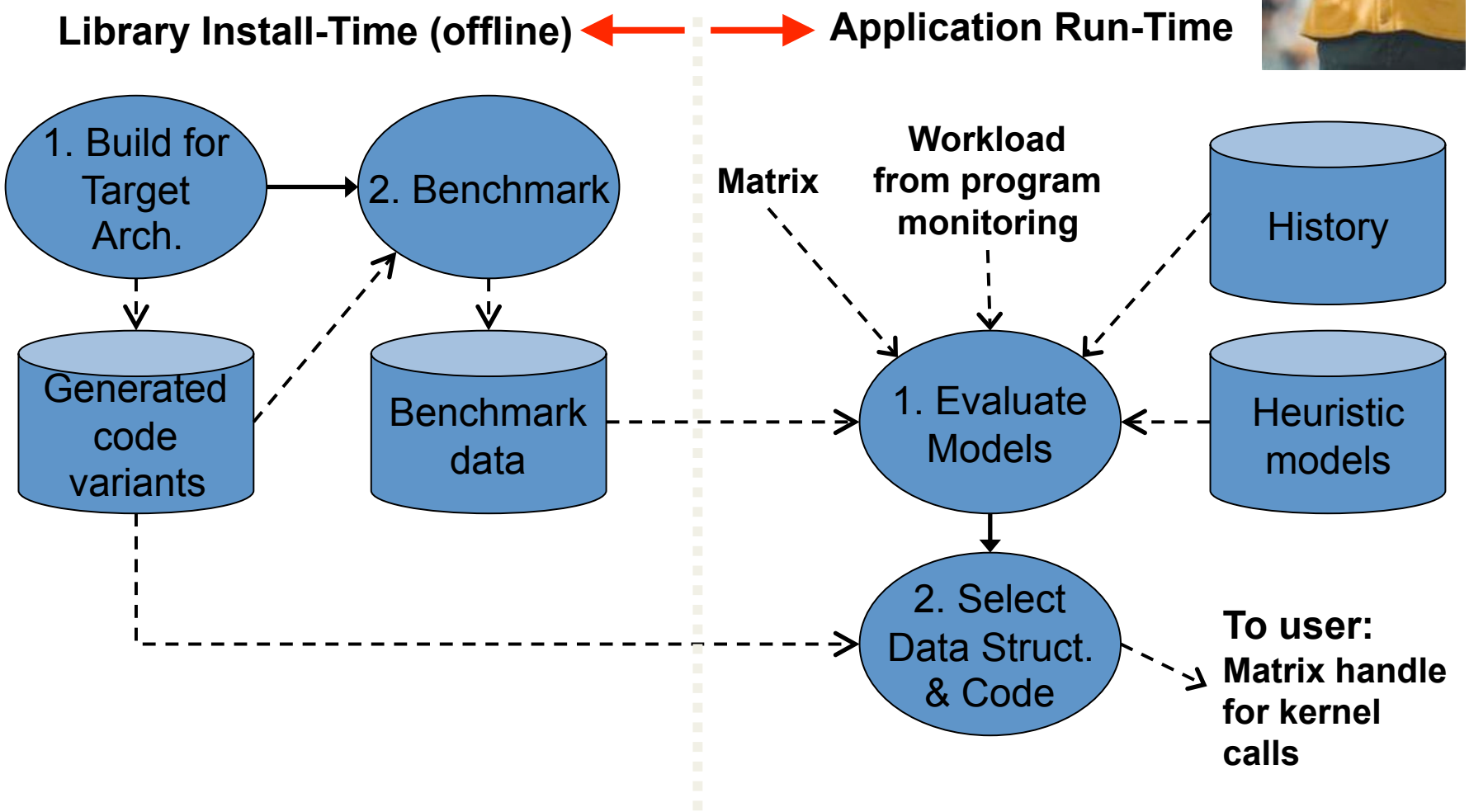
Itanium 2: 1.7x
Pentium 4: 2.1x
Power 4: 2.1x
Ultra 3: 3.3x

Optimized Sparse Kernel Interface - OSKI

- Provides sparse kernels automatically tuned for user's matrix & machine
  - BLAS-style functionality: SpMV, Ax & A^Ty, TrSV
  - Hides complexity of run-time tuning
  - Includes new, faster locality-aware kernels: A^TAx, A^kx
- Faster than standard implementations
  - Up to 4x faster matvec, 1.8x trisolve, 4x A^TA*x
- For "advanced" users & solver library writers
  - Available as stand-alone library (OSKI 1.0.1h, 6/07)
  - Available as PETSc extension (OSKI-PETSc .1d, 3/06)
  - Bebop.cs.berkeley.edu/oski
- Future work: add multicore

How the OSKI Tunes (Overvie





Extensibility: Advanced users may write & dynamically add “Code variants” and “Heuristic models” to system.

How to Call OSKI: 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 OSKI: 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 OSKI wrappers around this data */
oski_matrix_t A_tunable = oski_CreateMatCSR(ptr, ind, val, num_rows,
   num_cols, SHARE_INPUTMAT, …);
oski_vecview_t x_view = oski_CreateVecView(x, num_cols, UNIT_STRIDE);
oski_vecview_t y_view = oski_CreateVecView(y, num_rows, UNIT_STRIDE);

/* 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 OSKI: 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 OSKI wrappers around this data */
oski_matrix_t A_tunable = oski_CreateMatCSR(ptr, ind, val, num_rows,
   num_cols, SHARE_INPUTMAT, …);
oski_vecview_t x_view = oski_CreateVecView(x, num_cols, UNIT_STRIDE);
oski_vecview_t y_view = oski_CreateVecView(y, num_rows, UNIT_STRIDE);

/* Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
   oski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view); /* Step 2 */
```

How to Call OSKI: Tune with Explicit Hints

- User calls "tune" routine
  - May provide explicit tuning hints (OPTIONAL)

```
oski_matrix_t A_tunable = oski_CreateMatCSR( … );
   /* … */
/* Tell OSKI we will call SpMV 500 times (workload hint) */
oski_SetHintMatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view, 500);
/* Tell OSKI we think the matrix has 8x8 blocks (structural hint) */
oski_SetHint(A_tunable, HINT_SINGLE_BLOCKSIZE, 8, 8);

oski_TuneMat(A_tunable); /* Ask OSKI to tune */

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

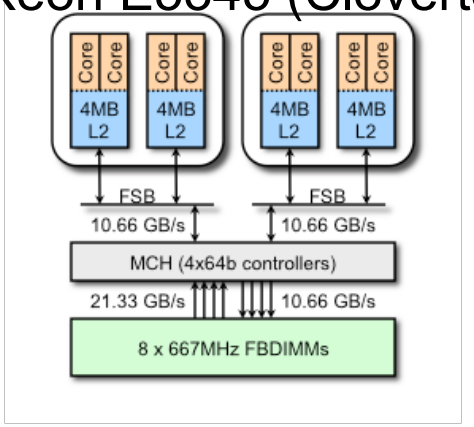
How the User Calls OSKI: Implicit Tuning

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

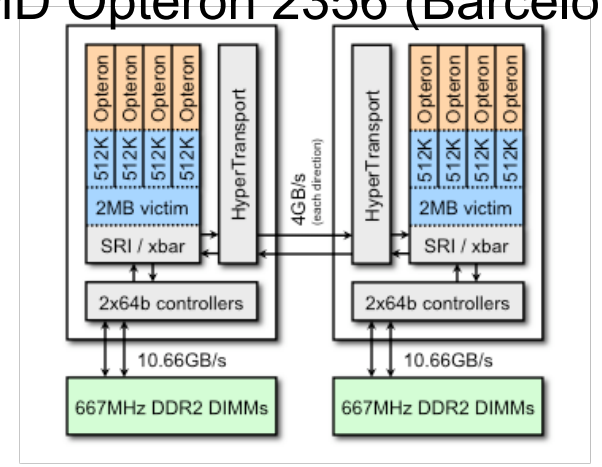
```
oski_matrix_t A_tunable = oski_CreateMatCSR( … );
   /* … */

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

Multicore SMPs Used for Tuning SpMV

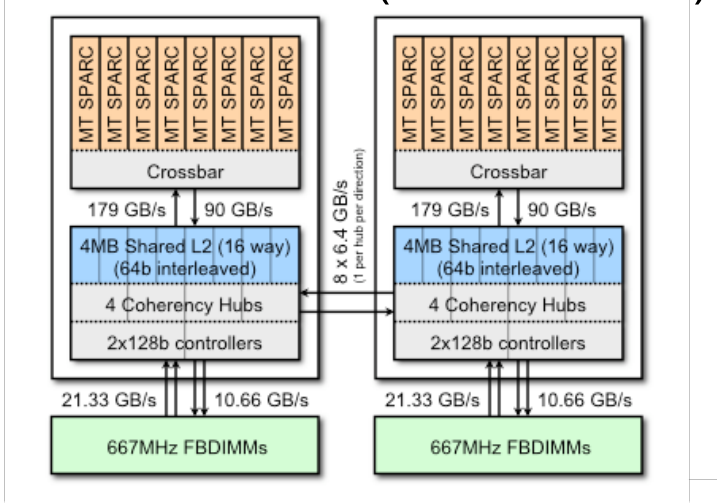
Intel Xeon E5345 (Clovertown)



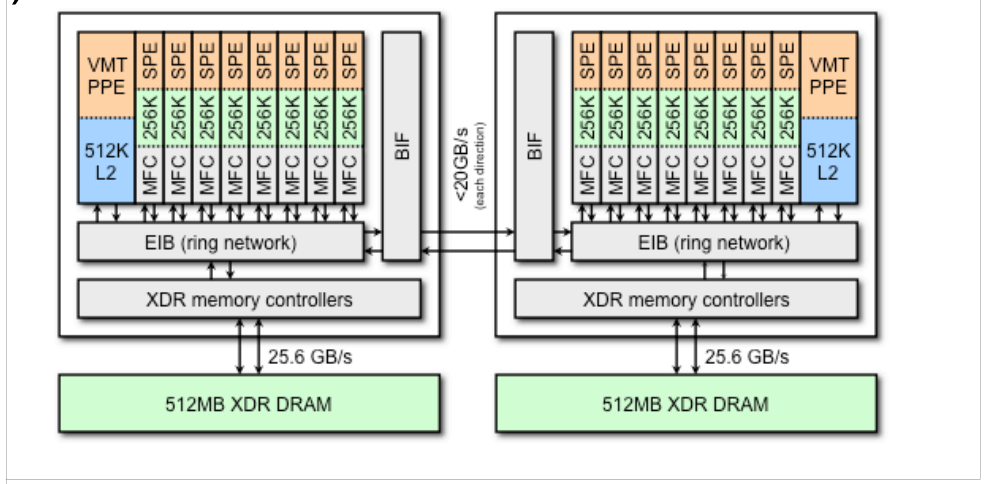
AMD Opteron 2356 (Barcelona)



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

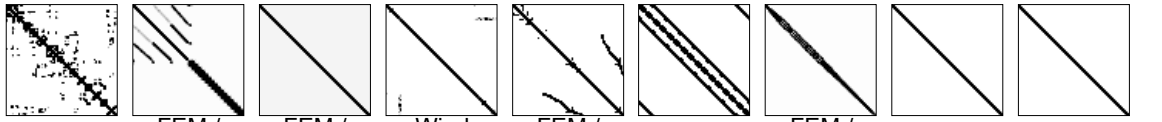
Set of 14 test matrices

- 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

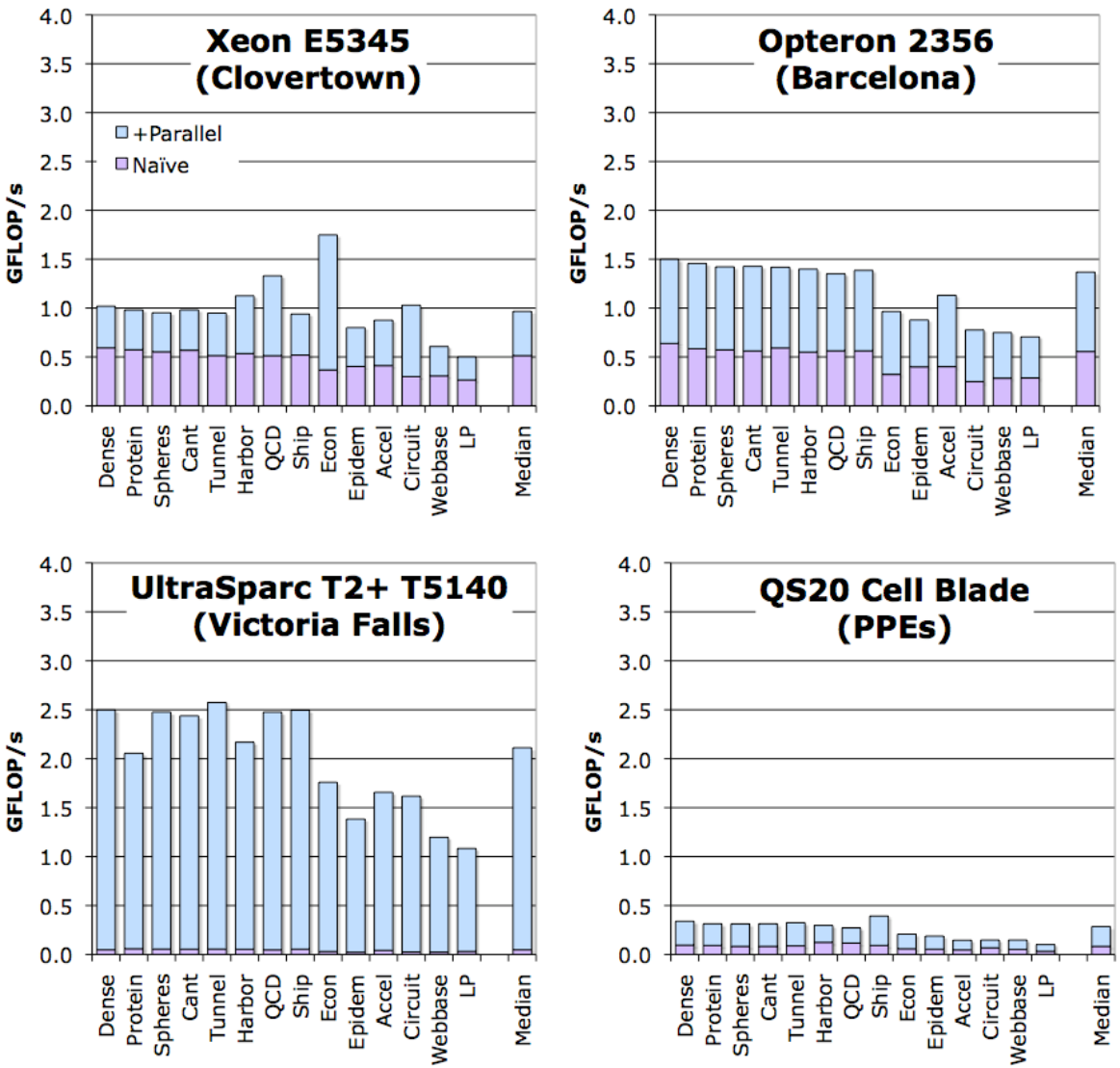
Poorly Structured hodgepodge

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

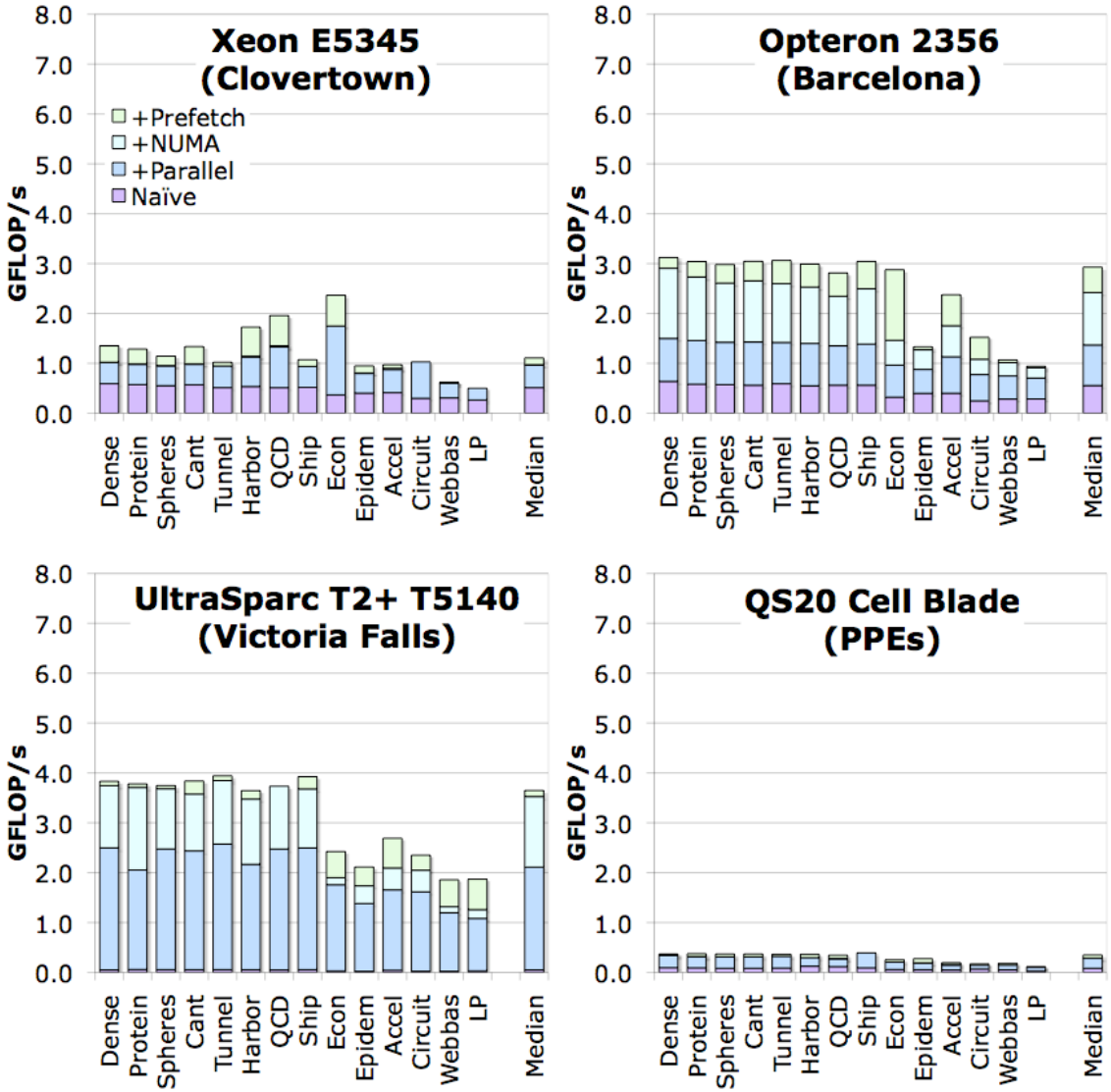
8 8

128 16

Naïve Pthreads

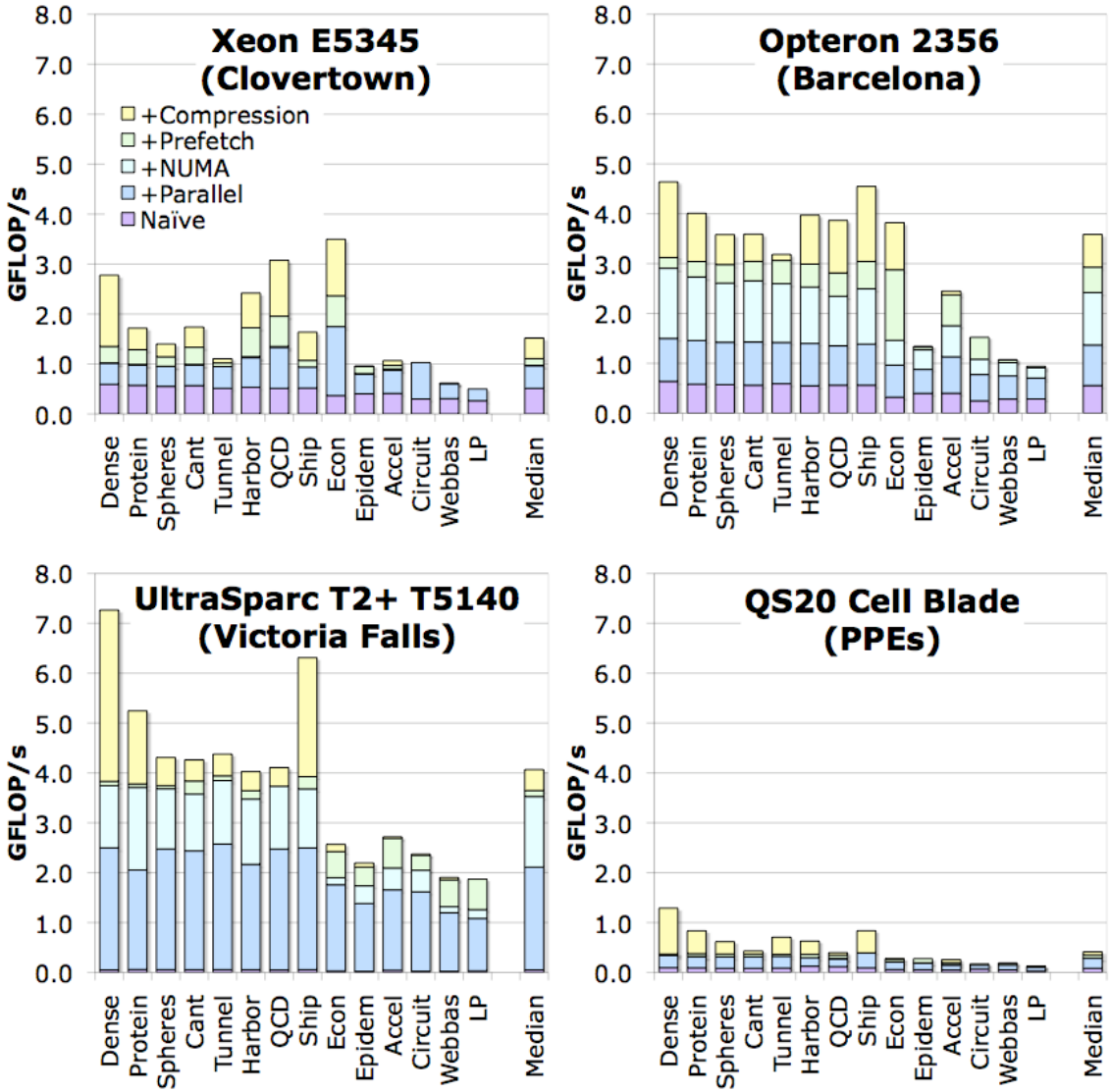
Naïve

SpMV Performance: NUMA and Software Prefetching



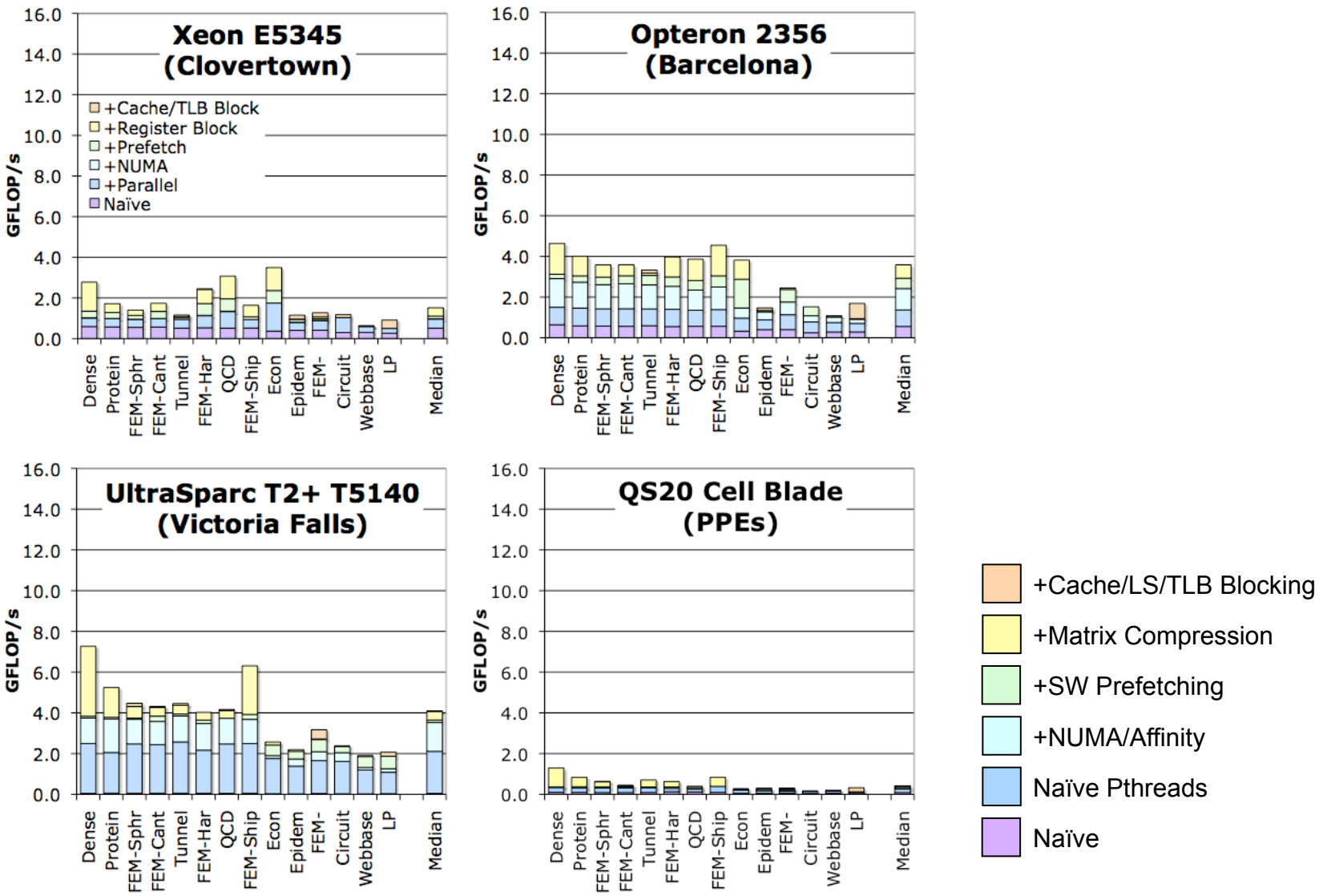
- NUMA-aware allocation is essential on NUMA SMPs.
- Explicit software prefetching can boost bandwidth and change cache replacement policies
- used **exhaustive** search

SpMV Performance: "Matrix Compression"

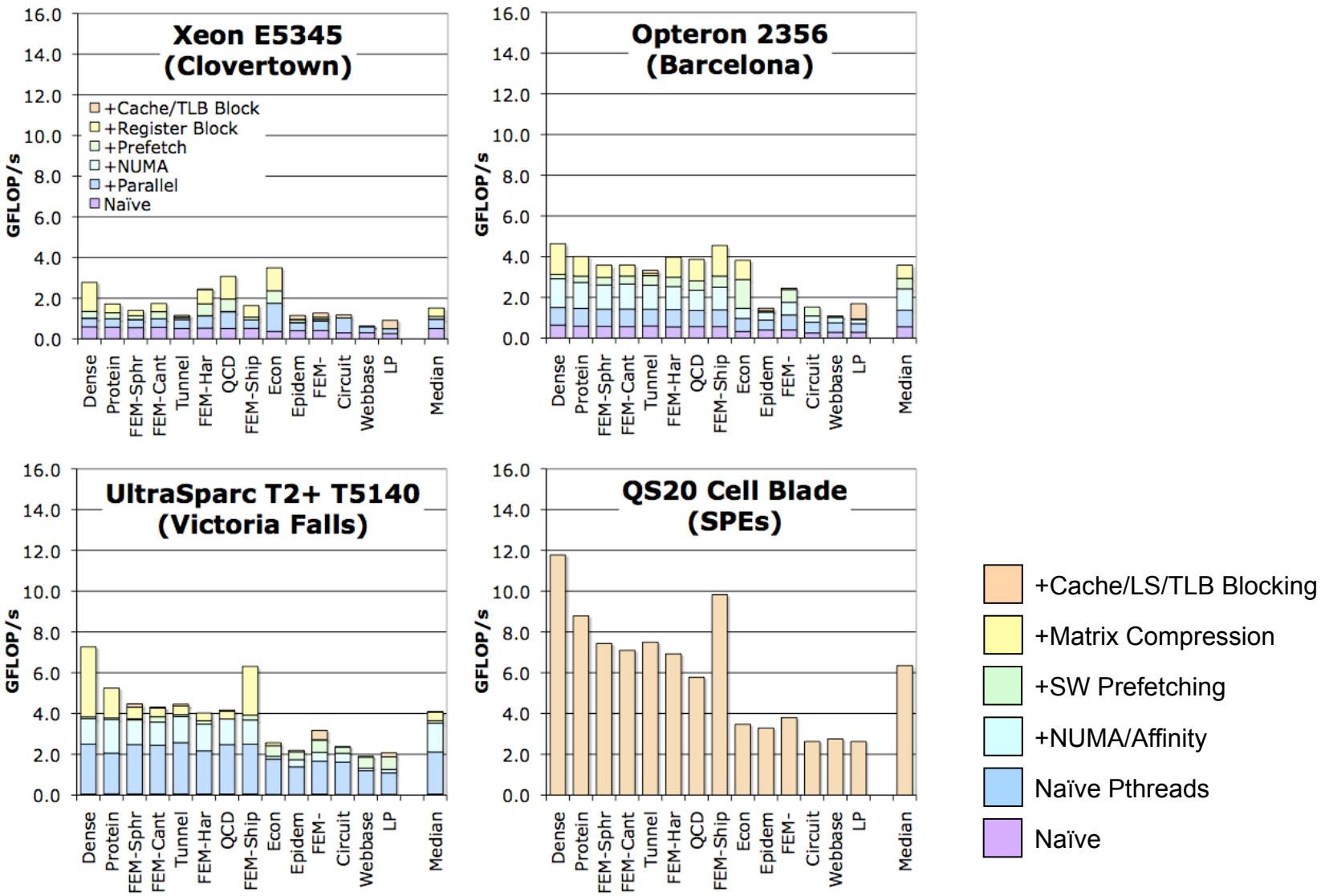


- Compression includes
  - register blocking
  - other formats
  - smaller indices
- Use **heuristic** rather than search

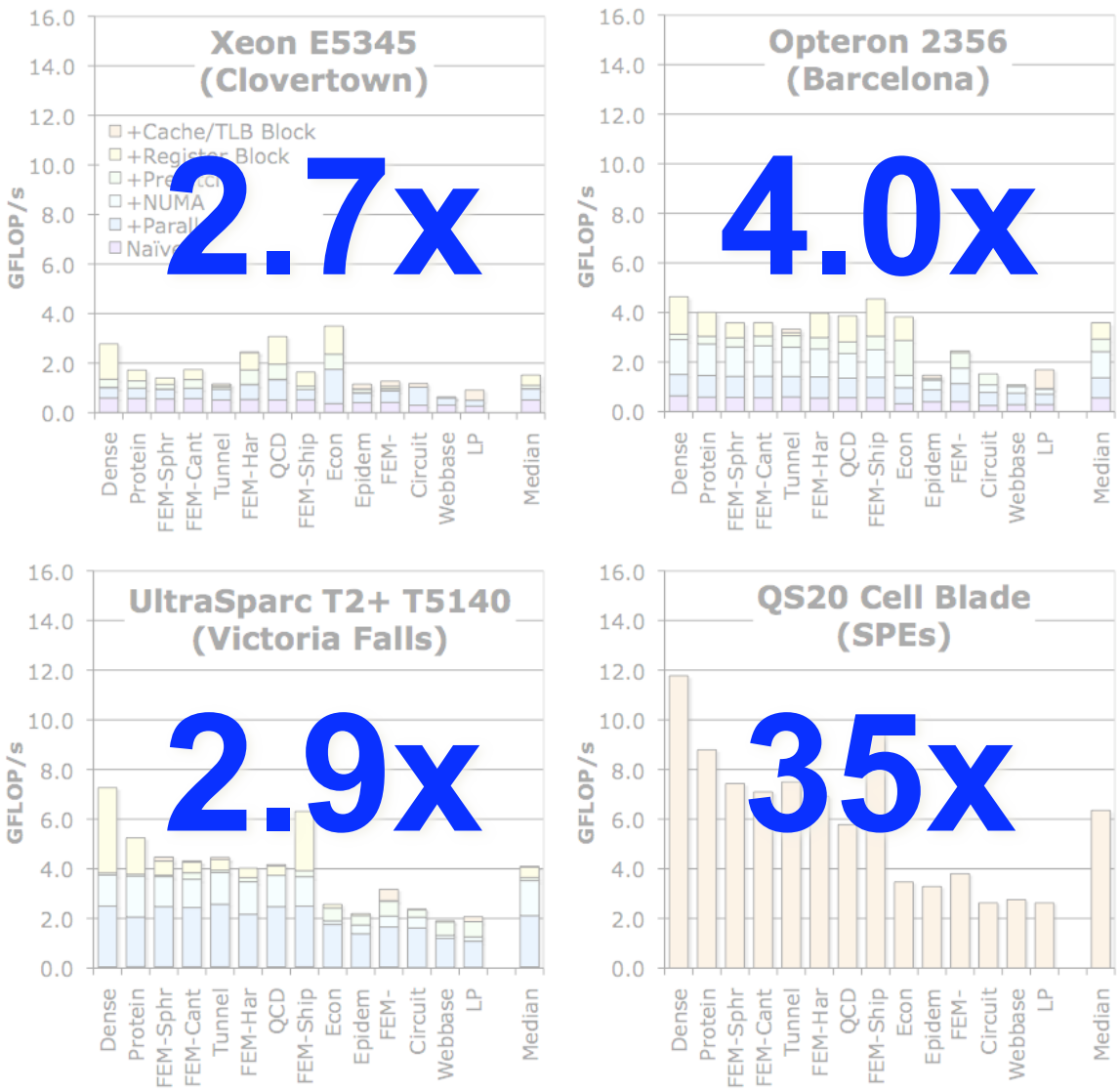
SpMV Performance: cache and TLB blocking



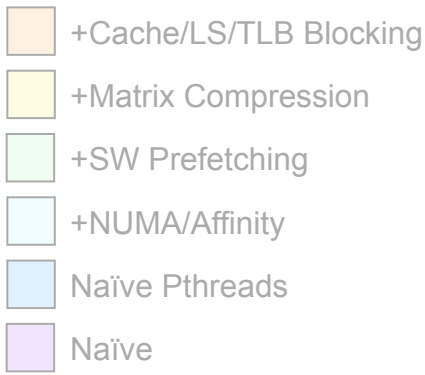
SpMV Performance: Architecture specific optimizations



SpMV Performance: max speedup



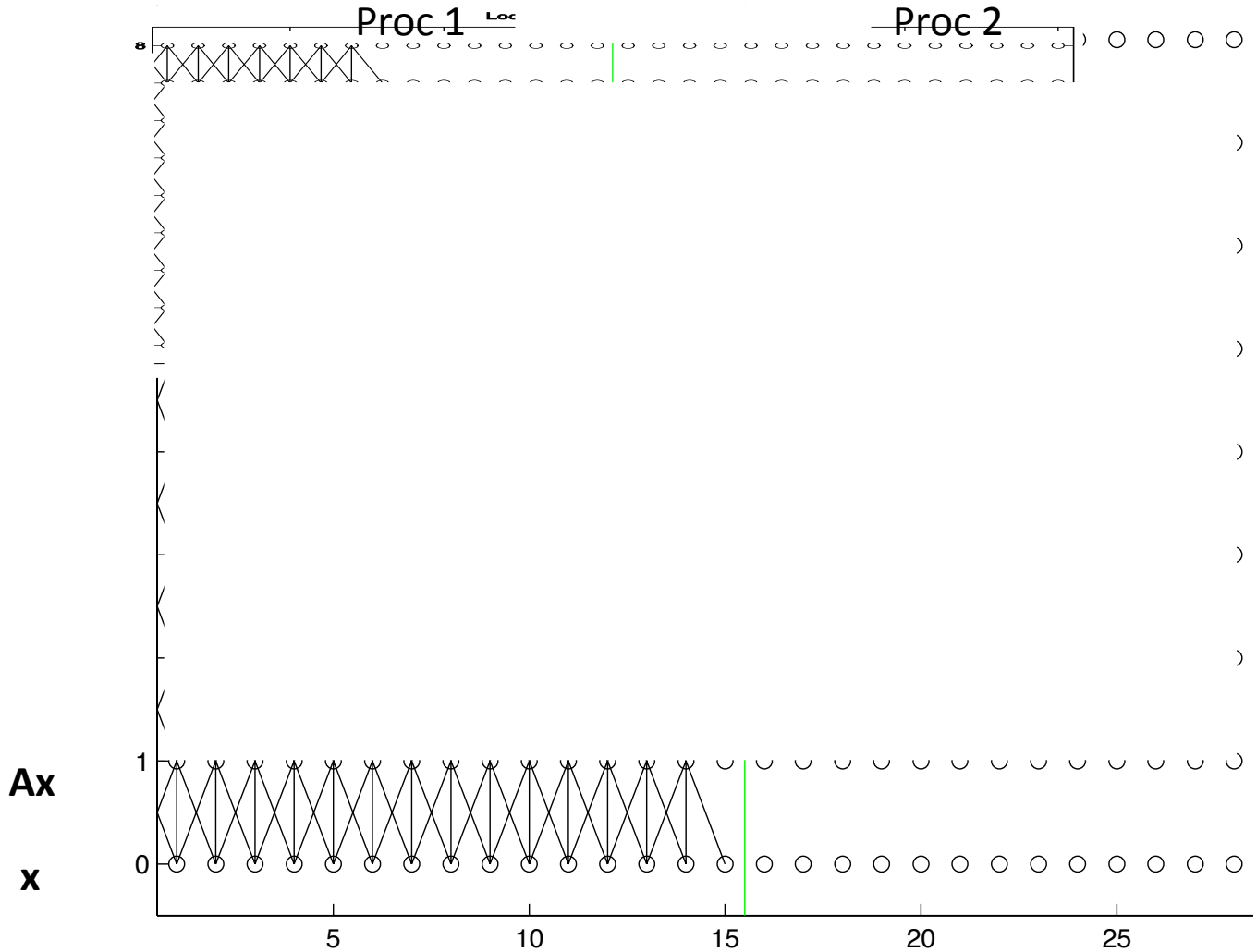
- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?



Avoiding Communication in Sparse Linear Algebra

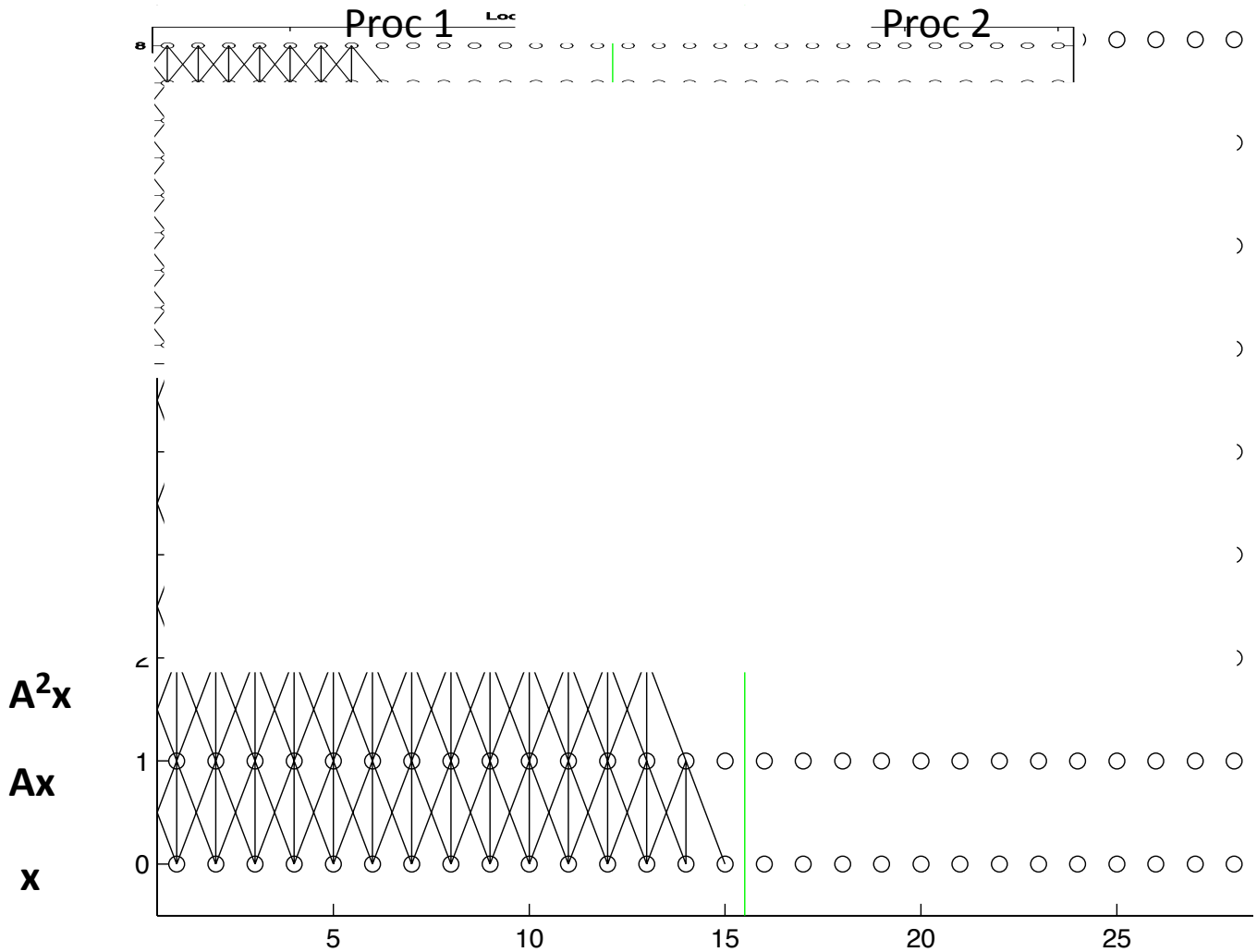
- k-steps of typical iterative solver for Ax=b or Ax=λx
  - Does k SpMVs with starting vector (eg with b, if solving Ax=b)
  - Finds "best" solution among all linear combinations of these k+1 vectors
  - Many such "Krylov Subspace Methods"
    - Conjugate Gradients, GMRES, Lanczos, Arnoldi, …
- Goal: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

Locally Dependent Entries for [x,Ax], A tridiagonal, 2 processors



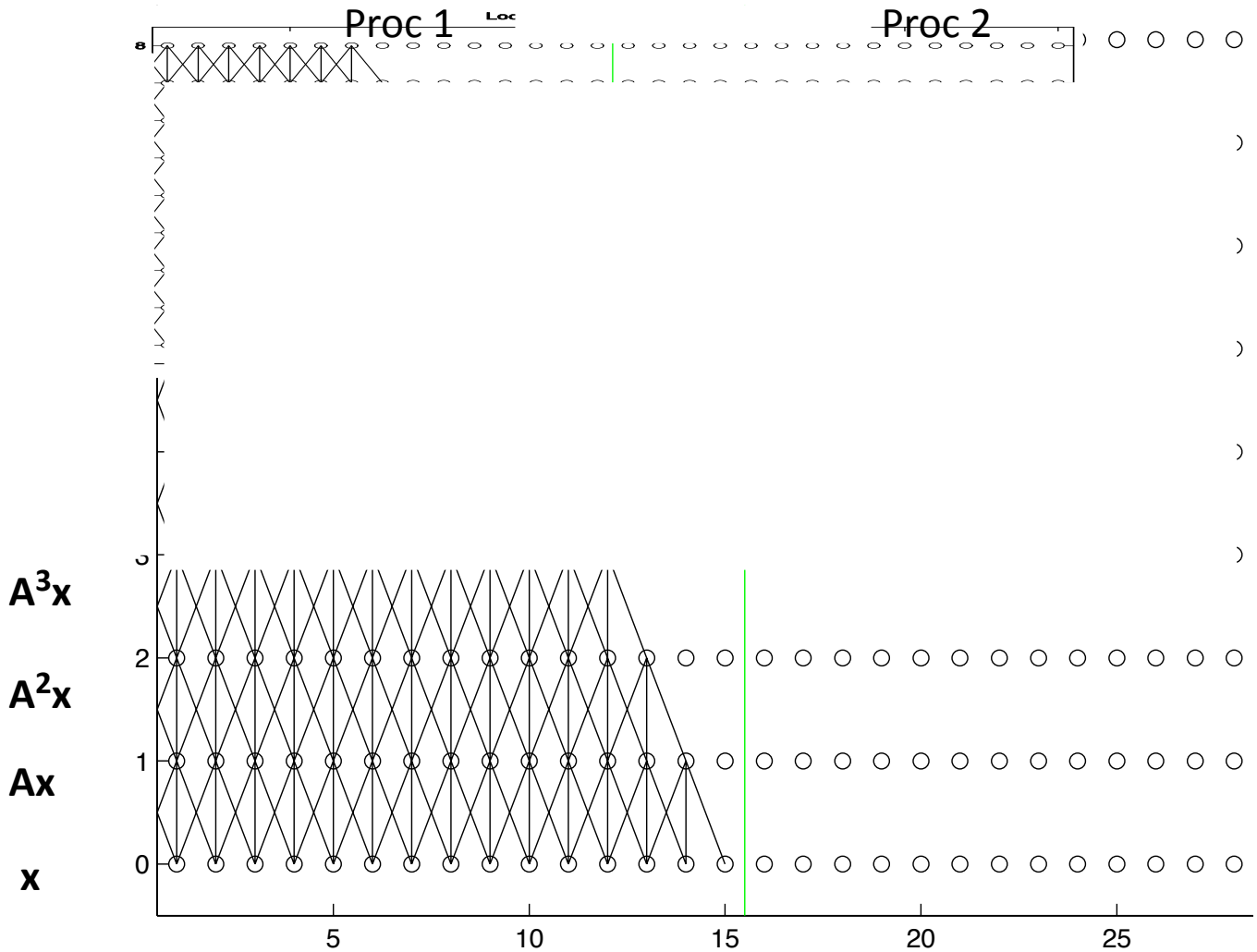
Can be computed without communication

Locally Dependent Entries for [x,Ax,A^2x], A tridiagonal, 2 processors



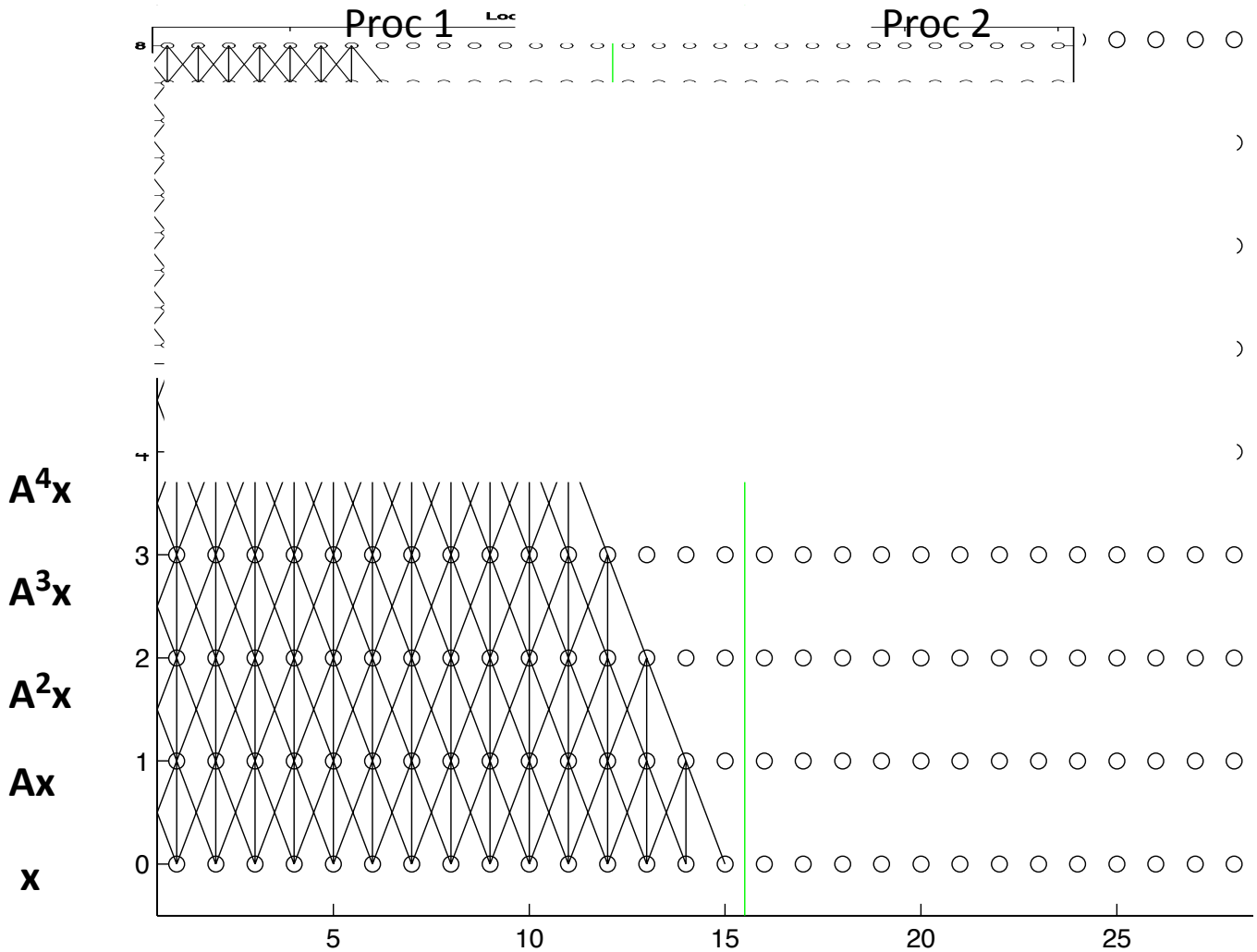
Can be computed without communication

Locally Dependent Entries for [x,Ax,…,A^3x], A tridiagonal, 2 processors



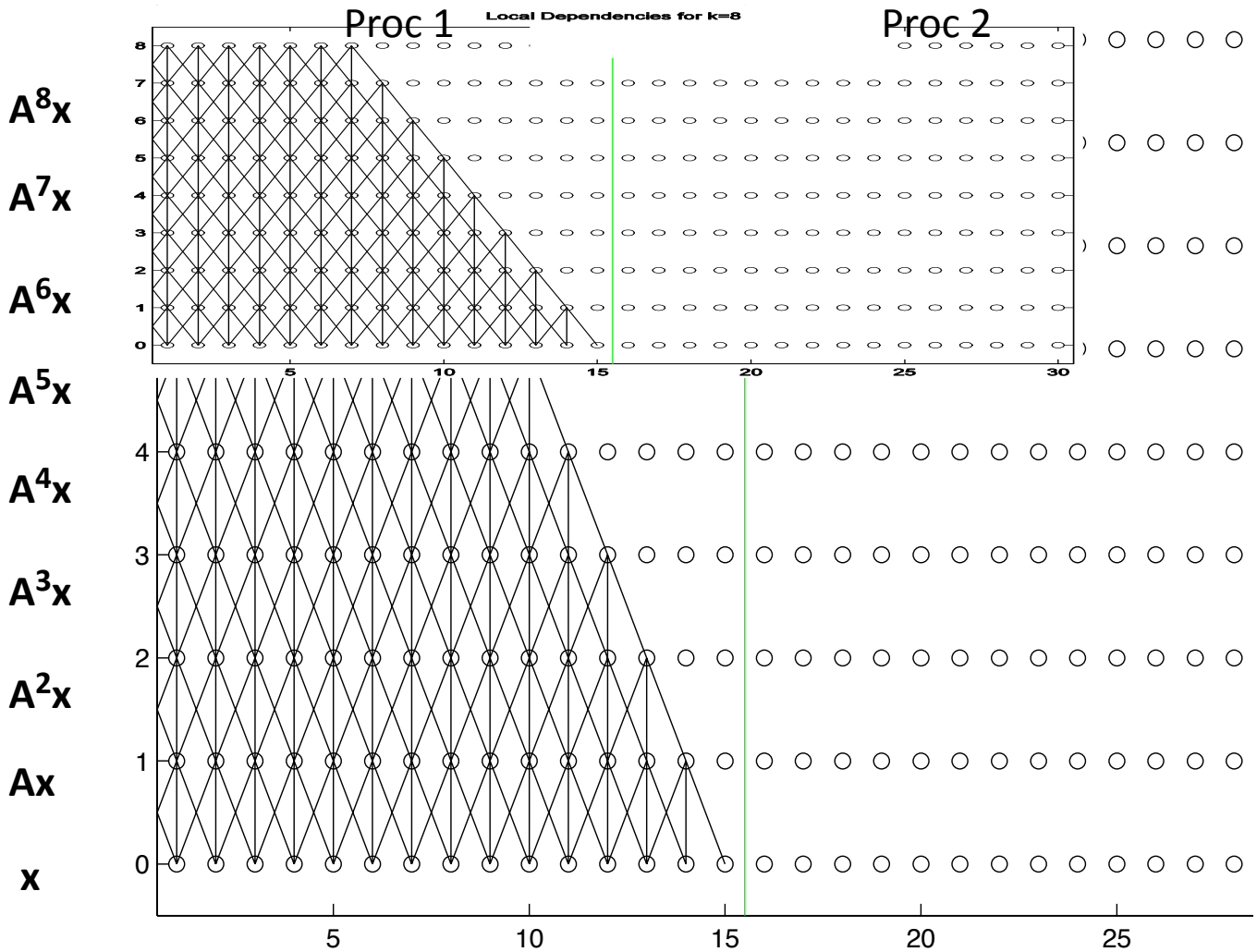
Can be computed without communication

Locally Dependent Entries for [x,Ax,…,A^4x], A tridiagonal, 2 processors



Can be computed without communication

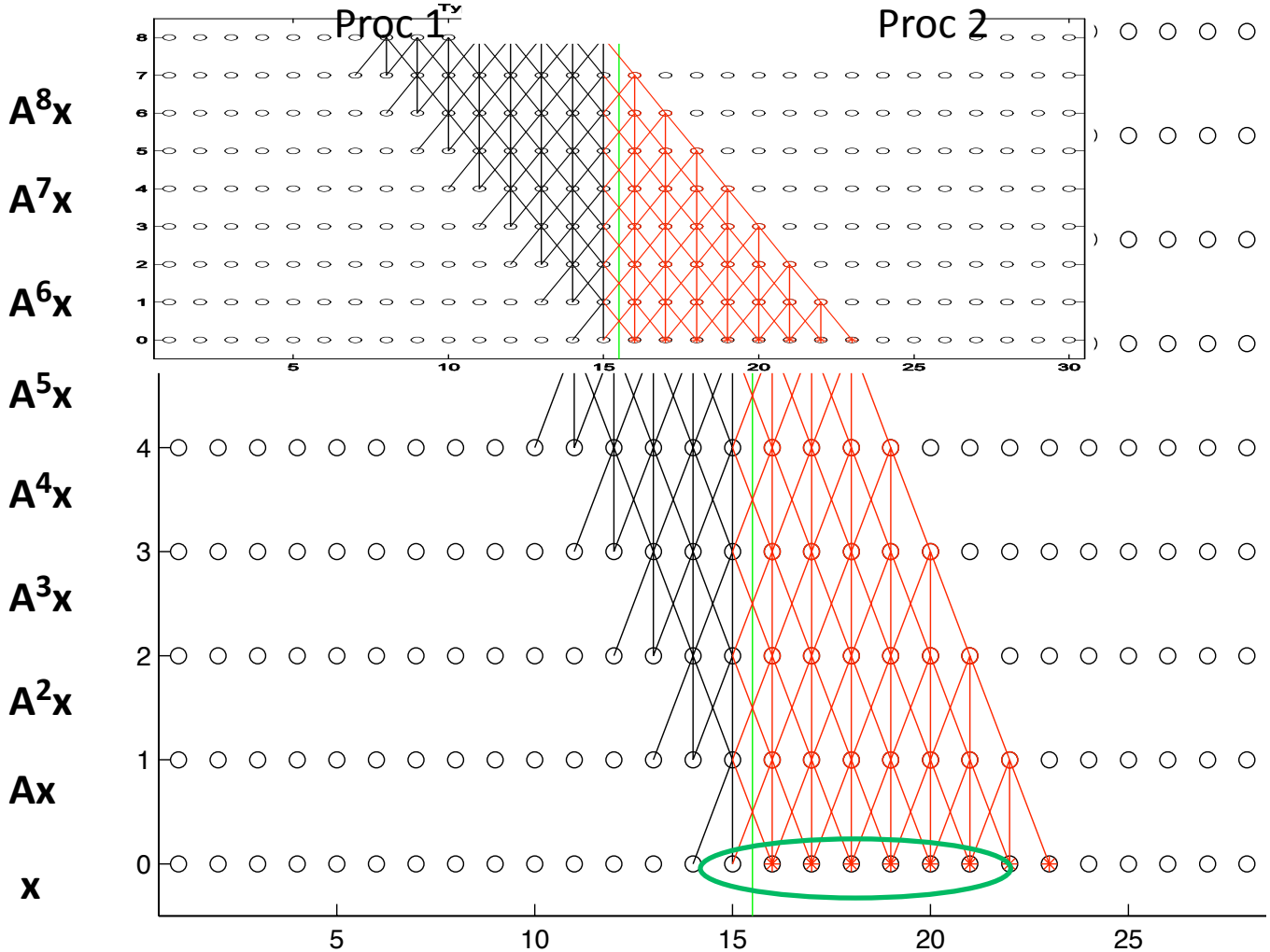
Locally Dependent Entries for $[x, Ax, \ldots, A^8x]$, A tridiagonal, 2 processors



Can be computed without communication

k=8 fold reuse of A

Remotely Dependent Entries for [x,Ax,…,A^{8}x], A tridiagonal, 2 processors

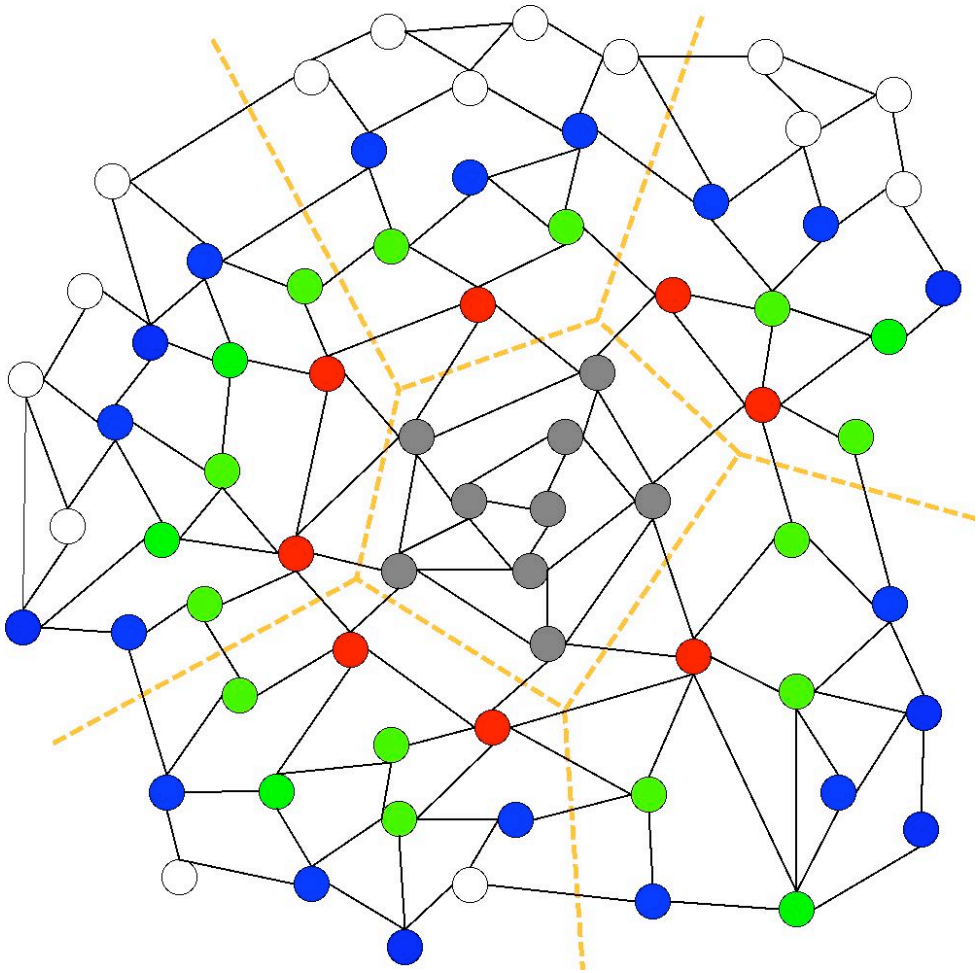


One message to get data needed to compute remotely dependent entries, ***not k=8***

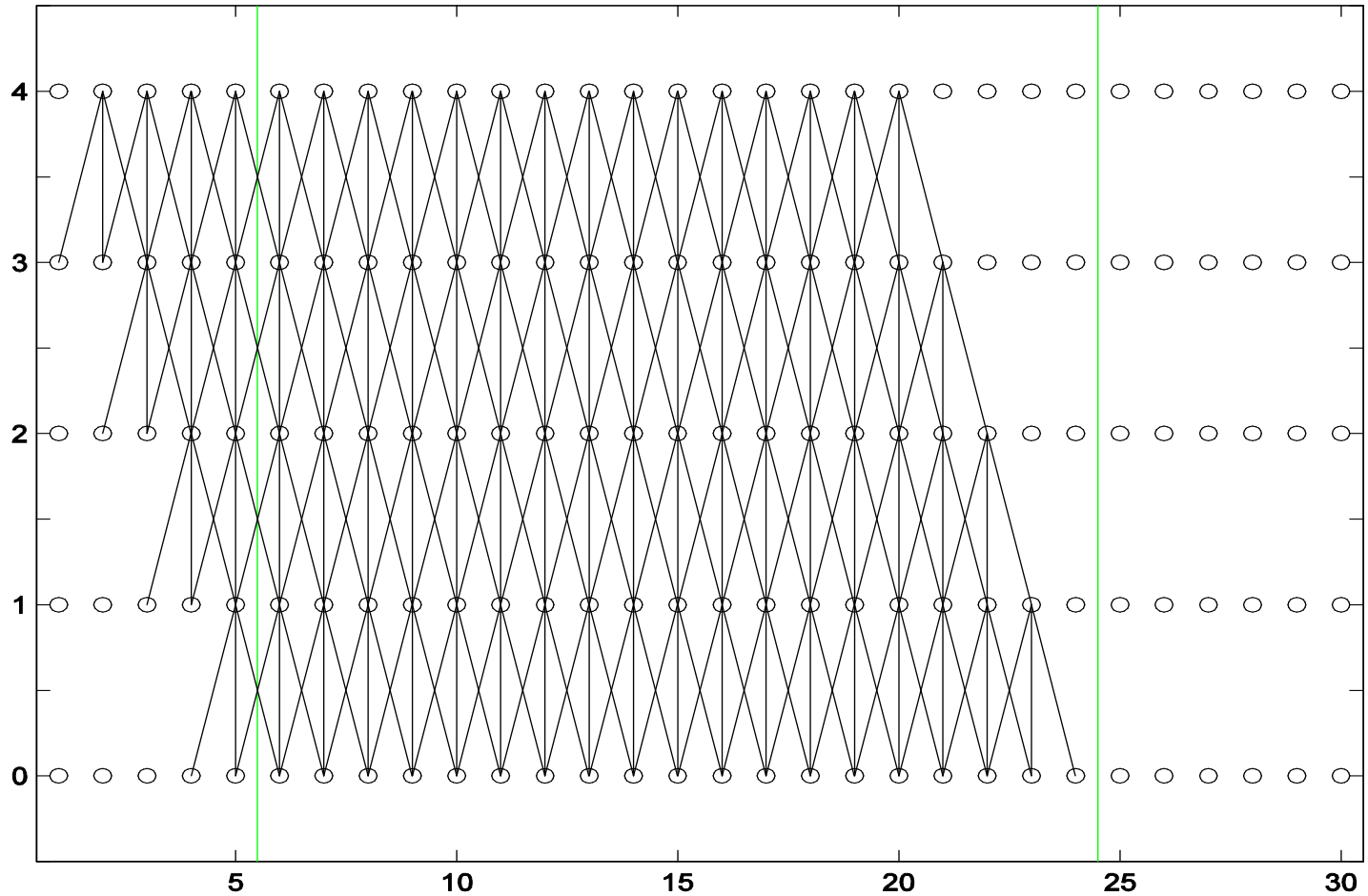
Minimizes number of messages = latency cost

Price: redundant work ∝ "surface/volume ratio"

Remotely Dependent Entries for $[x, Ax, A^2x, A^3x]$, A irregular, multiple processors



Sequential [x,Ax,…,A^4x], with memory hierarchy

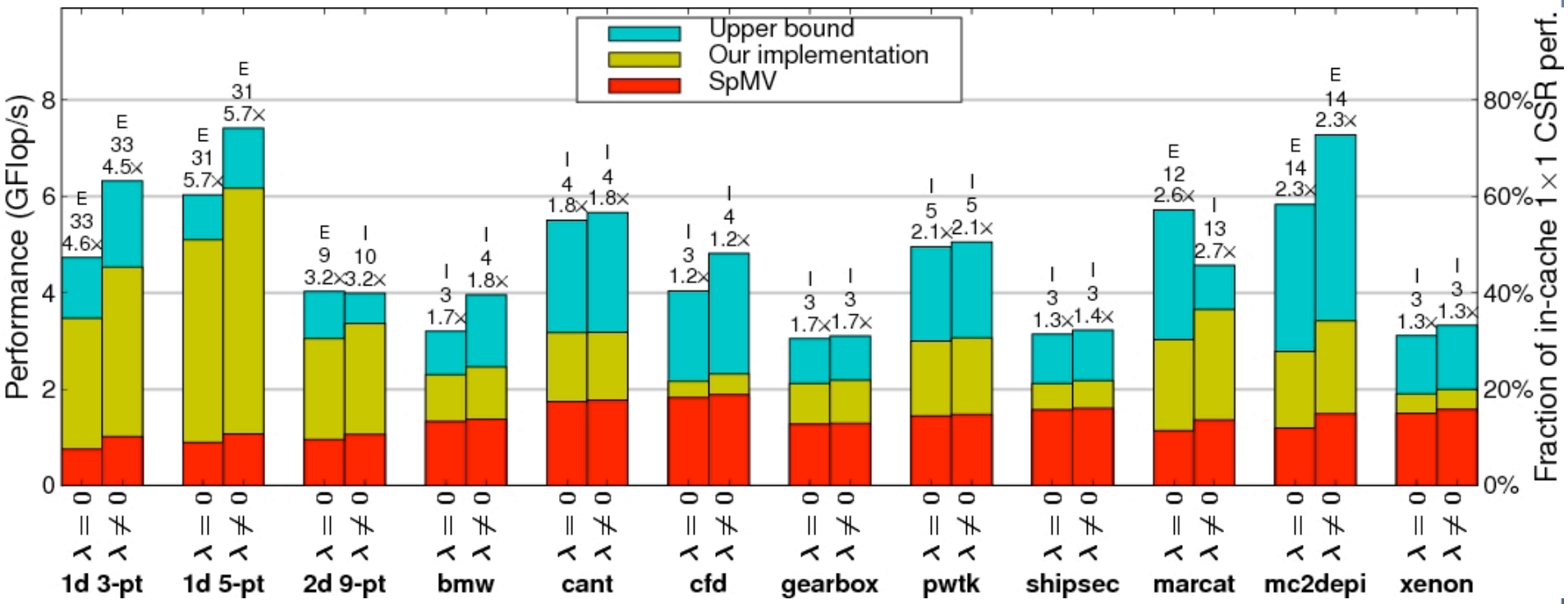


One read of matrix from slow memory, ***not k=4***

Minimizes words moved = bandwidth cost

No redundant work

Performance results on 8-Core Clovertown



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
```

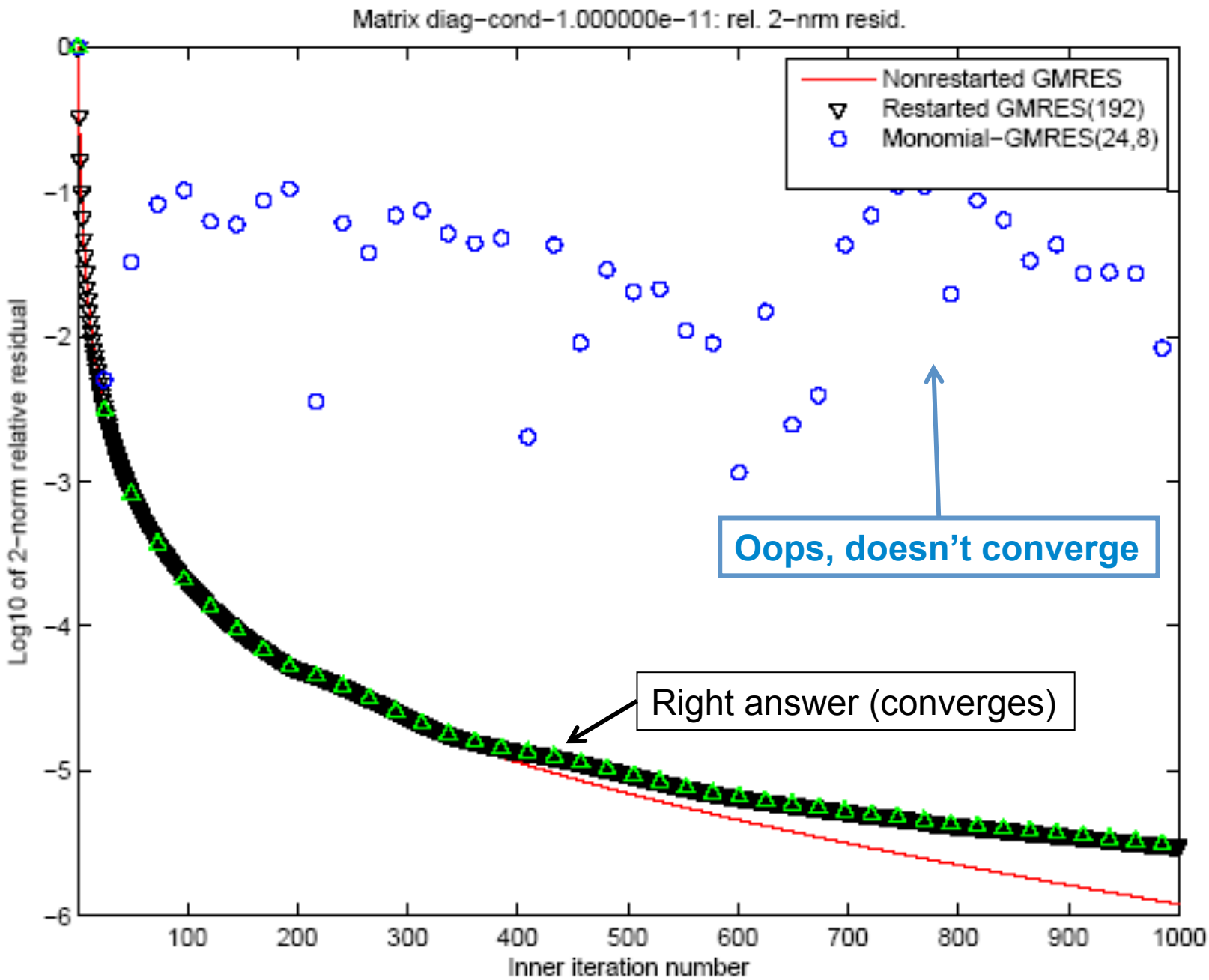
Communication cost =
k copies of A, vectors from slow to fast memory

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 …


Matrix diag−cond−1.000000e−11: rel. 2−nrm resid.
Nonrestarted GMRES
Restarted GMRES(192)
Monomial−GMRES(24,8)
Oops, doesn't converge
Right answer (converges)
Log10 of 2−norm relative residual
Inner iteration number
0
−1
−2
−3
−4
−5
−6
100
200
300
400
500
600
700
800
900
1000

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

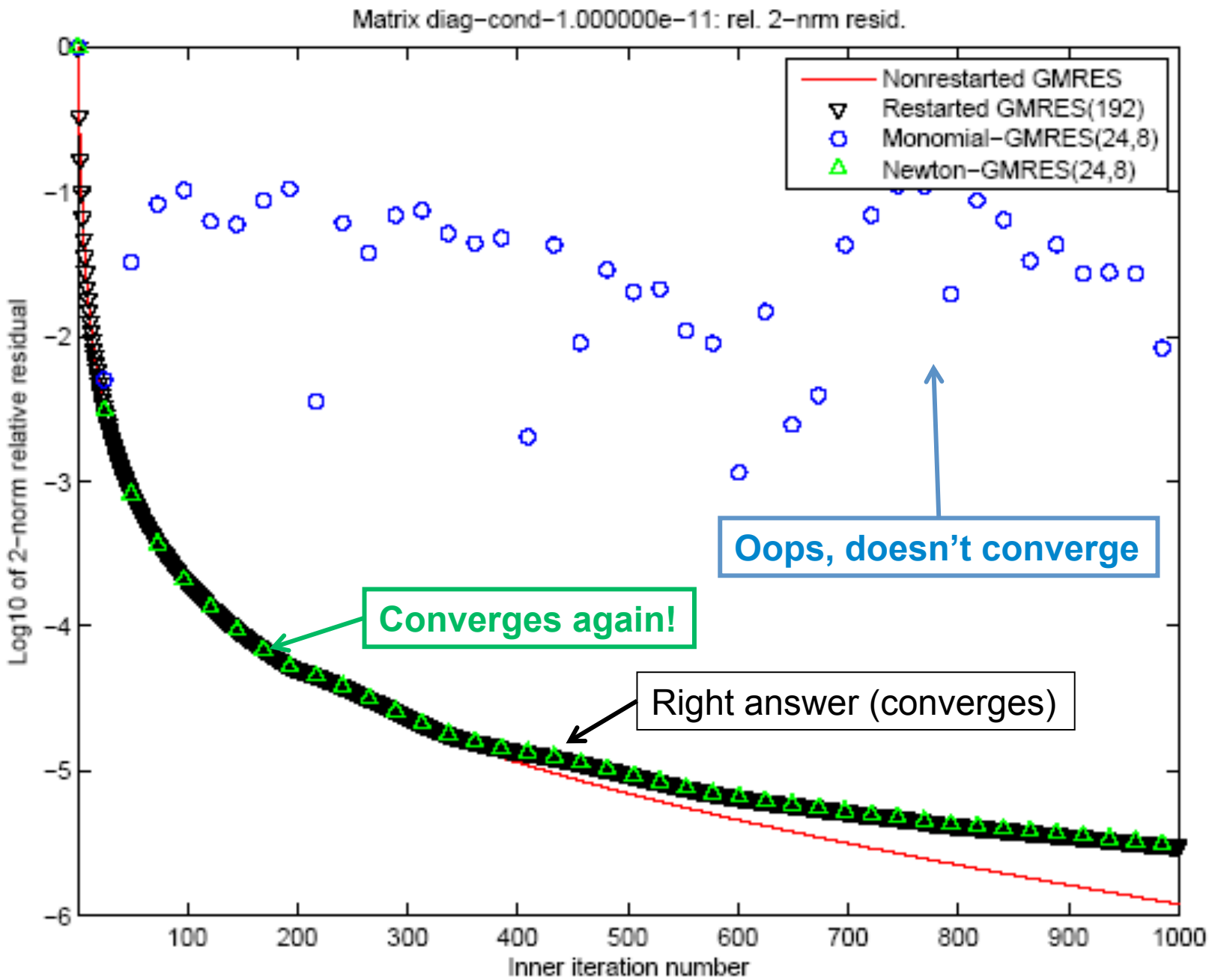
Build H from R

… slightly different R from before

solve LSQ problem with H for x

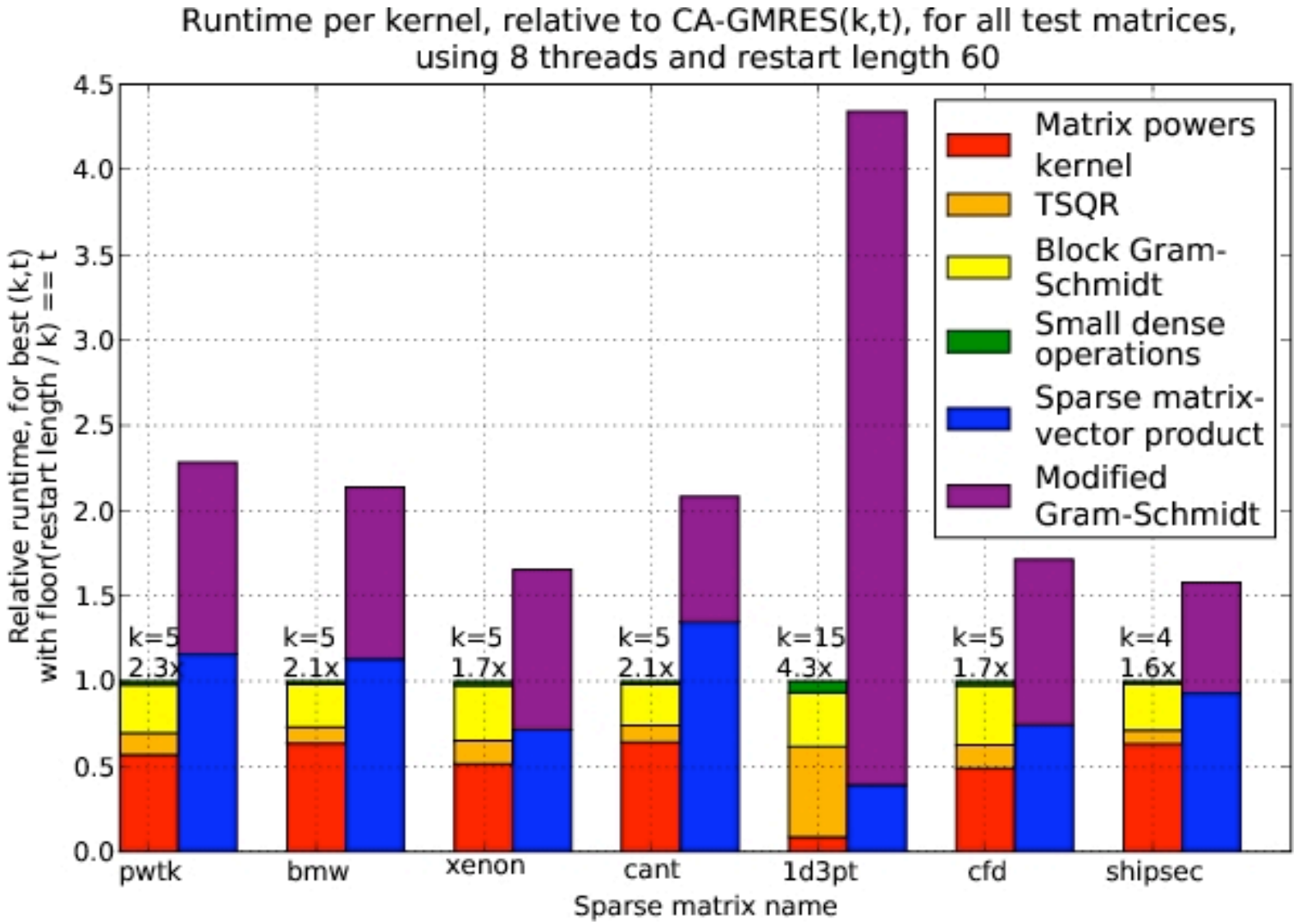
Communication cost still optimal:

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


Matrix diag−cond−1.000000e−11: rel. 2−nrm resid.
Nonrestarted GMRES
Restarted GMRES(192)
Monomial−GMRES(24,8)
Newton−GMRES(24,8)
Log10 of 2−norm relative residual
Inner iteration number
0
−1
−2
−3
−4
−5
−6
100
200
300
400
500
600
700
800
900
1000
Oops, doesn't converge
Converges again!
Right answer (converges)

Speed ups on 8-core Clovertown

CA-GMRES = Communication-Avoiding GMRES

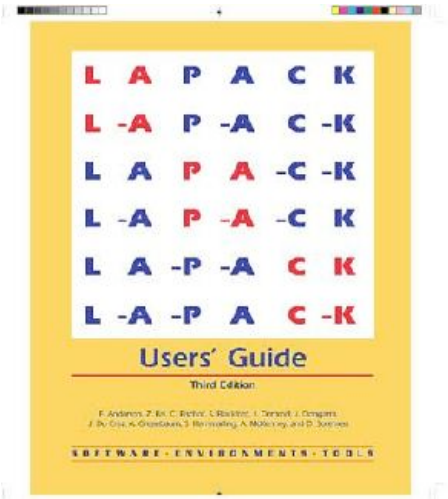


Paper by Mohiyuddin, Hoemmen, D. to appear in Supercomputing09

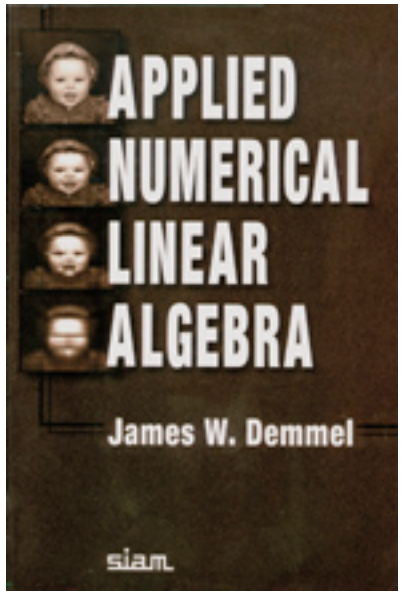
Sparse Conclusions

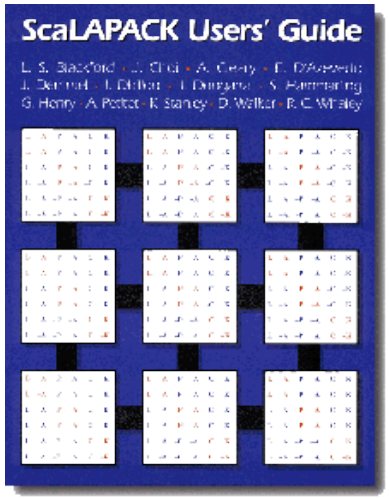
- Fast code must minimize communication
  - Especially for sparse matrix computations because communication dominates
- Generating fast code for a single SpMV
  - Design space of possible algorithms must be searched at run-time, when sparse matrix available
  - Design space should be searched automatically
- 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 (stay tuned for M. Hoemmen's PhD thesis…)
- For more information, see bebop.cs.berkeley.edu

Organizing Linear Algebra Motifs - in books and on-line

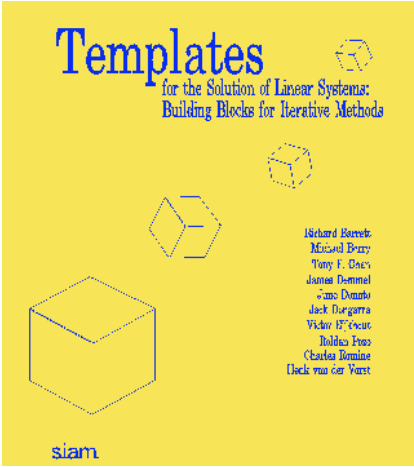


www.netlib.org/lapack



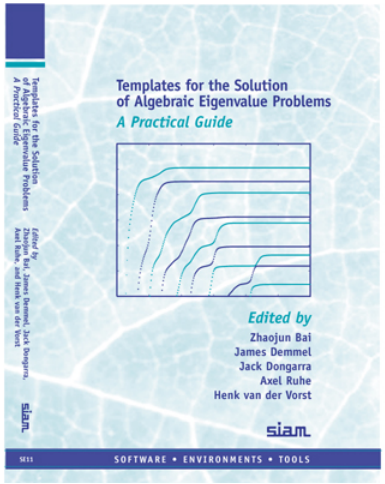


www.netlib.org/scalapack



www.netlib.org/templates

gams.nist.gov



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