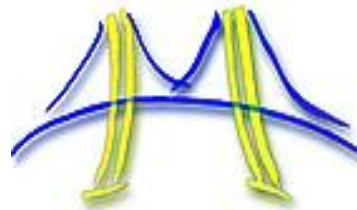


# PARLab Parallel Boot Camp



## Computational Patterns and Autotuning

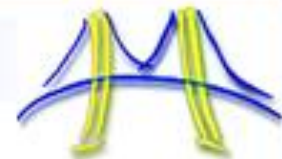
Jim Demmel

EECS and Mathematics

University of California, Berkeley



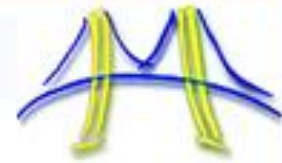
# Outline



- Productive parallel computing depends on recognizing and exploiting known useful patterns
  - Mathematical, Computational (7 Motifs), Structural
- Optimizing (some of) the 7 Motifs
  - To minimize time, 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
  - Related courses:
    - CS267: [www.cs.berkeley.edu/~demmel/cs267\\_Spr11](http://www.cs.berkeley.edu/~demmel/cs267_Spr11)
    - Ma221: “Advanced Matrix Computations”, this semester
    - CS294: “Communication Avoiding Algorithms,” this semester
  - 10-hour short course: [www.ba.cnr.it/ISSNLA2010/Courses.htm](http://www.ba.cnr.it/ISSNLA2010/Courses.htm)
  - Papers at [bebop.cs.berkeley.edu](http://bebop.cs.berkeley.edu)



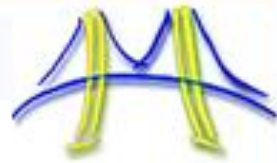
# “7 Motifs” of High Performance Computing



- 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:  $A_{\text{new}}(i,j) = 4 * A(i,j) - A(i-1,j) - A(i+1,j) - A(i,j-1) - A(i,j+1)$
  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:  $A_{\text{new}}(i,j) = 4 * A(i,j) - A(i-1,j) - A(i+1,j) - A(i,j-1) - A(i,j+1)$

## 4. Operations on Unstructured Grids

- Ex: Similar, but list of neighbors varies from entry to entry

## 5. Spectral Methods

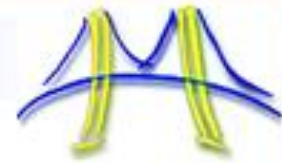
- Ex: Fast Fourier Transform (FFT)

## 6. Particle Methods

- Ex: Compute electrostatic forces on  $n$  particles

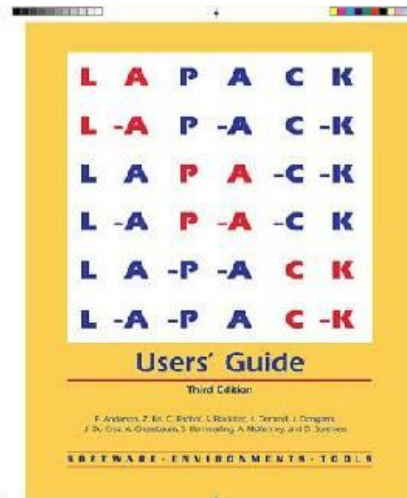
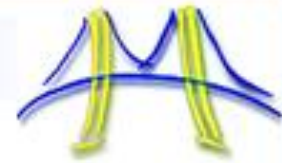
## 7. Monte Carlo

- Ex: Many independent simulations using different inputs



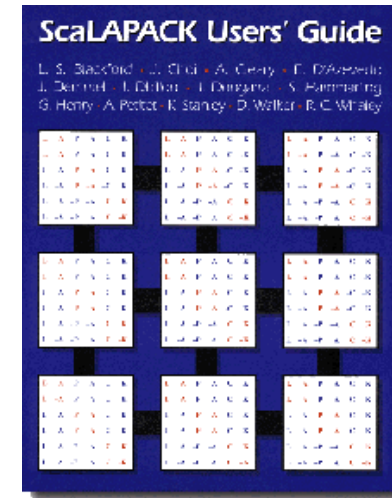
- How to use it
  - What problems does it solve?
  - How to choose solution approach, if more than one?
- How to find the best software available now
  - Best: fastest? most accurate? fewest keystrokes?
- How are the best implementations built?
  - What is the “design space” (wrt math and CS)?
  - How do we search for best (autotuning)?
- Open problems, current work, thesis problems...

# Organizing Linear Algebra Motifs - in books and on-line

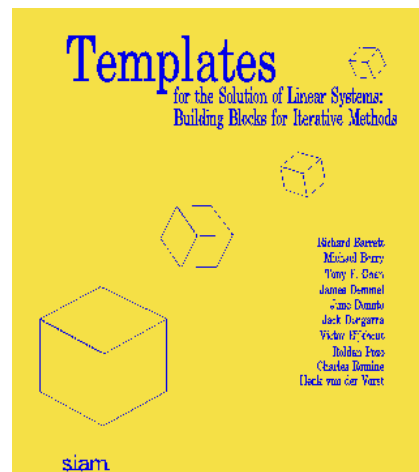


[www.netlib.org/lapack](http://www.netlib.org/lapack)

[gams.nist.gov](http://gams.nist.gov)



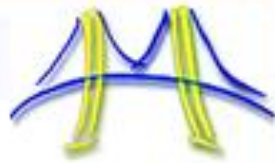
[www.netlib.org/scalapack](http://www.netlib.org/scalapack)



[www.netlib.org/templates](http://www.netlib.org/templates)



[www.cs.utk.edu/~dongarra/etemplates](http://www.cs.utk.edu/~dongarra/etemplates)



## “Communication Avoiding” algorithms

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

} communication
- Exponentially growing gaps between
  - Time\_per\_flop  $\ll$  1/Network BW  $\ll$  Network Latency
    - Improving 59%/year vs 26%/year vs 15%/year
  - Time\_per\_flop  $\ll$  1/Memory BW  $\ll$  Memory Latency
    - Improving 59%/year vs 23%/year vs 5.5%/year
- Goal : reorganize motifs to *avoid* communication
  - Between all memory hierarchy levels
    - $L1 \longleftrightarrow L2 \longleftrightarrow \text{DRAM} \longleftrightarrow \text{network, etc}$
  - Not just *overlapping* communication and arithmetic (speedup  $\leq 2x$ )
  - Very large speedups possible




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.”

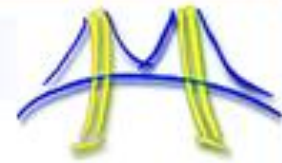
FY 2010 Congressional Budget, Volume 4, FY2010 Accomplishments, Advanced Scientific Computing Research (ASCR), pages 65-67.

**CA-GMRES (Hoemmen, Mohiyuddin, Yelick, JD)**  
**“Tall-Skinny” QR (Grigori, Hoemmen, Langou, JD)**

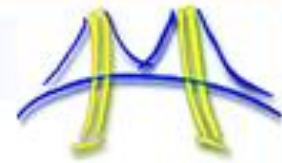




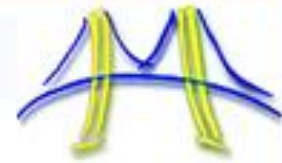
# Obstacle to avoiding communication: Low “computational intensity”



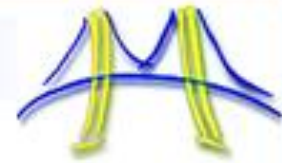
- Let  $f$  = #arithmetic operations in an algorithm
- Let  $m$  = #words of data needed
- Def:  $q = f/m$  = computational intensity
- If  $q$  small, say  $q=1$ , so one op/word, then algorithm can't run faster than memory speed
- But if  $q$  large, so many ops/word, then algorithm can (potentially) 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  $q$



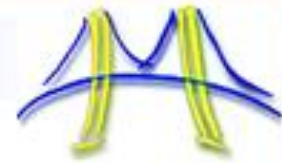
# DENSE LINEAR ALGEBRA MOTIF



- 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 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 ...



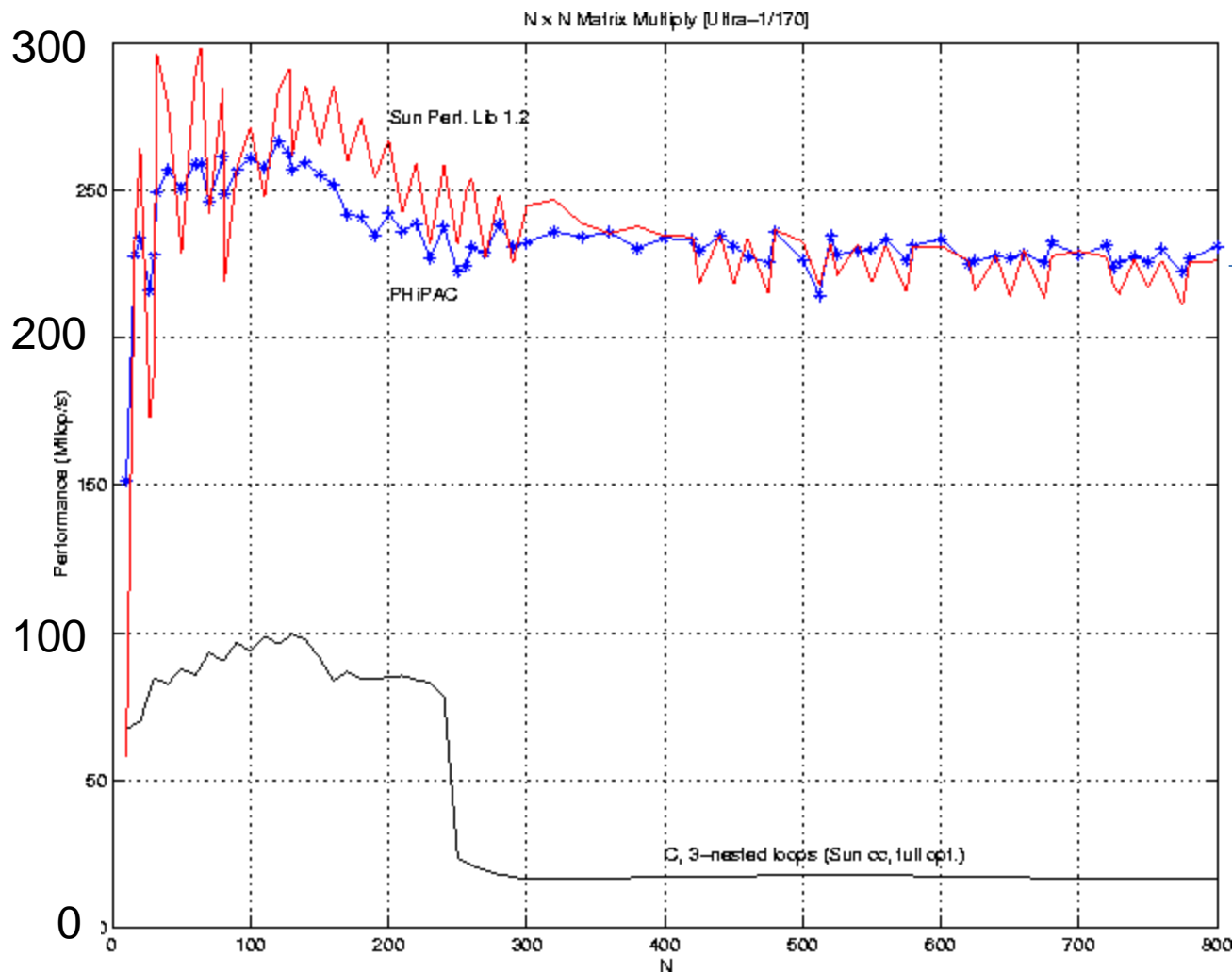
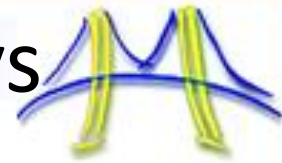
- So the BLAS-2 were invented (1984-1986)
  - Standard library of 25 operations (mostly) on matrix/vector pairs
    - Ex:  $y = \alpha \cdot A \cdot x + \beta \cdot y$  (“GEMV”),  $A = A + \alpha \cdot x \cdot y^T$  (“GER”),  $y = T^{-1} \cdot x$  (“TRSV”)
  - Why BLAS 2 ? 2 nested loops, do  $O(n^2)$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



- 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”

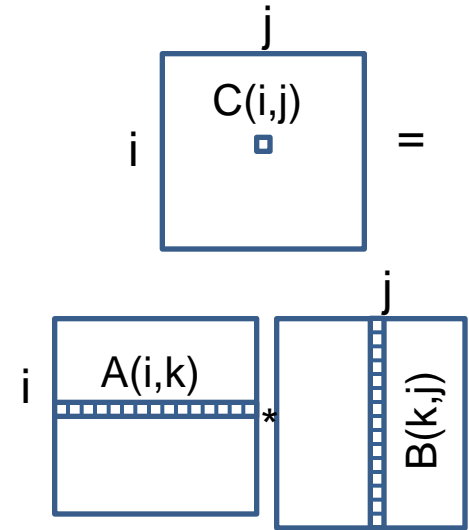
- Replace usual 3 nested loops ...

for  $i=1$  to  $n$

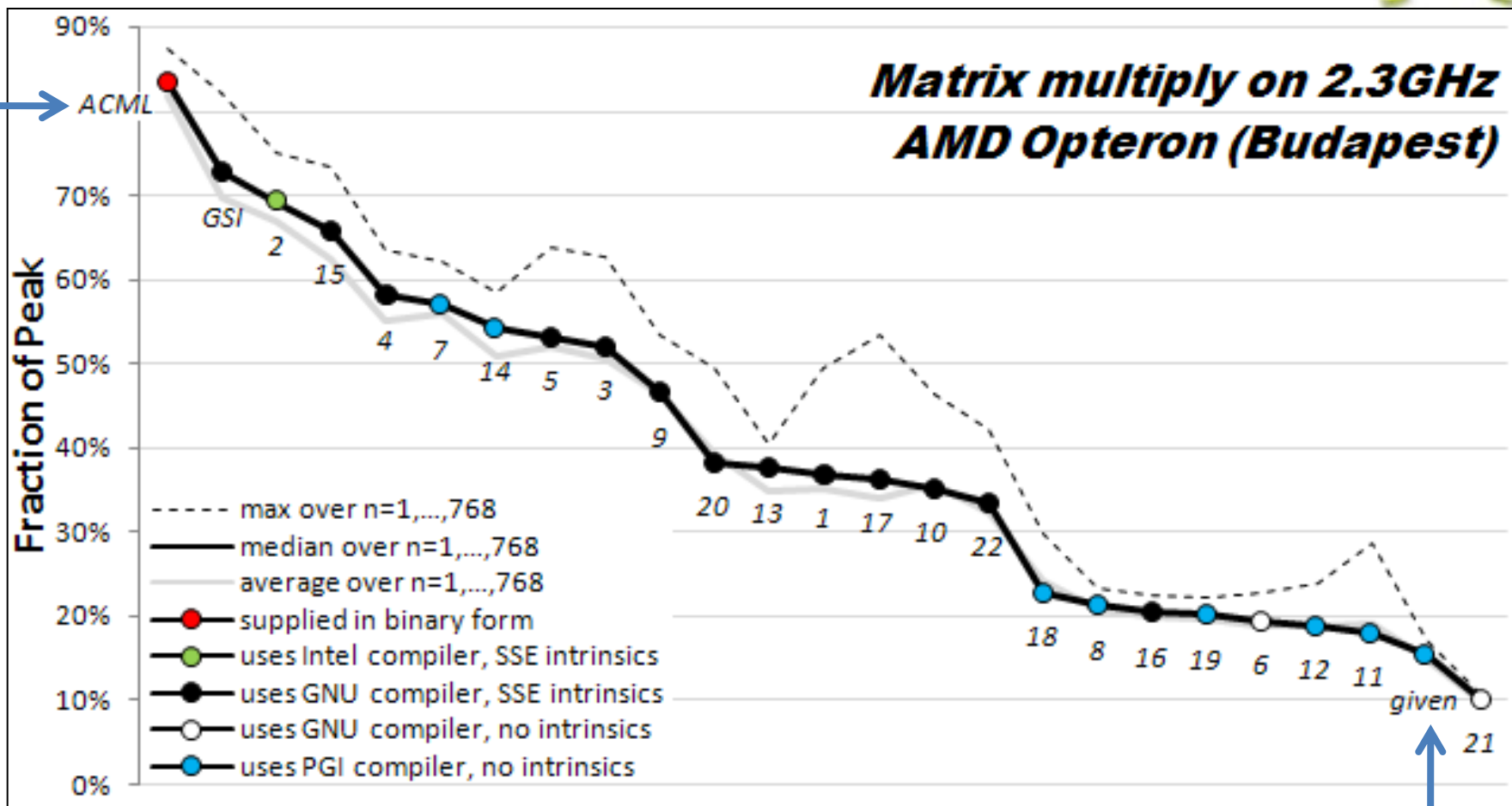
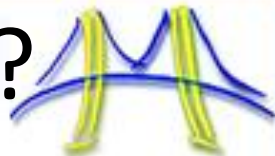
for  $j=1$  to  $n$

for  $k=1$  to  $n$

$$C(i,j) = C(i,j) + A(i,k)*B(k,j)$$



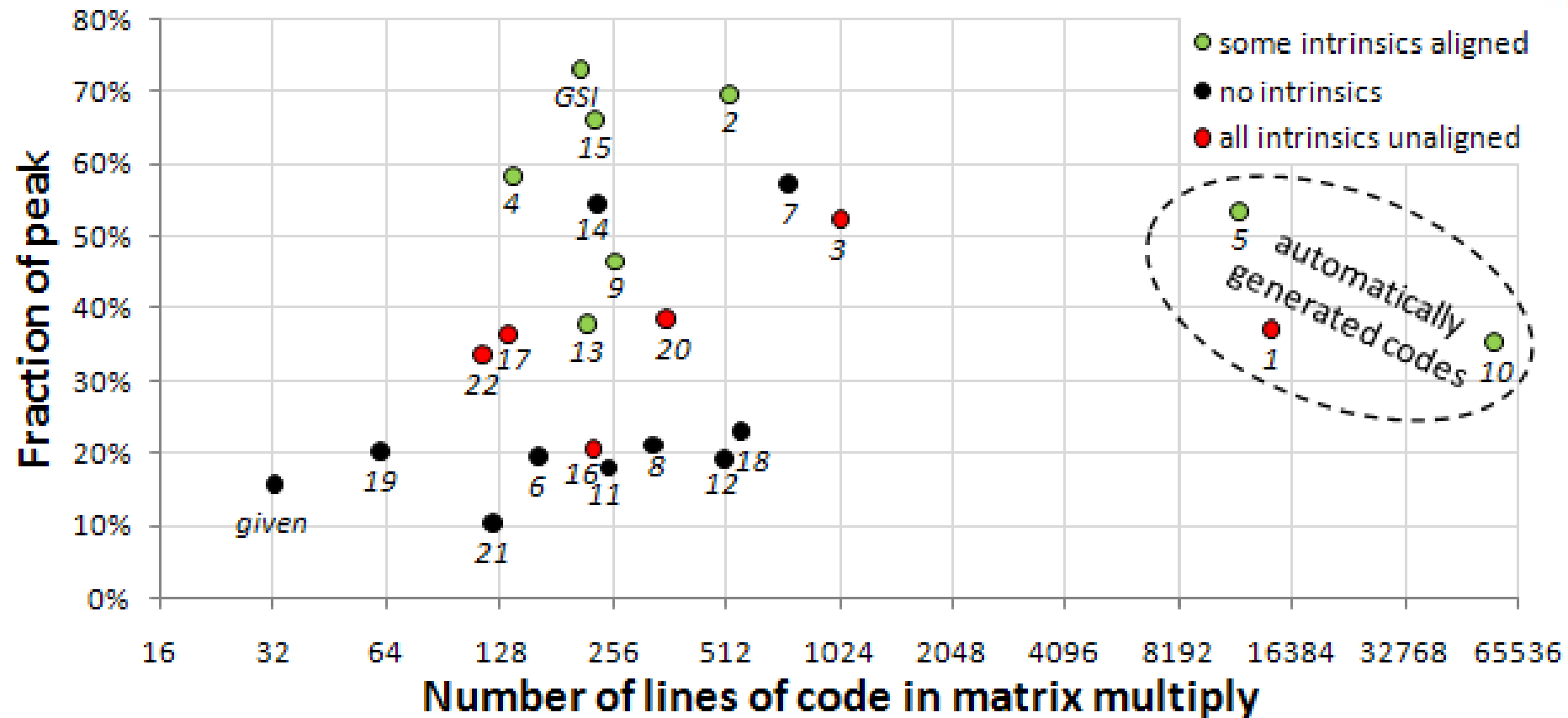
# How hard is hand-tuning, anyway?



- 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/](http://www.cs.berkeley.edu/~volkov/cs267.sp09/hw1/results/)
- Naïve matmul: just 2% of peak

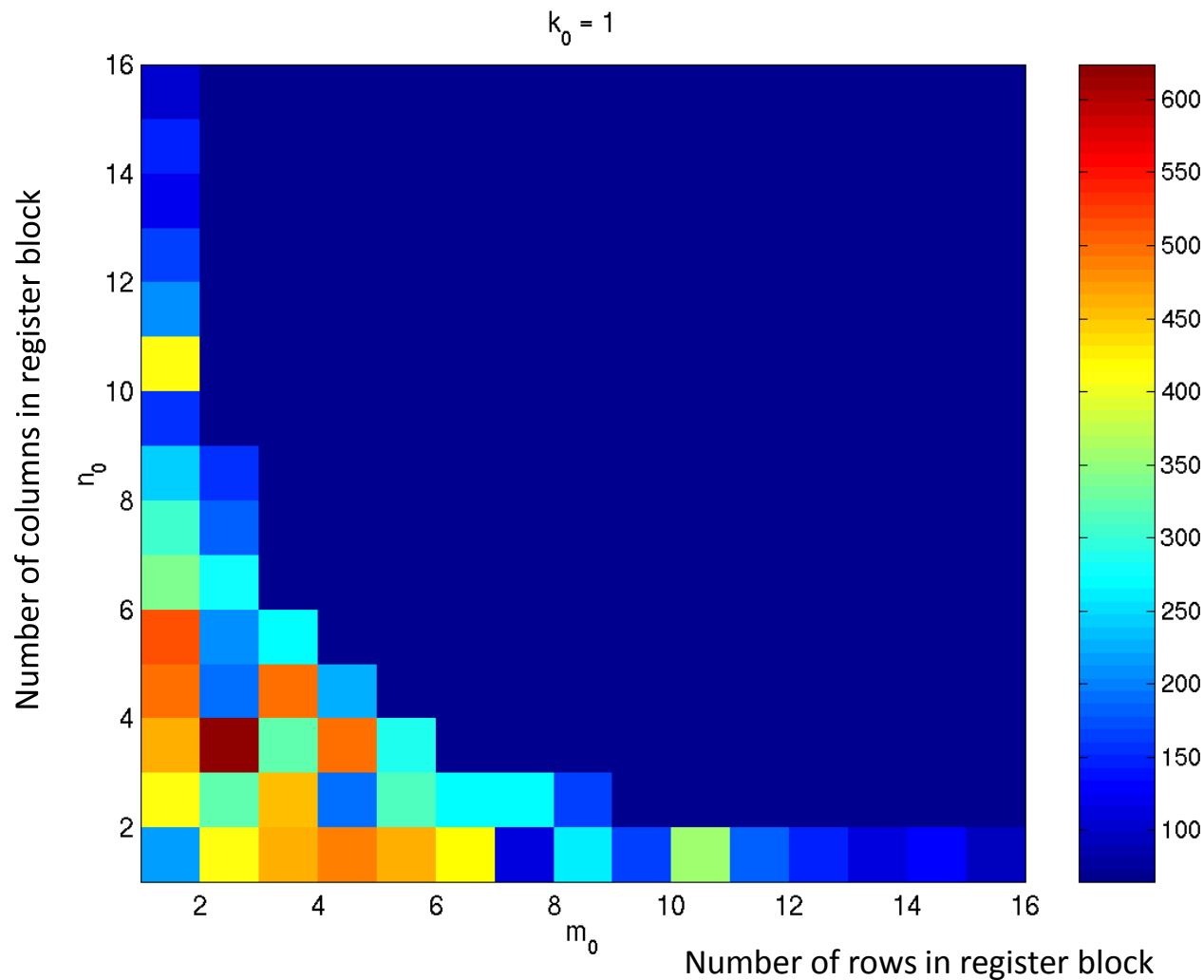
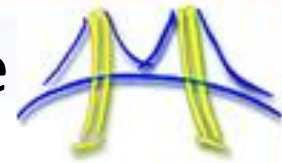


# How hard is hand-tuning, anyway?





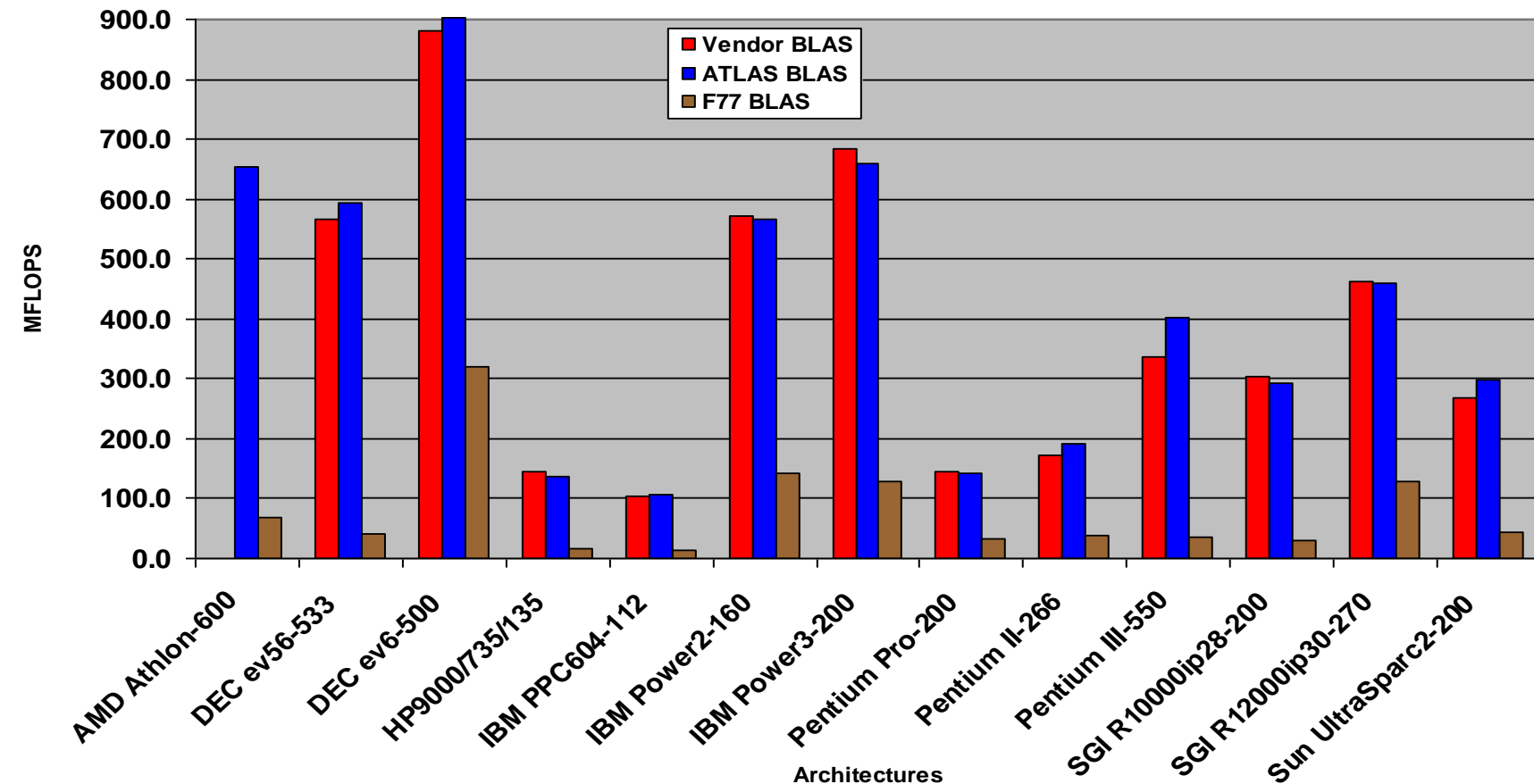
# What part of the Matmul Search Space Looks Like



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)

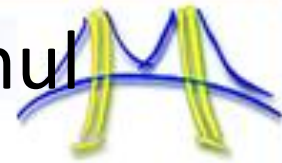
Source: Jack Dongarra



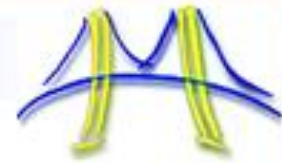
- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.
- ATLAS written by C. Whaley, inspired by PHiPAC, by Asanovic, Bilmes, Chin, D.



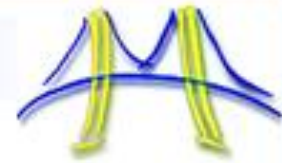
# Lower bounds on Communication for Matmul



- 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)

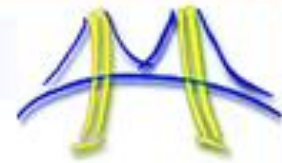


- LAPACK – “Linear Algebra PACKage” - uses BLAS-3 (1989 – now)
  - Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of each row to other rows – BLAS-1
    - Need to reorganize GE (and everything else) to use BLAS-3 instead
  - Contents of current LAPACK (summary)
    - Algorithms we can turn into (nearly) 100% BLAS 3 for large n
      - Linear Systems: solve  $Ax=b$  for  $x$
      - Least Squares: choose  $x$  to minimize  $\sqrt{\sum_i r_i^2}$  where  $r=Ax-b$
    - Algorithms that are only up to ~50% BLAS 3, rest BLAS 1 & 2
      - “Eigenproblems”: Find  $\lambda$  and  $x$  where  $Ax = \lambda x$
      - Singular Value Decomposition (SVD):  $A^T Ax = \sigma^2 x$
    - 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](http://www.netlib.org/lapack)

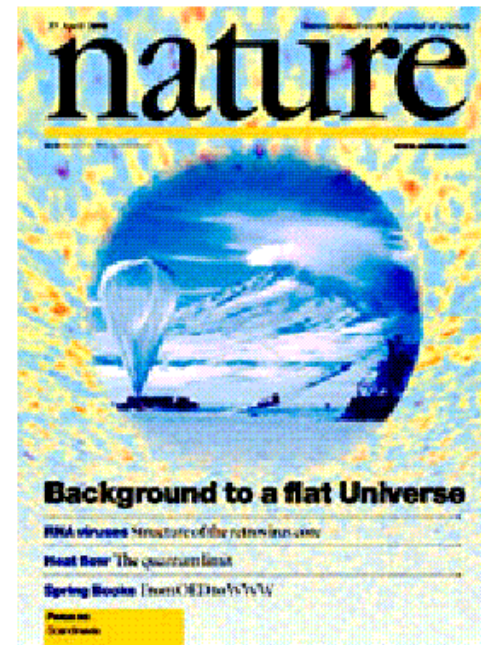


- Is LAPACK parallel?
  - Only if the BLAS are parallel (possible in shared memory)
- ScaLAPACK – “Scalable LAPACK” (1995 – now)
  - For distributed memory – uses MPI
  - More complex data structures, algorithms than LAPACK
    - Only subset of LAPACK’s functionality available
    - Work in progress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

# Success Stories for Sca/LAPACK

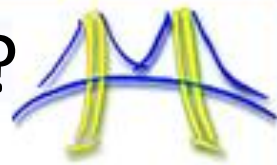


- Widely used
  - Adopted by Mathworks, Cray, Fujitsu, HP, IBM, IMSL, Intel, NAG, NEC, SGI, ...
  - >100M web hits(in 2009, 56M in 2006) @ Netlib (incl. CLAPACK, LAPACK95)
- New science discovered through the solution of dense matrix systems
  - Nature article on the flat universe used ScaLAPACK
  - 1998 Gordon Bell Prize
  - [www.nersc.gov/news/reports/newNERSCresults050703.pdf](http://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



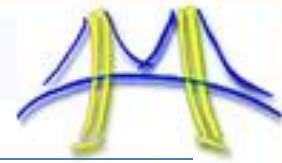
## Do Sca/LAPACK Minimize Communication?

- Can extend lower bound for matmul to all “direct methods” of linear algebra
- Lower bounds on #words\_moved (bandwidth\_cost) and #messages (latency\_cost) for
  - BLAS, LU, QR, Eig, SVD, compositions...
  - Dense and Sparse matrices
  - Parallel and sequential
  - 2 levels and hierarchies
- *Almost none* of Sca/LAPACK attains these lower bounds
- New (mostly dense) algorithms that do attain them
  - Large measured and modeled speedups
- Time to reengineer all these algorithms!
- (Partially extends to Strassen-like algorithms)





# TSQR: QR of a Tall, Skinny matrix



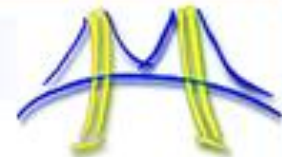
$$W = \begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix}$$

$$\begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} Q_{01} & R_{01} \\ Q_{11} & R_{11} \end{pmatrix}$$

$$\begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} = \begin{pmatrix} Q_{02} & R_{02} \end{pmatrix}$$



# TSQR: QR of a Tall, Skinny matrix



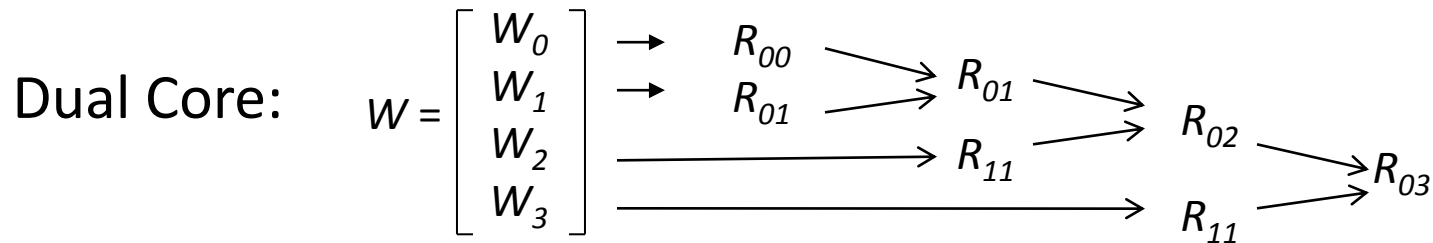
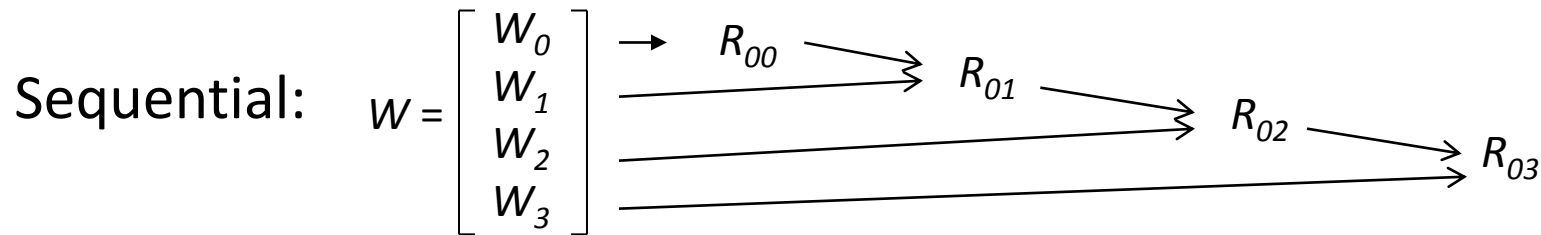
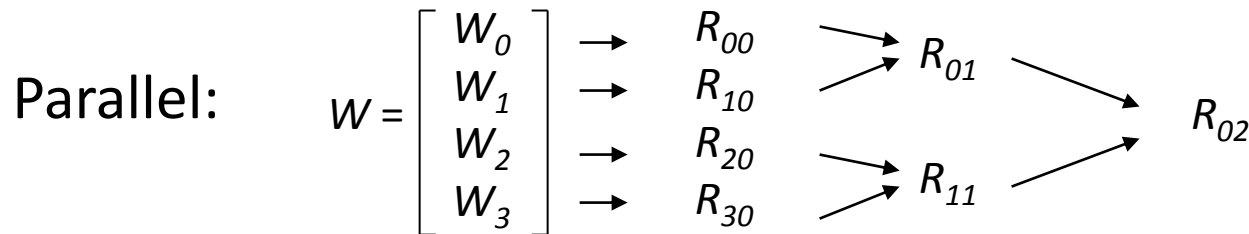
$$W = \begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix} = \begin{pmatrix} Q_{00} & R_{00} \\ Q_{10} & R_{10} \\ Q_{20} & R_{20} \\ Q_{30} & R_{30} \end{pmatrix} = \begin{pmatrix} Q_{00} & & & \\ & Q_{10} & & \\ & & Q_{20} & \\ & & & Q_{30} \end{pmatrix} \cdot \begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix}$$

$$\begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} Q_{01} & R_{01} \\ Q_{11} & R_{11} \end{pmatrix} = \begin{pmatrix} Q_{01} & \\ & Q_{11} \end{pmatrix} \cdot \begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix}$$

$$\begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} = \begin{pmatrix} Q_{02} & R_{02} \end{pmatrix}$$

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

# TSQR: An Architecture-Dependent Algorithm

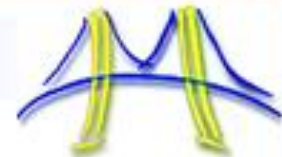


Multicore / Multisocket / Multirack / Multisite / Out-of-core: ?

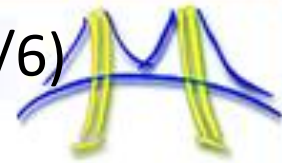
Can choose reduction tree dynamically



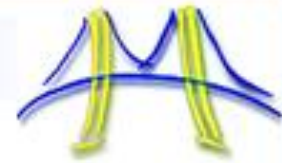
# TSQR Performance Results



- 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 et al)
  - Cloud – early result – up and running
- 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



- Communication-Avoiding for everything (open problems...)
- 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 at [icl.cs.utk.edu/plasma/](http://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](http://www.cs.berkeley.edu/~volkov)
  - Lower, matching upper bounds (SPAA’11 paper, at [bebop.cs.berkeley.edu](http://bebop.cs.berkeley.edu))
  - MAGMA – Matrix Algebra on GPU and Multicore Architectures
    - Release 1.0 at [icl.cs.utk.edu/magma/](http://icl.cs.utk.edu/magma/)
- How much code generation can we automate?
  - MAGMA , and FLAME ([www.cs.utexas.edu/users/flame/](http://www.cs.utexas.edu/users/flame/))



# SPARSE LINEAR ALGEBRA MOTIF

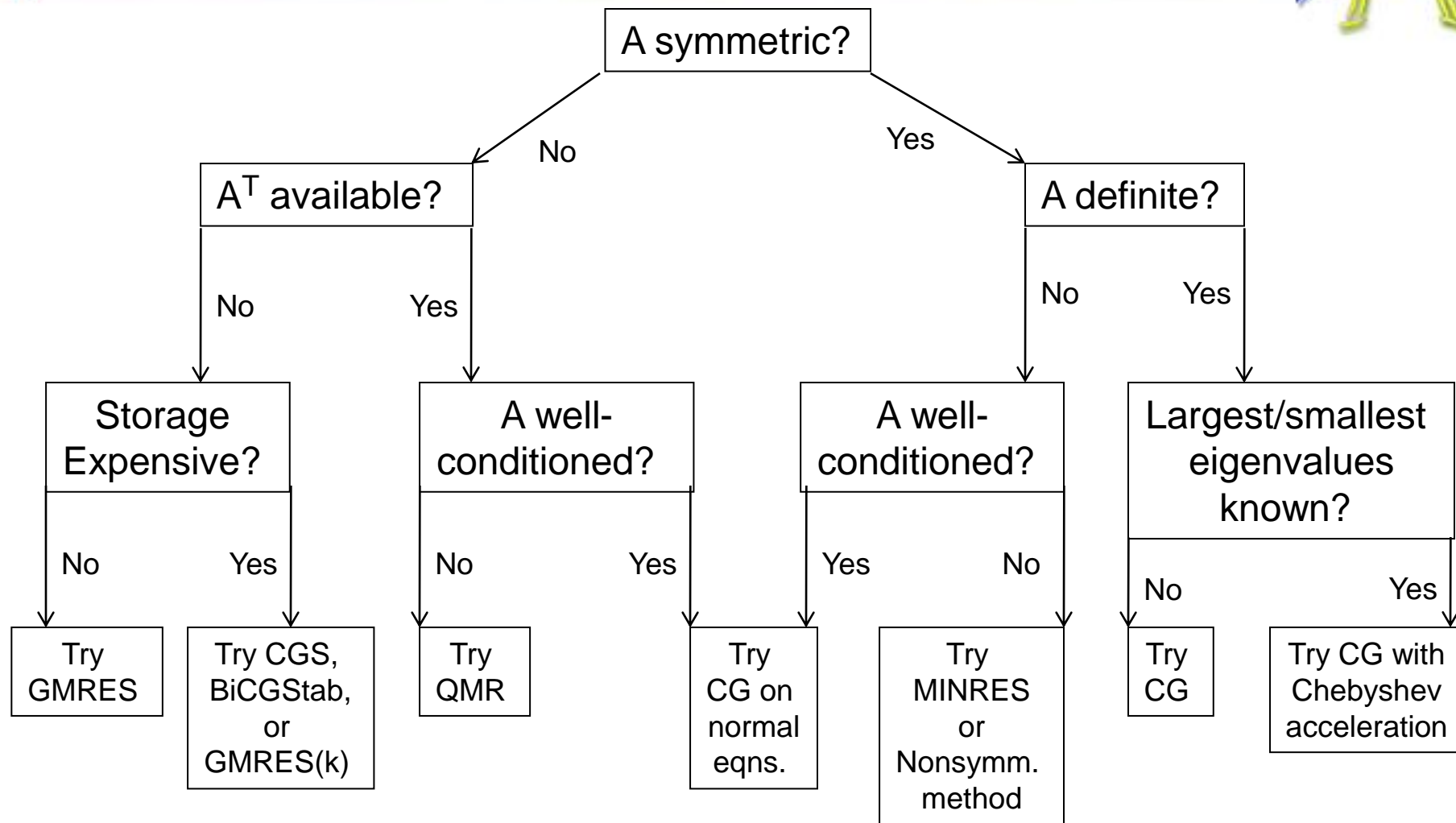
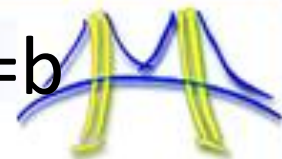


# Sparse Matrix Computations

- Similar problems to dense matrices
  - $Ax=b$ , Least squares,  $Ax = \lambda x$ , 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](http://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](http://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



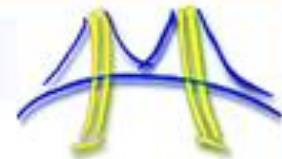
# Choosing a Krylov Subspace Method for $Ax=b$



- All depend on SpMV
- See [www.netlib.org/templates](http://www.netlib.org/templates) for  $Ax=b$
- See [www.cs.ucdavis.edu/~bai/ET/contents.html](http://www.cs.ucdavis.edu/~bai/ET/contents.html) for  $Ax=\lambda x$  and SVD



# Sparse Outline

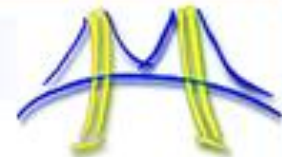


- Approaches to Automatic Performance Tuning
- Results for sparse matrix kernels
  - Sparse Matrix Vector Multiplication (SpMV)
  - Sequential and Multicore results
- OSKI = Optimized Sparse Kernel Interface
- Tuning Entire Sparse Solvers
  - Avoiding Communication
- What is a sparse matrix?



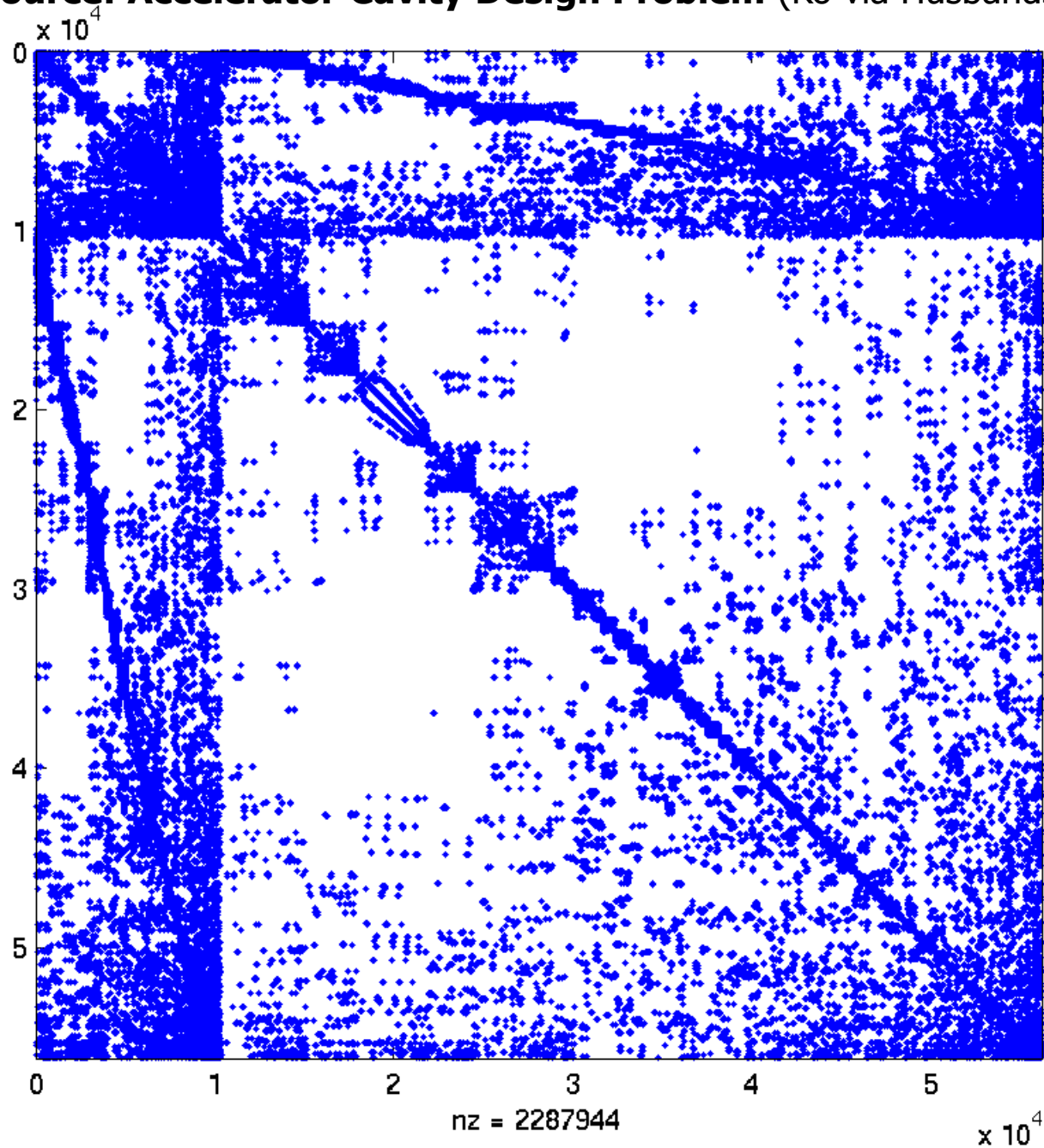


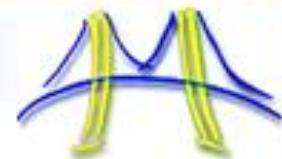
# Approaches to Automatic Performance Tuning



- **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](http://www.spiral.net), [www.fftw.org](http://www.fftw.org)

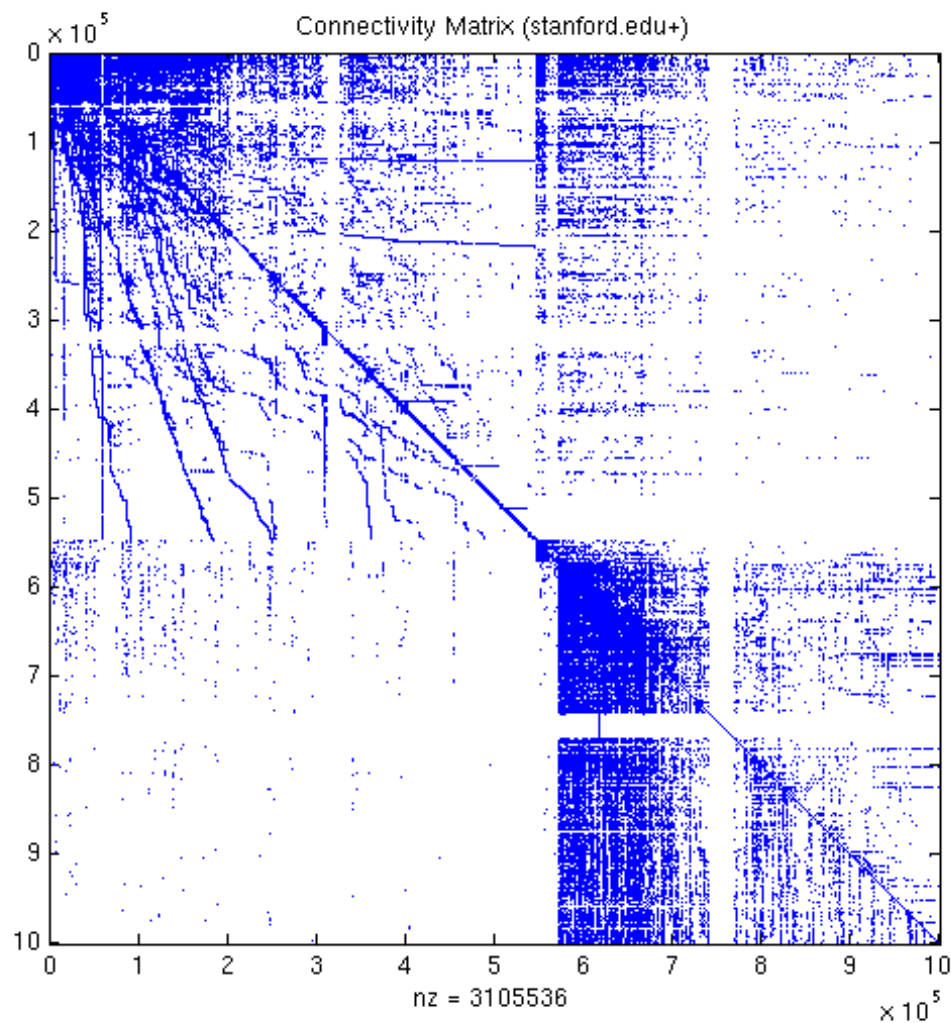
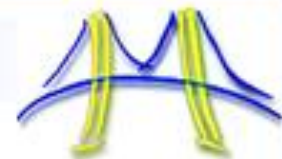
# Source: Accelerator Cavity Design Problem (Ko via Husbands)



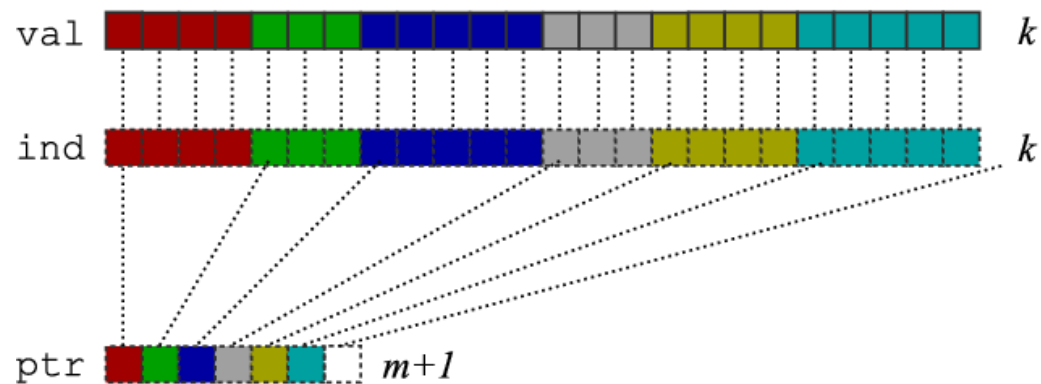
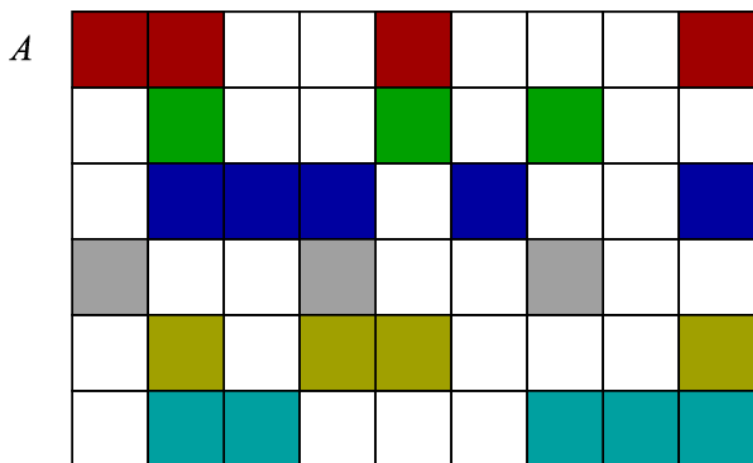
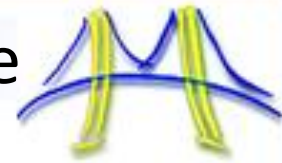
[illegible]



# A Sparse Matrix You Use Every Day



# SpMV with Compressed Sparse Row (CSR) Storage



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

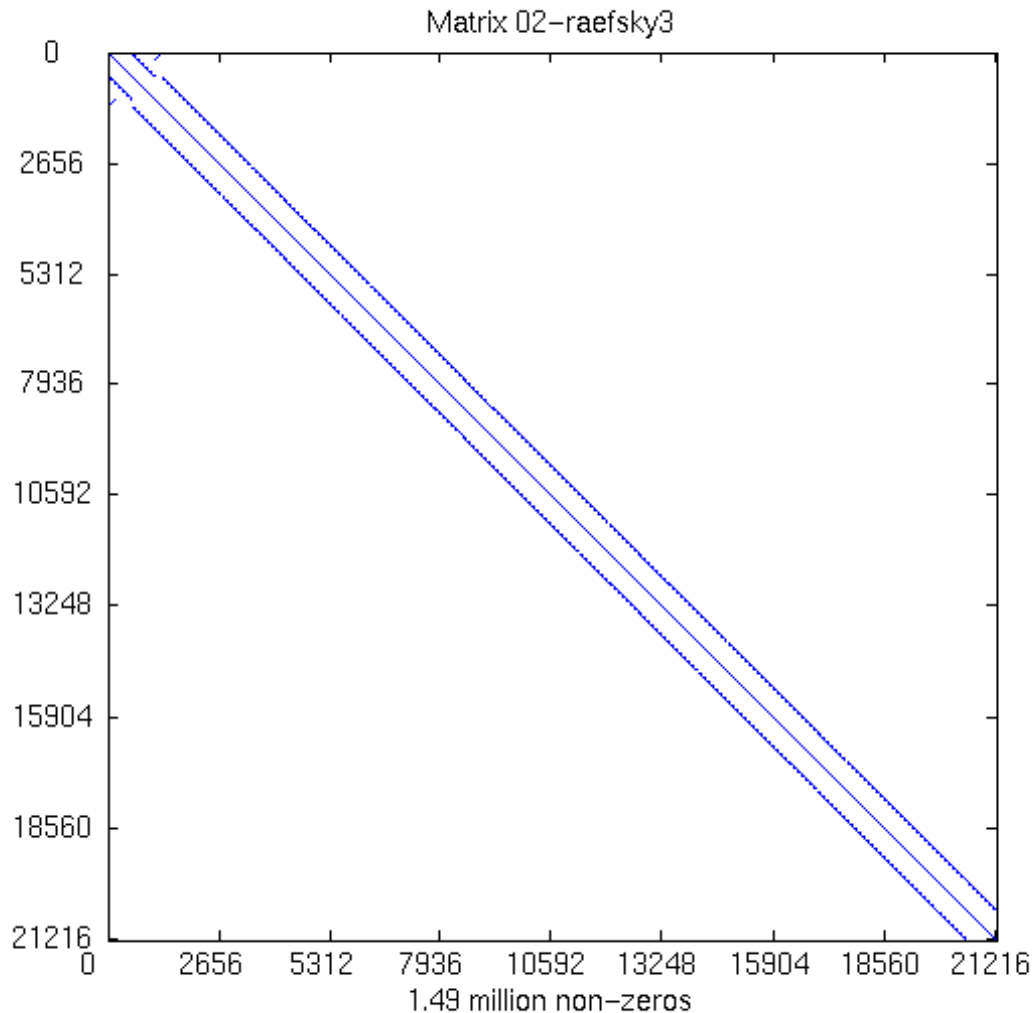
for each row  $i$

for  $k=\text{ptr}[i]$  to  $\text{ptr}[i+1]$  do

$y[i] = y[i] + \text{val}[k]*x[\text{ind}[k]]$

Only 2 flops per  
2 mem\_refs:  
Limited by getting  
data from memory

# Example: The Difficulty of Tuning

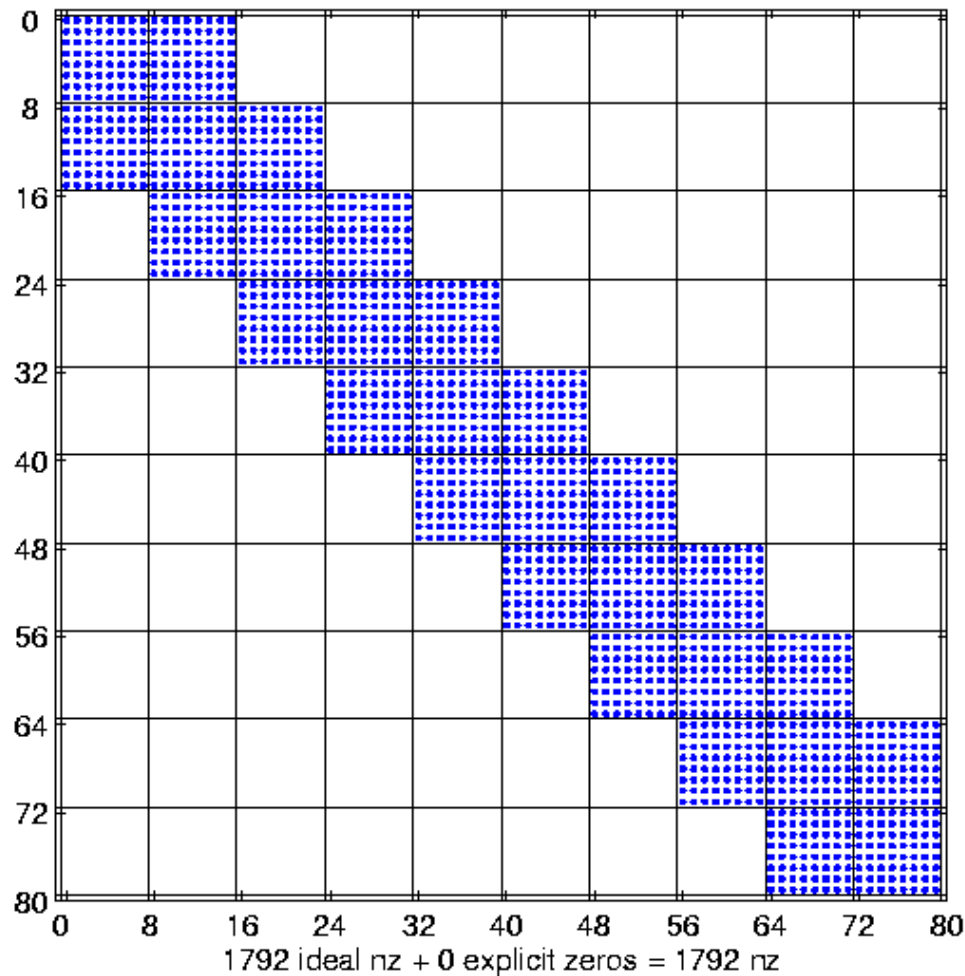


- $n = 21200$
- $\text{nnz} = 1.5 \text{ M}$
- kernel: SpMV
- Source: NASA structural analysis problem



# Example: The Difficulty of Tuning

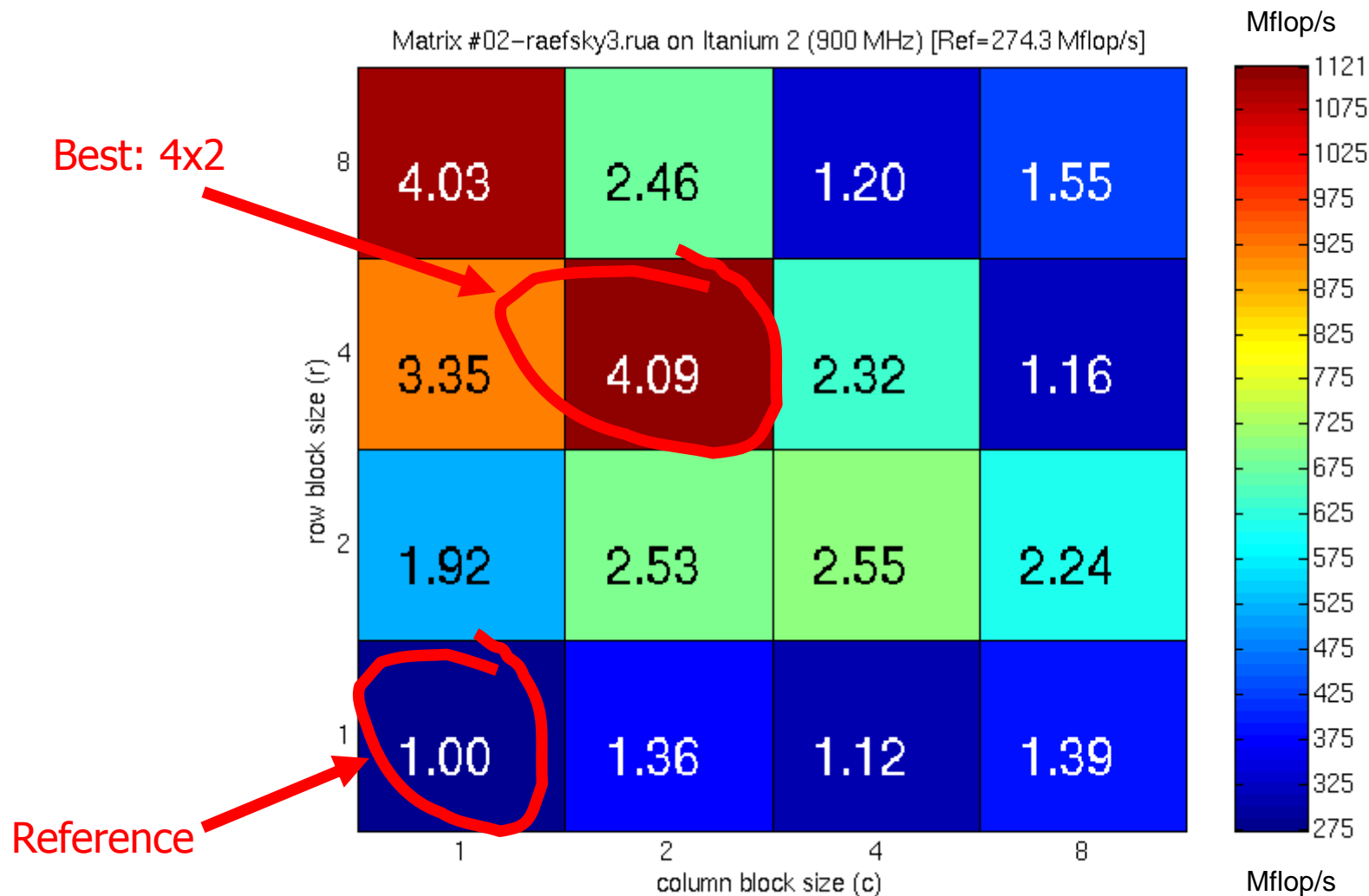
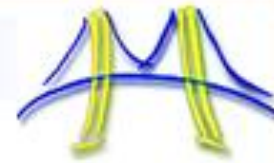
Matrix 02-raefsky3



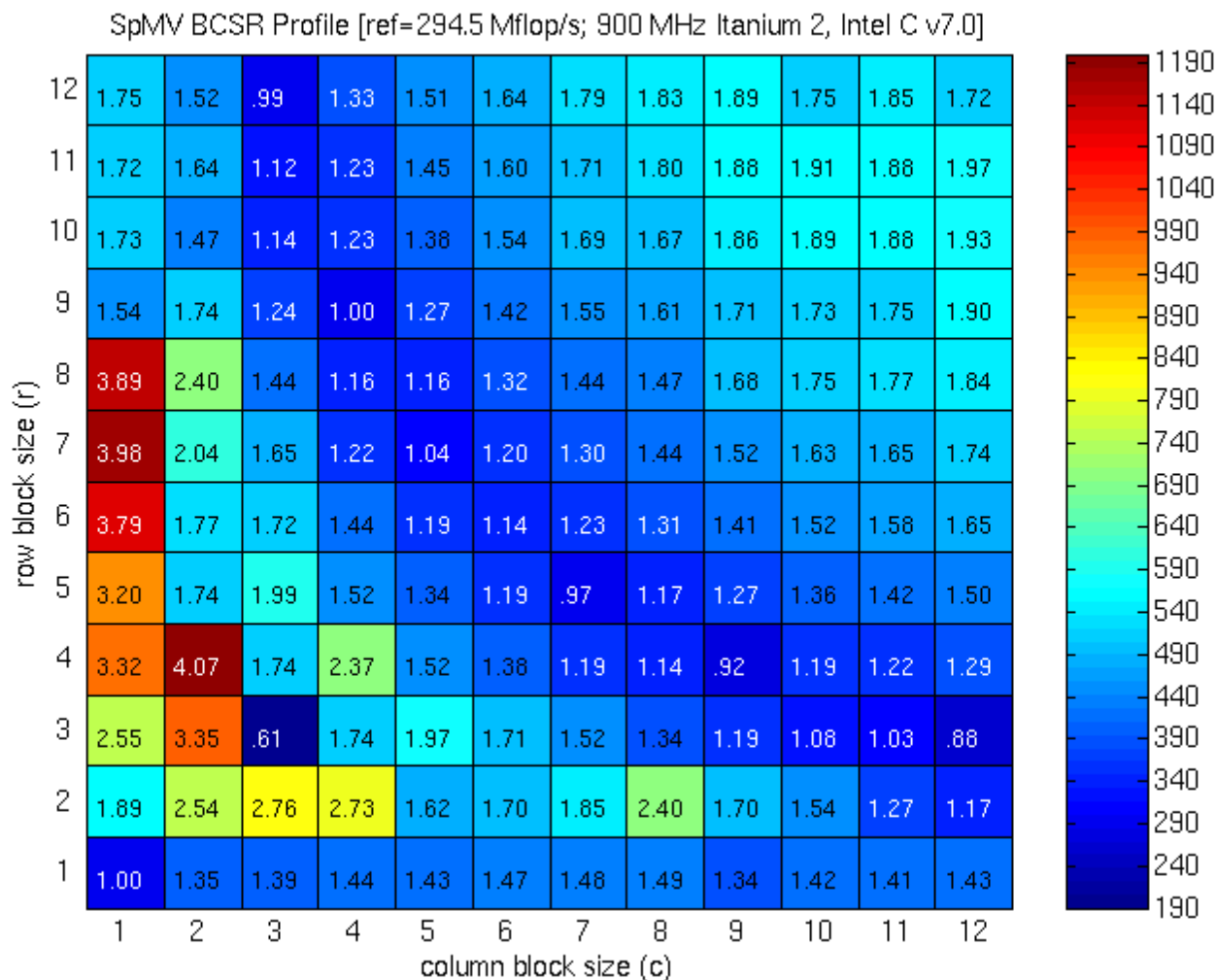
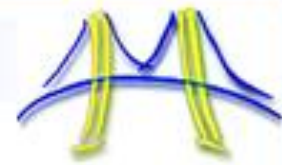
- $n = 21200$
- $nnz = 1.5 \text{ M}$
- kernel: SpMV
- Source: NASA structural analysis problem
- **8x8** dense substructure: exploit this to limit #mem\_refs



# Speedups on Itanium 2: The Need for Search



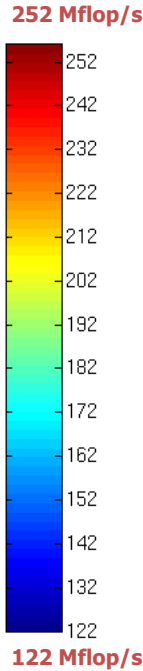
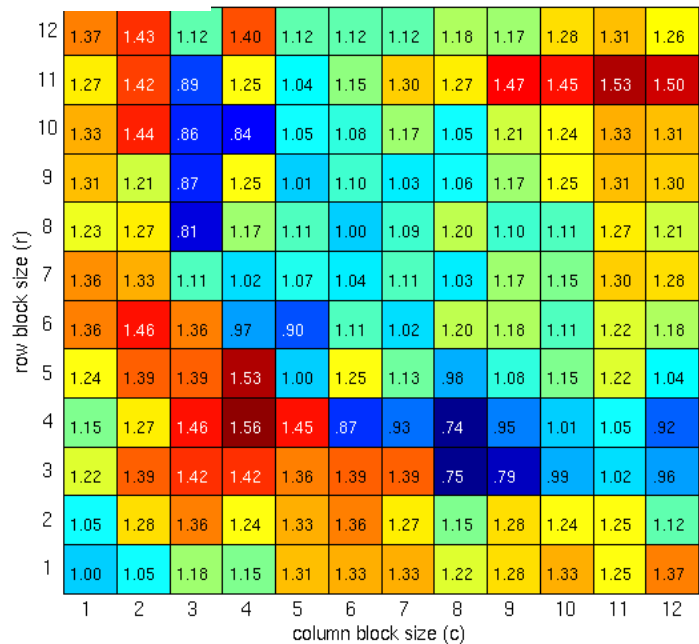
# Register Profile: Itanium 2



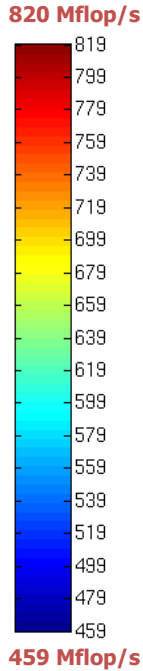
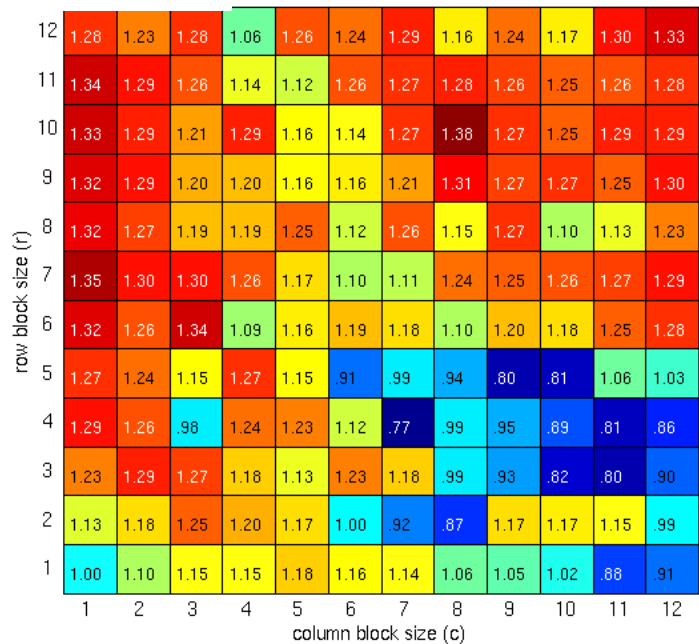
1190 Mflop/s

190 Mflop/s

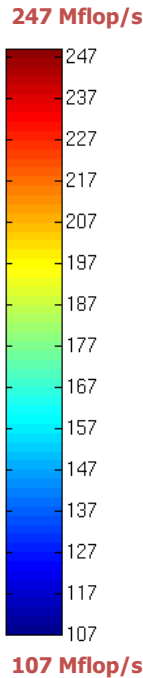
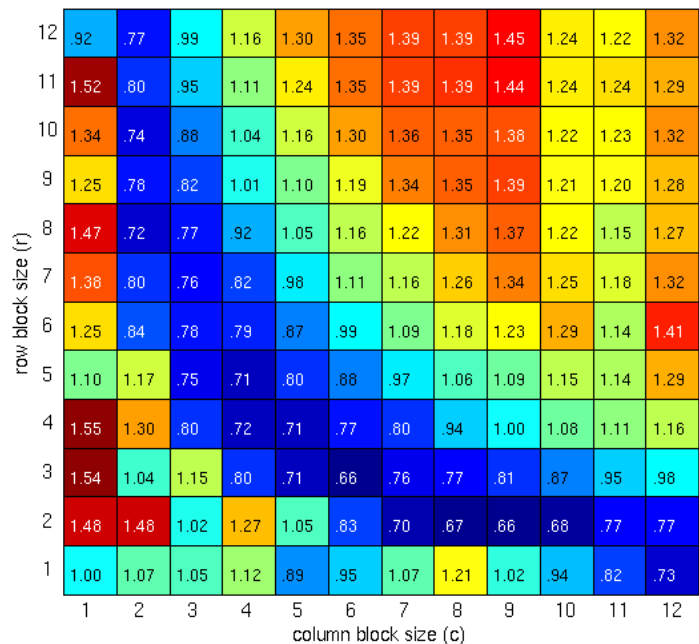
**Power3 - 17%**      profile [ref=163.9 Mflop/s; 375 MHz Power3, IBM xlc v5]



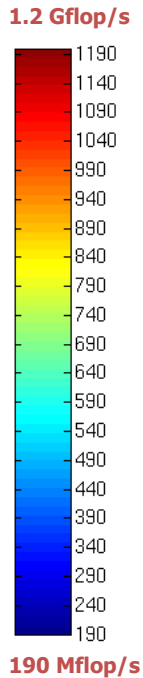
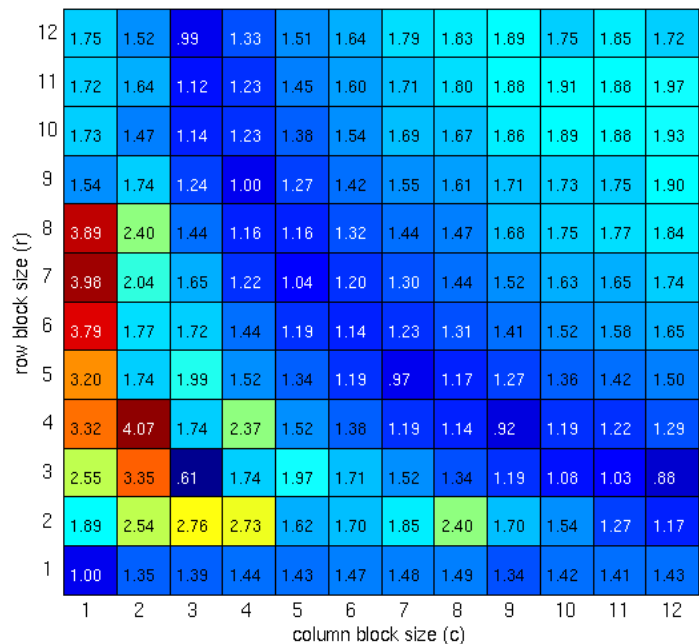
**Power4 - 16%**      ifile [ref=594.9 Mflop/s; 1.3 GHz Power4, IBM xlc v6]



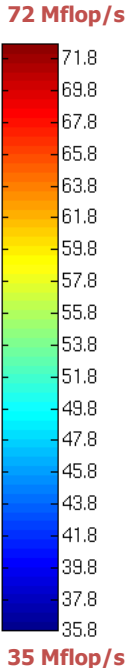
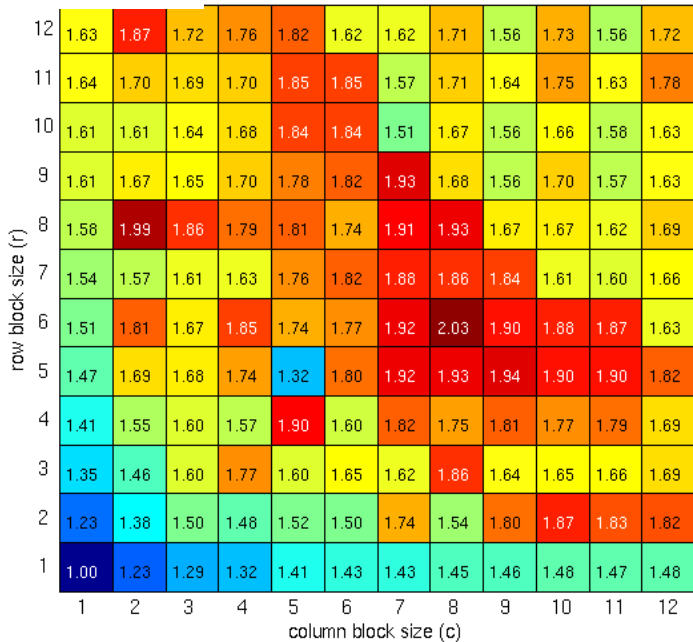
**Itanium 1 - 8%**      profile [ref=161.2 Mflop/s; 800 MHz Itanium, Intel C v7]



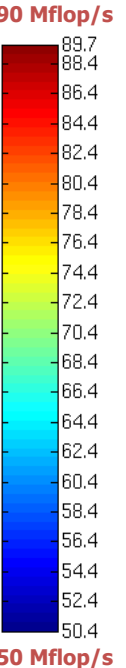
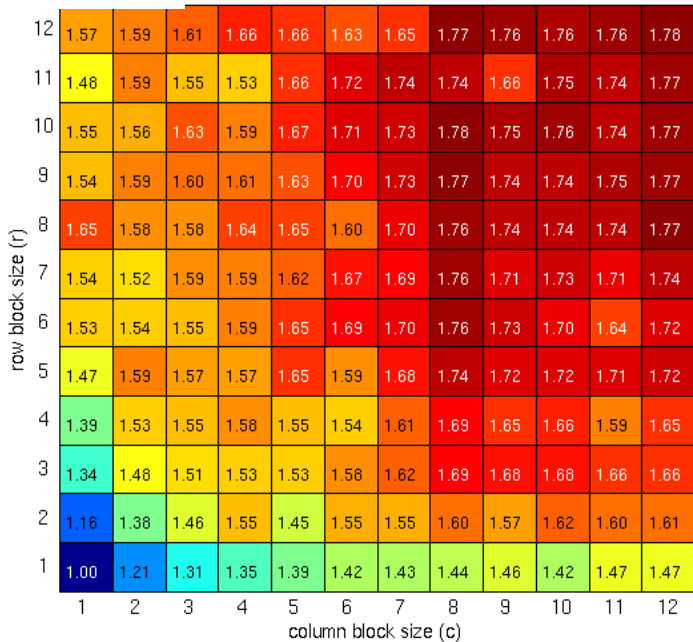
**Itanium 2 - 33%**      ifile [ref=294.5 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]



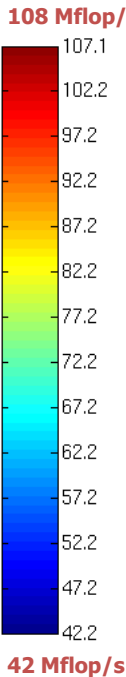
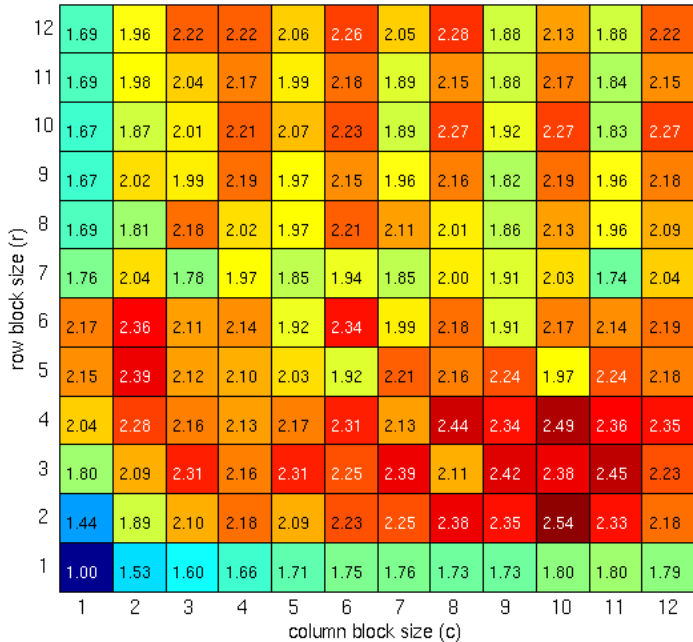
Ultra 2i - 11% ofile [ref=35.8 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]



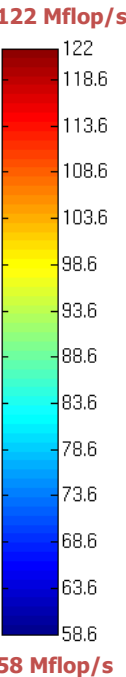
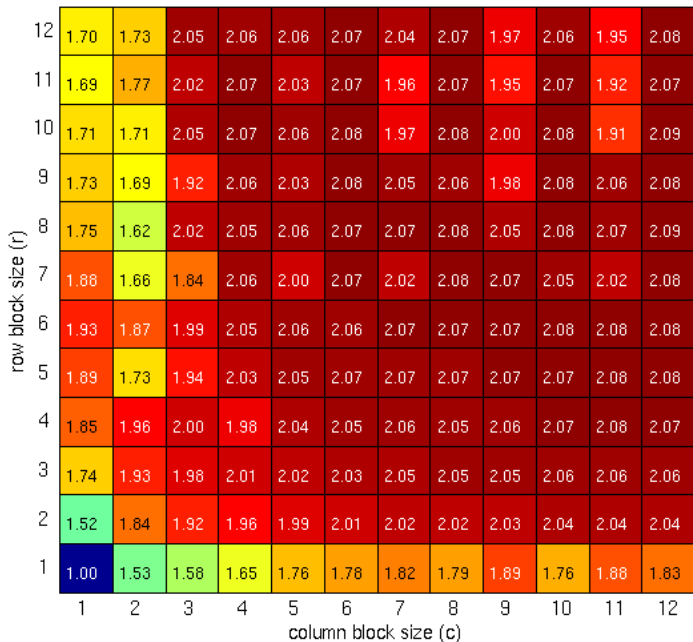
Ultra 3 - 5% Profile [ref=50.3 Mflop/s; 900 MHz Sun Ultra 3, Sun C v6.0]



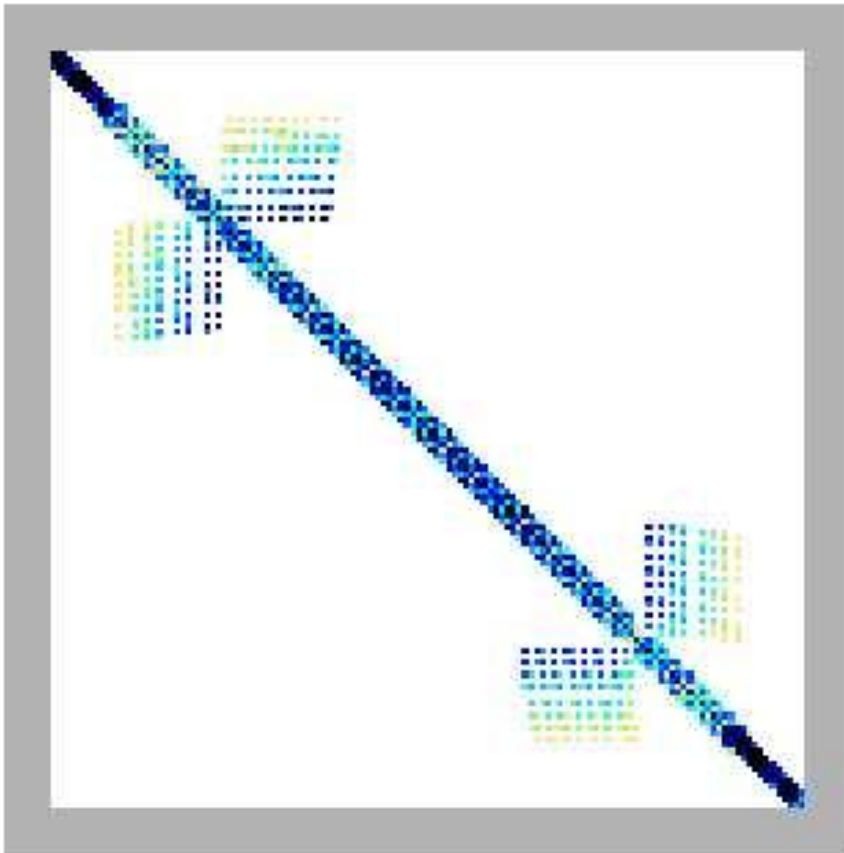
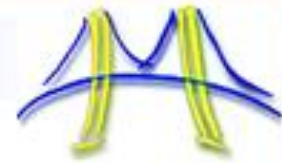
Pentium III - 21% =42.1 Mflop/s; 500 MHz Pentium III, Intel C v7.0]



Pentium III-M - 15% =58.6 Mflop/s; 800 MHz Pentium III-M, Intel C v7.0]

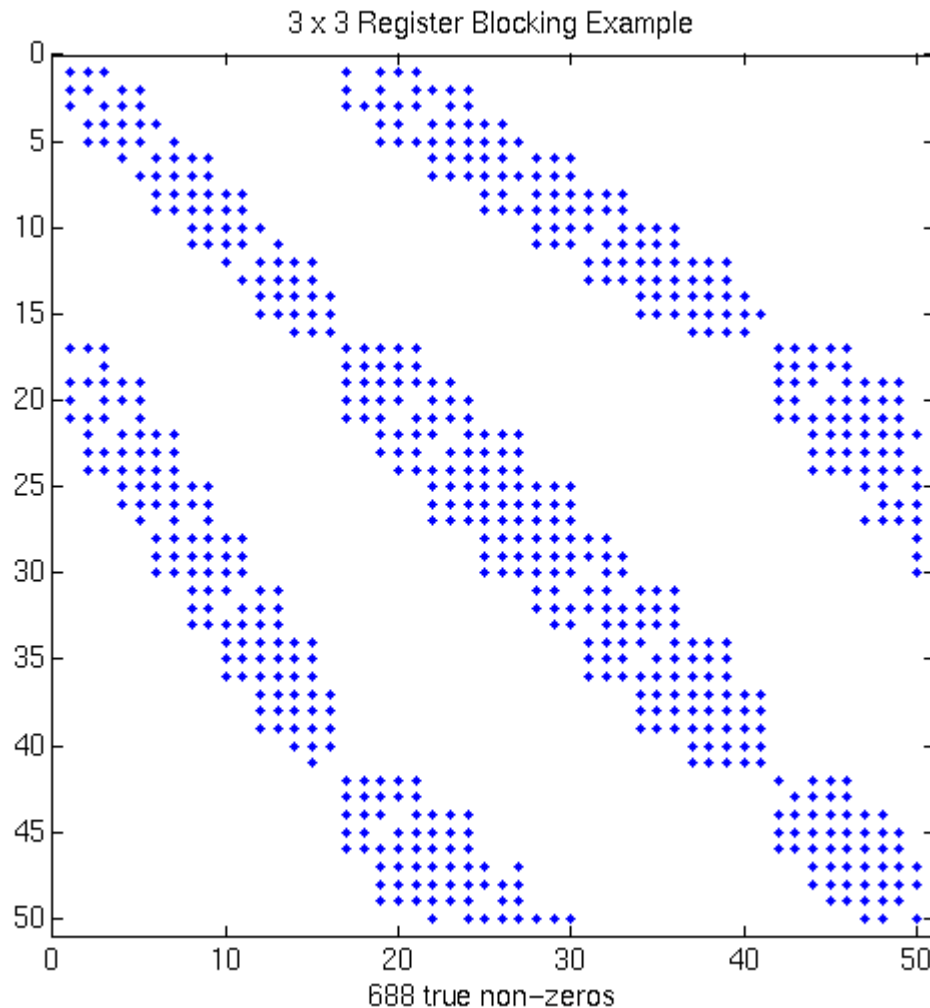
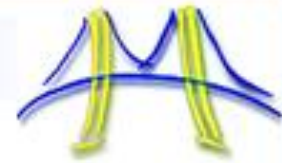


# Another example of tuning challenges



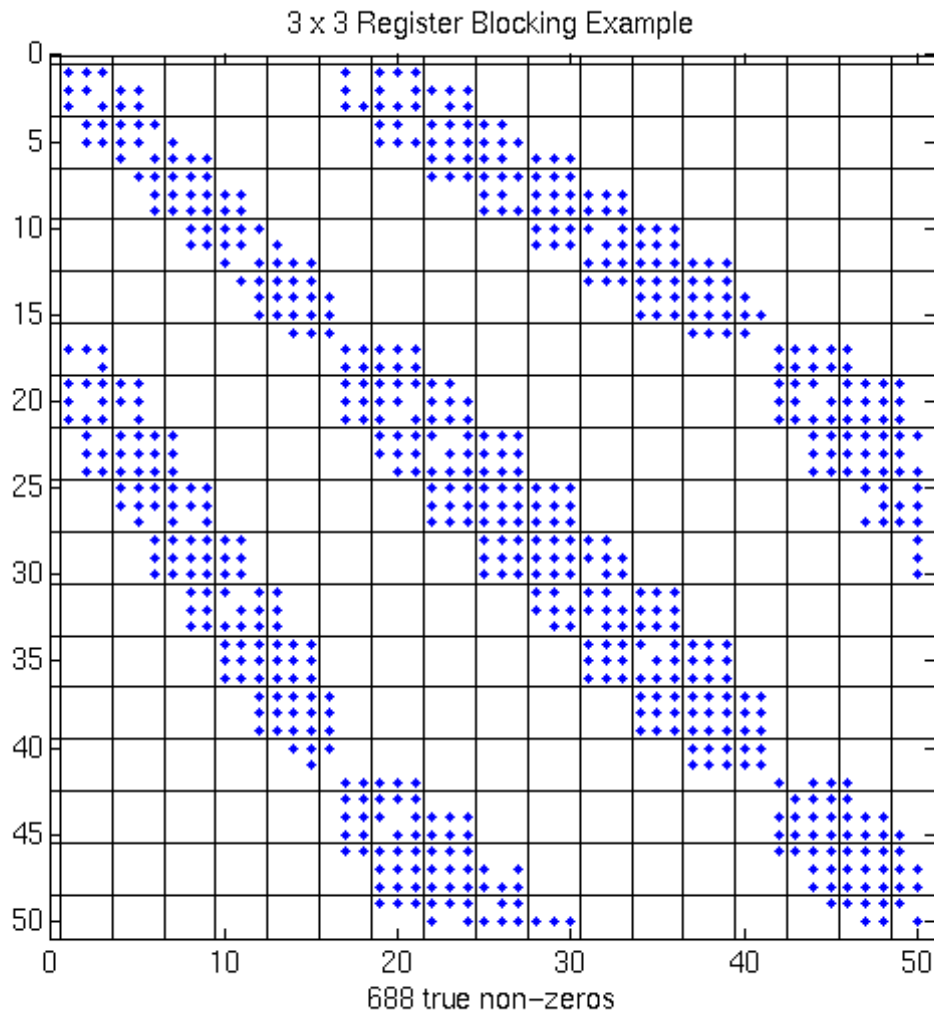
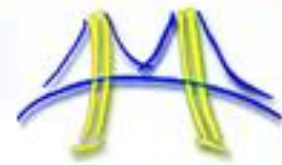
- More complicated non-zero structure in general
- $N = 16614$
- $NNZ = 1.1M$

# Zoom in to top corner



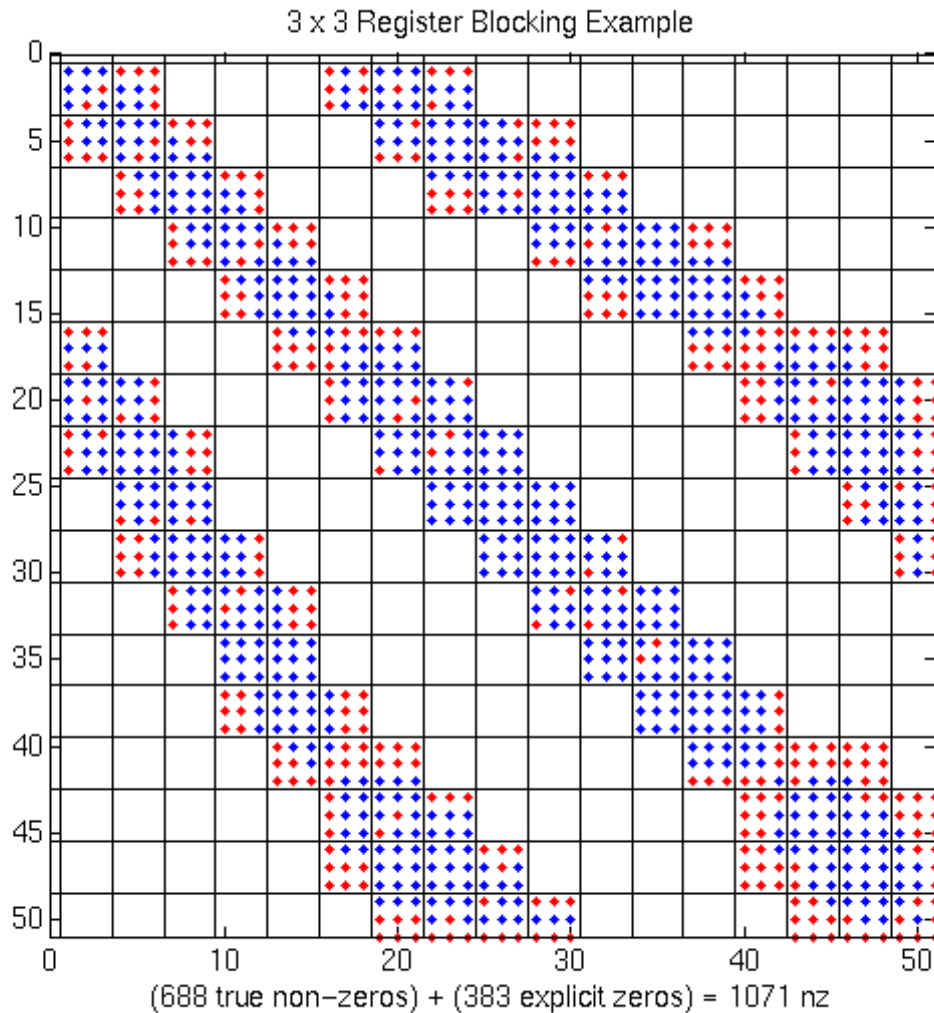
- More complicated non-zero structure in general
- $N = 16614$
- $NNZ = 1.1M$

# 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!

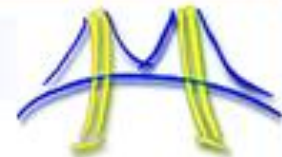


- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
  - Fill-in explicit zeros
  - Unroll 3x3 block multiplies
  - “Fill ratio” = 1.5



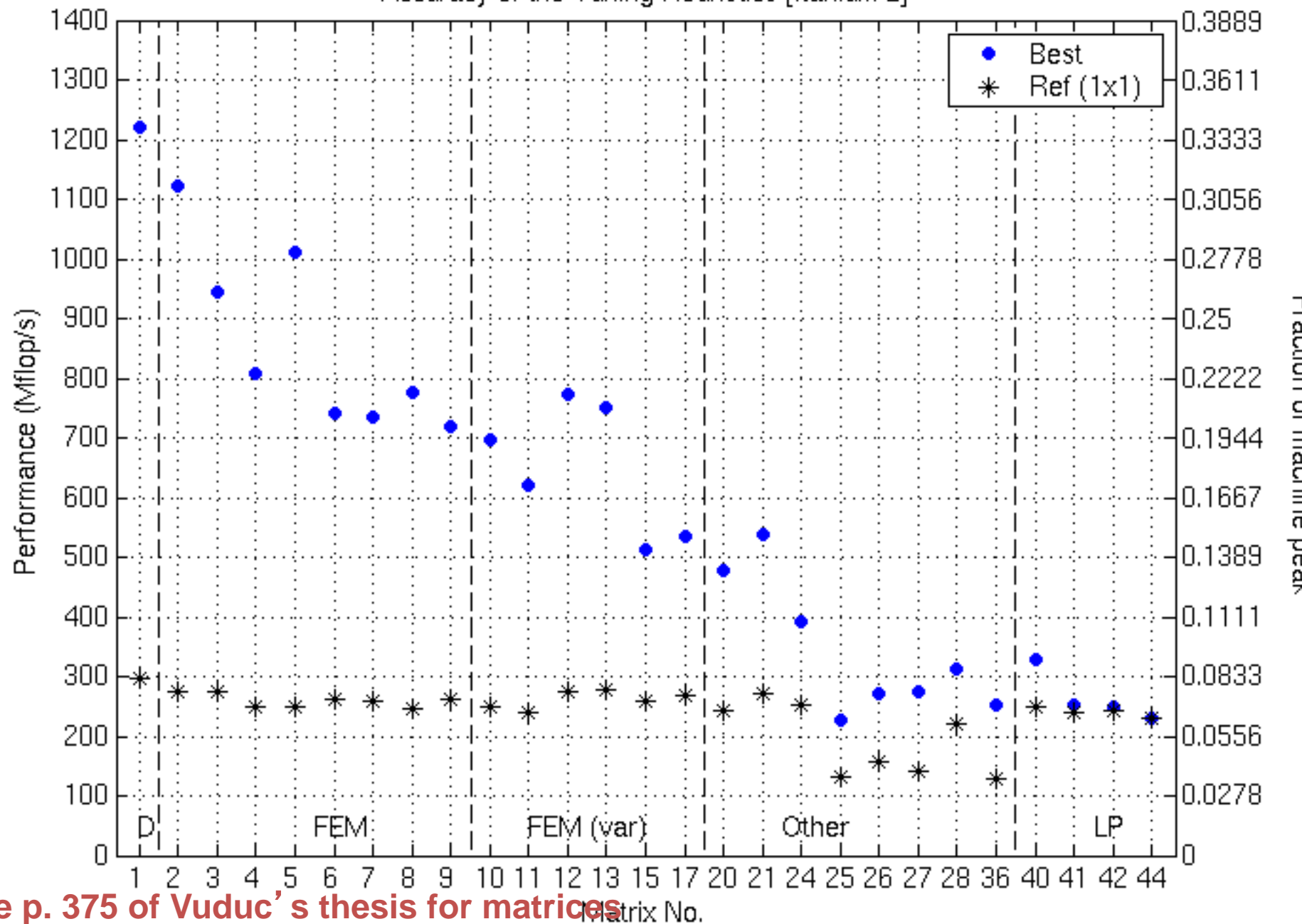


# Selecting Register Block Size $r \times c$



- **Off-line benchmark**
  - Precompute **Mflops( $r, c$ )** using dense  $A$  for each  $r \times c$
  - Once per machine/architecture
- **Run-time “search”**
  - Sample  $A$  to estimate **Fill( $r, c$ )** for each  $r \times c$
  - Control cost =  $O(s \cdot \text{nnz})$  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**  $\sim \text{Fill}(r, c) / \text{Mflops}(r, c)$
- **Cost of tuning**
  - Lower bound: convert matrix in 5 to 40 unblocked SpMV
  - Heuristic: 1 to 11 SpMVs
- **Tuning only useful when we do many SpMVs**
  - Common case, eg in sparse solvers

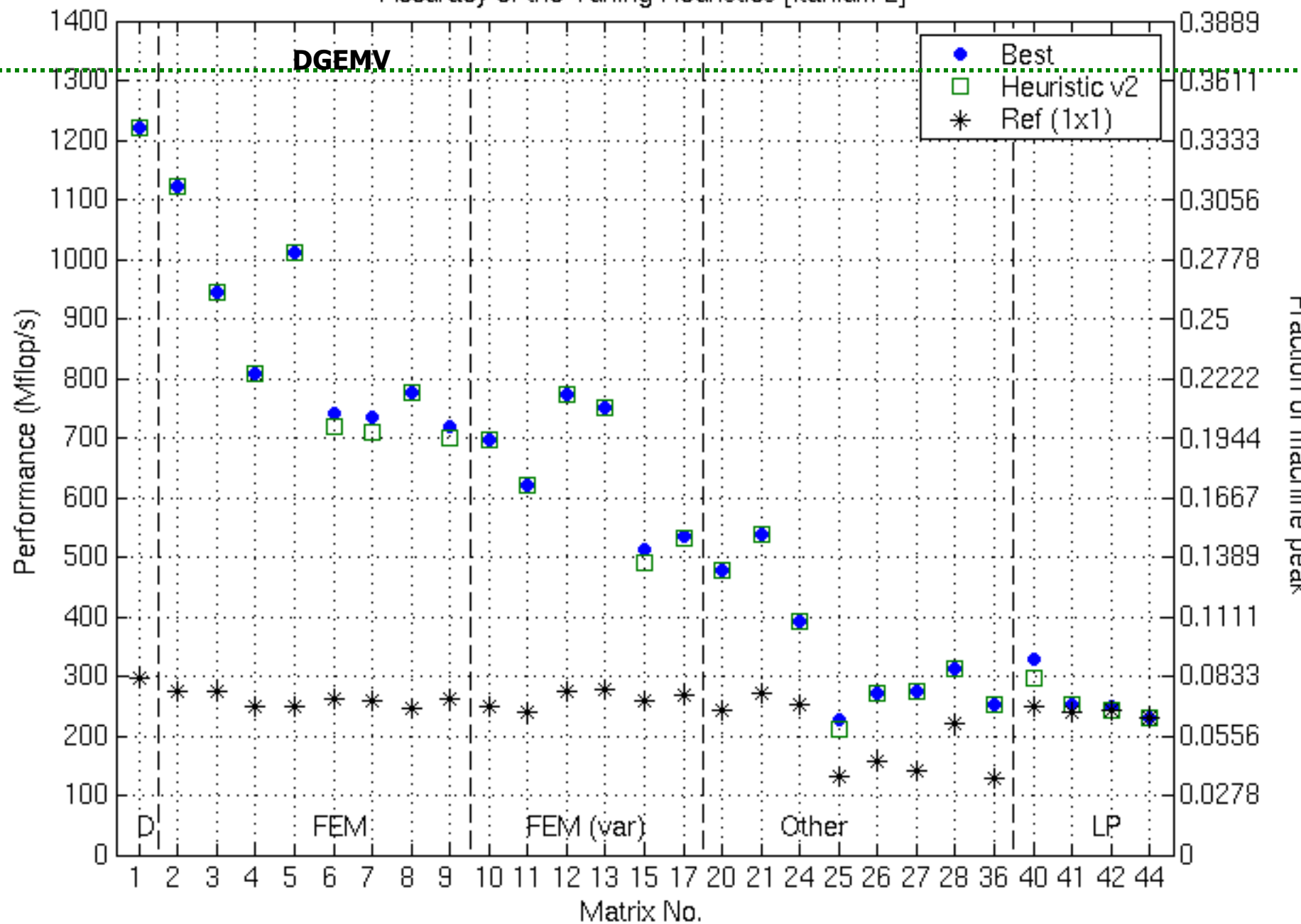
Accuracy of the Tuning Heuristics [Itanium 2]



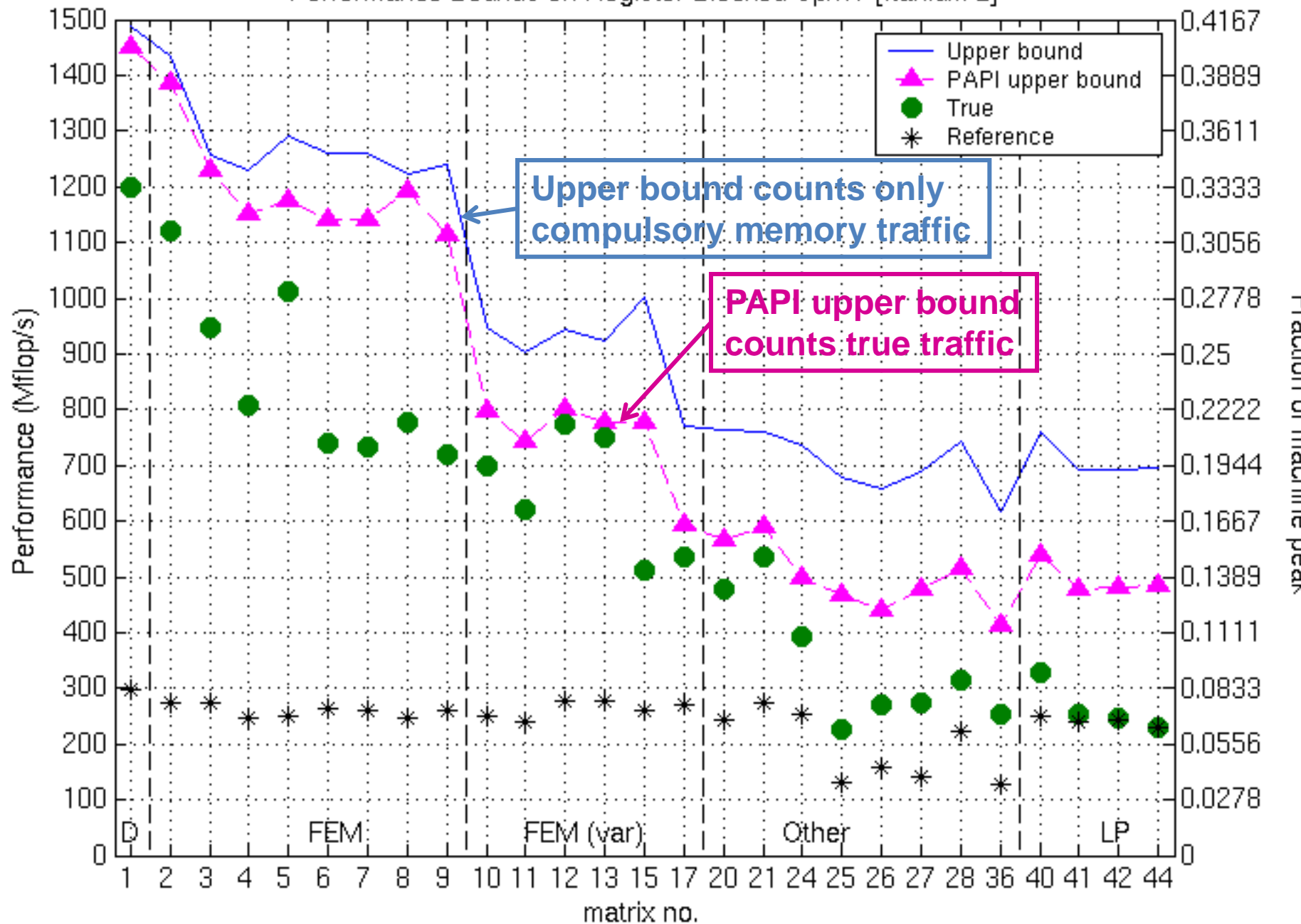
See p. 375 of Vuduc's thesis for matrices

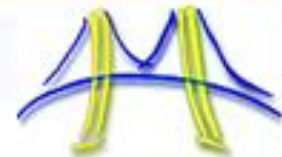
NOTE: "Fair" flops used (ops on explicit zeros not counted as "work")

Accuracy of the Tuning Heuristics [Itanium 2]



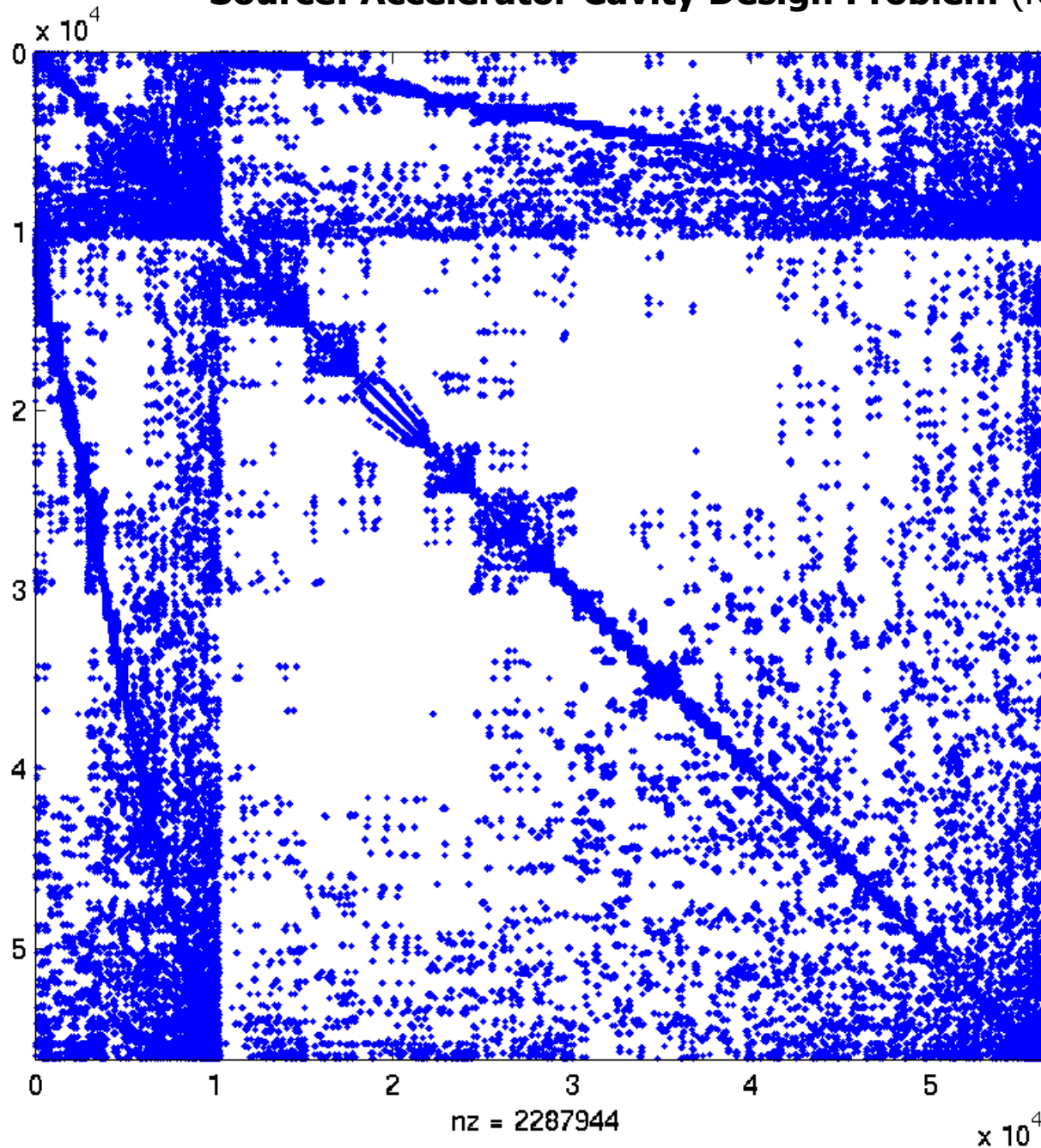
Performance Bounds on Register Blocked SpMV [Itanium 2]





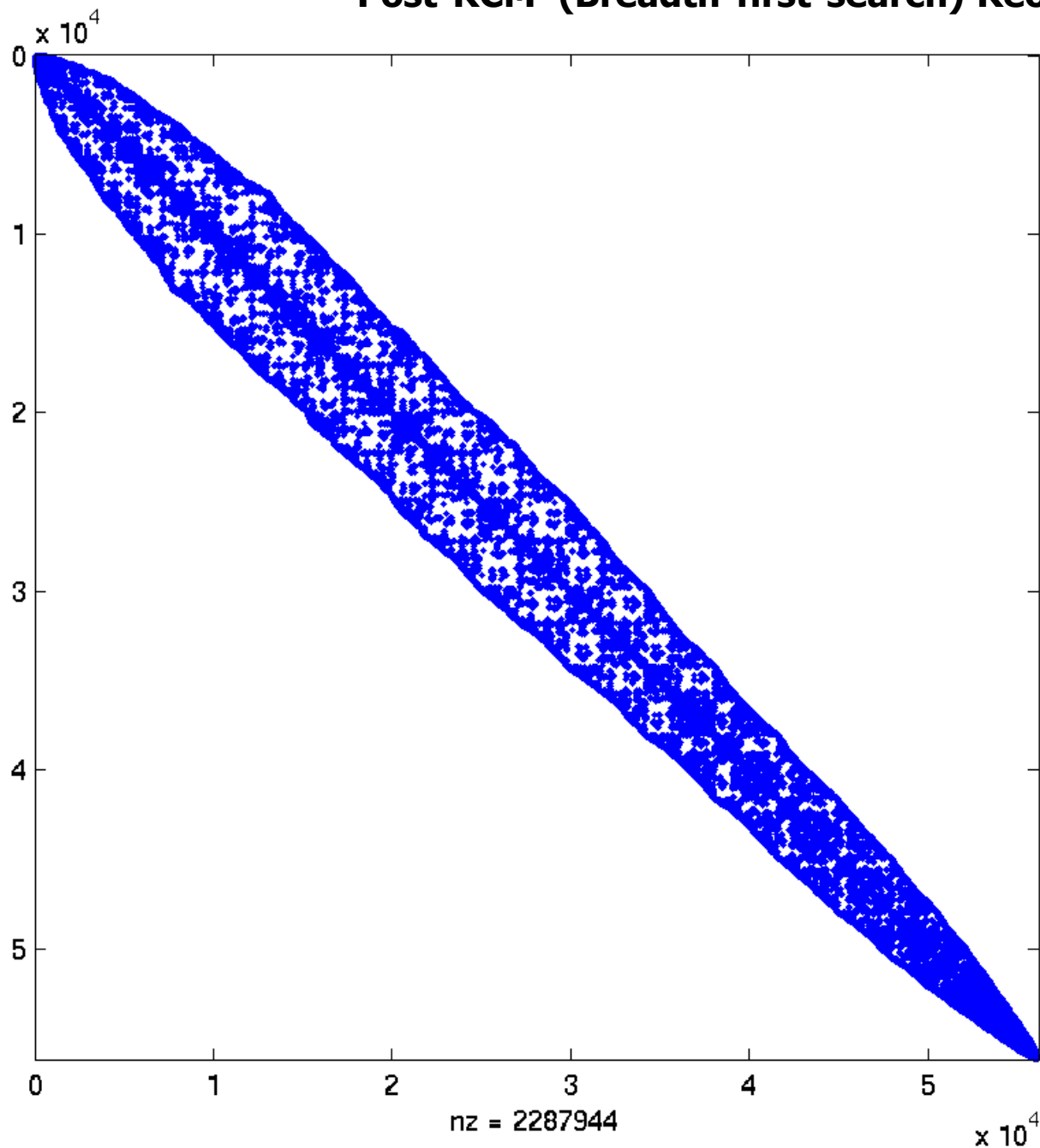
- Optimizations for SpMV
  - **Register blocking (RB)**: up to **4x** over CSR
  - **Variable block splitting**: **2.1x** over CSR, **1.8x** over RB
  - **Diagonals**: **2x** over CSR
  - **Reordering** to create dense structure + **splitting**: **2x** over CSR
  - **Symmetry**: **2.8x** over CSR, 2.6x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

## Source: Accelerator Cavity Design Problem (Ko via Husbands)



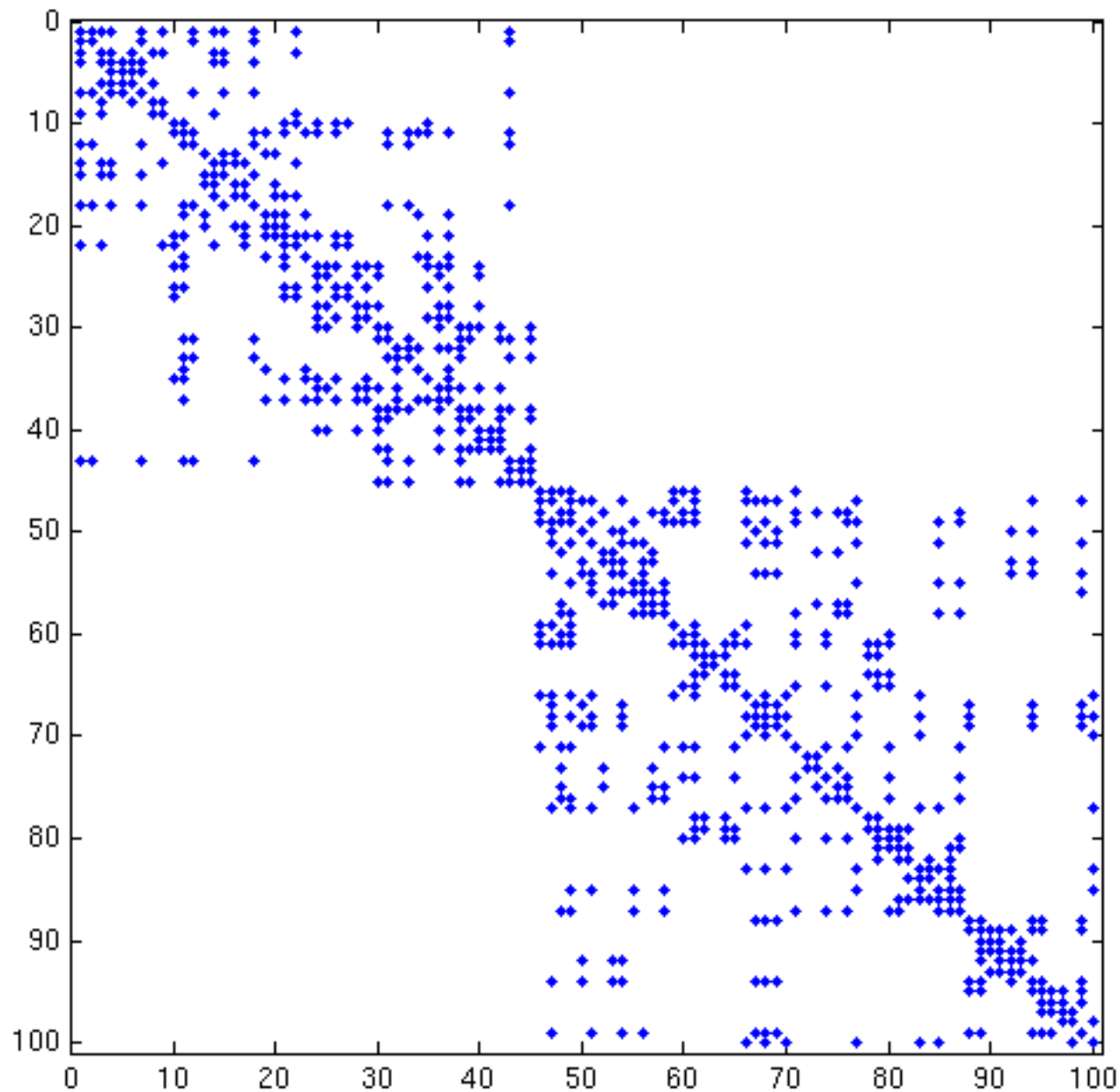
Can we reorder the rows and columns to create dense blocks, to accelerate SpMV?

## Post-RCM (Breadth-first-search) Reordering



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

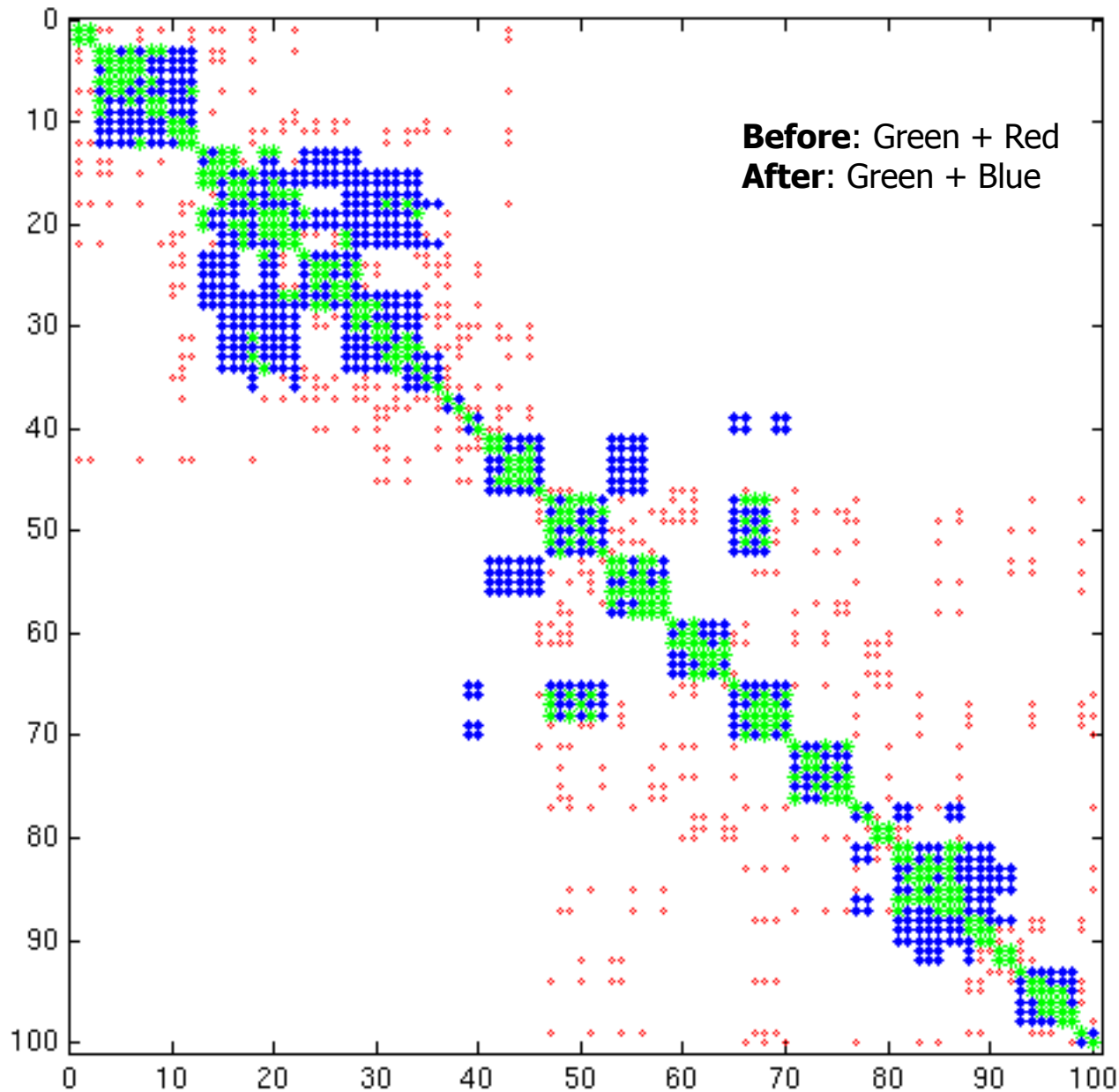
## 100x100 Submatrix Along Diagonal



Here is the top 100x100 submatrix before RCM

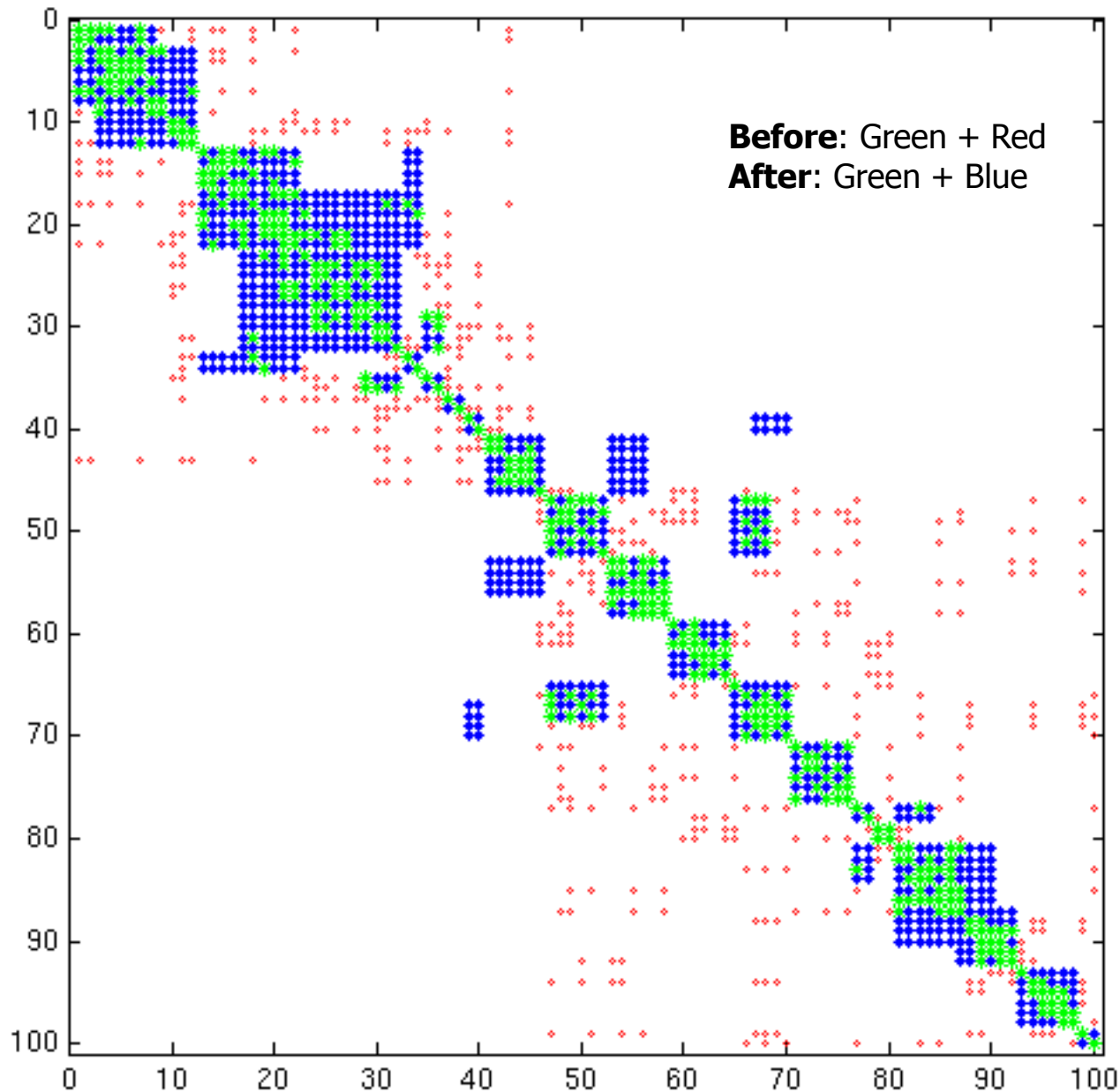


## “Microscopic” Effect of RCM Reordering



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):

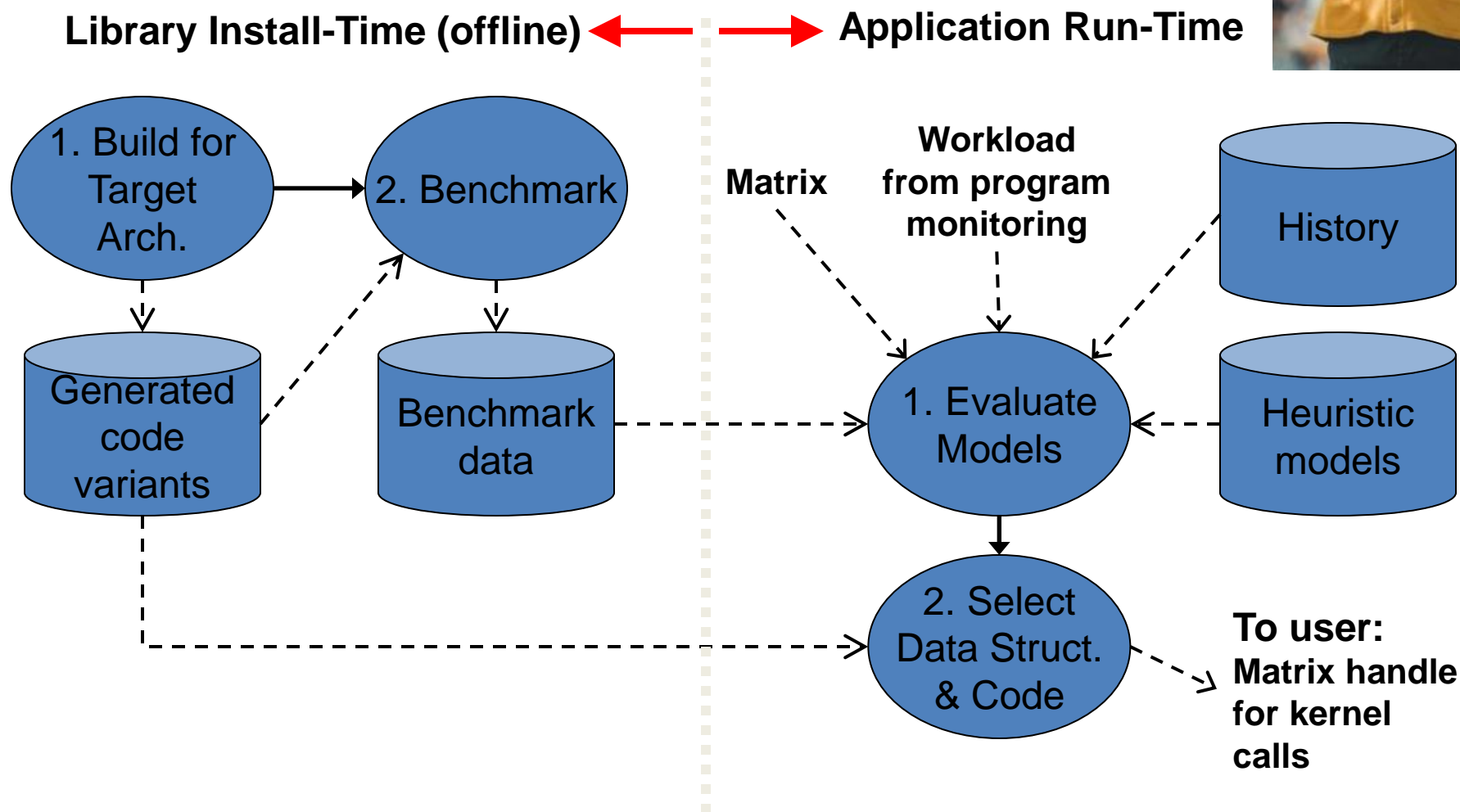
Itanium 2: 1.7x  
Pentium 4: 2.1x  
Power 4: 2.1x  
Ultra 3: 3.3x

# Optimized Sparse Kernel Interface - OSKI



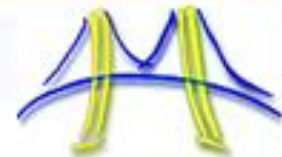
- Provides sparse kernels automatically tuned for user's matrix & machine
  - BLAS-style functionality: SpMV,  $Ax$  &  $A^T y$ , TrSV
  - Hides complexity of run-time tuning
  - Includes new, faster locality-aware kernels:  $A^T A x$ ,  $A^k x$
- Faster than standard implementations
  - Up to 4x faster matvec, 1.8x trisolve, 4x  $A^T A * x$
- 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](http://Bebop.cs.berkeley.edu/oski)
- Current work: adding multicore, other optimizations - pOSKI

# How OSKI Tunes (Overview)



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

# How to Call OSKI: Basic Usage

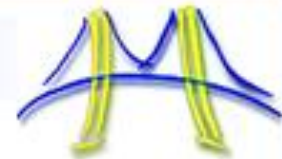


- May gradually migrate existing apps
  - Step 1: “Wrap” existing data structures
  - Step 2: Make BLAS-like kernel calls

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

```
/* Compute  $y = \beta \cdot y + \alpha \cdot A \cdot x$ , 500 times */  
for( i = 0; i < 500; i++ )  
    my_matmult( ptr, ind, val,  $\alpha$ , x,  $\beta$ , y );
```

# How to Call OSKI: Basic Usage

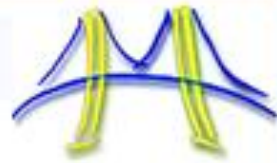


- May gradually migrate existing apps
  - Step 1: “Wrap” existing data structures
  - Step 2: Make BLAS-like kernel calls

```
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  $y = \beta \cdot y + \alpha \cdot A \cdot x$ , 500 times */
for( i = 0; i < 500; i++ )
    my_matmult( ptr, ind, val,  $\alpha$ , x,  $\beta$ , 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  $y = \beta \cdot y + \alpha \cdot A \cdot x$ , 500 times */
for( i = 0; i < 500; i++ )
    oski_MatMult(A_tunable, OP_NORMAL,  $\alpha$ , x_view,  $\beta$ , 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,  $\alpha$ , x_view,  $\beta$ , 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,  $\alpha$ , x_view,  $\beta$ , y_view);
```



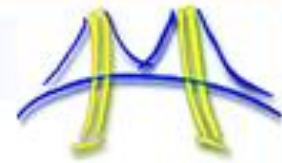
# How the User Calls OSKI: Implicit Tuning



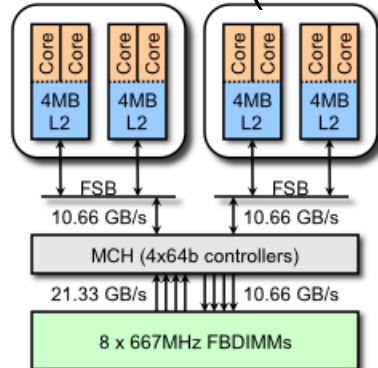
- Ask library to infer workload
  - Library profiles all kernel calls
  - May periodically re-tune

```
oski_matrix_t A_tunable = oski_CreateMatCSR( ... );  
/* ... */  
  
for( i = 0; i < 500; i++ ) {  
    oski_MatMult(A_tunable, OP_NORMAL,  $\alpha$ , x_view,  $\beta$ , y_view);  
    oski_TuneMat(A_tunable); /* Ask OSKI to tune */  
}
```

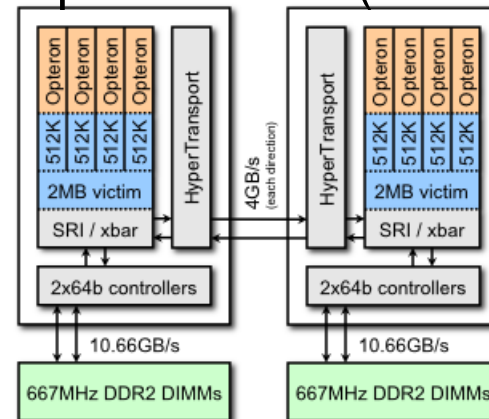
# Multicore SMPs Used for Tuning SpMV



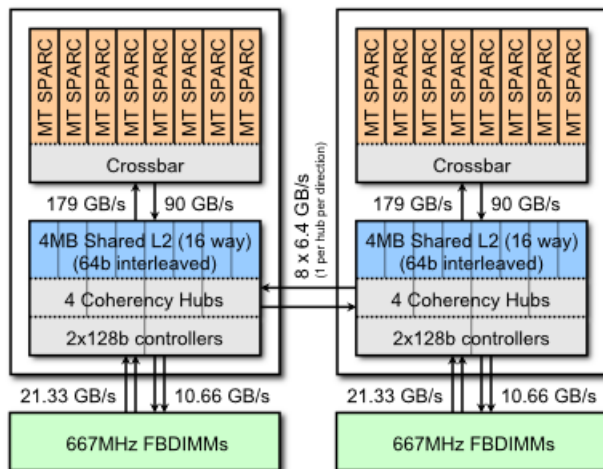
Intel Xeon E5345 (Clovertown)



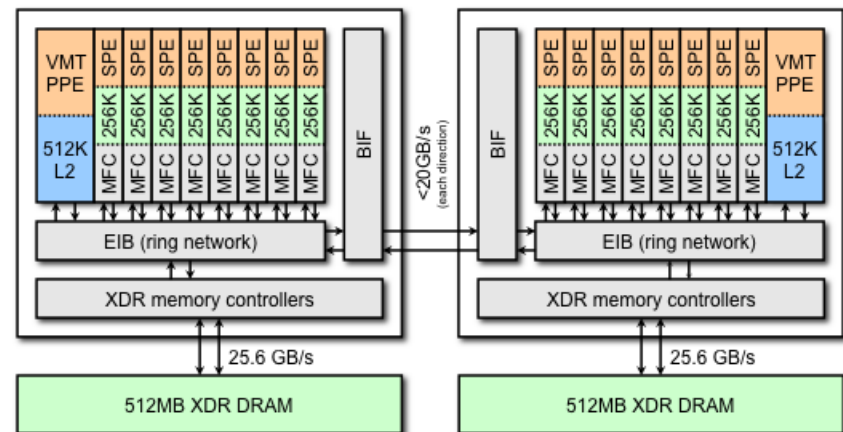
AMD Opteron 2356 (Barcelona)



Sun T2+ T5140 (Victoria Falls)

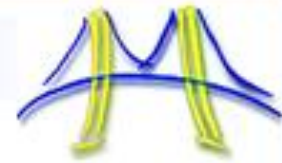


IBM QS20 Cell Blade





# 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

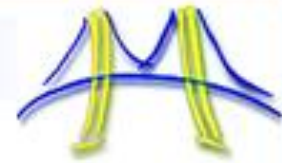
## Sun T2+ T5140 (Victoria Falls)

- 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



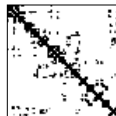
- All bigger than the caches of our SMPs

2K x 2K Dense matrix  
stored in sparse format

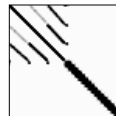


Dense

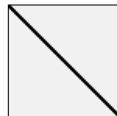
Well Structured  
(sorted by nonzeros/row)



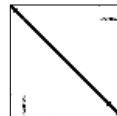
Protein



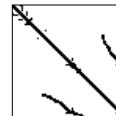
FEM /  
Spheres



FEM /  
Cantilever



Wind  
Tunnel



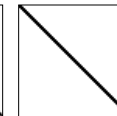
FEM /  
Harbor



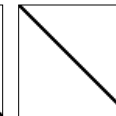
QCD



FEM /  
Ship



Economics



Epidemiology

Poorly Structured  
hodgepodge



FEM /  
Accelerator



Circuit



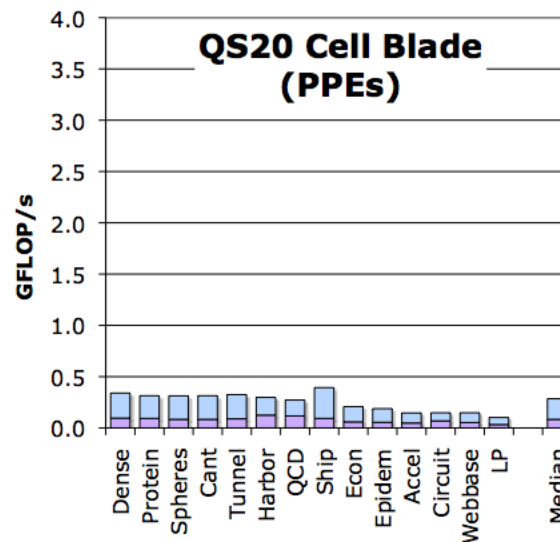
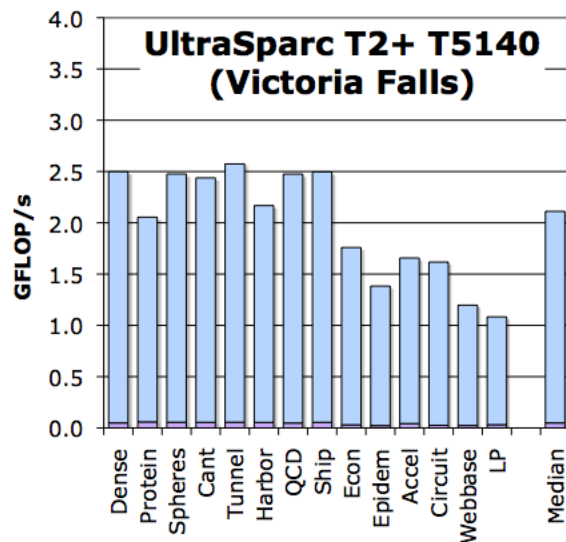
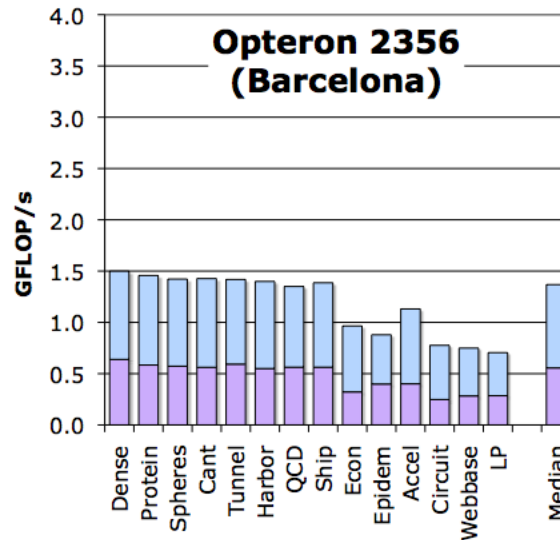
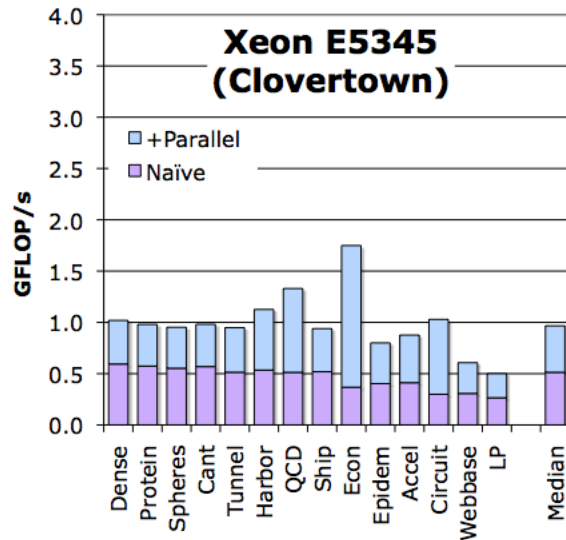
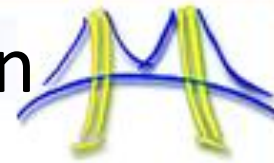
webbase

Extreme Aspect Ratio  
(linear programming)



LP

# SpMV Performance: Naive parallelization



- Out-of-the box SpMV performance on a suite of 14 matrices

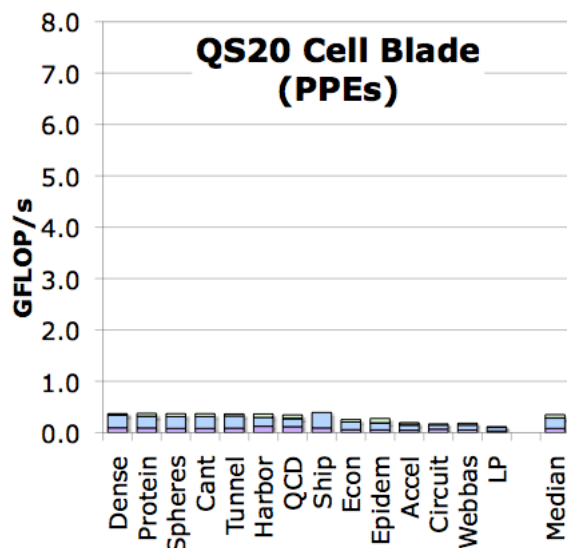
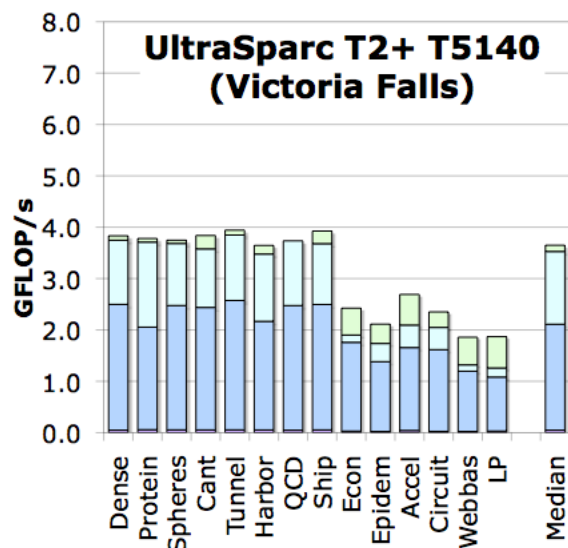
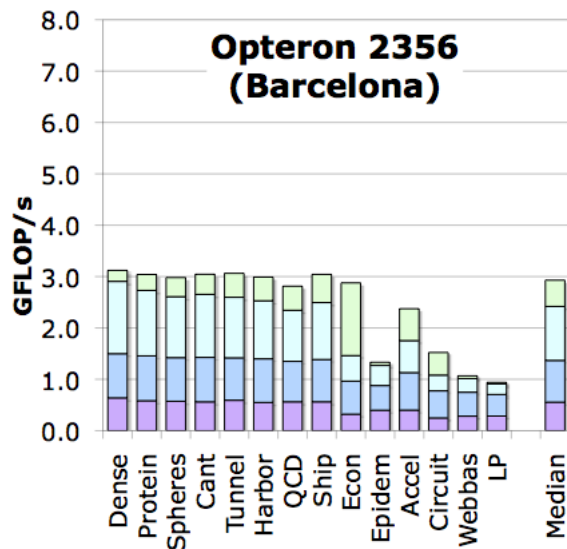
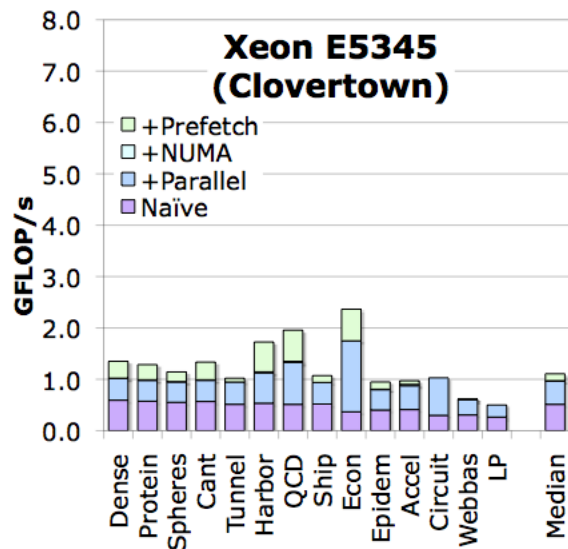
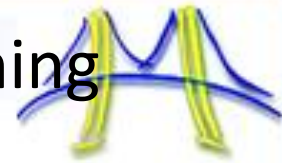
- Scalability isn't great:  
Compare to # threads

8 8  
128 16

Naïve Pthreads  
Naïve

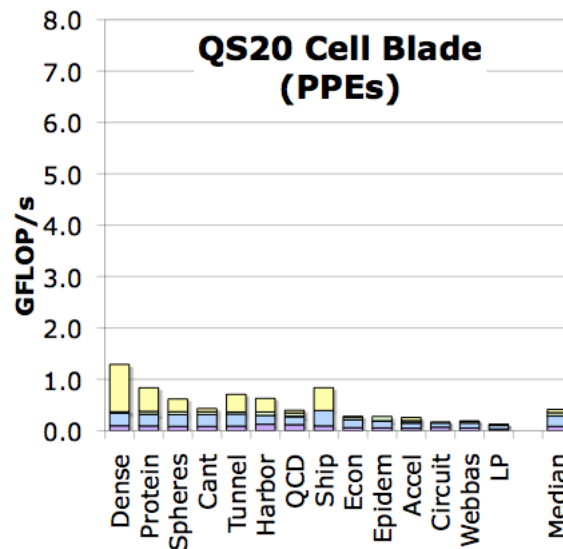
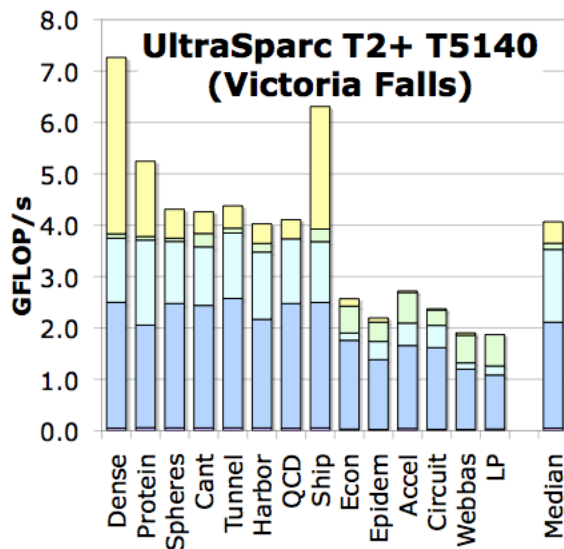
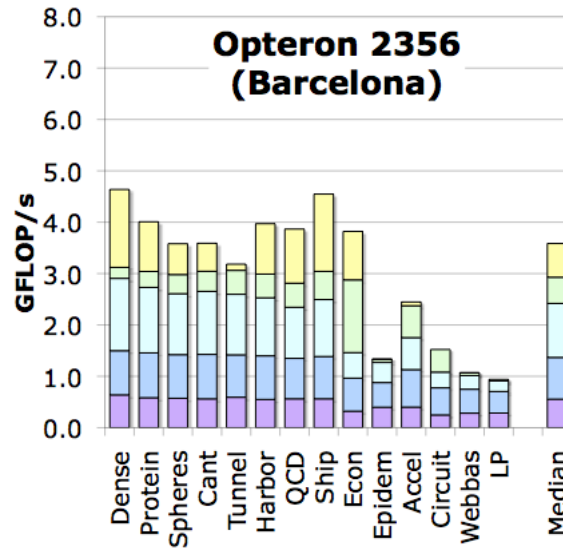
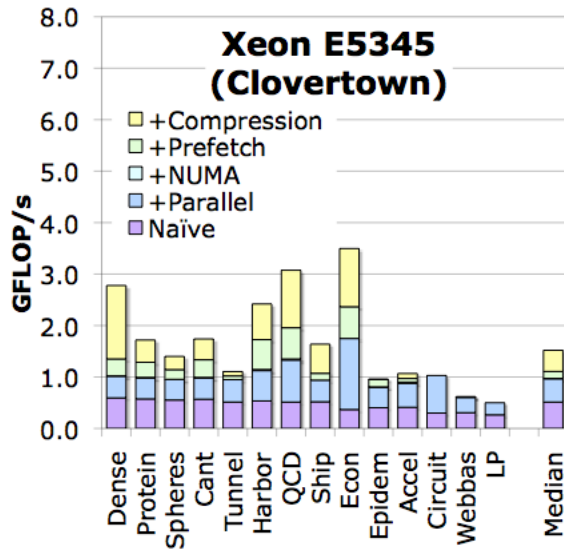
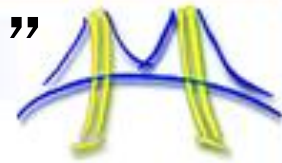


# SpMV Performance: NUMA and Software Prefetching



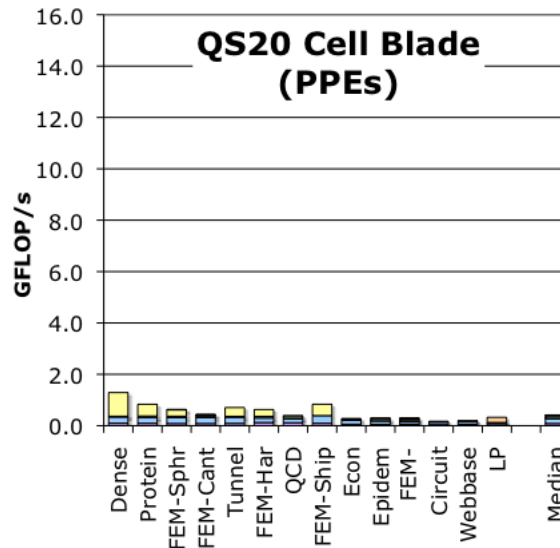
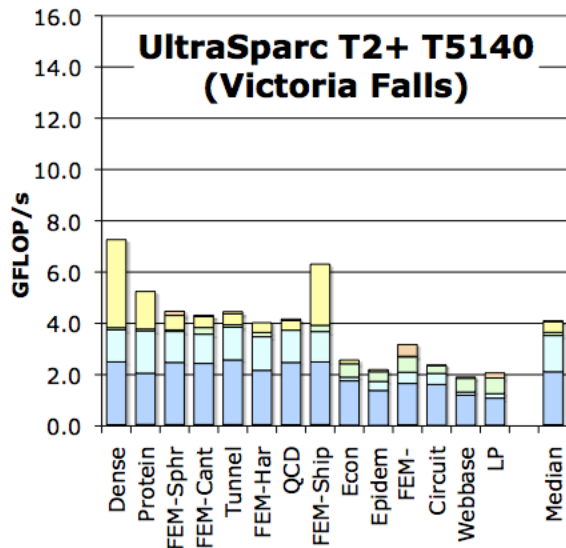
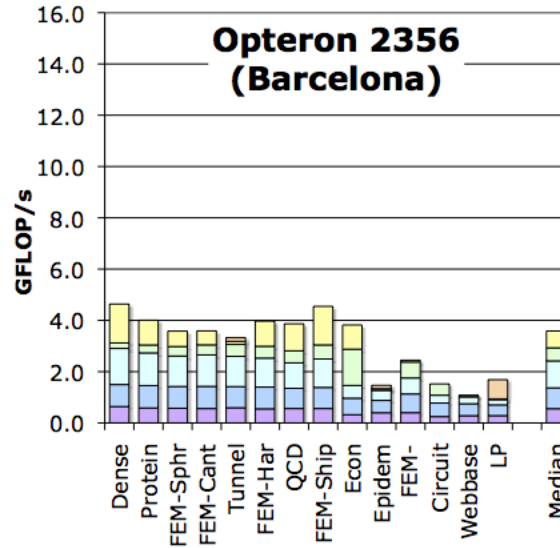
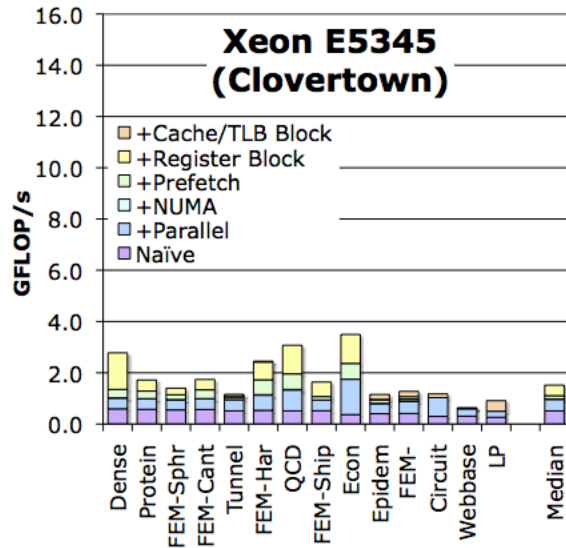
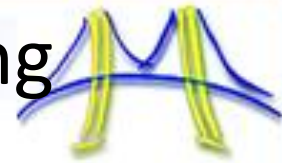
- ❖ NUMA-aware allocation is essential on NUMA SMPs.
- ❖ Explicit software prefetching can boost bandwidth and change cache replacement policies
- ❖ used **exhaustive** search

# SpMV Performance: “Matrix Compression”



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

# SpMV Performance: cache and TLB blocking

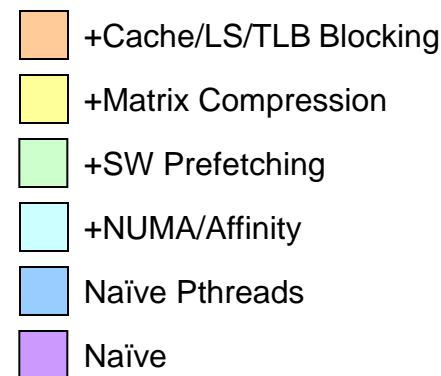
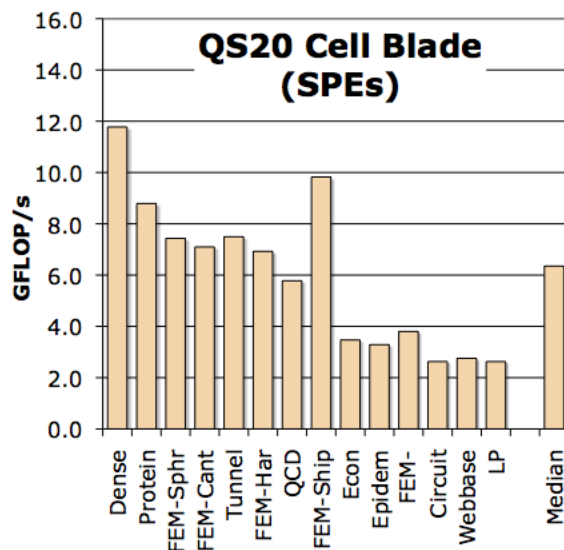
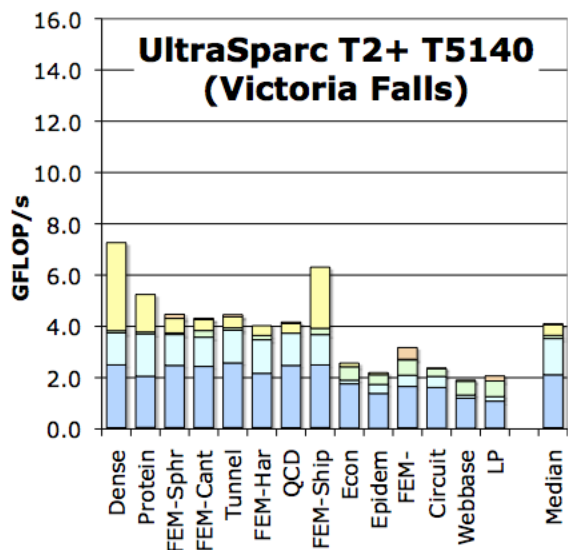
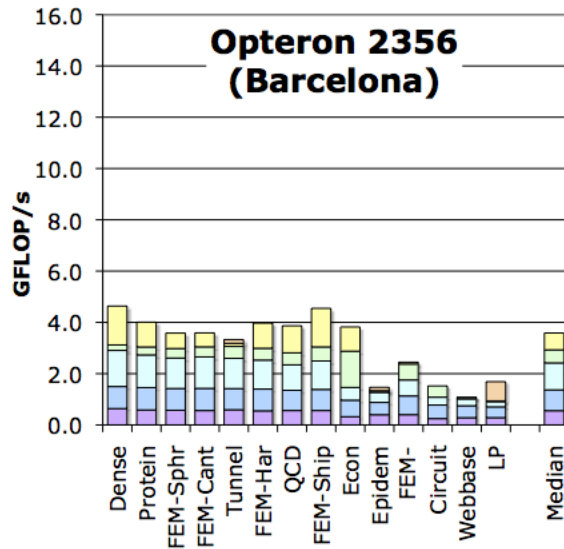
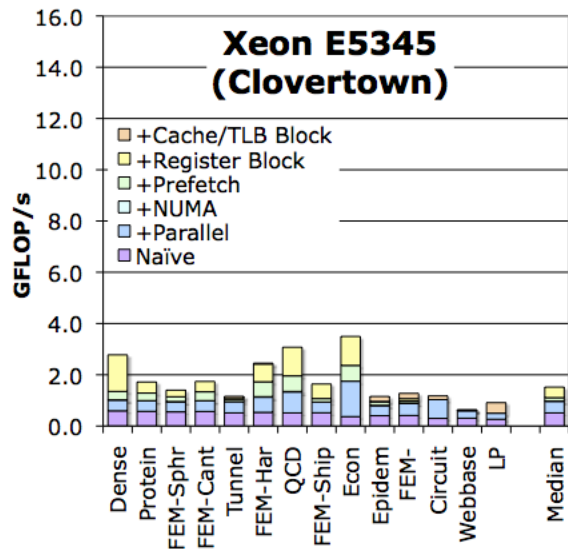
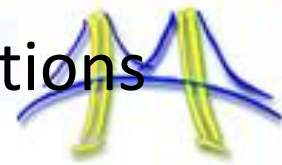


- +Cache/LS/TLB Blocking
- +Matrix Compression
- +SW Prefetching
- +NUMA/Affinity
- Naïve Pthreads
- Naïve

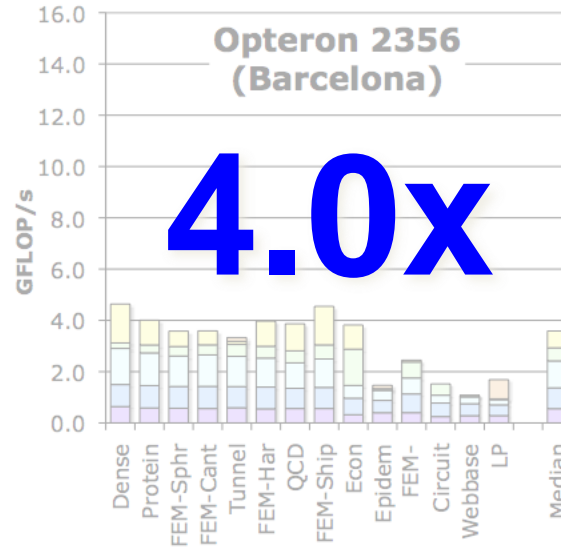
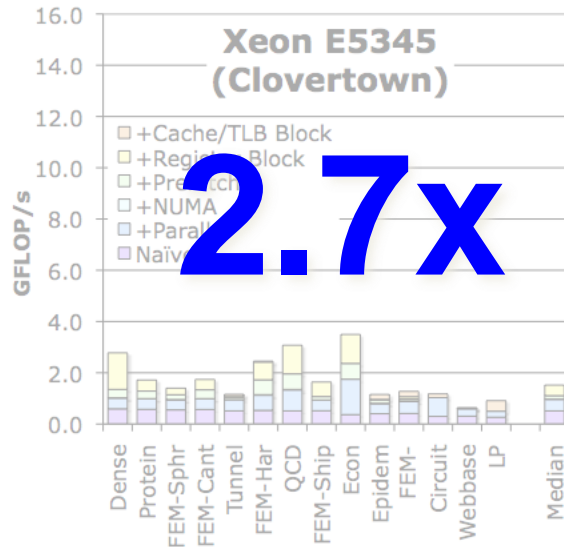
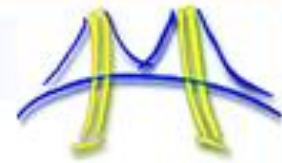




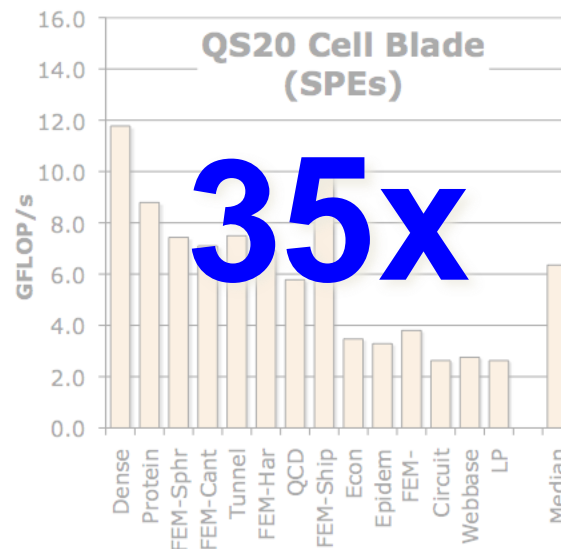
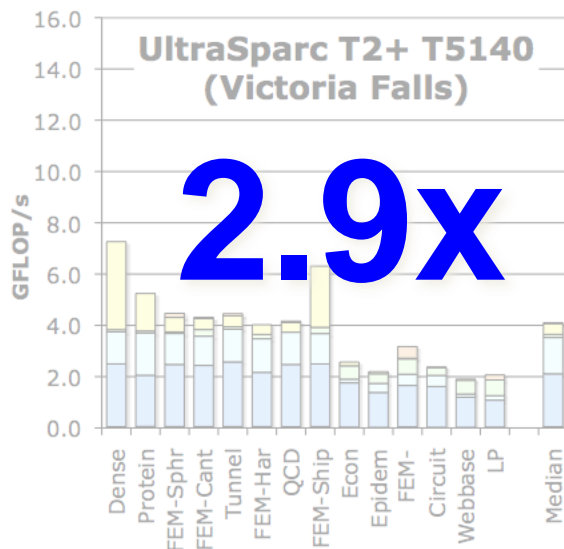
# SpMV Performance: Architecture specific optimizations



# SpMV Performance: max speedup



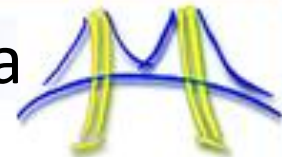
- Fully auto-tuned SpMV performance across the suite of matrices
- Included SPE/local store optimized version
- Why do some optimizations work better on some architectures?



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

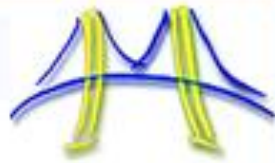


# Avoiding Communication in Sparse Linear Algebra



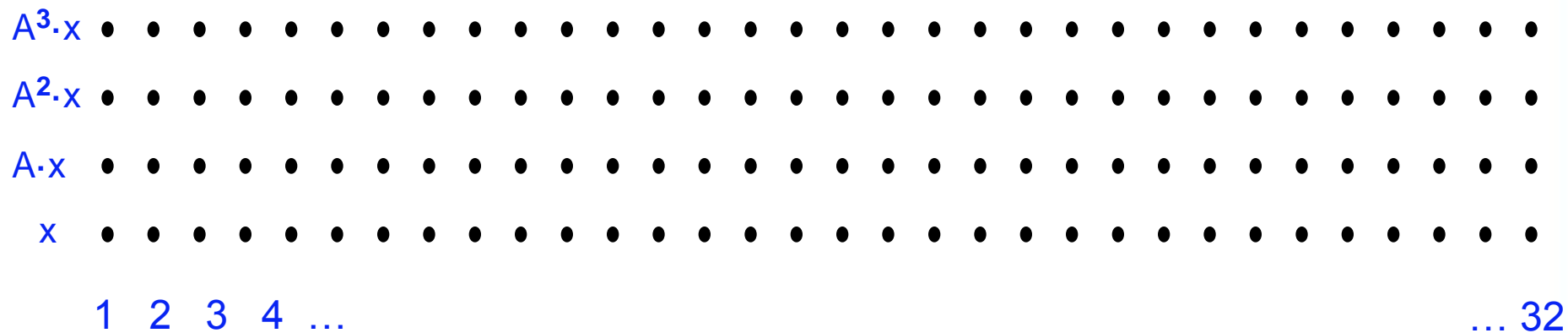
- k-steps of typical iterative solver for  $Ax=b$  or  $Ax=\lambda x$ 
  - Does k SpMV with starting vector (eg with b, if solving  $Ax=b$ )
  - Finds “best” solution among all linear combinations of these k+1 vectors
  - Many such “Krylov Subspace Methods”
    - Conjugate Gradients, GMRES, Lanczos, 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

# Communication Avoiding Kernels:



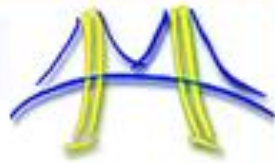
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



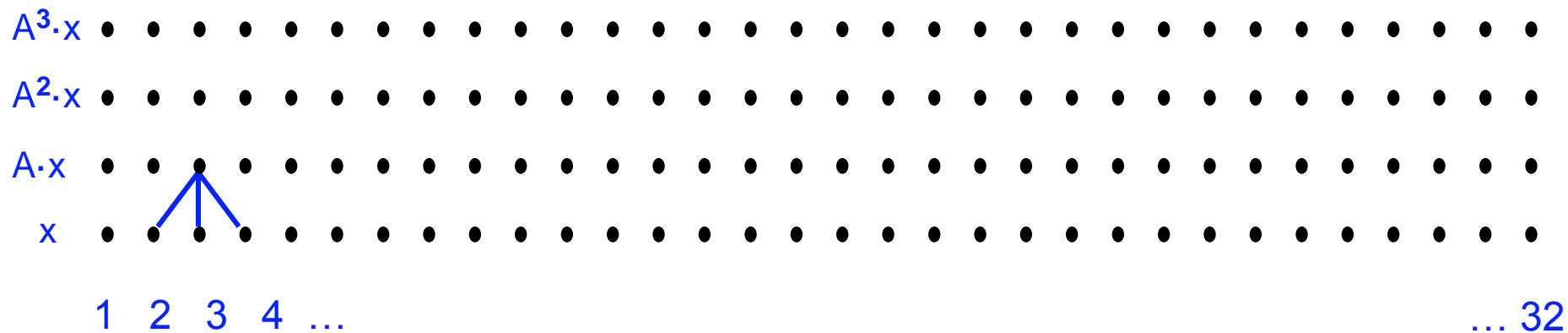
- Example:  $A$  tridiagonal,  $n=32$ ,  $k=3$
- Works for any “well-partitioned”  $A$

# Communication Avoiding Kernels:



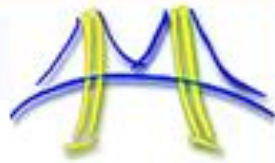
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



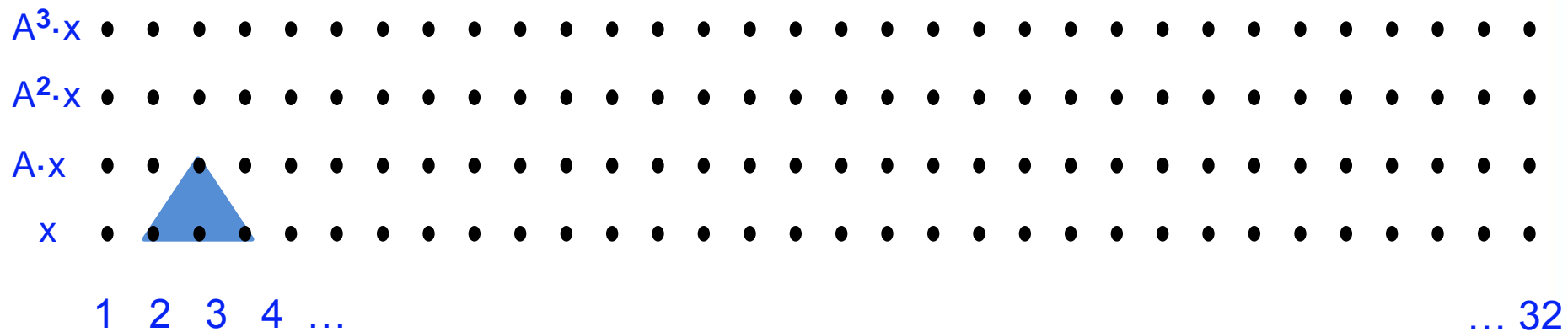
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



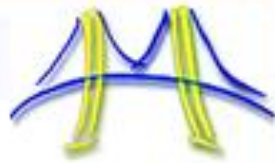
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



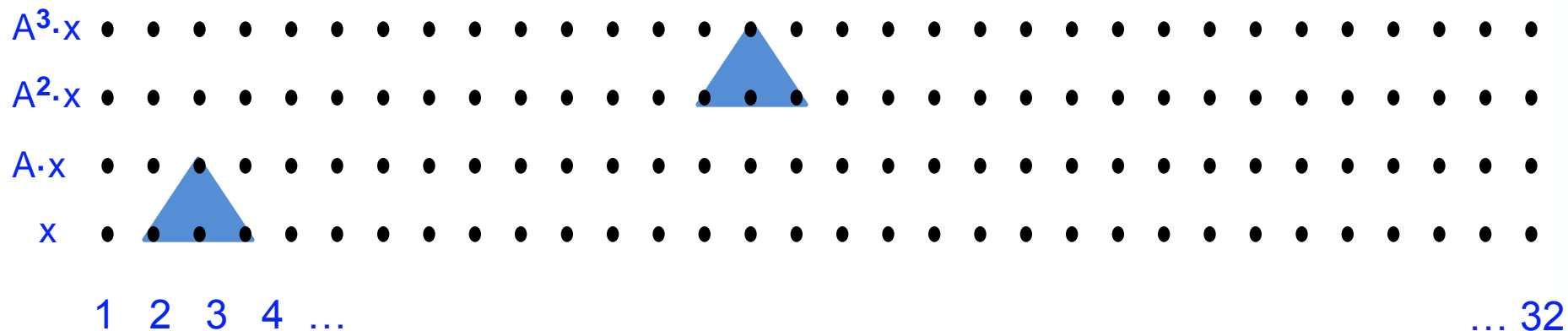
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



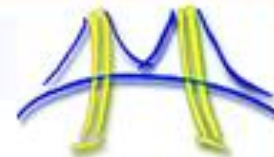
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



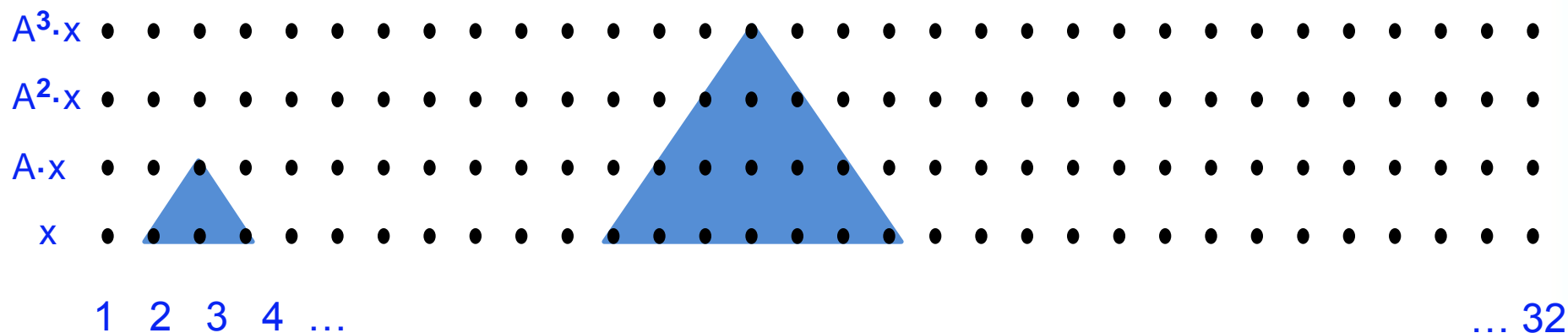
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



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

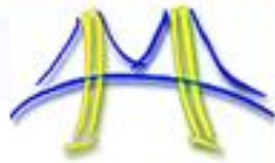
- Replace  $k$  iterations of  $y = A \cdot x$  with  $[Ax, A^2x, \dots, A^kx]$



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

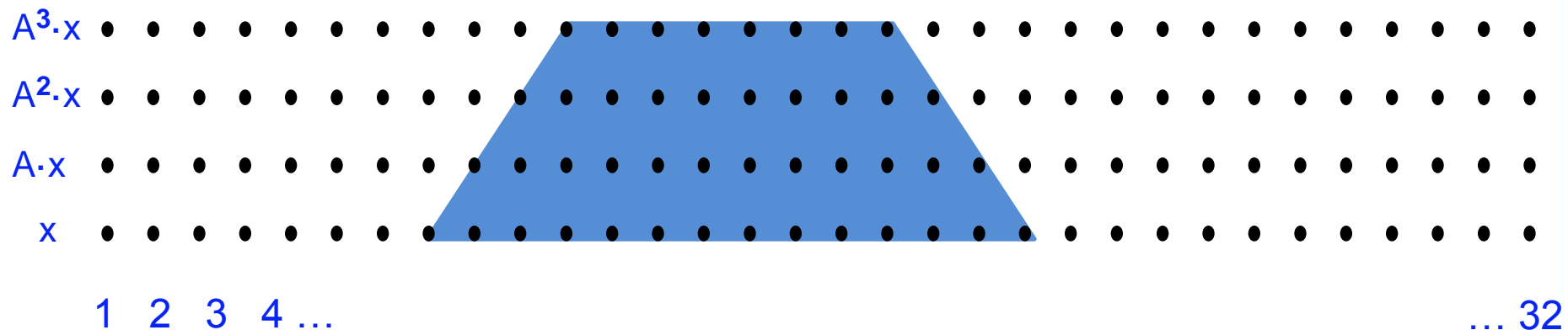


# Communication Avoiding Kernels:



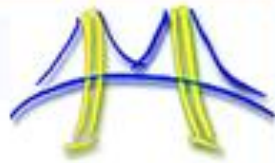
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



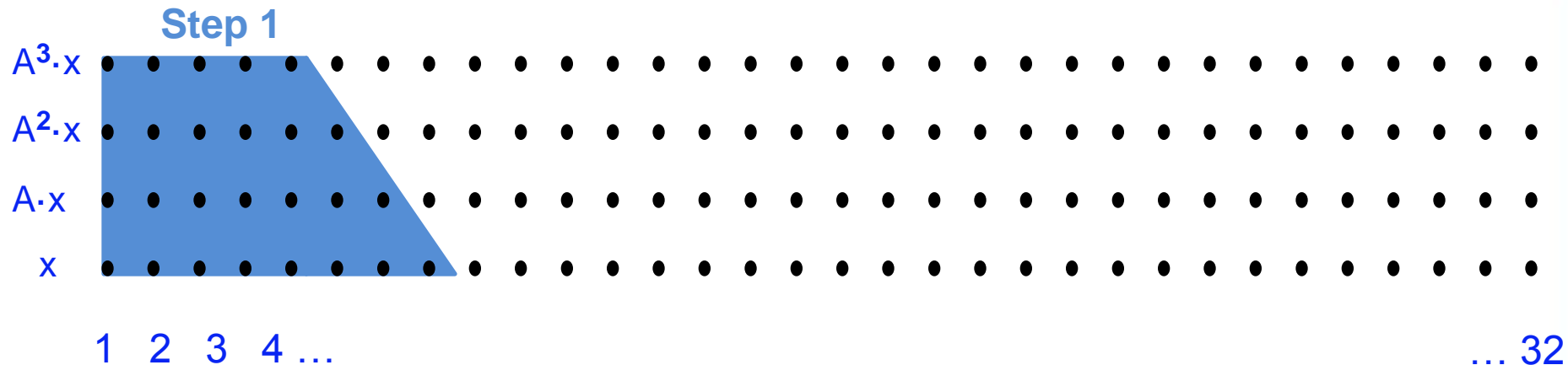
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



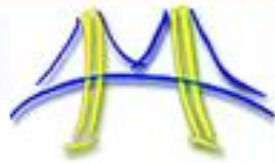
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



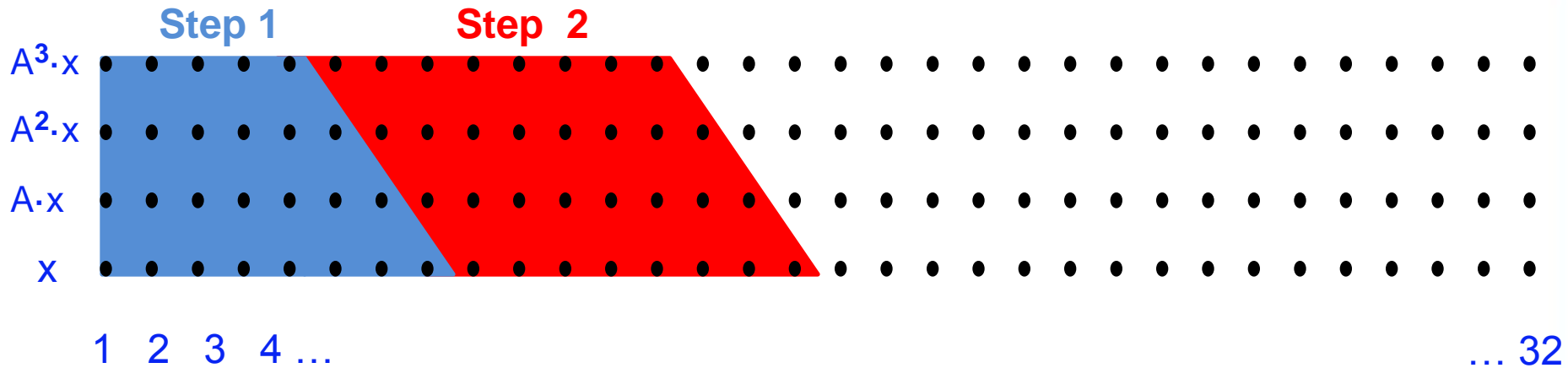
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



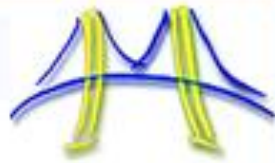
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



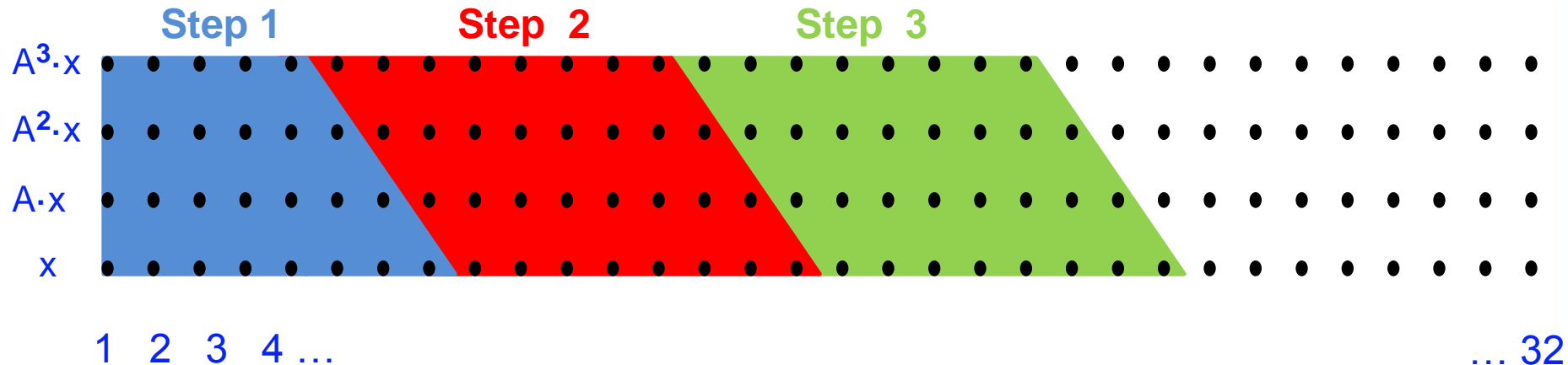
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



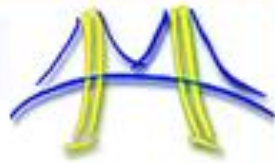
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



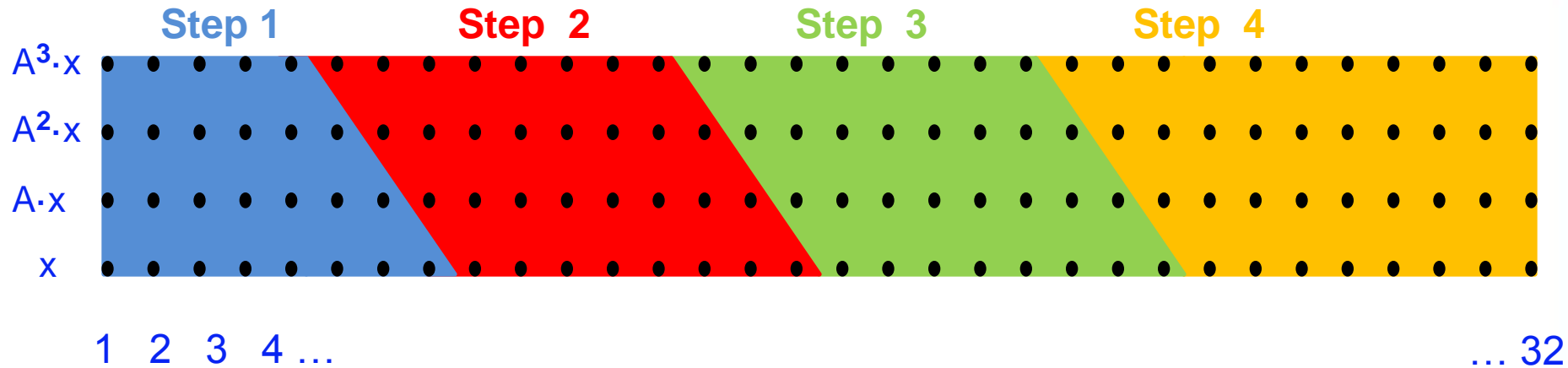
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



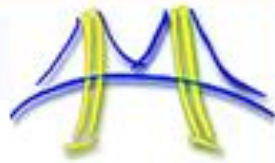
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



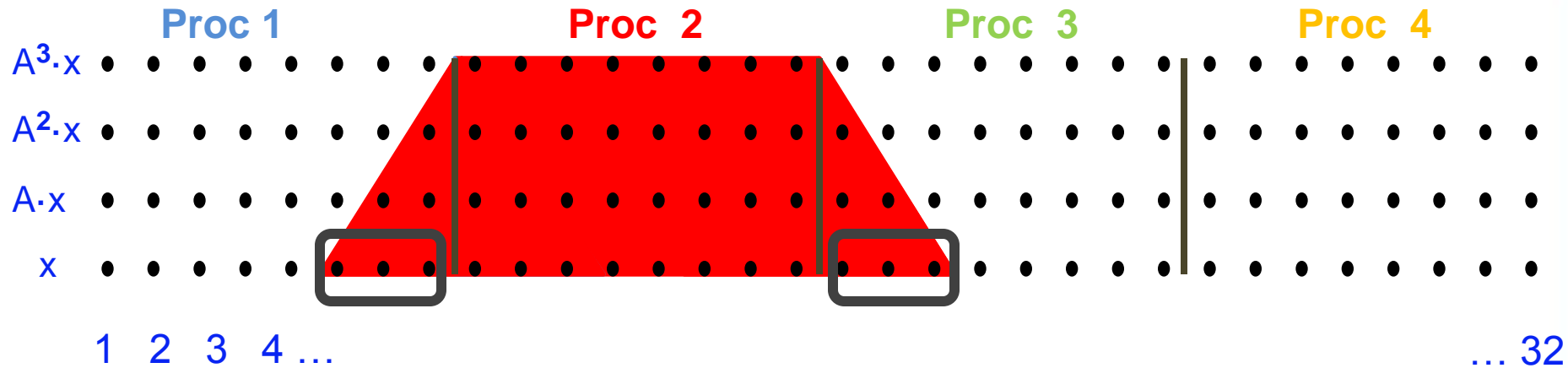
- Example: A tridiagonal,  $n=32$ ,  $k=3$

# Communication Avoiding Kernels:



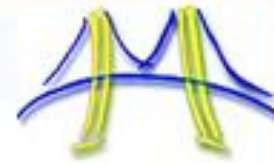
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$

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



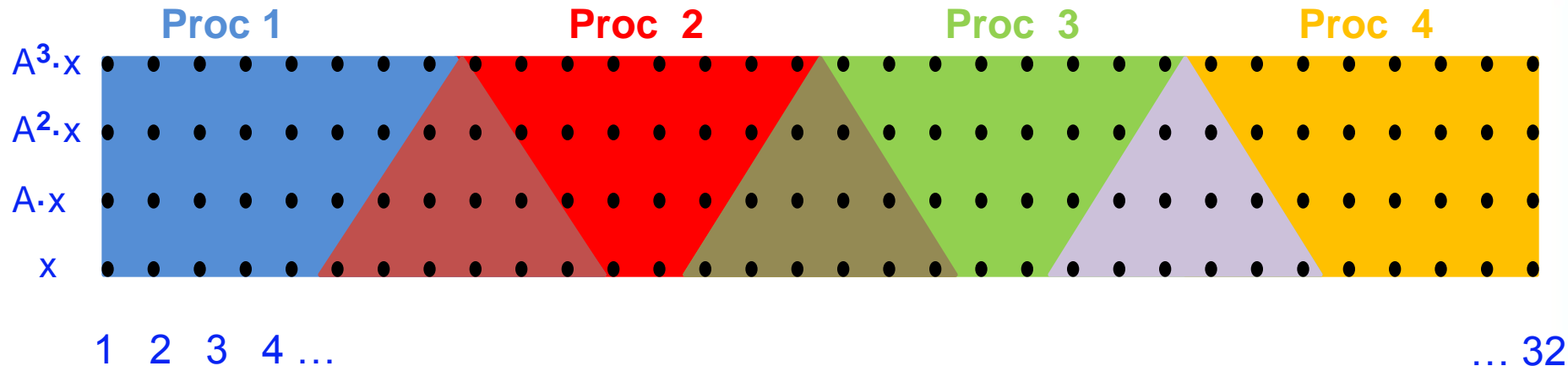
- Example: A tridiagonal,  $n=32$ ,  $k=3$
- Each processor communicates once with neighbors

# Communication Avoiding Kernels:



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

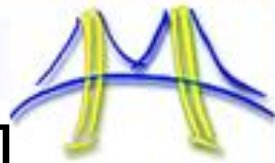
- Replace  $k$  iterations of  $y = A \cdot x$  with  $[Ax, A^2x, \dots, A^kx]$
- Parallel Algorithm



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

# Communication Avoiding Kernels:

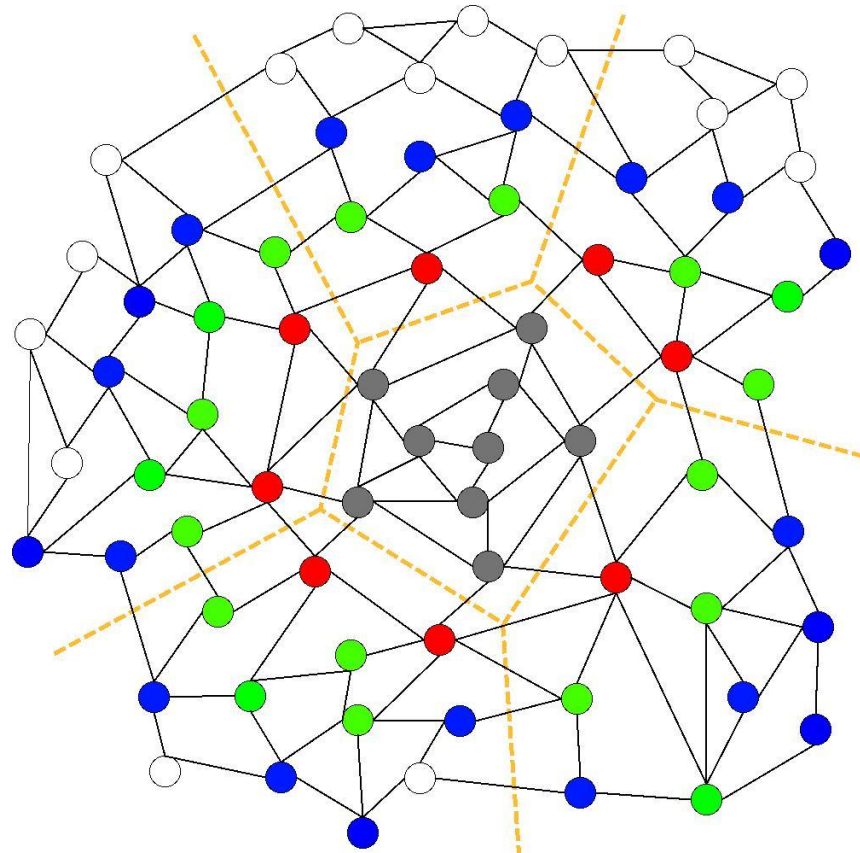
The Matrix Powers Kernel :  $[Ax, A^2x, \dots, A^kx]$



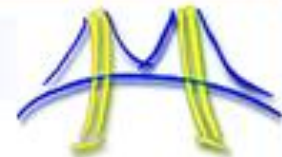
Same idea works for general sparse matrices

Partitioning by rows  $\rightarrow$   
Graph partitioning

Processing left to right  $\rightarrow$   
Traveling Salesman Problem







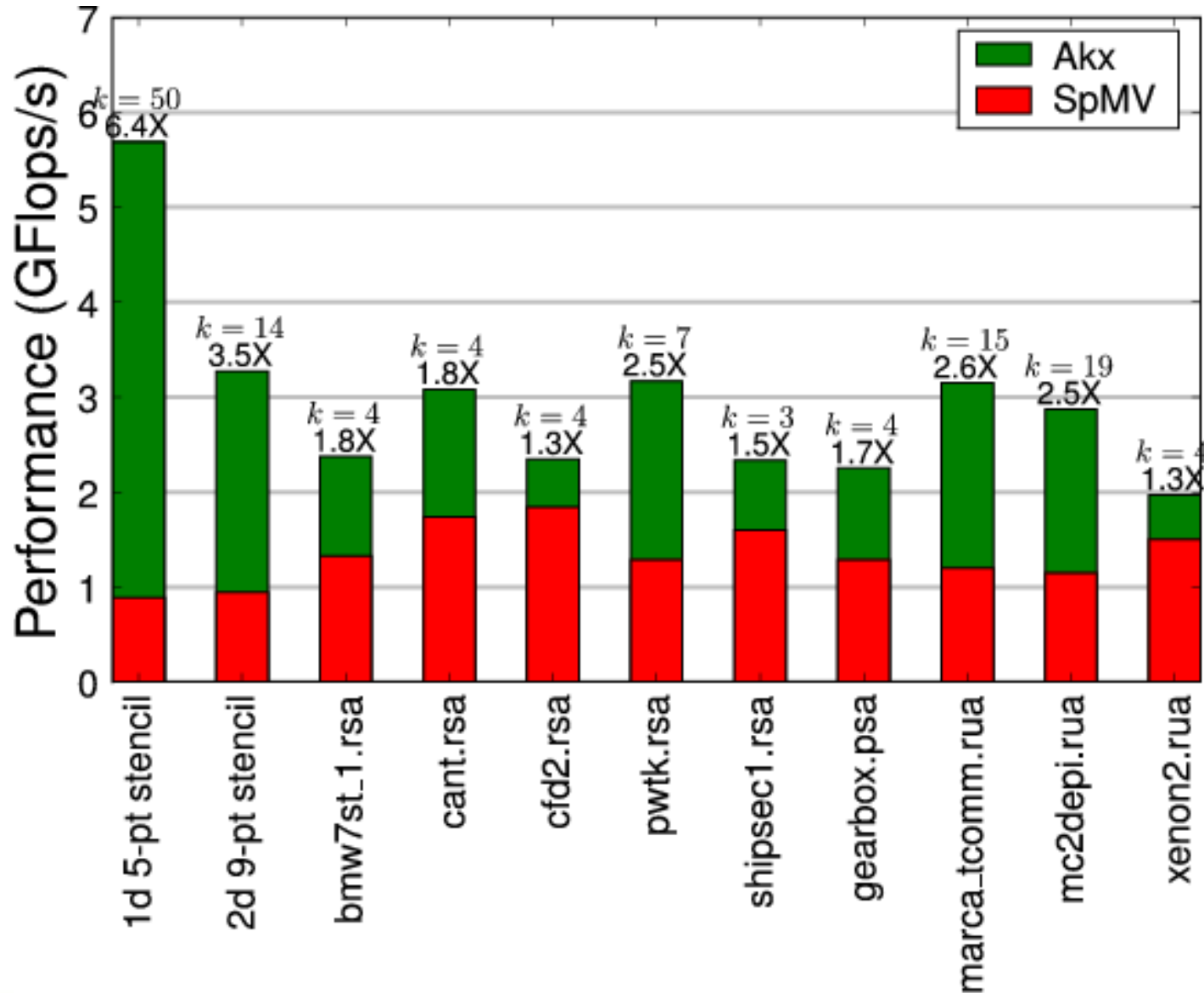
# What about multicore?

- 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

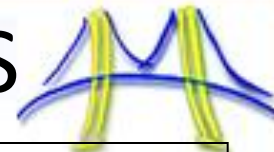
# Speedups on Intel Clovertown (8 core)



Test matrices include stencils and practical matrices  
See SC09 paper on [bebop.cs.berkeley.edu](http://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
```

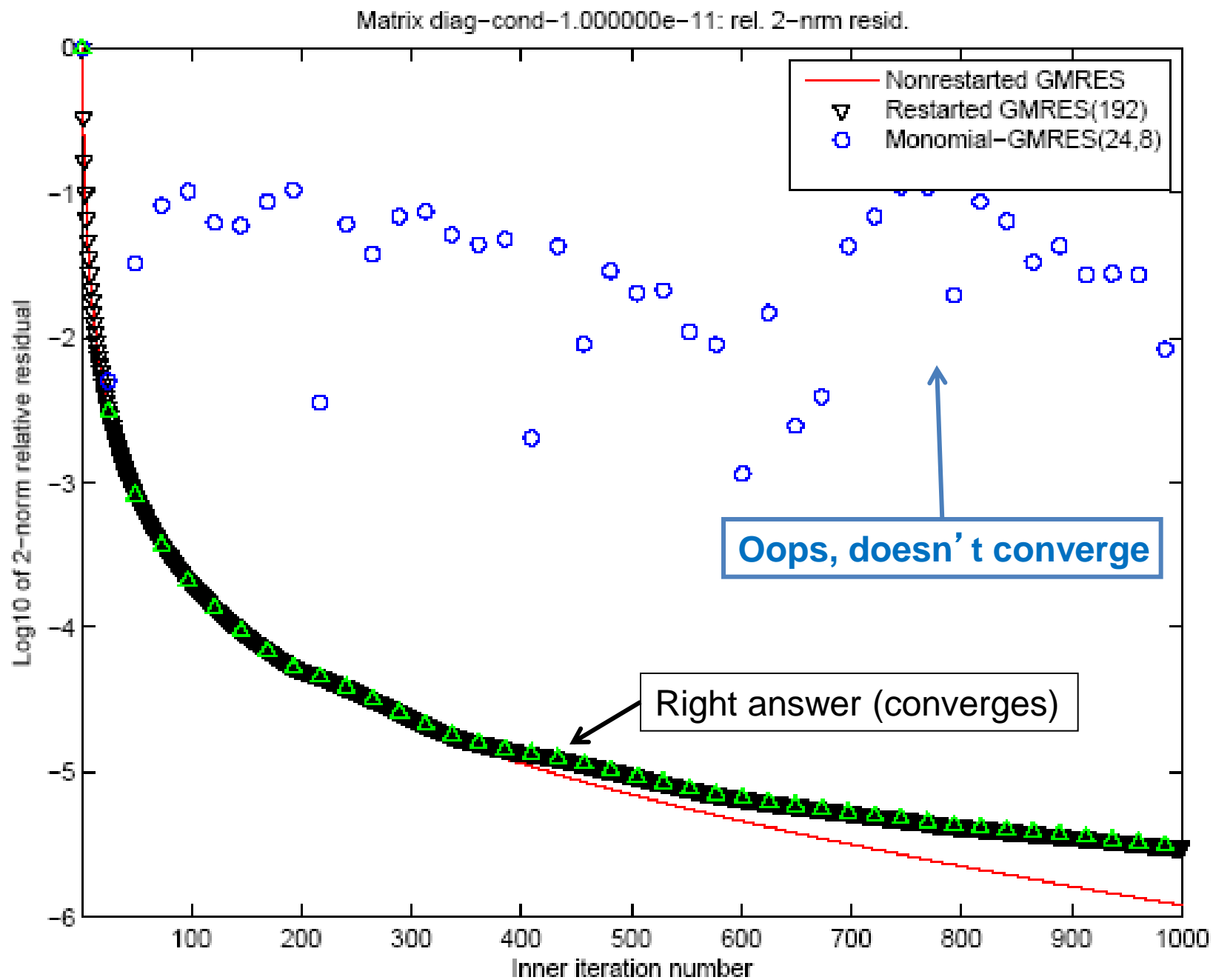
Communication cost =  
**k copies** of A, vectors from  
slow to fast memory

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
```

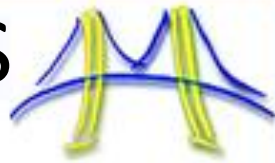
Communication cost =  
**O(1) copy** of A, vectors from  
slow to fast memory

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



# Minimizing Communication of GMRES

## (and getting the right answer)



Communication-Avoiding GMRES, ver. 2

$$W = [ v, p_1(A)v, p_2(A)v, \dots, p_k(A)v ]$$

... where  $p_i(A)v$  is a degree- $i$  polynomial in  $A$  multiplied by  $v$

... polynomials chosen to keep vectors independent

$$[Q, R] = \text{TSQR}(W)$$

... “Tall Skinny QR”

... new optimal QR discussed before

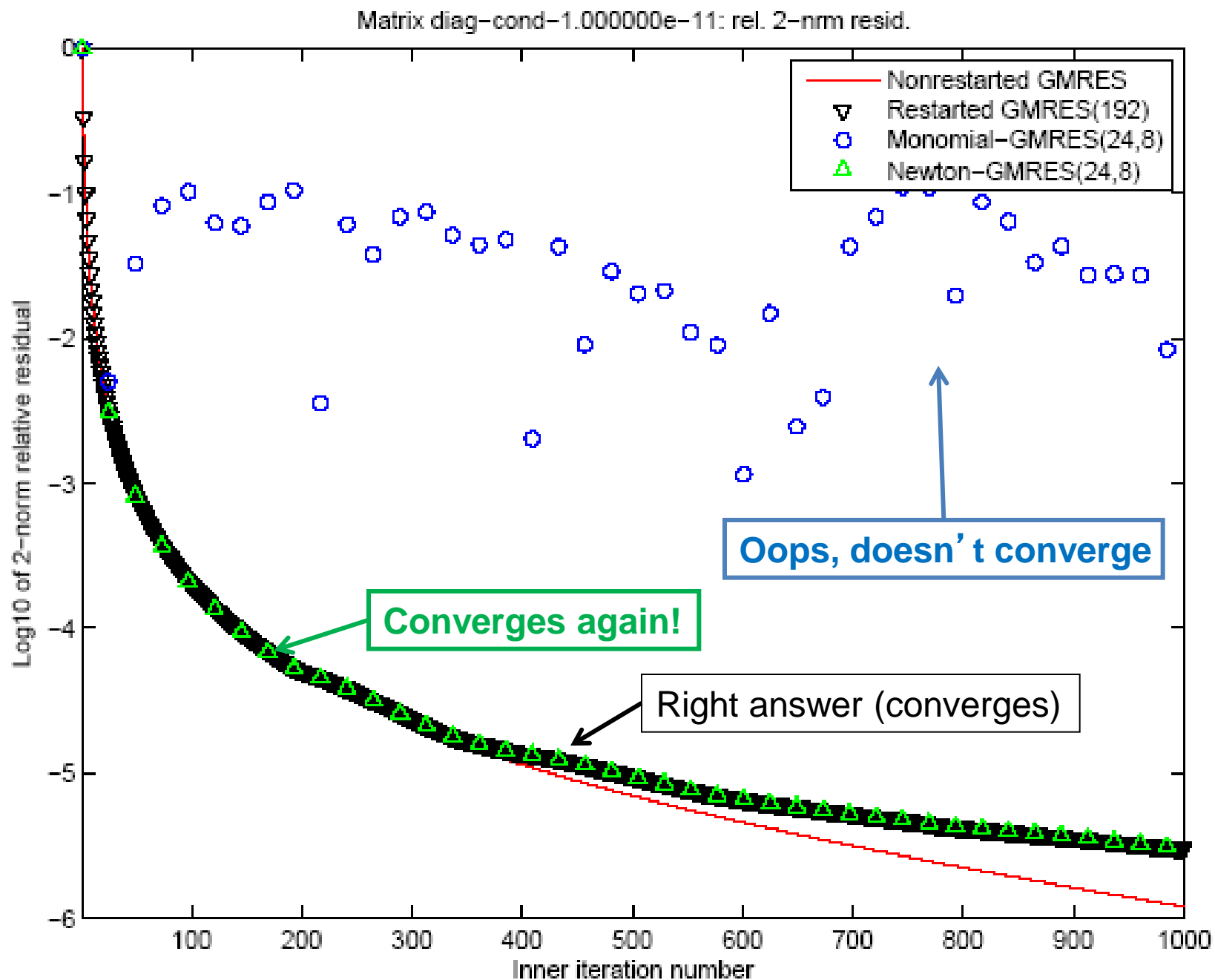
Build  $H$  from  $R$

... slightly different  $R$  from before

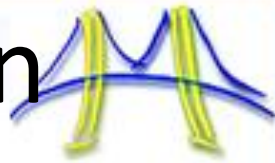
solve LSQ problem with  $H$  for  $x$

Communication cost still optimal:

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

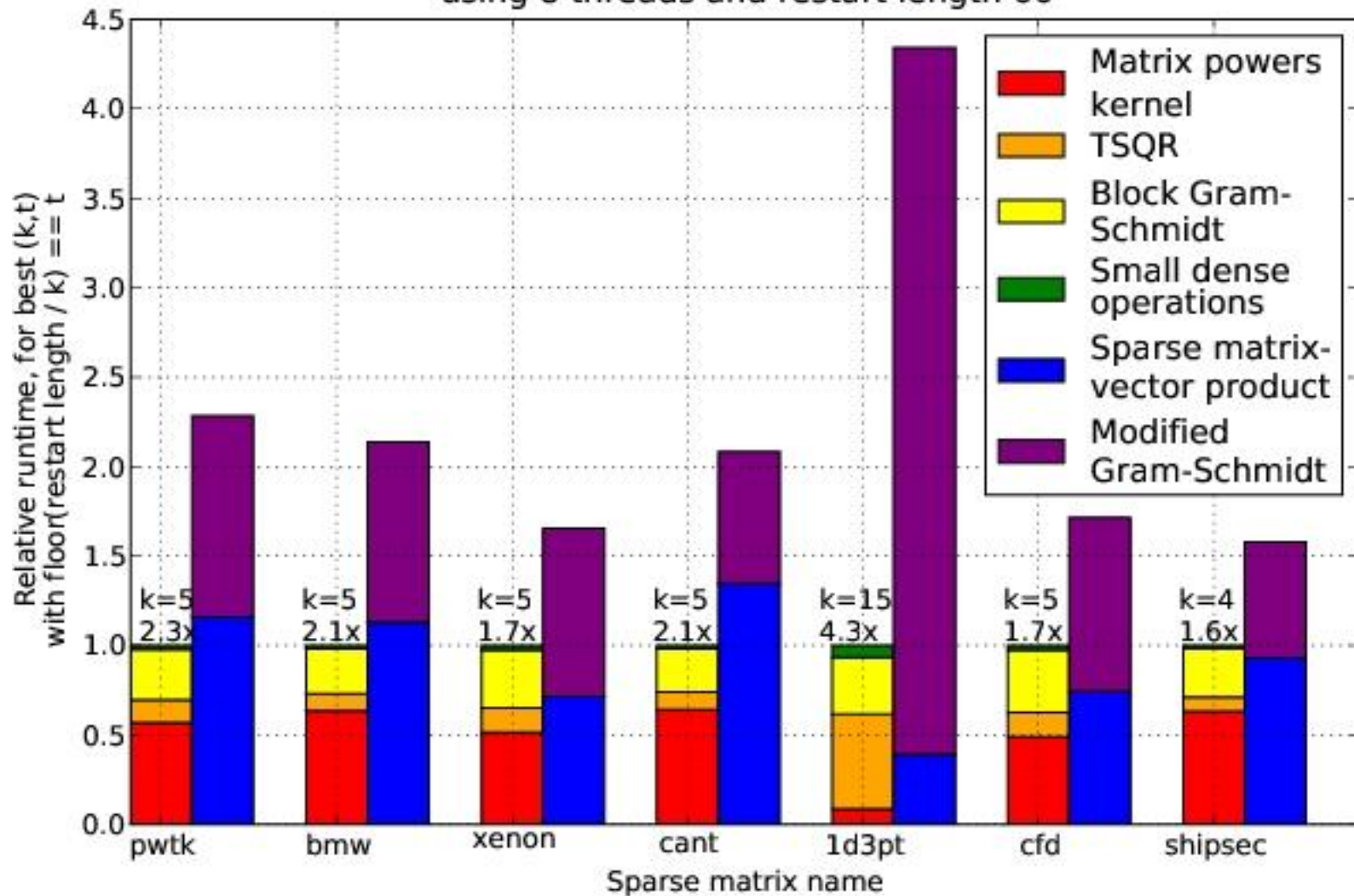


# Speed ups on 8-core Clovertown

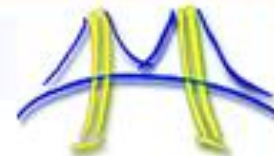


CA-GMRES = Communication-Avoiding GMRES

Runtime per kernel, relative to CA-GMRES(k,t), for all test matrices,  
using 8 threads and restart length 60



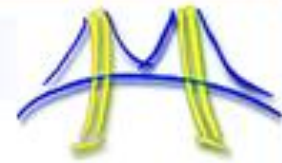
Paper by Mohiyuddin, Hoemmen, D. to appear in Supercomputing09



# Summary of what is known (1/2), open

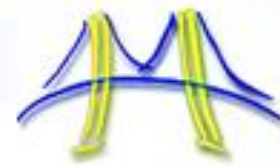
- GMRES
  - Can independently choose  $k$  to optimize speed, restart length  $r$  to optimize convergence
  - Need to “co-tune” Akx kernel and TSQR
  - Know how to use more stable polynomial bases
  - Proven speedups
- Can similarly reorganize other Krylov methods
  - Arnoldi and Lanczos, for  $Ax = \lambda x$  and for  $Ax = \lambda Mx$
  - Conjugate Gradients (CG), for  $Ax = b$
  - Biconjugate Gradients (BiCG), CG Squared (CGS), BiCGStab for  $Ax=b$
  - Other Krylov methods?





## Summary of what is known (2/2), open

- Preconditioning:  $MAx = Mb$ 
  - Need different communication-avoiding kernel:  $[x, Ax, MAx, AMAx, MAMAx, AMAMAx, \dots]$
  - For which preconditioners  $M$  can we minimize communication?
    - Easy: diagonal  $M$
    - A little harder: more general sparse  $M$
    - Works (in principle) for Hierarchically Semi-Separable  $M$
    - How does it work in practice?
- See Mark Hoemmen's PhD thesis for details
  - [bebop.cs.berkeley.edu](http://bebop.cs.berkeley.edu)



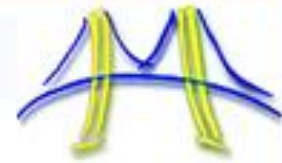
# What is a sparse matrix?

		<i>Structure</i>		
		<i>Static</i>		<i>Dynamic</i>
		<i>Implicit</i>	<i>Explicit</i>	<i>Implicit</i>
<i>Values</i>	<i>Static</i>	<b>LBM, Stencils</b> on structured grids	<b>Laplacian of a Graph</b>	-
	<i>Explicit</i>	<b>CBIR's SpMV</b> extremely large & complex stencil	<b>Standard SpMV</b> e.g. CSR	-
	<i>Dynamic</i>	-	-	<b>PIC Histograms</b> 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](http://bebop.cs.berkeley.edu)

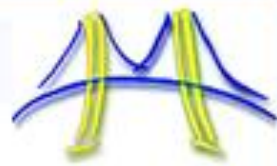


# STRUCTURED GRID MOTIF

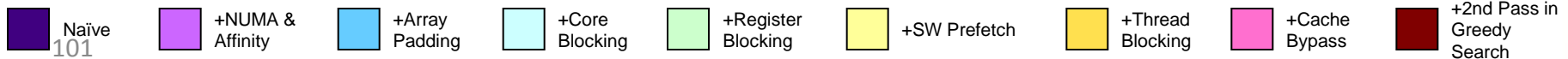
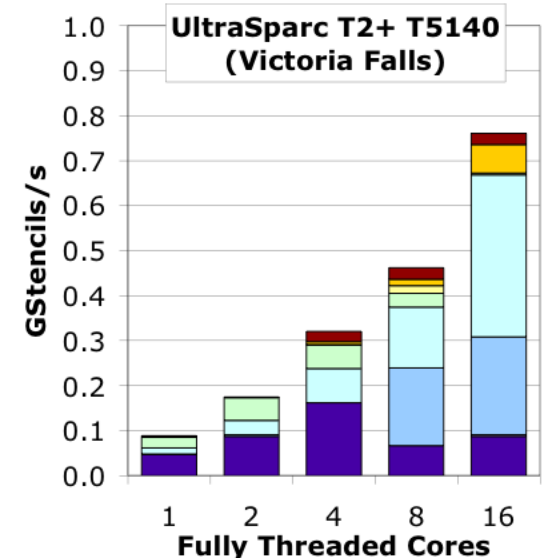
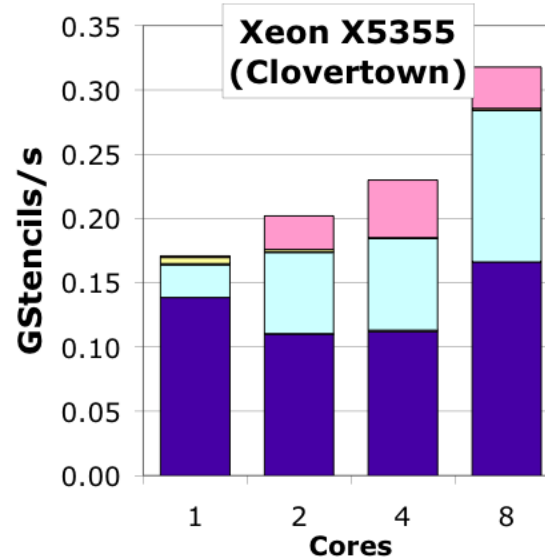
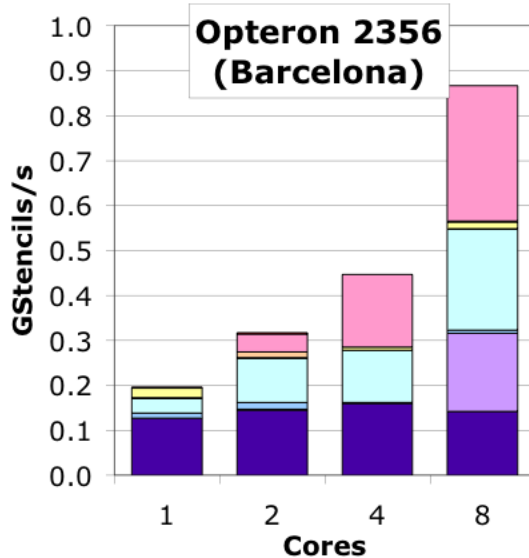
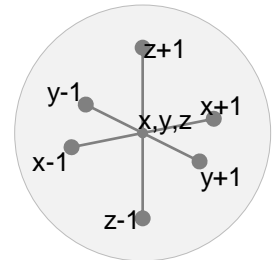
Source: Sam Williams

# Structured Grids

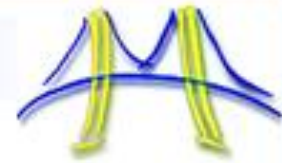
## Finite Difference Operators



- 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

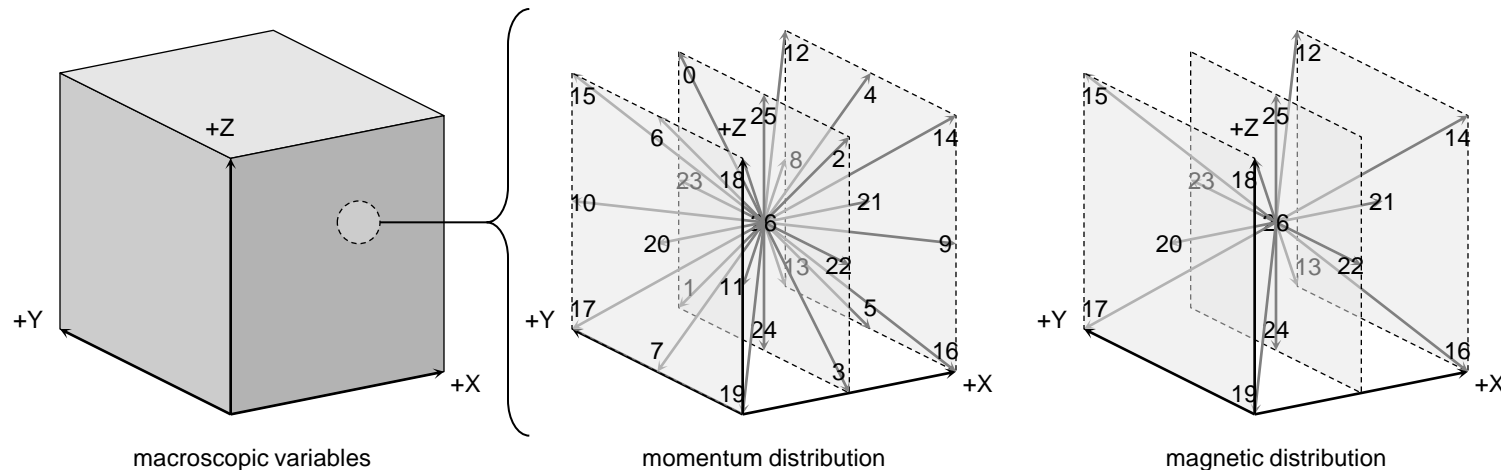


# Structured Grids

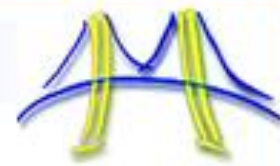


## Lattice Boltzmann Methods

- 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  $\sim 1300$  flops and  $\sim 1200$  bytes of memory traffic

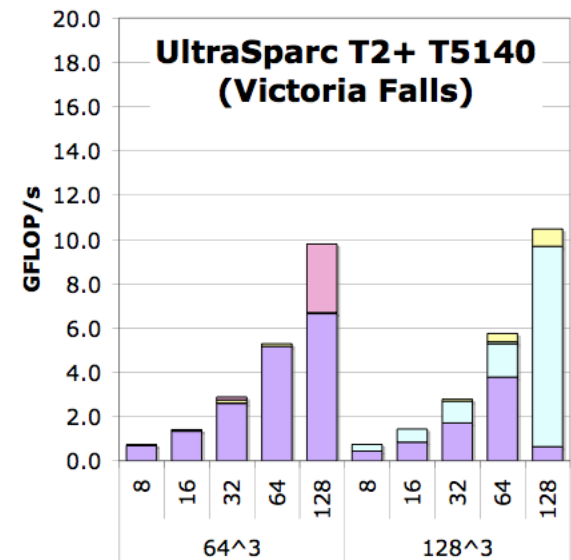
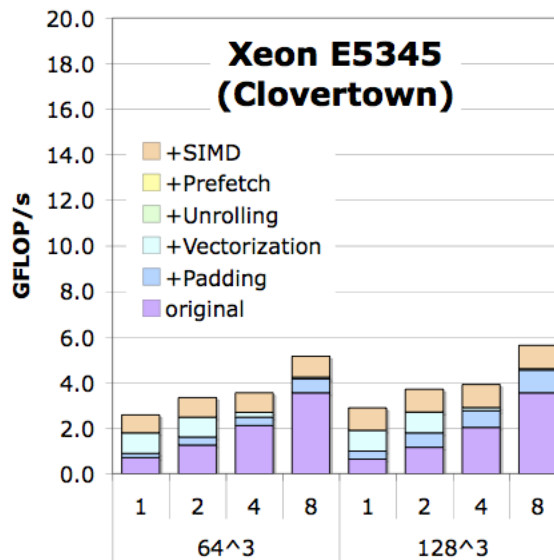
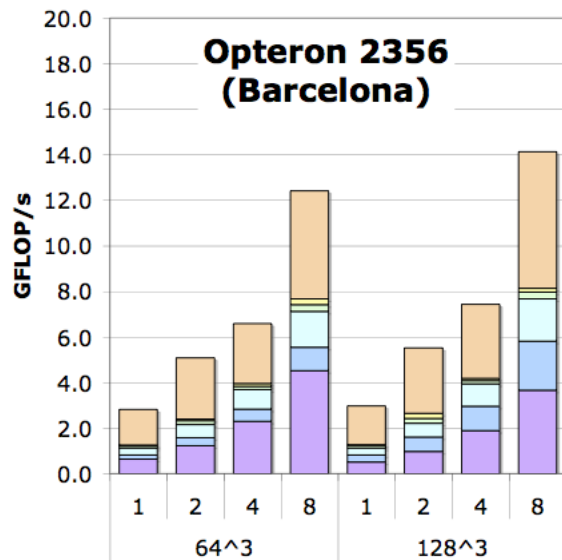


# Structured Grids



## Lattice Boltzmann Methods

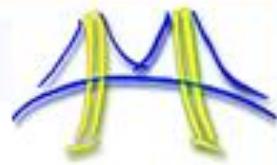
- ❖ Challenged by:
  - The higher flop:byte ratio of  $\sim 1.0$  is still bandwidth-limiting
  - TLB locality (touch 150 pages per lattice update)
  - cache associativity (150 disjoint lines)
  - efficient SIMDization
- ❖ easy to (correctly) parallelize
- ❖ **explicit SIMDization & SW prefetch are dependent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



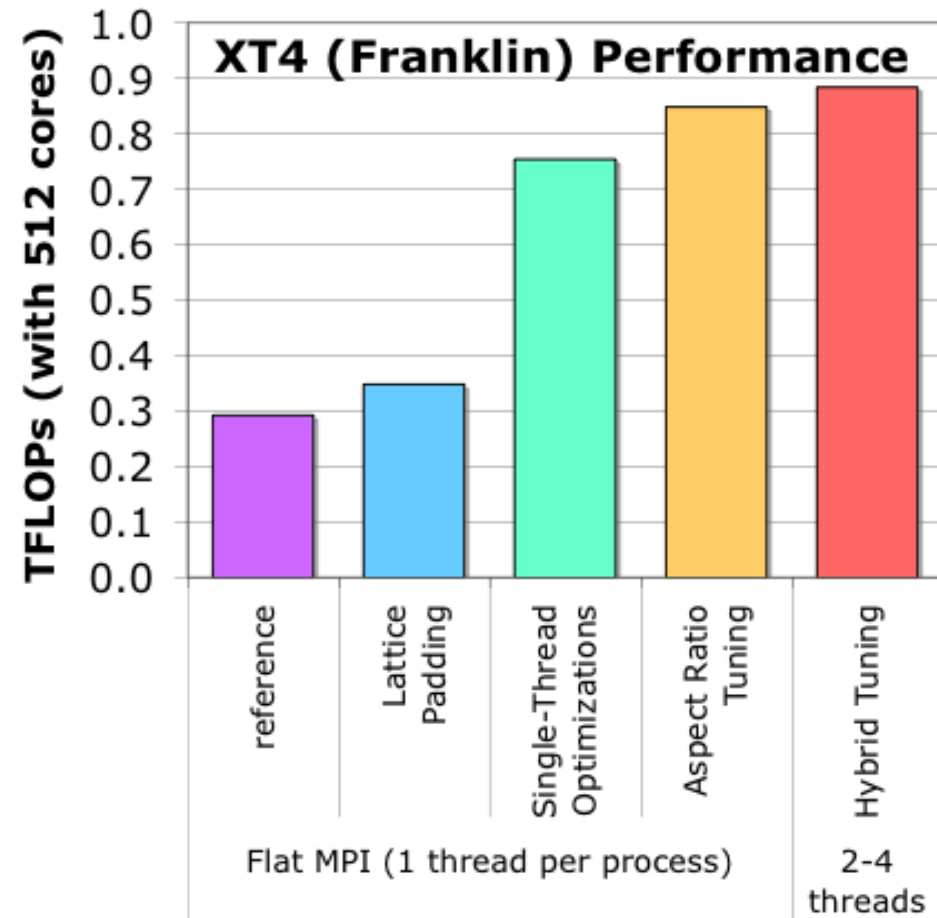
Reference + NUMA    +Padding    +Vectorization    +Unrolling    +SW Prefetch    +Explicit SIMDization    +small pages

# Structured Grids

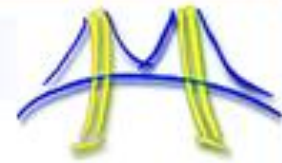
## Lattice Boltzmann Methods



- **Distributed Memory & Hybrid**
- MPI, MPI+threads, MPI+OpenMP (SPMD, SPMD<sup>2</sup>, 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)



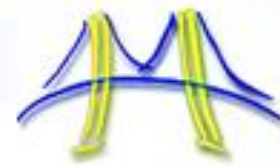




# PARTICLE METHOD MOTIF

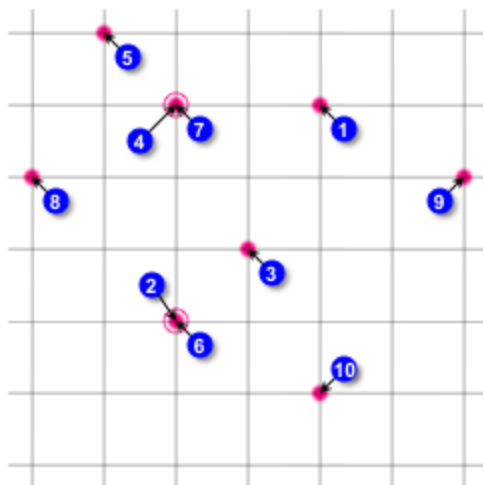
Source: Sam Williams

# Particle Methods

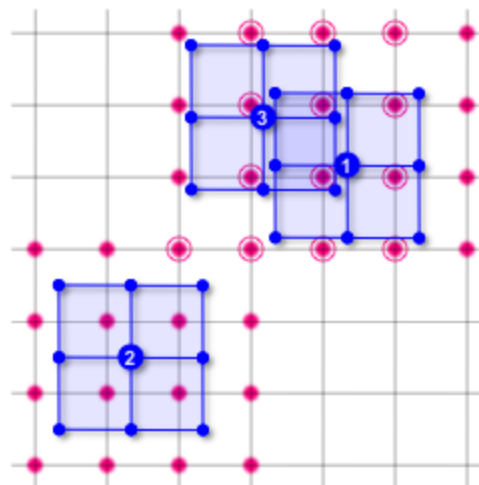


## Particle-In-Cell

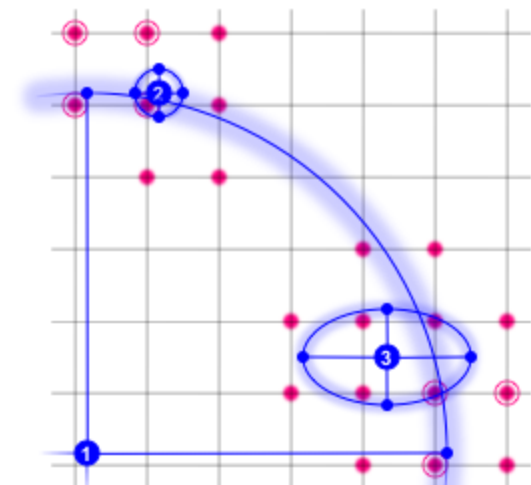
- Rather than calculating  $O(N^2)$  forces, calculate impact of particles on field and field on particles  $\rightarrow O(N)$ 
  - particle-to-grid interpolation (scatter-add)  $\lll$  most challenging step
  - Poisson solve
  - grid-to-particle/push interpolation (gather)  $\lll$  EP
- Used in a number of simulations including Heart and Fusion
- Trivial simplification would be a 2D histogram
- **These codes can be challenging to parallelize in shared memory**



(a)

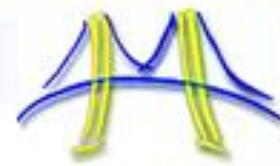


(b)



(c)

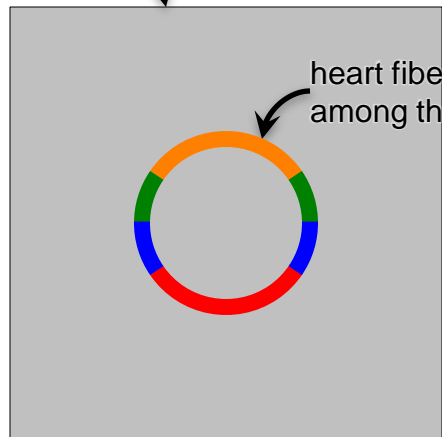
# Particle Methods



## Particle-In-Cell

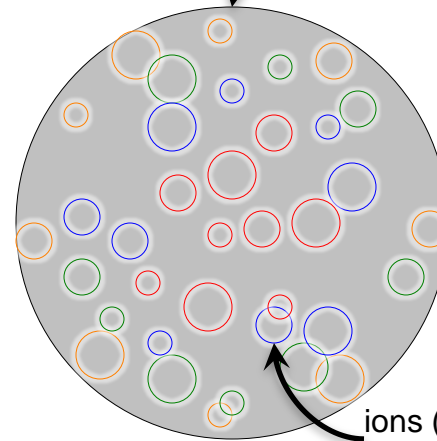
- ❖ PIC codes can be particularly challenging to parallelize
- ❖ PIC codes can have different grid topologies, different particle characteristics, and different particle distributions
- ❖ To mitigate these differences, we tune over 5 synchronization approaches (3 locking, FP atomics, none) and 5 partial grid replication strategies (none, partitioned, overlapping partitions, dynamic, full)

one fluid grid shared  
among all threads



heart fibers partitioned  
among threads

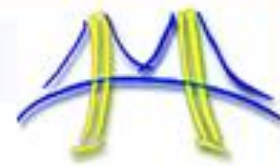
one charge grid shared  
among all threads



ions (rings) partitioned  
among threads



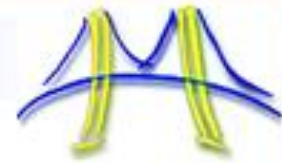
# Particle Methods



## Particle-In-Cell

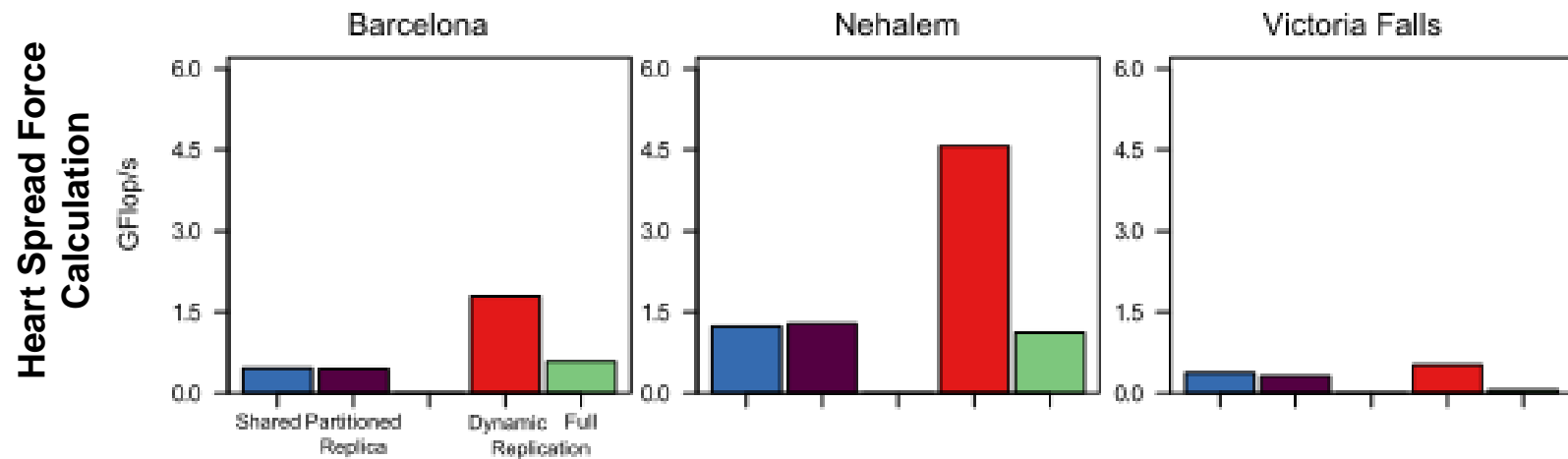
- ❖ Results: vastly less memory than MPI on a node with some performance gains
- ❖ Clearly, optimization is dependent on the distribution of particles, available memory, and architecture

# Particle Methods

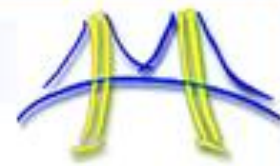


## Particle-In-Cell

- ❖ Results: vastly less memory than MPI on a node with some performance gains
- ❖ Clearly, optimization is dependent on the distribution of particles, available memory, and architecture

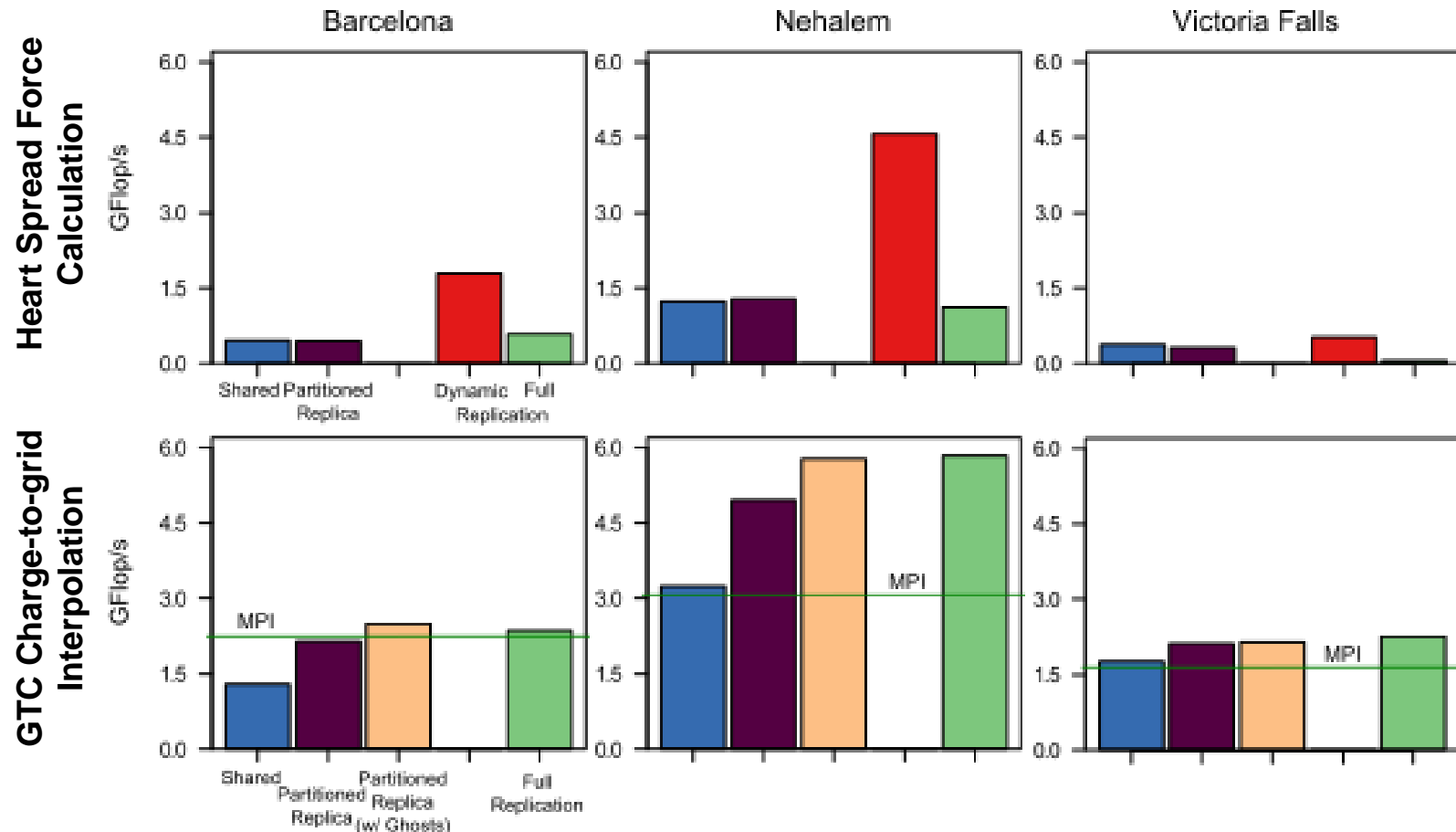


# Particle Methods



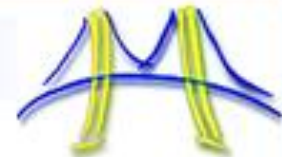
## Particle-In-Cell

- ❖ Results: vastly less memory than MPI on a node with some performance gains
- ❖ Clearly, optimization is dependent on the distribution of particles, available memory, and architecture



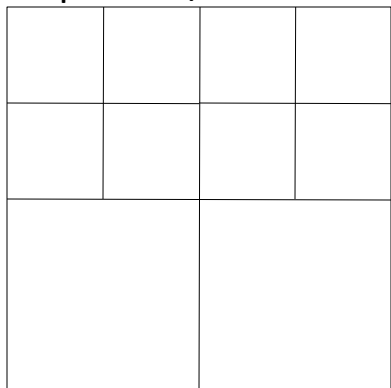


# Particle Methods

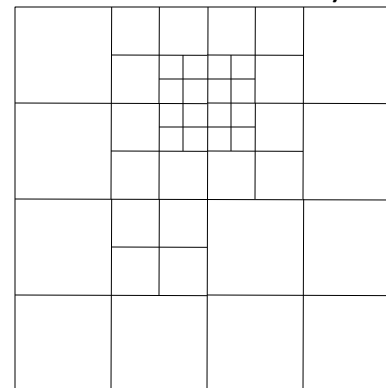


## Fast Multipole Method

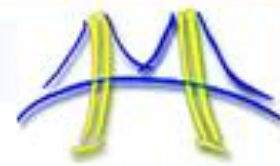
- Alternate (tree-based) approach for calculating forces
- Kernel Independent FMM (KIFMM) is challenged by 7 computational phases (kernels) including list computations and tree traversals.
- List computations vary from those requiring direct particle-particle interactions, to those based on many small FFTs
- Different architectures (CPUs, GPUs...) may require different codes for each phase.
- Additionally, FMM is parameterized by the number of particles per box in the oct-tree.
  - More particles/box  $\Rightarrow$  more flops (direct calculations)
  - Fewer particle/box  $\Rightarrow$  fewer flops (but more work in tree traversals)



Fewer total flops,  
more tree traversals

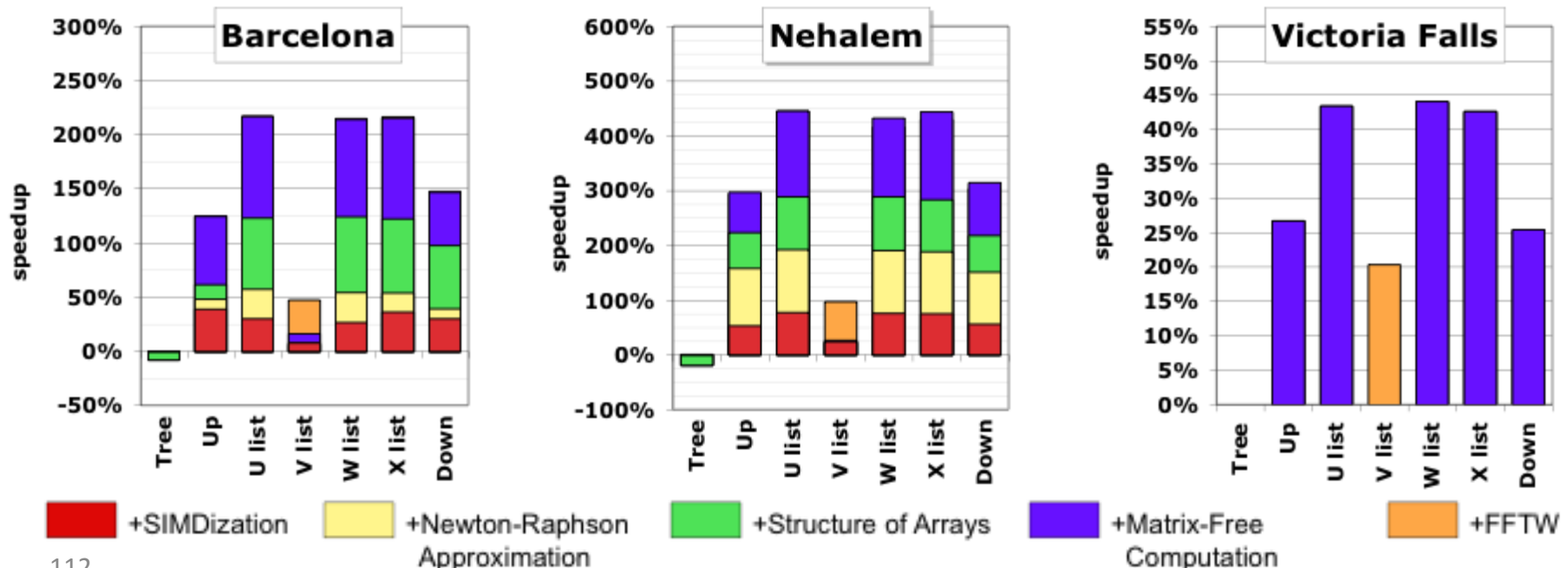


# Particle Methods



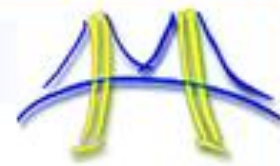
## Fast Multipole Method

- Different architectures showed speedups for different phases from conventional auto-tuning.
- Additionally, tuning algorithmic parameters showed different architectures preferred different sweet spots...
  - Nehalem's sweet spot was around 250 particles/box
  - GPUs required up to **4000 particles/box** to attain similar performance. That is, to cope with poor tree traversal performance GPU's had to perform **16x as many flop's**





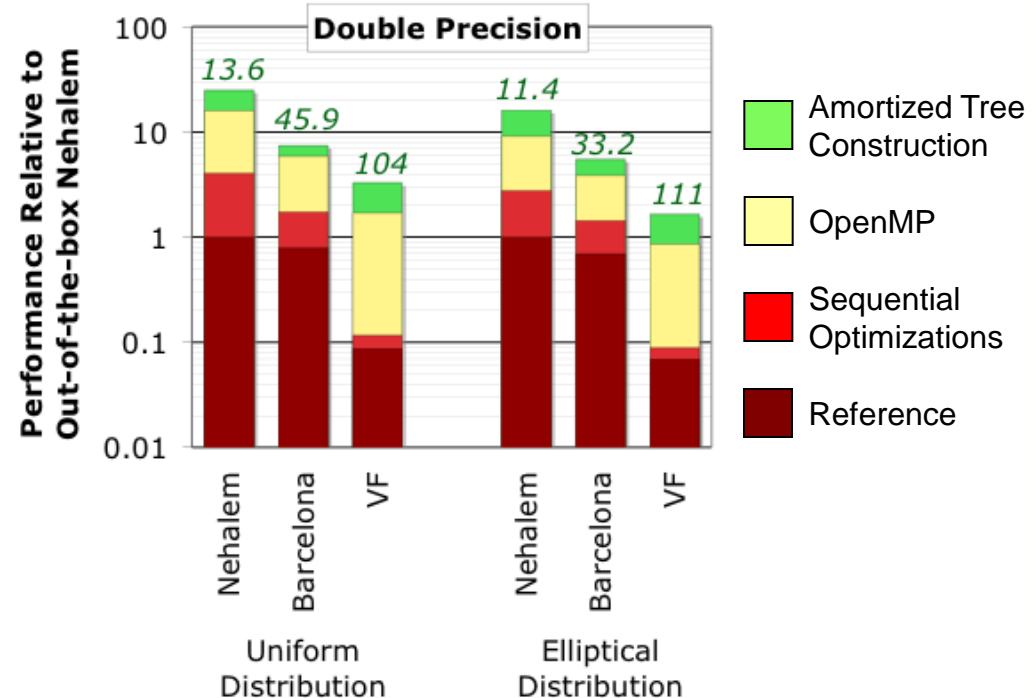
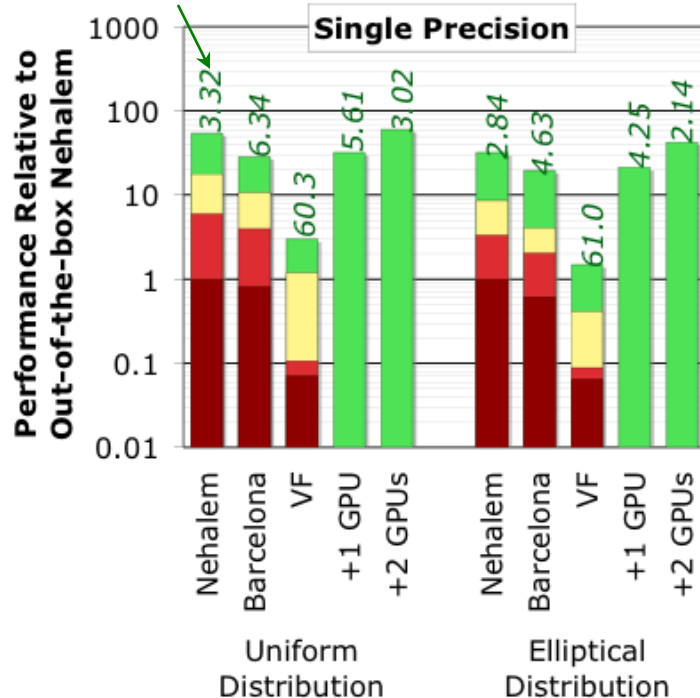
# Particle Methods



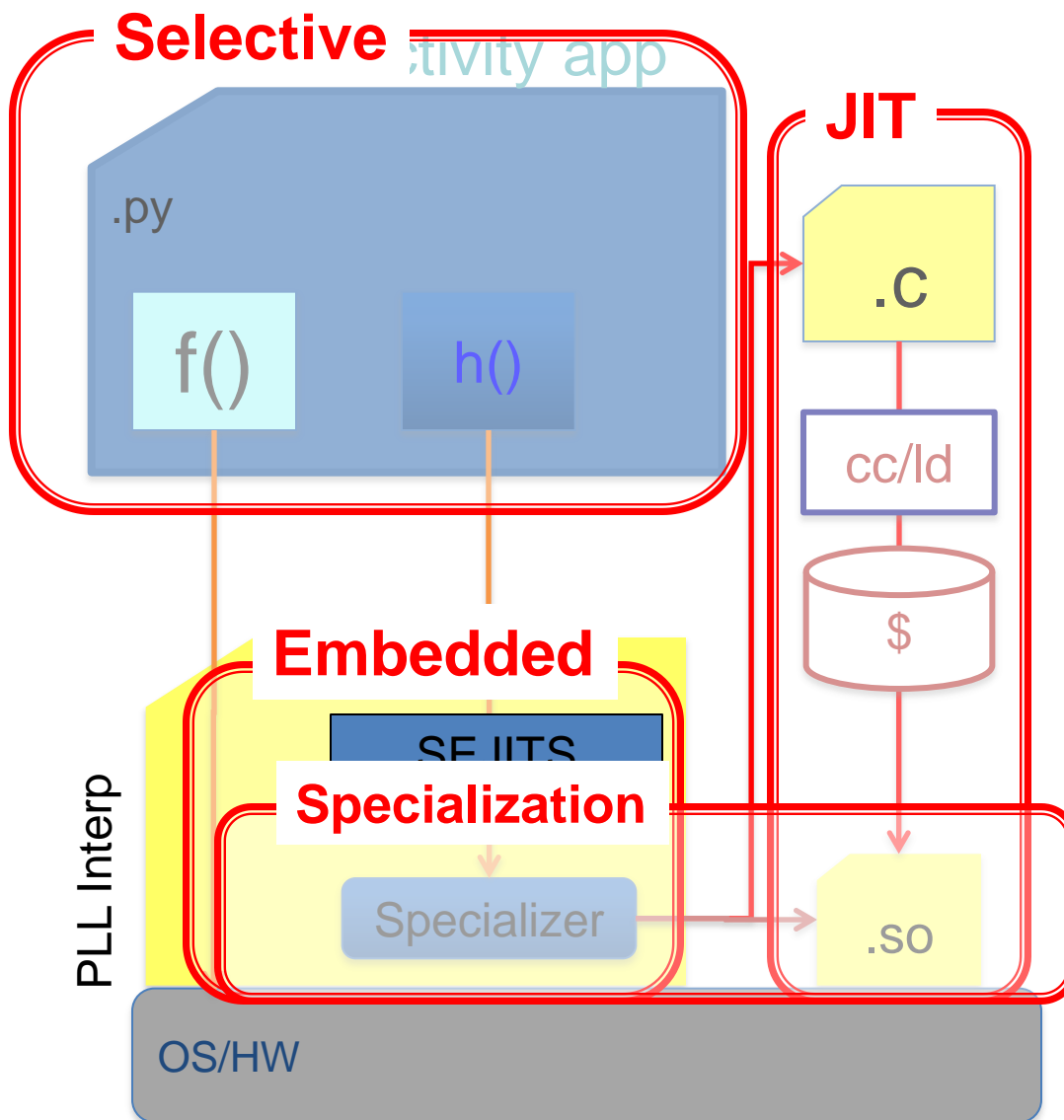
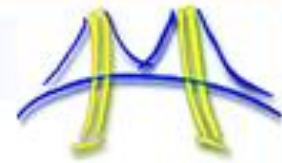
## Fast Multipole Method

- Different architectures showed speedups for different phases from conventional auto-tuning.
- Additionally, tuning algorithmic parameters showed different architectures preferred different sweet spots...
  - Nehalem's sweet spot was around 250 particles/box
  - GPUs required up to **4000 particles/box** to attain similar performance. That is, to cope with poor tree traversal performance GPU's had to perform **16x as many flop's**

runtimes in seconds

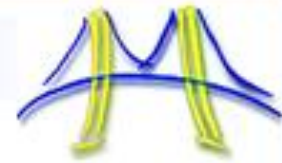


# Delivering Autotuning via SEJITS

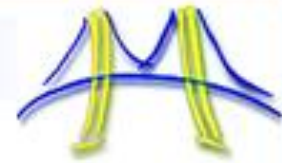




# Summary



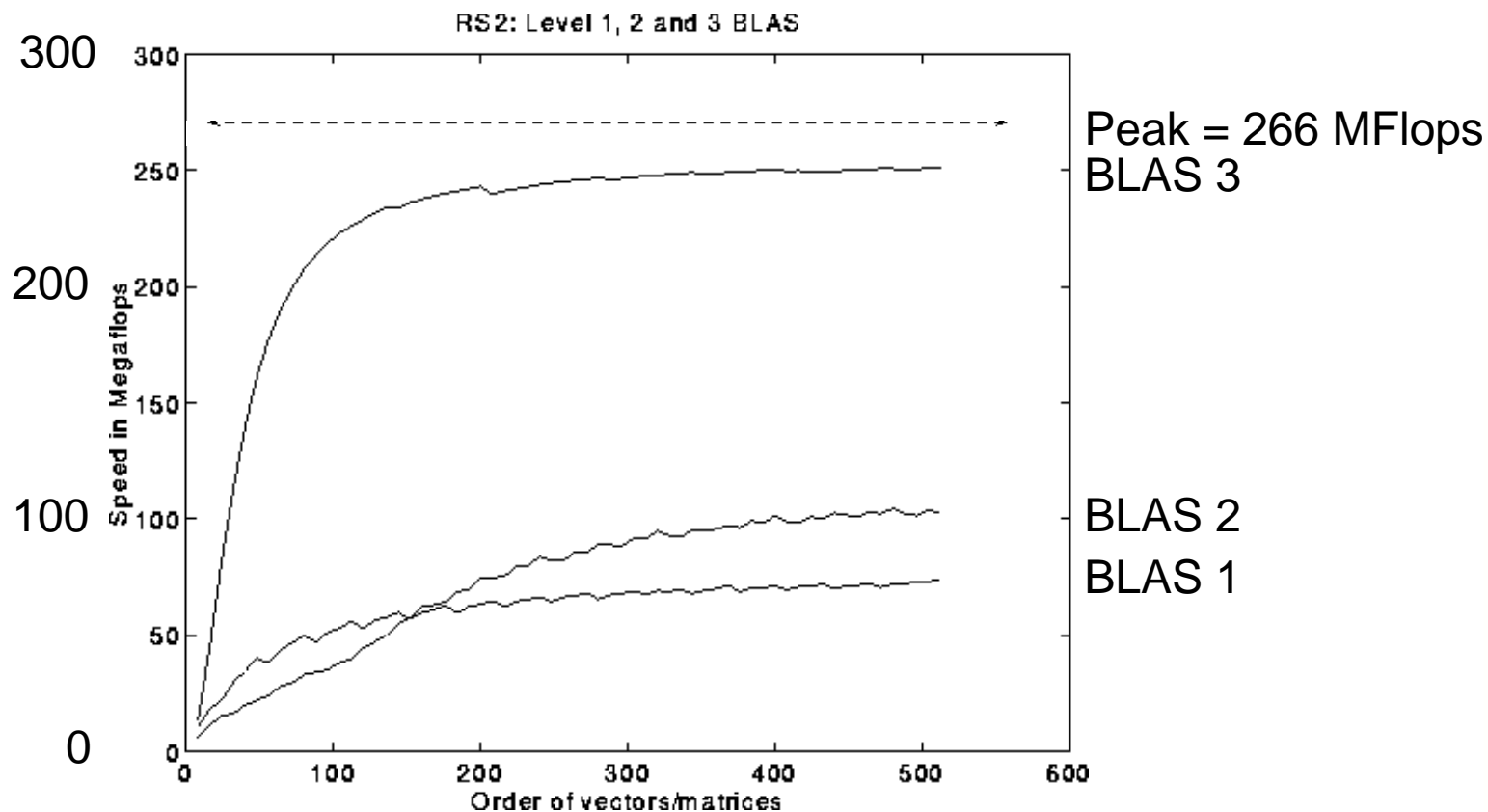
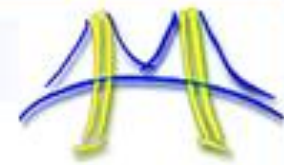
- “Design spaces” for algorithms and implementations are large and growing
- Finding the best algorithm/implementation by hand is hard and getting harder
- Ideally, we would have a database of “techniques” that would grow over time, and be searched automatically whenever a new input and/or machine comes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



Extra slides

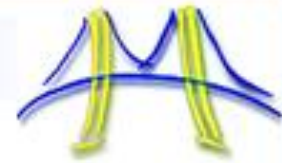


# BLAS 1/2/3 speeds on IBM RS6000/590

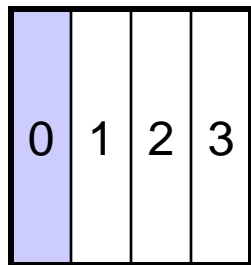


BLAS 3 (n-by-n matrix matrix multiply) vs  
BLAS 2 (n-by-n matrix vector multiply) vs  
BLAS 1 (saxpy of n vectors)

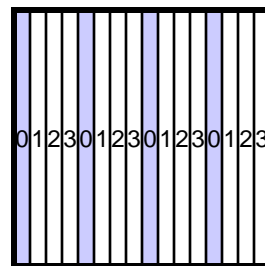
# Different Parallel Data Layouts for Matrices



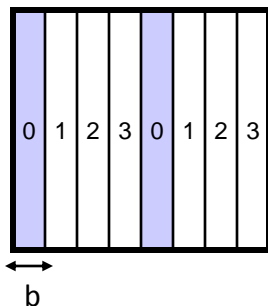
Bad load balance on many submatrices



1) 1D Column Blocked Layout

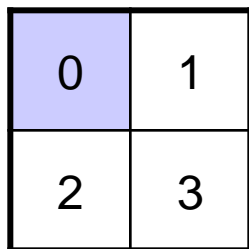


2) 1D Column Cyclic Layout

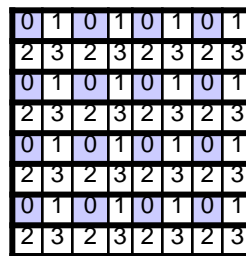


3) 1D Column Block Cyclic Layout

4) Row versions of the previous layouts



5) 2D Row and Column Blocked Layout

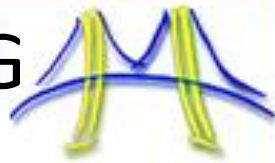


Generalizes others

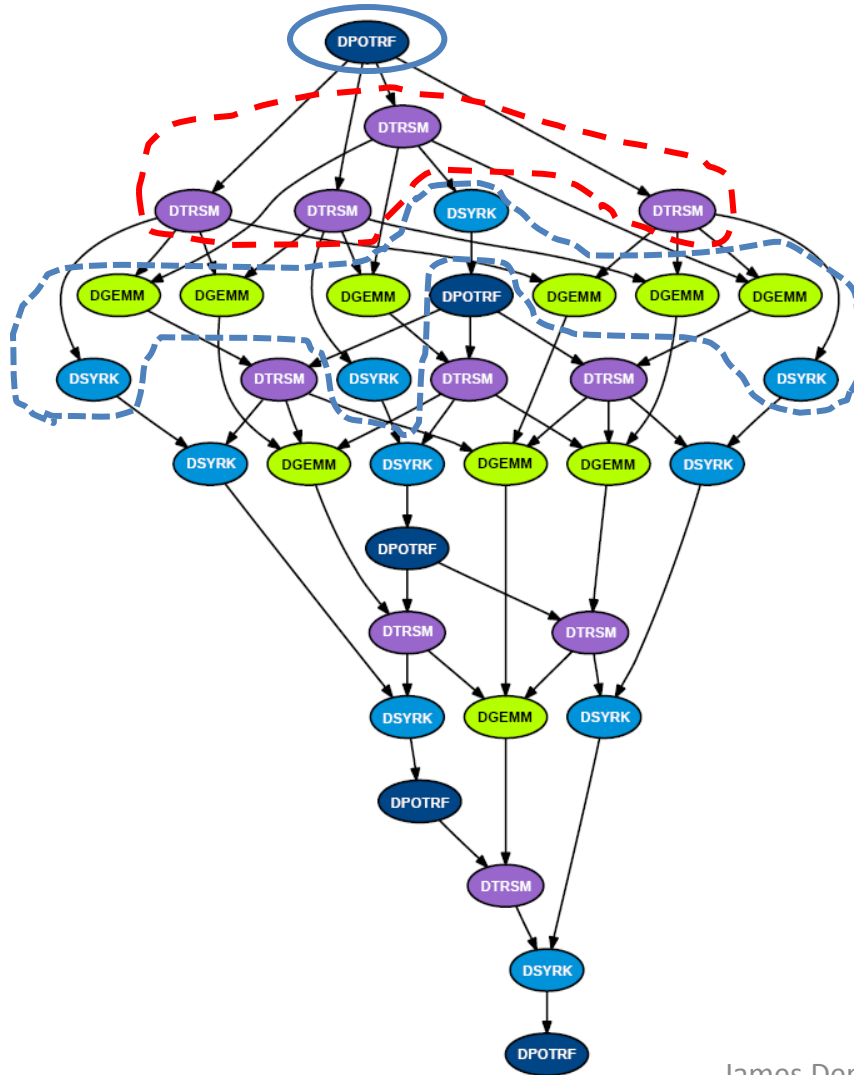
**Best load balance on submatrices**

6) 2D Row and Column Block Cyclic Layout

# 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

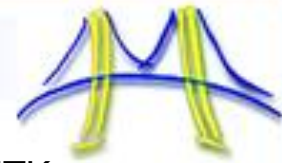


- Can process DAG in any order respecting dependencies
- What is the best schedule?
  - Static vs dynamic?
  - Programmer builds DAG vs compiler or run-time system?
  - Build and schedule whole DAG (size =  $O((n/b)^3)$ ) or just “front”
  - Use locality hints?

DAG courtesy of Jakub Kurzak, UTK

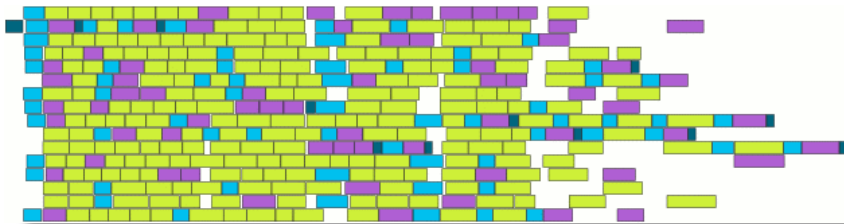
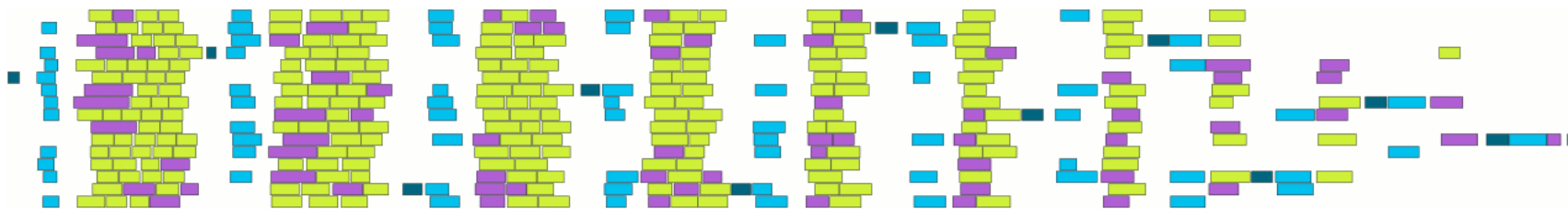


# Measured Results for Tiled Cholesky

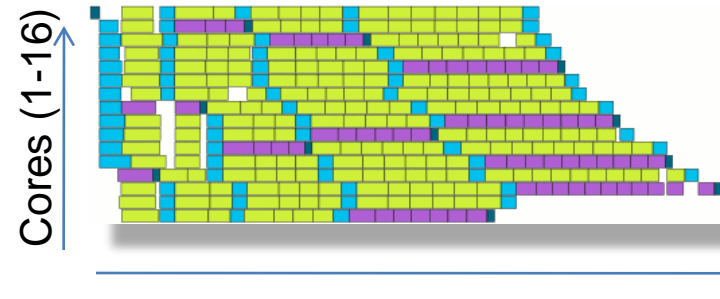


- Measured on Intel Tigerton 2.4 GHz
- Slide courtesy of Jakub Kurzak, UTK

Cilk ([www.cilk.com](http://www.cilk.com)): programmer defined spawn and sync



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

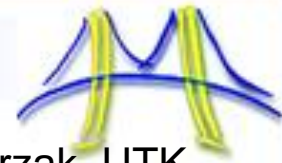


PLASMA: static schedule  
supplied by programmer

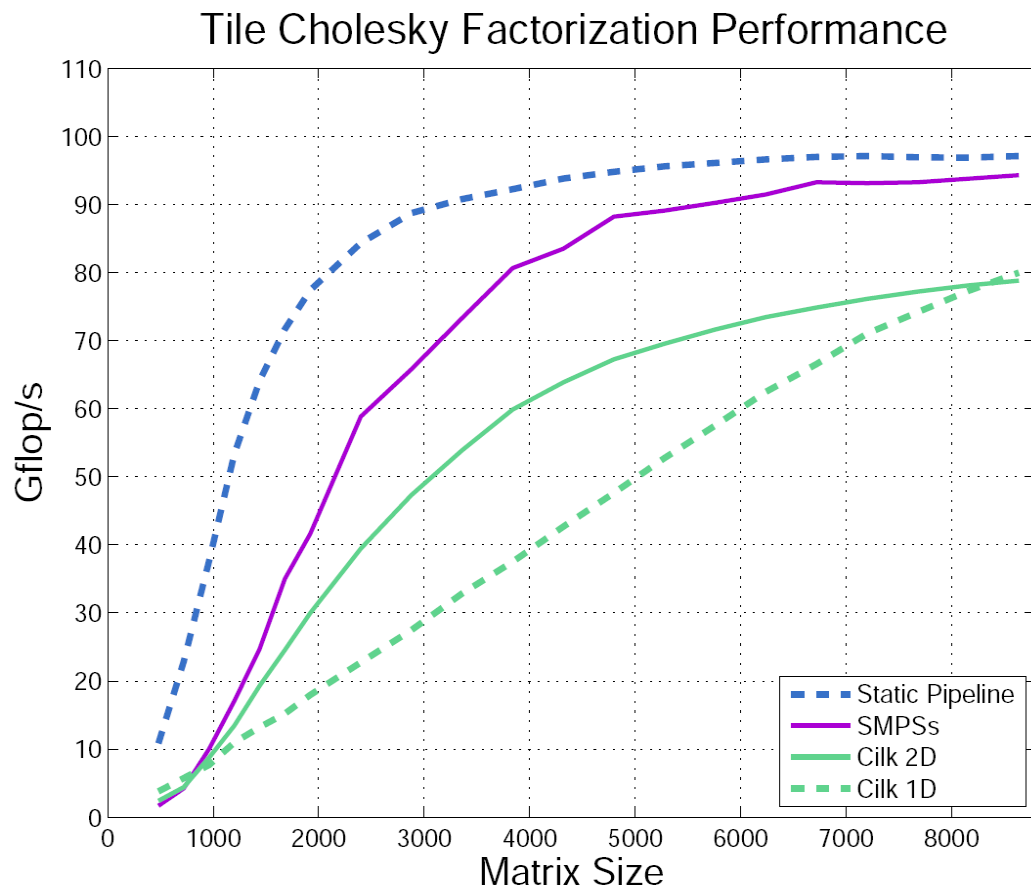




# More Measured Results for Tiled Cholesky



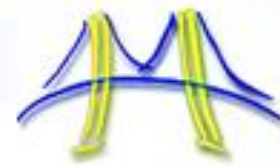
Slide courtesy of Jakub Kurzak, UTK



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

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

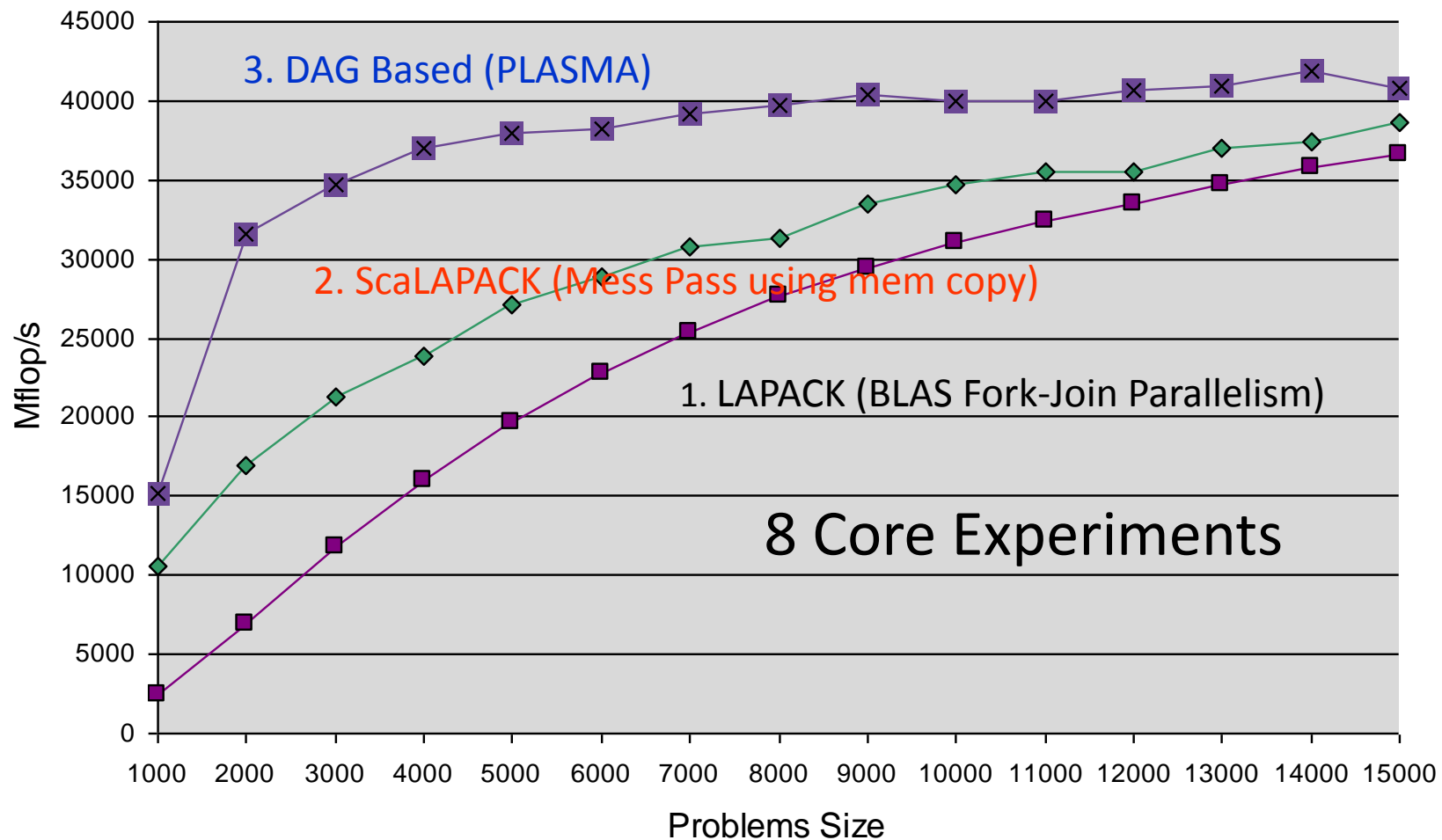
# Intel's Clovertown Quad Core

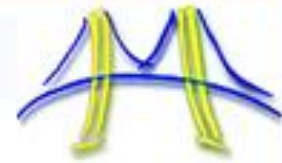


3 Implementations of LU factorization

Source: Jack Dongarra

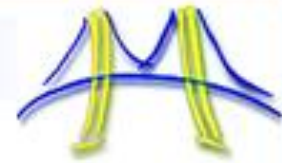
Quad core w/2 sockets per board, w/ 8 Threads





- PLASMA 2.0.0 released
  - Just Cholesky, QR, LU, using static scheduling
  - LU does not do partial pivoting – Stability?
  - [icl.cs.utk.edu/plasma/](http://icl.cs.utk.edu/plasma/)
- Future of PLASMA
  - Add dynamic scheduling, similar to SMPs
    - 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

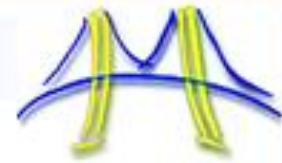
# Dense Linear Algebra on GPUs



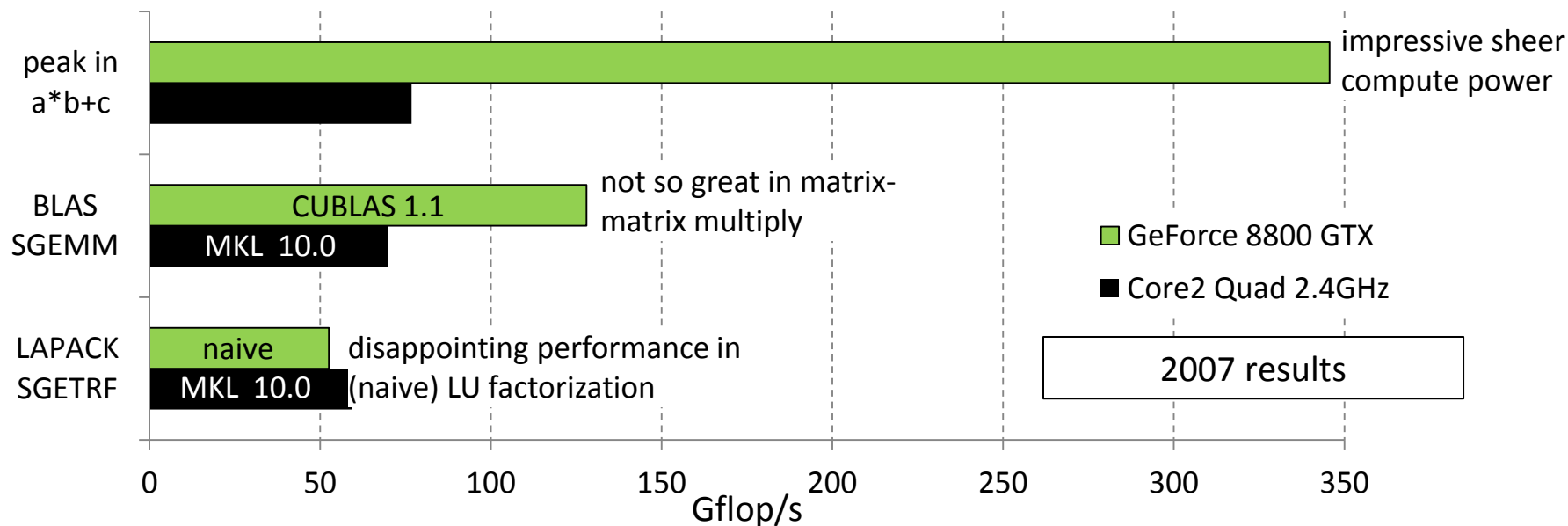
- Source: Vasily Volkov's SC08 paper
- New challenges
  - More complicated memory hierarchy
  - Not like “L1 inside L2 inside ...”,
    - Need to choose which memory to use carefully
    - Need to move data explicitly
  - GPU does some operations much faster than CPU, but not all
  - CPU and GPU like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



# Motivation



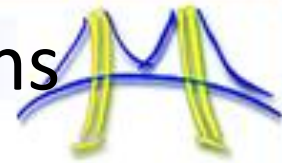
- NVIDIA released CUBLAS 1.0 in 2007: BLAS for GPUs
- Allows easy port of LAPACK to GPU (consider single precision only)



- 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

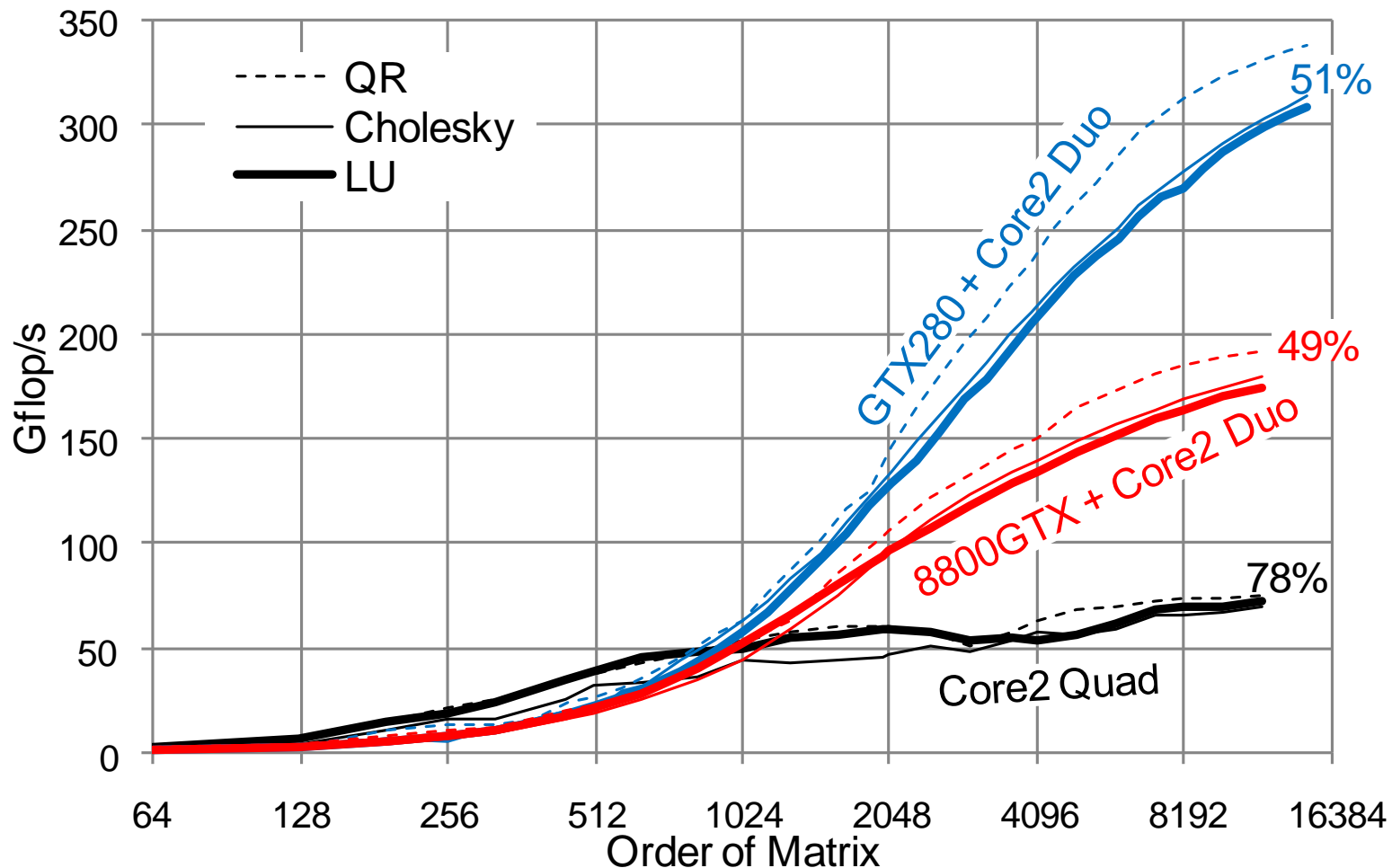
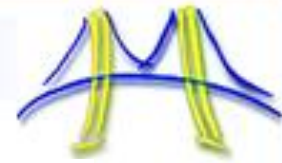


- Minimize communication with CPU memory
- Keep as much data in registers as possible
  - Largest, fastest on-GPU memory
  - Vector-only operations
- Use as little shared memory as possible
  - Smaller, slower than registers; use for communication, sharing only
  - Speed limit: 66% of peak with one shared 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

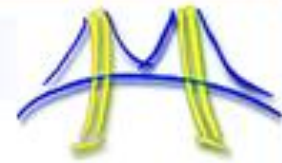
# Performance Results



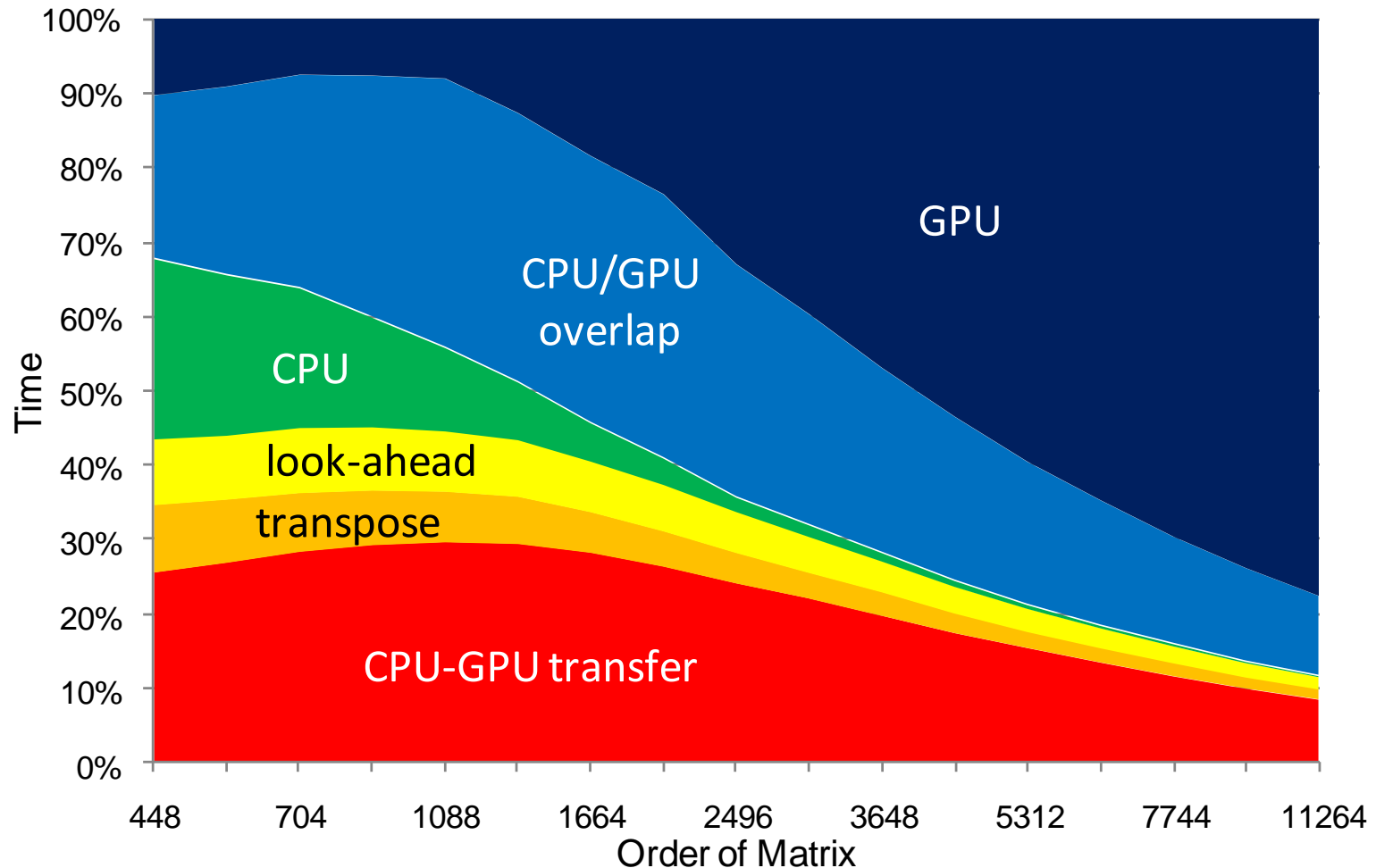
Our solution runs at ~50% of the system's peak (shown on the right)  
It is bound by SGEMM that runs at 60% of the GPU-only peak



# Where does the time go?

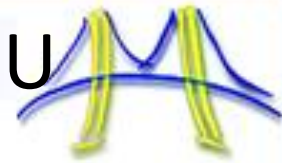


- Time breakdown for LU on 8800 GTX

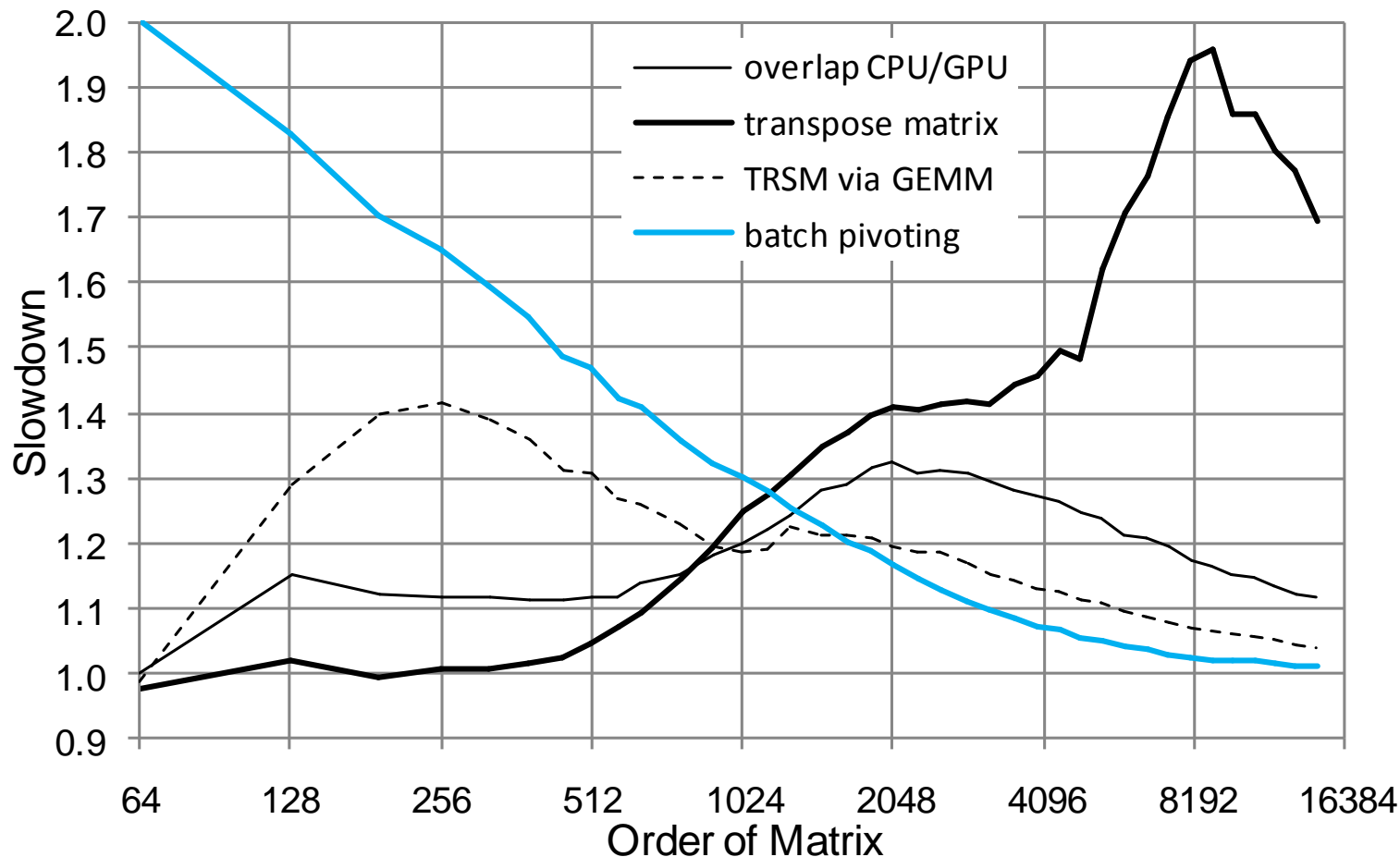




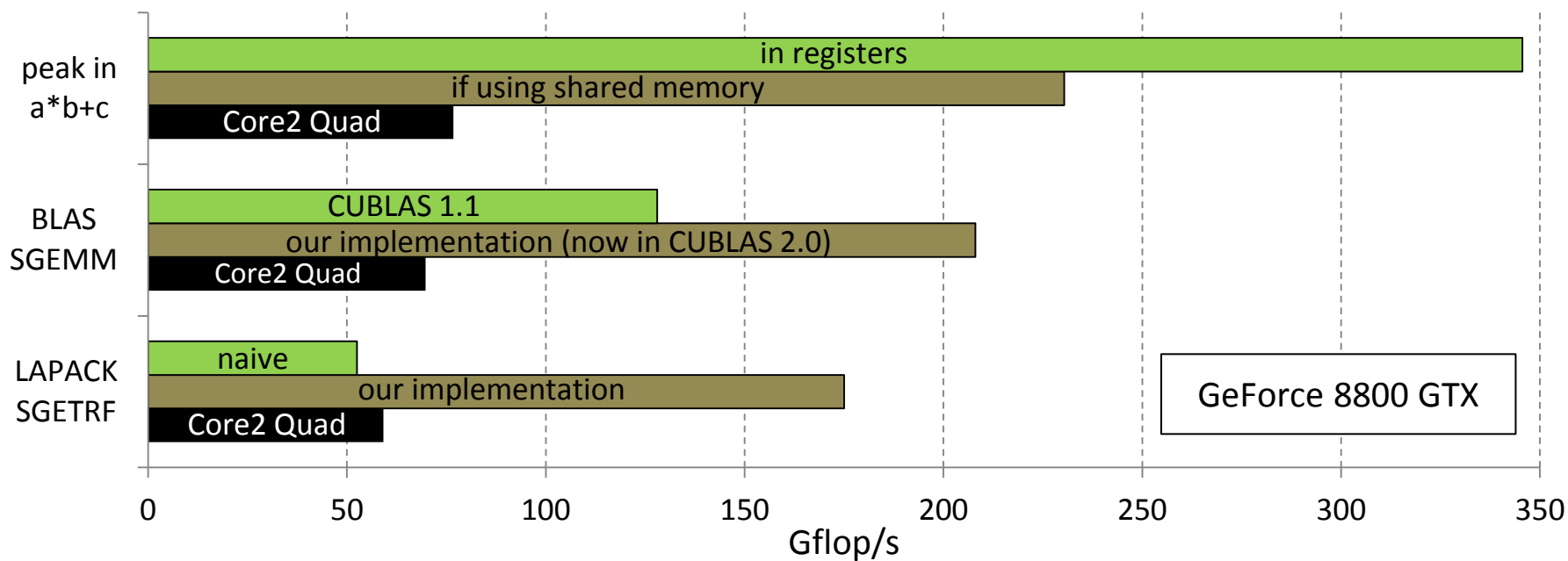
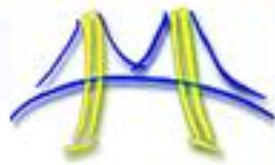
# Importance of various optimizations on GPU



- Slowdown when omitting one of the optimizations on GTX 280



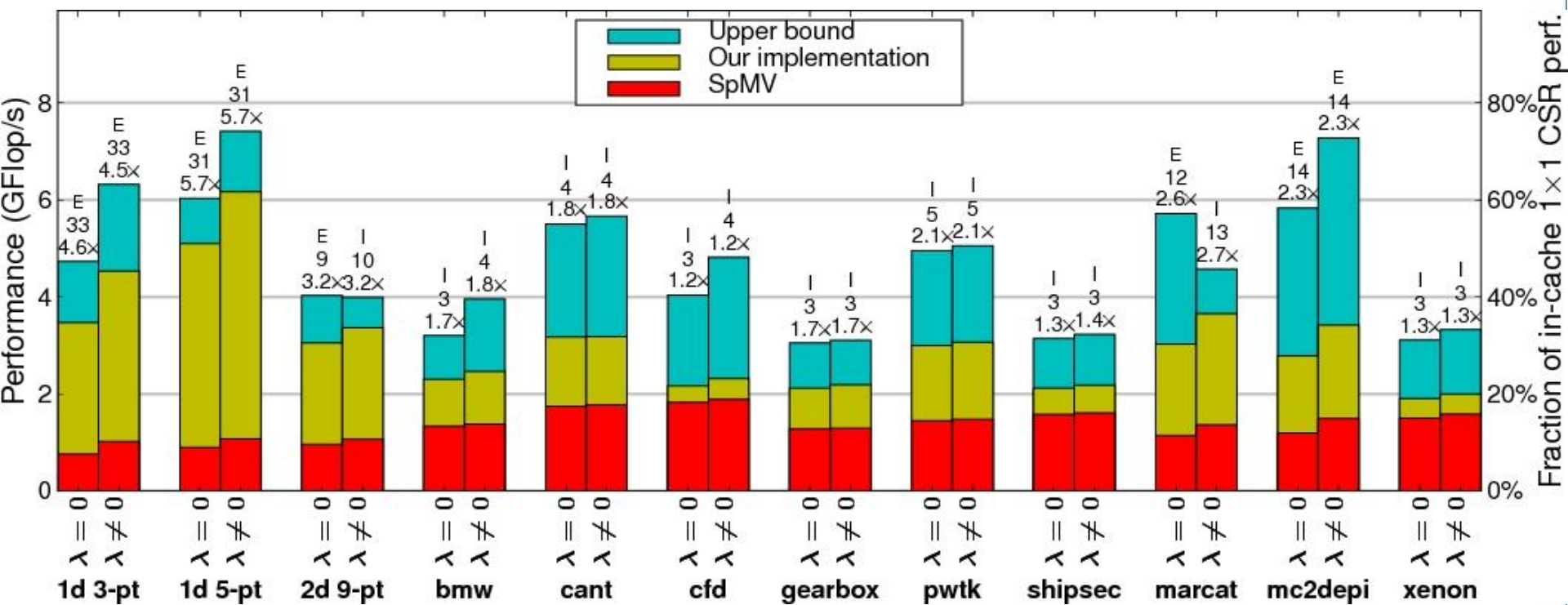
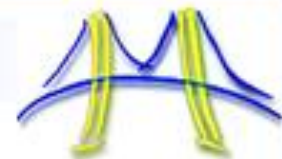
# Results for matmul, LU on NVIDIA



- 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

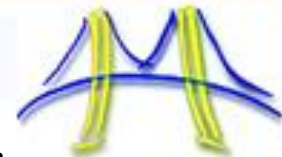


# Performance results on 8-Core Clovertown





QR of a tall, skinny matrix is bottleneck;  
Use TSQR instead: example using 4 procs



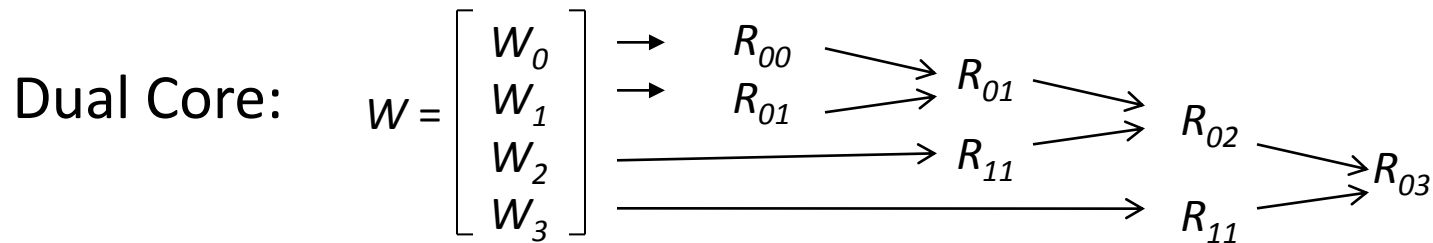
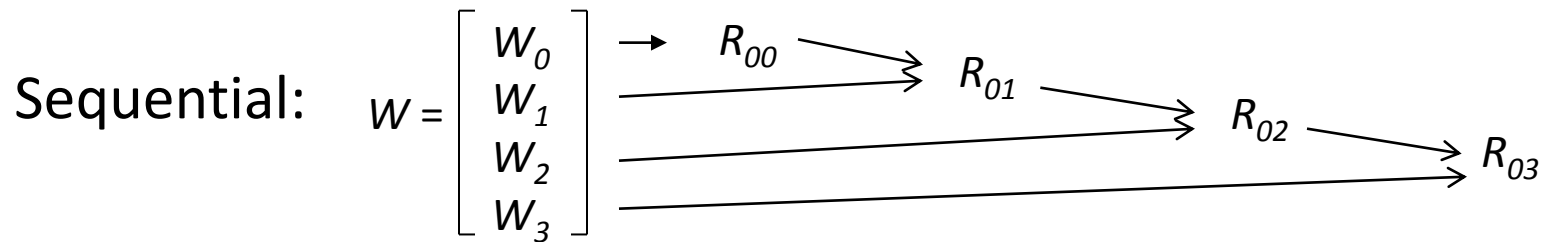
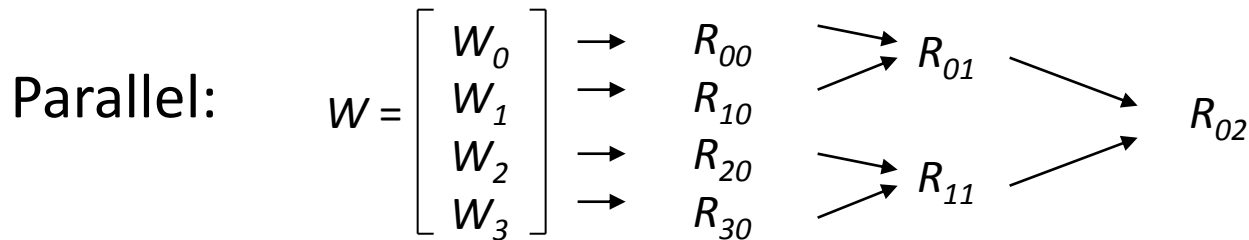
$$W = \begin{pmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{pmatrix} = \begin{pmatrix} Q_{00} \\ \hline Q_{10} \\ \hline Q_{20} \\ \hline Q_{30} \end{pmatrix} \cdot \begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix}$$

$$\begin{pmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{pmatrix} = \begin{pmatrix} Q_{01} \\ \hline Q_{11} \end{pmatrix} \cdot \begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix}$$

$$\begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} = Q_{02} R_{02}$$

Q is represented implicitly as a product (tree of factors)

# Minimizing Communication in TSQR

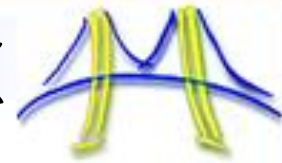


Multicore / Multisocket / Multirack / Multisite / Out-of-core: ?

Can choose reduction tree dynamically



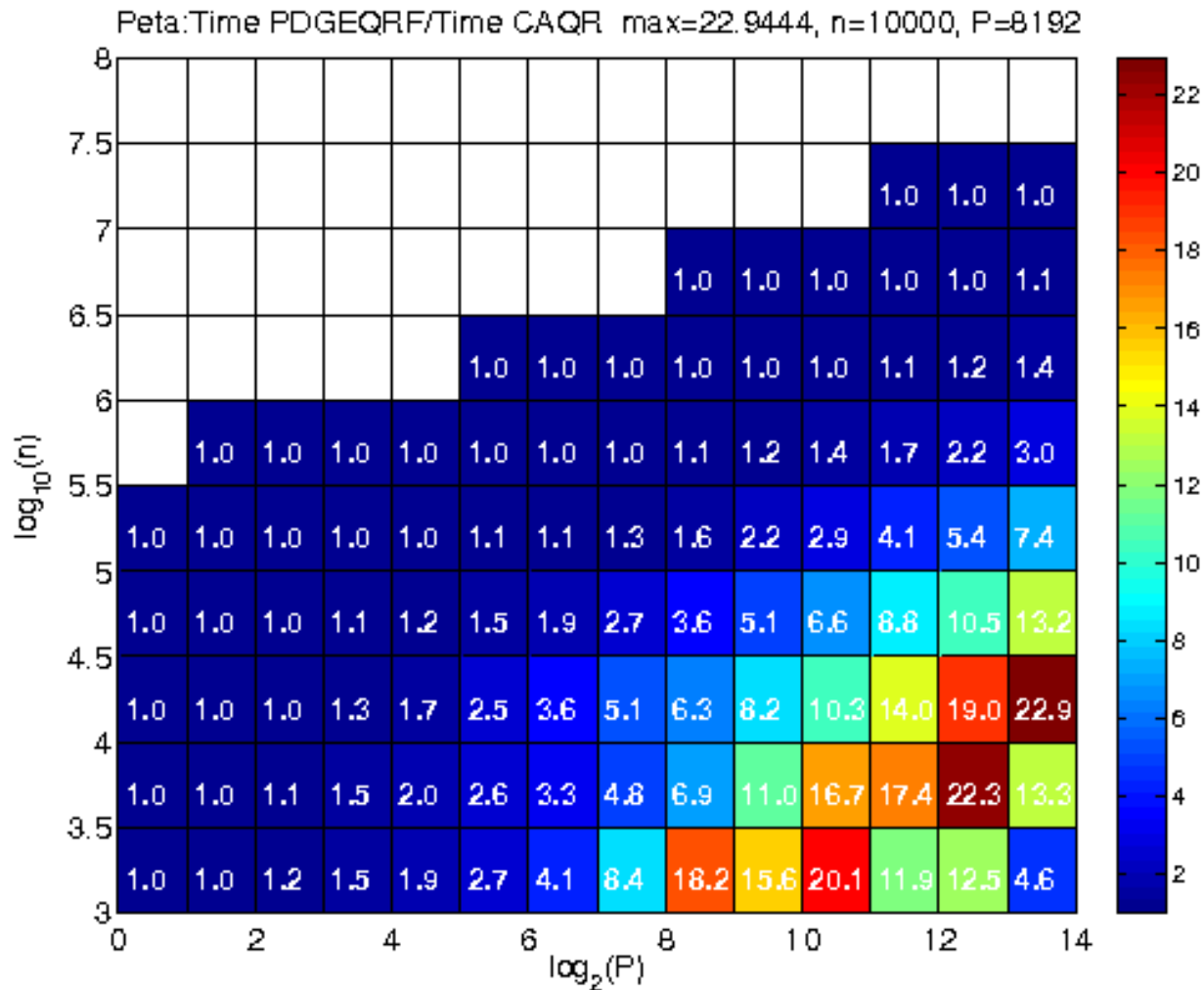
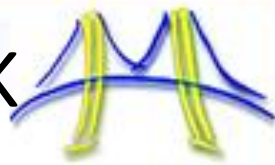
# Performance of TSQR vs Sca/LAPACK



- **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)
  - **Fermi GPU**– early results (work in progress)
    - **16x speedup** (vs LAPACK on Nehalem)
    - **6x speedup** versus conventional algorithm (tuned!) on Fermi
- **Sequential**
  - **Out-of-Core on PowerPC laptop**
    - $\infty$  x **speedup** vs LAPACK with virtual memory, which never finished
    - As little as 2x slowdown vs (predicted) infinite DRAM
- **Grid** – **4x speedup** on 4 cities (Dongarra et al)
- **Cloud** – early result – up and running using Mesos

# Modeled Speedups of CAQR vs ScaLAPACK

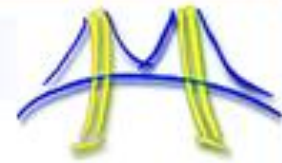
Up to **22.9x speedup** on modeled Petascale machine



Petascale machine with 8192 procs, each at 500 GFlops/s, a bandwidth of 4 GB/s.

$$\gamma = 2 \cdot 10^{-12} s, \alpha = 10^{-5} s, \beta = 2 \cdot 10^{-9} s / \text{word}.$$





# Additional OSKI Features

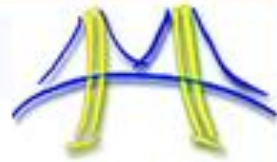
- Embedded scripting language for selecting customized, complex transformations
  - Mechanism to save/restore transformations

```
/* In "my_app.c" */  
fp = fopen("my_xform.txt", "rt");  
fgets(buffer, BUFSIZE, fp);  
  
oski_ApplyMatTransform(A_tunable,  
    buffer);  
oski_MatMult(A_tunable, ...);
```

```
# In file, "my_xform.txt"  
# Compute  $A_{\text{fast}} = P * A * P^T$  using  
  Pinar's reordering algorithm  
A_fast, P =  
    reorder_TSP(InputMat);  
# Split  $A_{\text{fast}} = A_1 + A_2$ , where  $A_1$  in 2x2  
  block format,  $A_2$  in CSR  
A1, A2 =  
    A_fast.extract_blocks(2, 2);  
  
return transpose(P) * (A1+A2) * P;
```



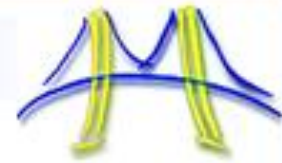
# Requested new functionality for OSKI



- ❖ Users want to be able to
  - Change non-zero pattern of the matrix
    - Matrix may change or be perturbed during computation
    - Currently not permitted in OSKI- would require retuning
  - Assemble a matrix from fragments
    - Common in finite element methods
    - Fragments may overlap



# Proposed OSKI Interface Changes



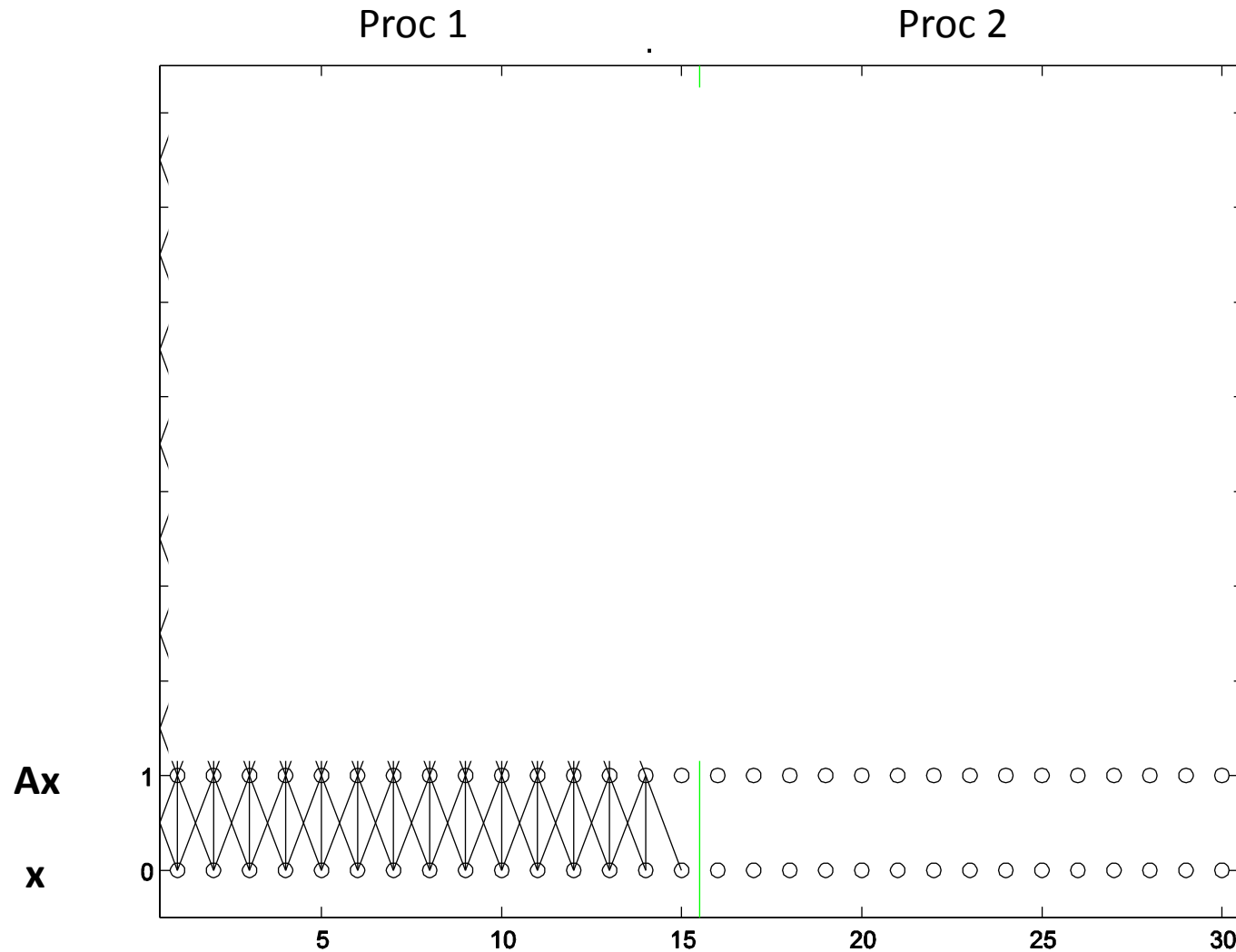
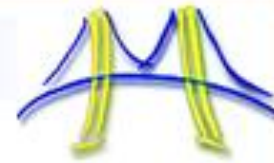
## ❖ New data structure: List of matrices

- Allows a matrix to be expressed as a sum of matrices ( $A=A_1+\dots+A_n$ )
  - Easily allows for assembly from fragments and variable splitting
  - Pattern update: represent the changed entry as the addition of another matrix

## ❖ Merge method

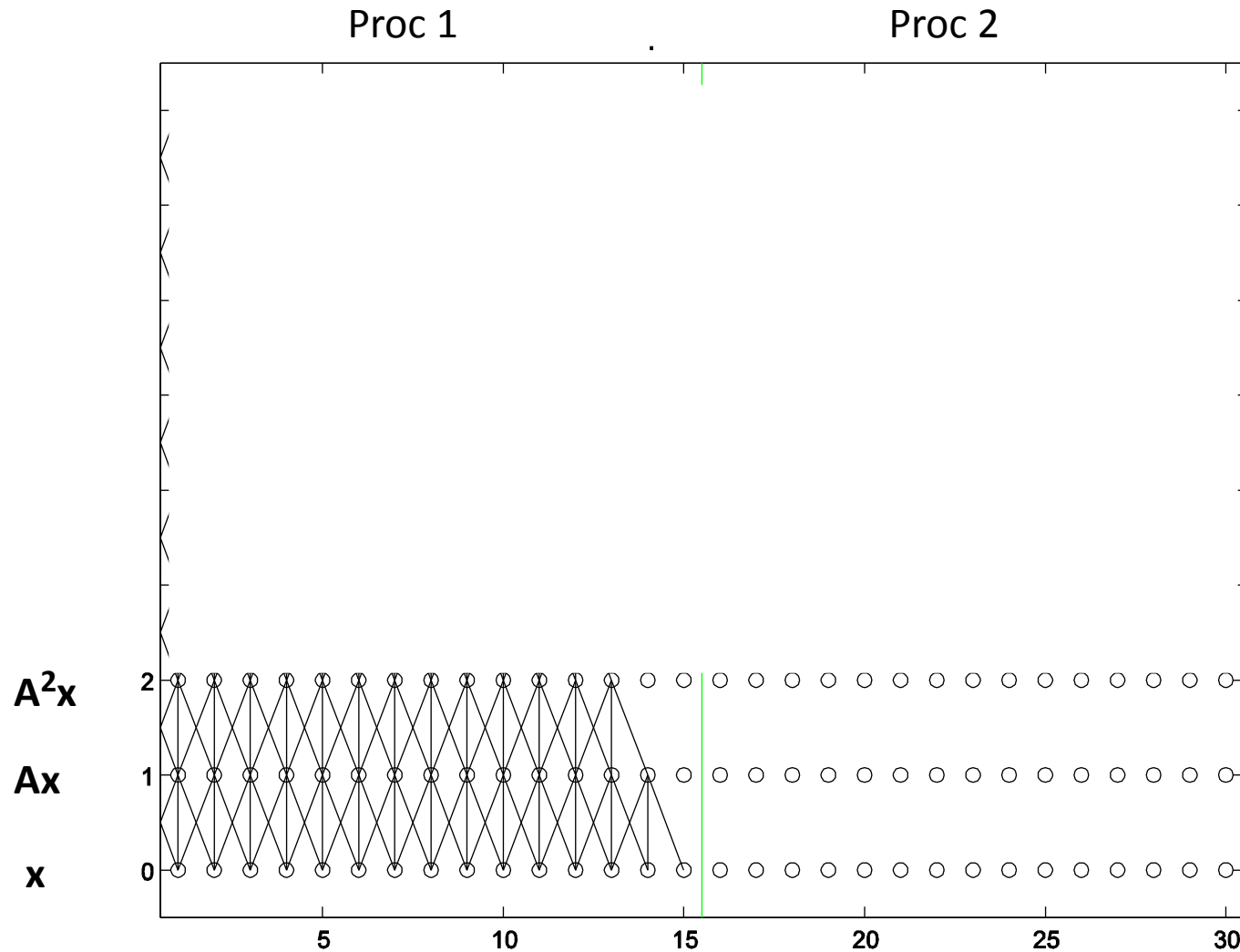
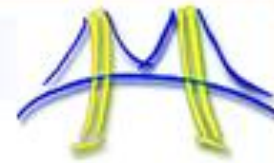
- Merges the list  $A_1+\dots+A_n$  into a single matrix
- User can decide when to merge matrices, or...
- In the future, merging may also be a tuning decision made by OSKI
  - Ex: combining matrices with only a few entries

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



Can be computed without communication

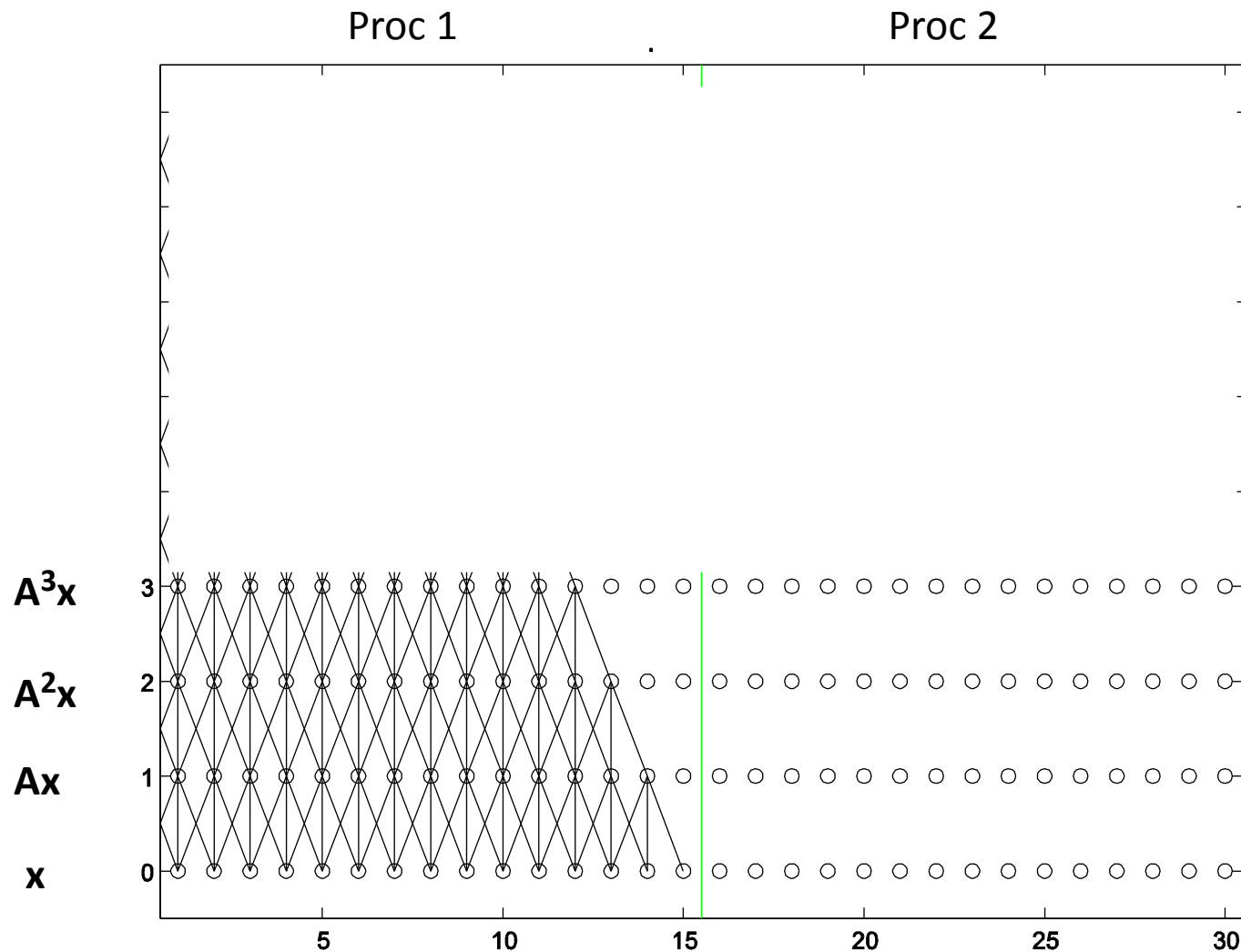
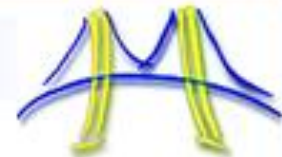
# Locally Dependent Entries for [ $x, Ax, A^2x$ ], $A$ tridiagonal, 2 processors



Can be computed without communication



# Locally Dependent Entries for $[x, Ax, \dots, A^3x]$ , $A$ tridiagonal, 2 processors



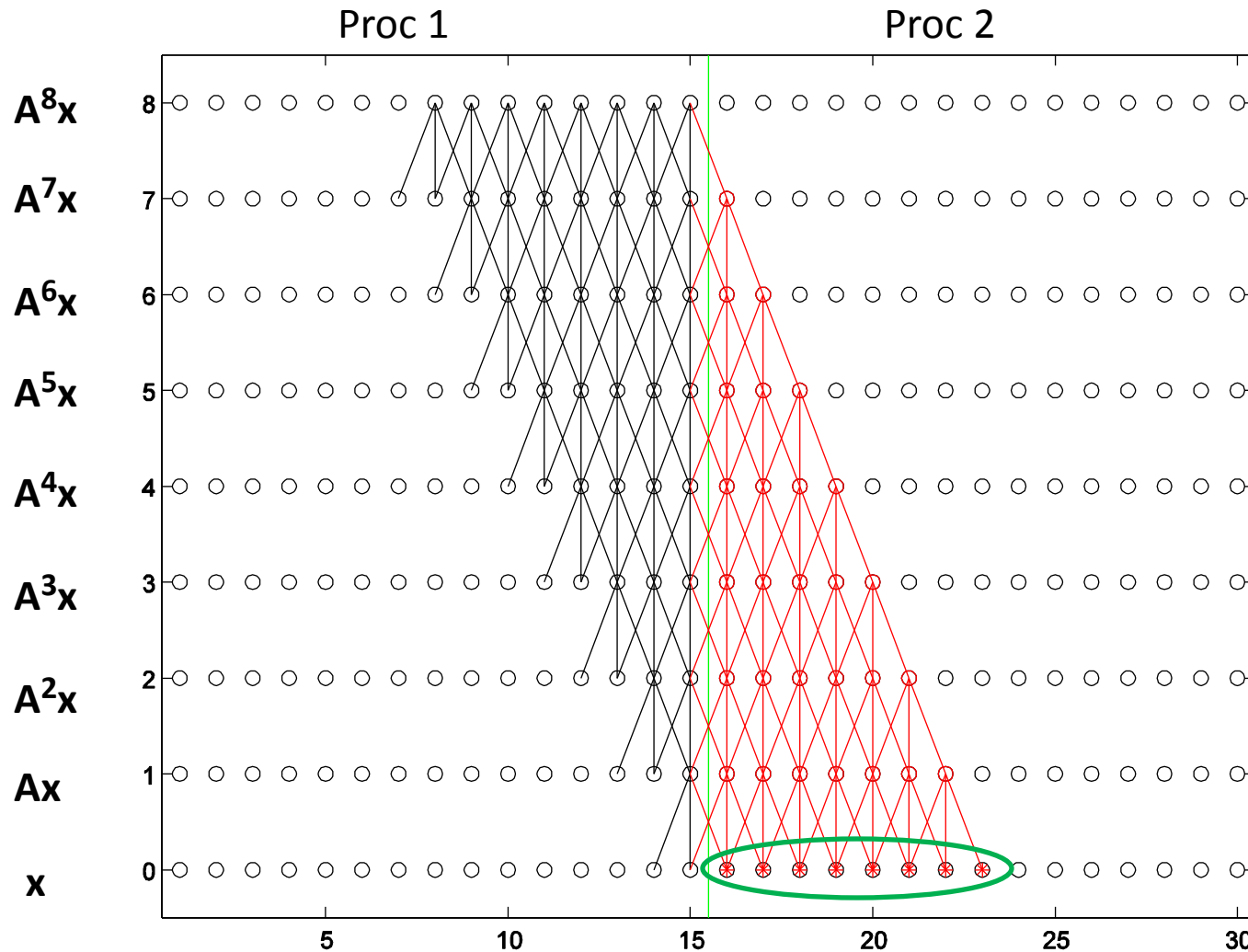
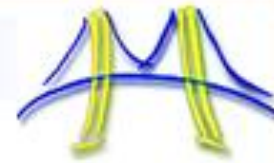
Can be computed without communication





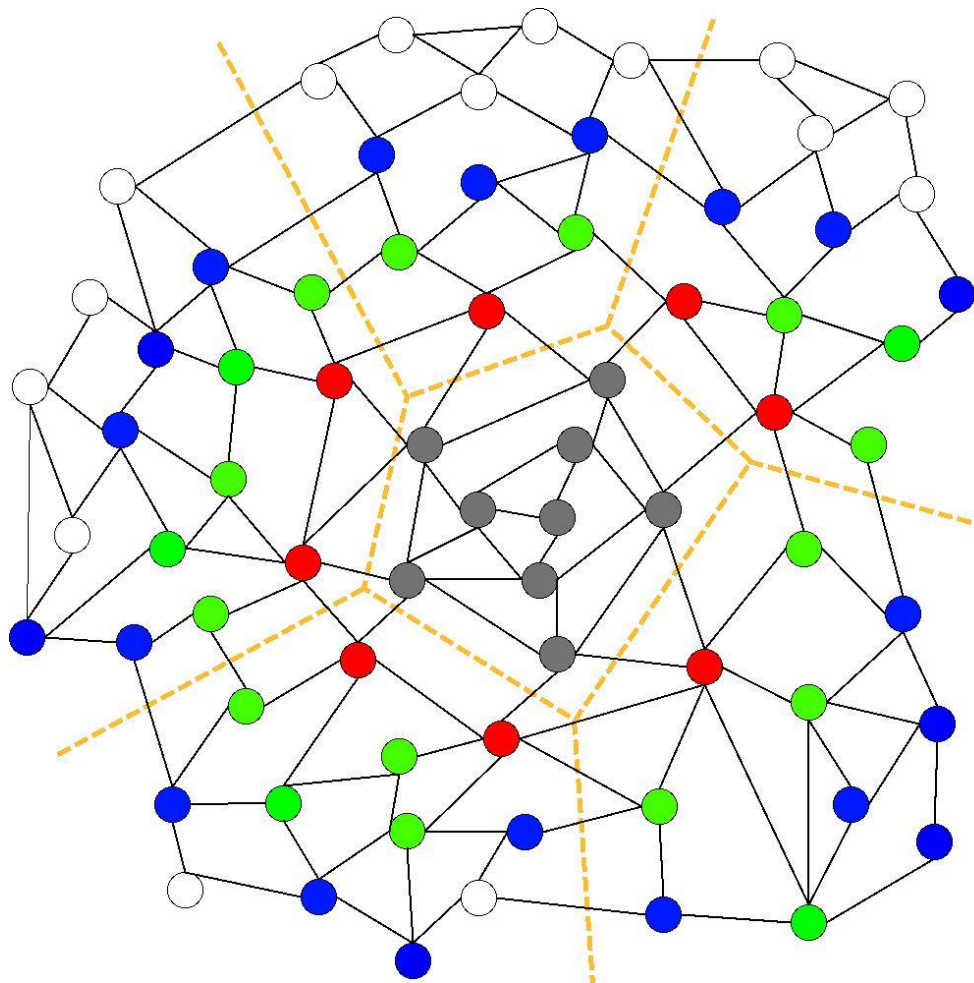
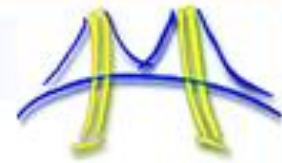


# Remotely Dependent Entries for $[x, Ax, \dots, A^8x]$ , $A$ tridiagonal, 2 processors

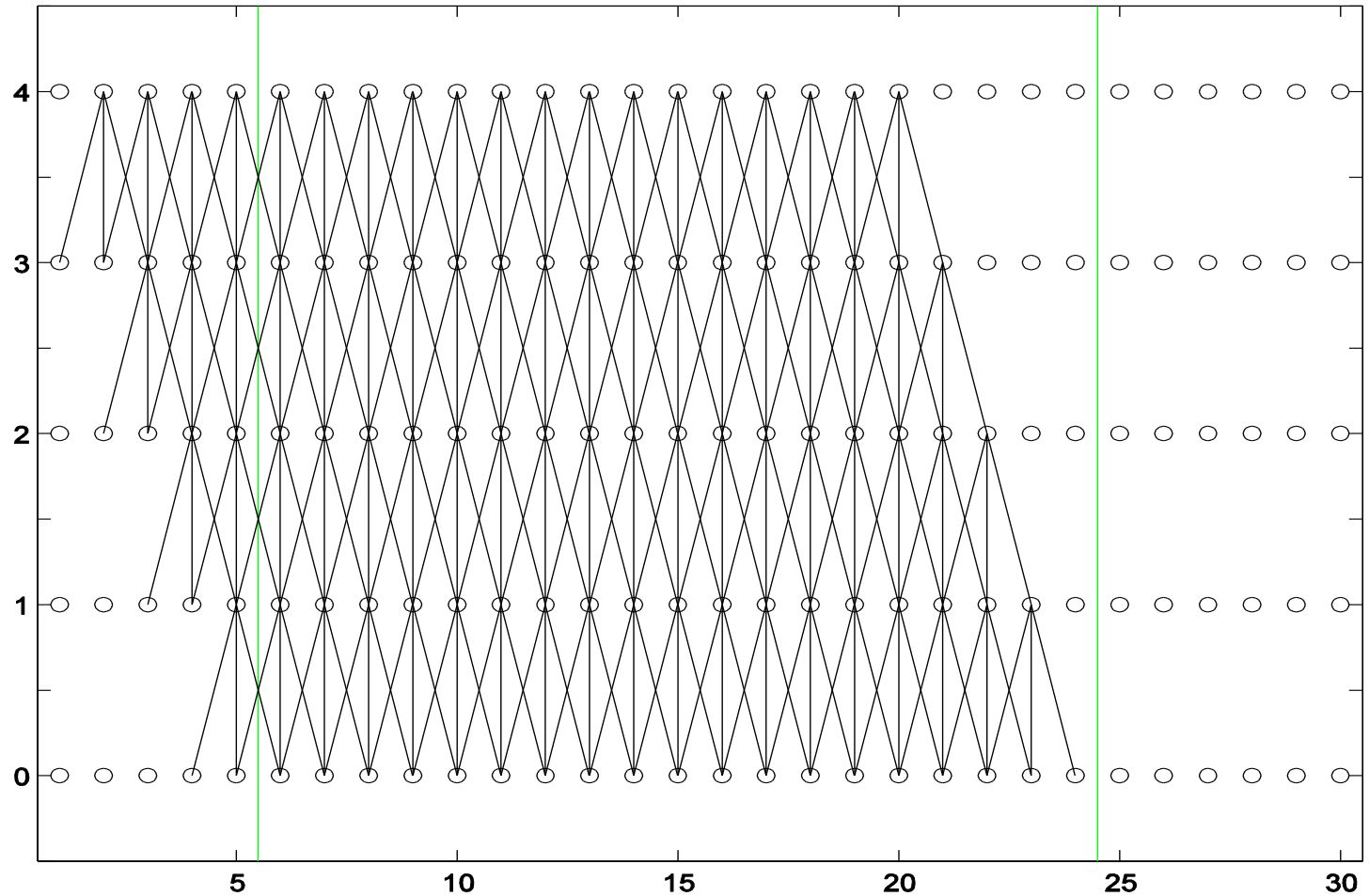
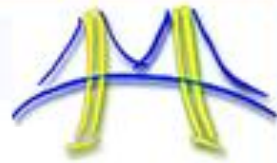


**One** message to get data needed to compute remotely dependent entries, **not**  $k=8$   
 Minimizes number of messages = latency cost  
 Price: **redundant work**  $\propto$  “surface/volume ratio”

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



# Sequential $[x, Ax, \dots, A^4x]$ , with memory hierarchy

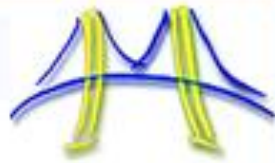


***One*** read of matrix from slow memory, ***not***  $k=4$

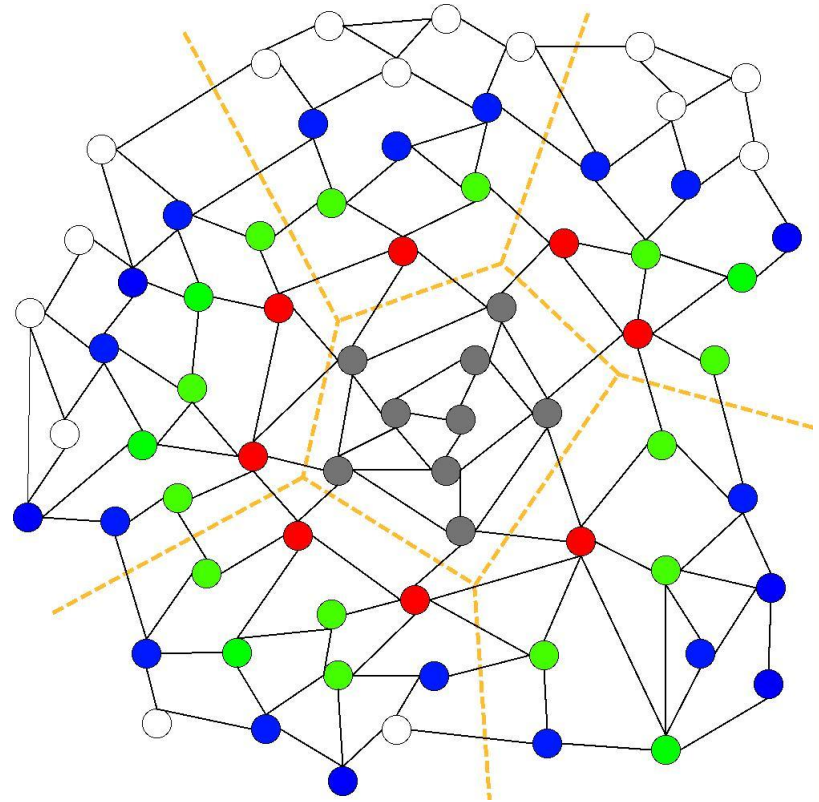
Minimizes words moved = bandwidth cost

No redundant work

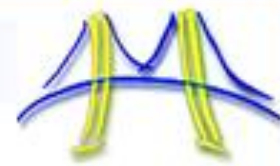
# In what order should the sequential algorithm process a general sparse matrix?



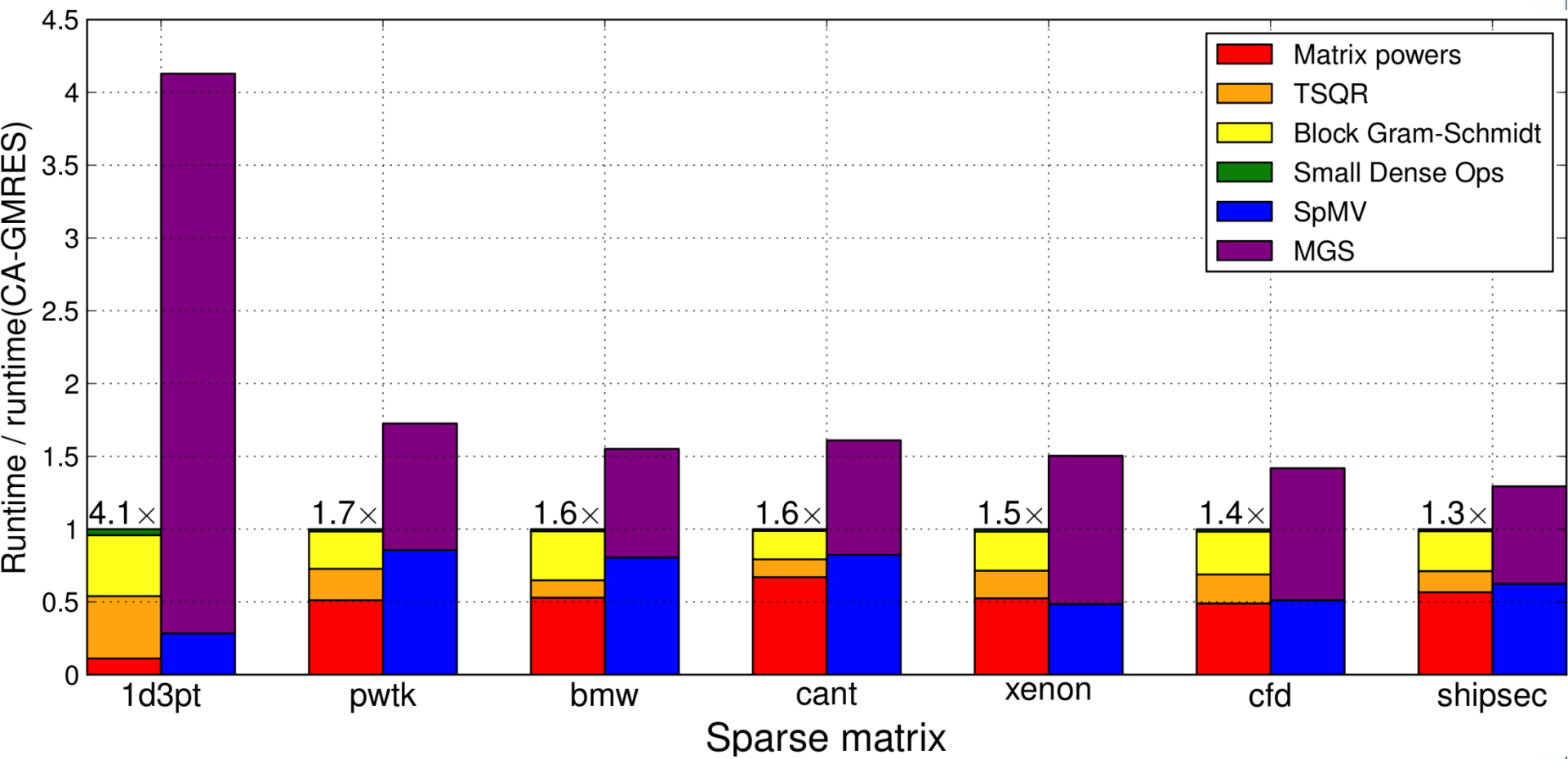
- Process band matrix from left to right, to reuse data already in fast memory
- Best order not obvious for a general matrix: can formulate as Traveling Salesman Problem (TSP)



- One vertex per matrix partition
- Weight of edge  $(j, k)$  is memory cost of processing partition  $k$  right after partition  $j$
- TSP: find lowest cost “tour” visiting all vertices



# Nehalem Speedups



# Our Pattern Language 2.0

Productivity Layer

Efficiency Layer

## 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

**Choose you high level architecture? Guided decomposition**

Task Decomposition ↔ Data Decomposition

Group Tasks    Order groups    data sharing    data access

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

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

**Identify the key computational patterns – what are my key computations?**

### Guided instantiation

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

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

Event Based	Data Parallelism	Pipeline	Task Parallelism	Digital Circuits
Divide and Conquer	Geometric Decomposition	Discrete Event	Graph algorithms	

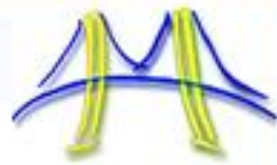
**Utilize Supporting Structures – how do I implement my concurrency? Guided mapping**

		Shared Queue	Master/worker
Fork/Join	Distributed Array	Shared Hash Table	Loop Parallelism
CSP	Shared Data	SPMD	BSP

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

Thread Creation/destruction	Message passing	Speculation	Barriers	Semaphores
Process Creation/destruction	Collective communication	Transactional memory	Mutex	

# 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	$N^{3/2}$ ( $N^{4/3}$ )	$N^{1/2}$ ( $1/3$ ) $\cdot \log N$	$N$	$N$
• Red/Black SOR	$N^{3/2}$ ( $N^{4/3}$ )	$N^{1/2}$ ( $N^{1/3}$ )	$N$	$N$
• Sparse LU	$N^{3/2}$ ( $N^2$ )	$N^{1/2}$	$N \cdot \log N$ ( $N^{4/3}$ )	$N$
• FFT	$N \cdot \log N$	$\log N$	$N$	$N$
• Multigrid	$N$	$\log^2 N$	$N$	$N$
• Lower bound	$N$	$\log N$	$N$	

For more details, Ma221 offered this semester!