

Automatic Performance Tuning of Sparse Matrix Kernels

Berkeley Benchmarking and OPTimization (BeBOP) Project
<http://www.cs.berkeley.edu/~richie/bebop>

James Demmel, Katherine Yelick
Richard Vuduc, Shoaib Kamil, Rajesh Nishtala, Benjamin Lee
Atilla Gyulassy, Chris Hsu
University of California, Berkeley

January 24, 2003

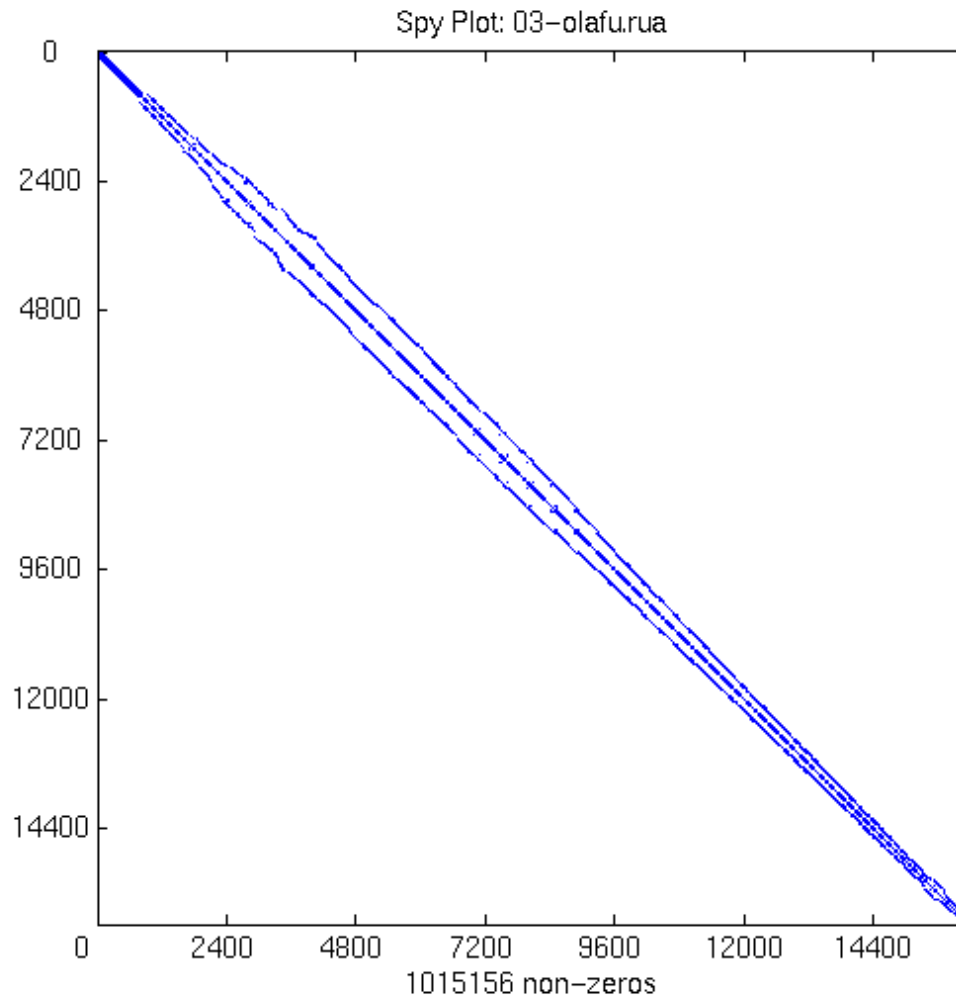
Outline

- Performance tuning challenges
 - Demonstrate complexity of tuning
- Automatic performance tuning
 - Overview of techniques and results
 - New results for Itanium 2
- Structure of the Google matrix
 - What optimizations are likely to pay-off?
 - Preliminary experiments: 2x speedups possible on Itanium 2

Tuning Sparse Matrix Kernels

- Sparse tuning issues
 - Typical uniprocessor performance < 10% machine peak
 - Indirect, irregular memory references—poor locality
 - High bandwidth requirements, poor instruction mix
 - Performance depends on architecture, kernel, *and matrix*
 - How to select data structures, implementations? at run-time?
- Our approach: for each kernel,
 - *Identify and generate* a space of implementations
 - *Search* to find the fastest (models, experiments)
- Early success: SPARSITY
 - sparse matrix-vector multiply (SpMV) [Im & Yelick '99]

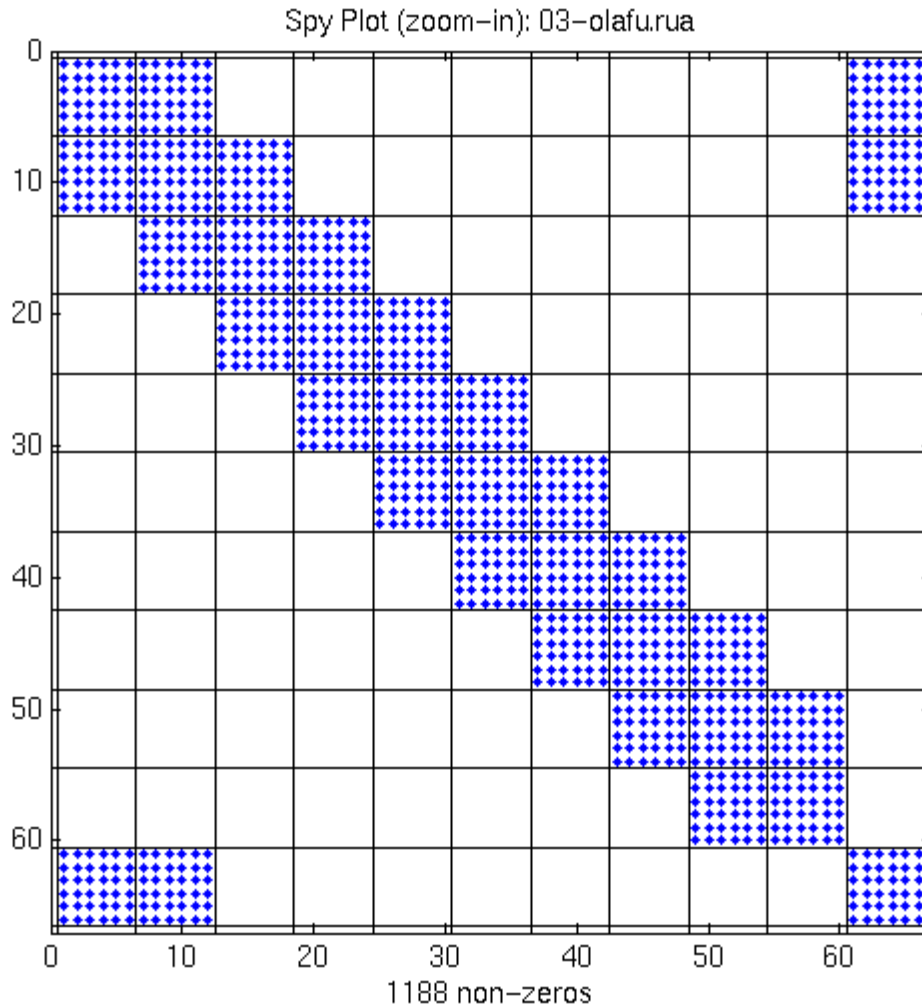
Sparse Matrix Example



- $n = 16146$
- $\text{nnz} = 1.0\text{M}$
- kernel: SpMV

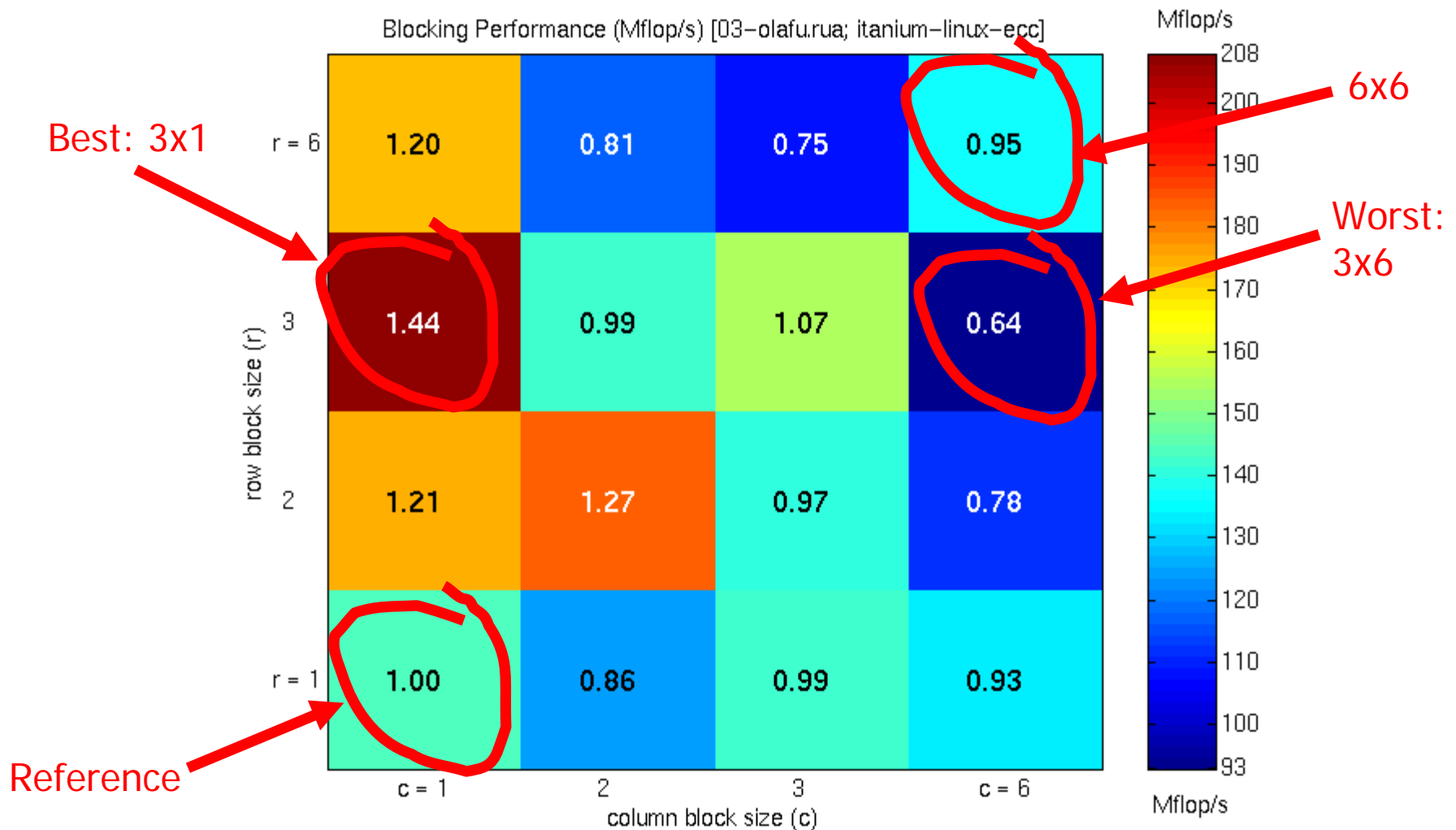
- Source: NASA structural analysis problem

Sparse Matrix Example (enlarged submatrix)

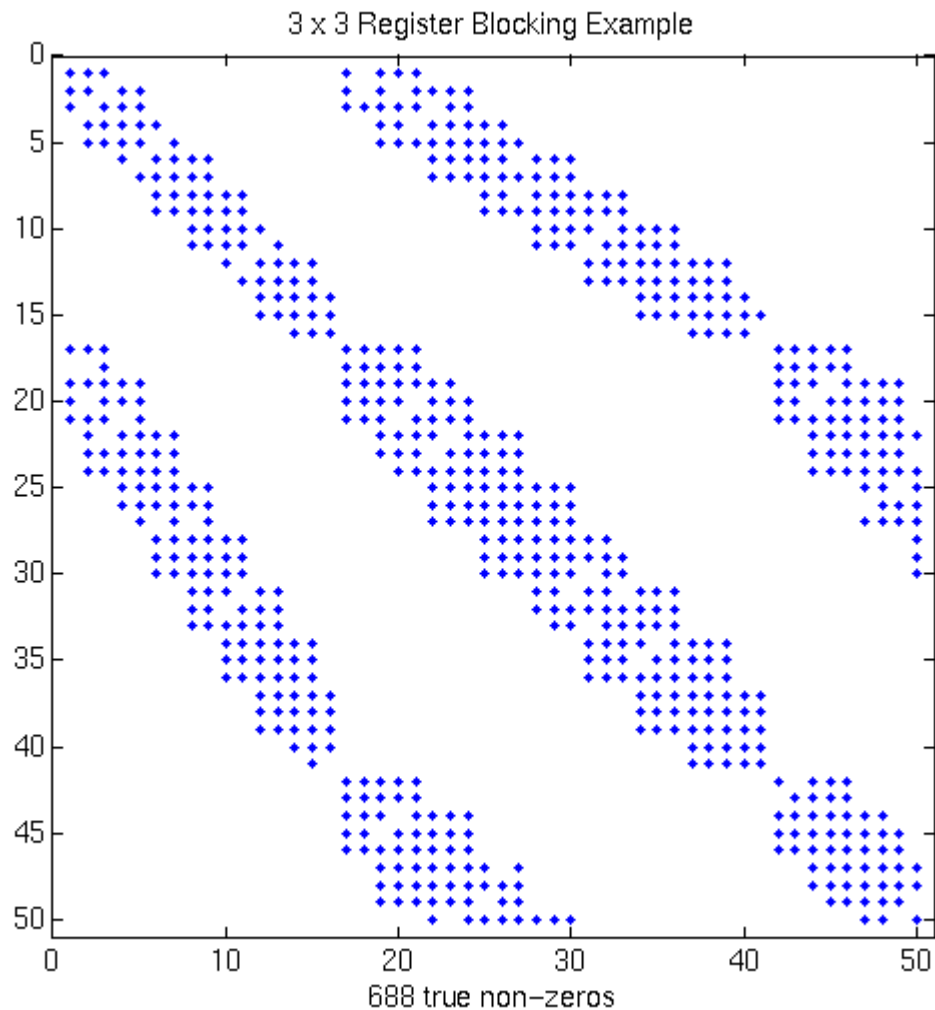


- $n = 16146$
- $nnz = 1.0M$
- kernel: SpMV
- Natural 6x6 dense block structure

Speedups on Itanium: The Need for Search

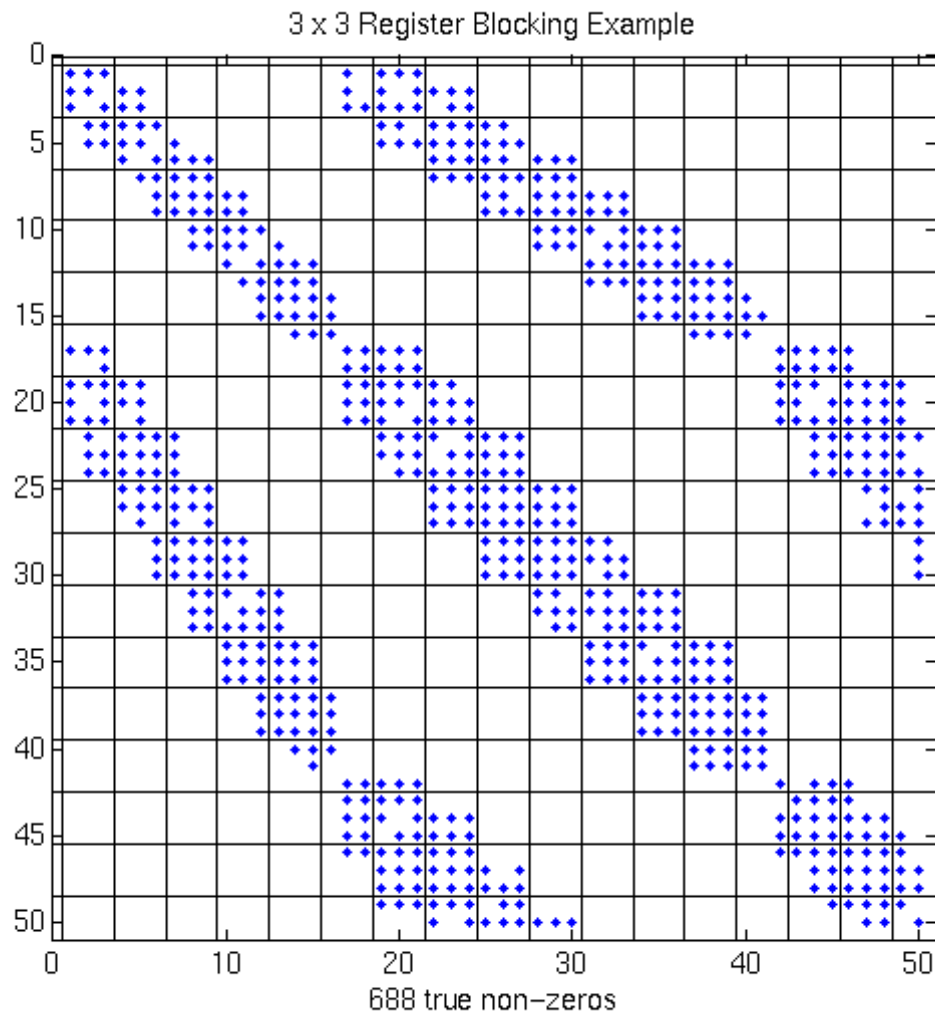


Filling-In Zeros to Improve Efficiency



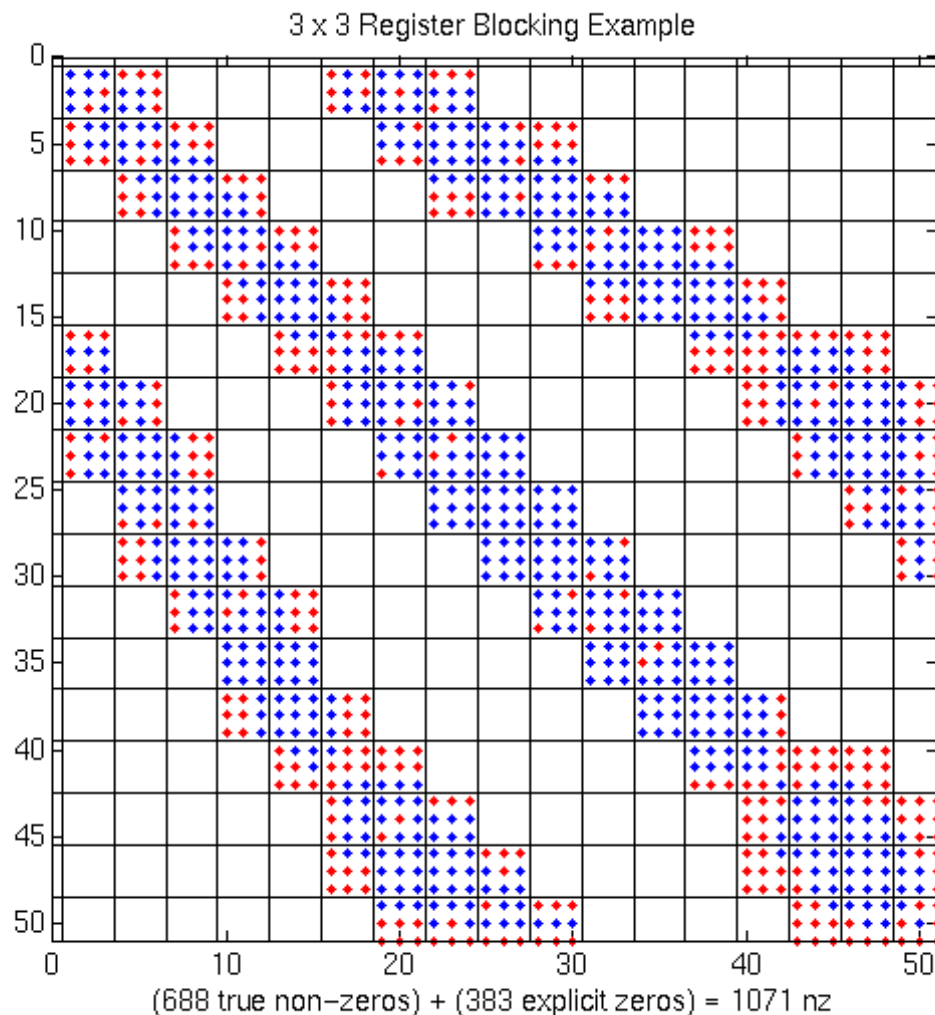
- More complicated non-zero structure in general

Filling-In Zeros to Improve Efficiency



- More complicated non-zero structure in general
- One SPARSITY technique: uniform register-level blocking
- Example: 3x3 blocking
 - Logical 3x3 grid

Filling-In Zeros to Improve Efficiency

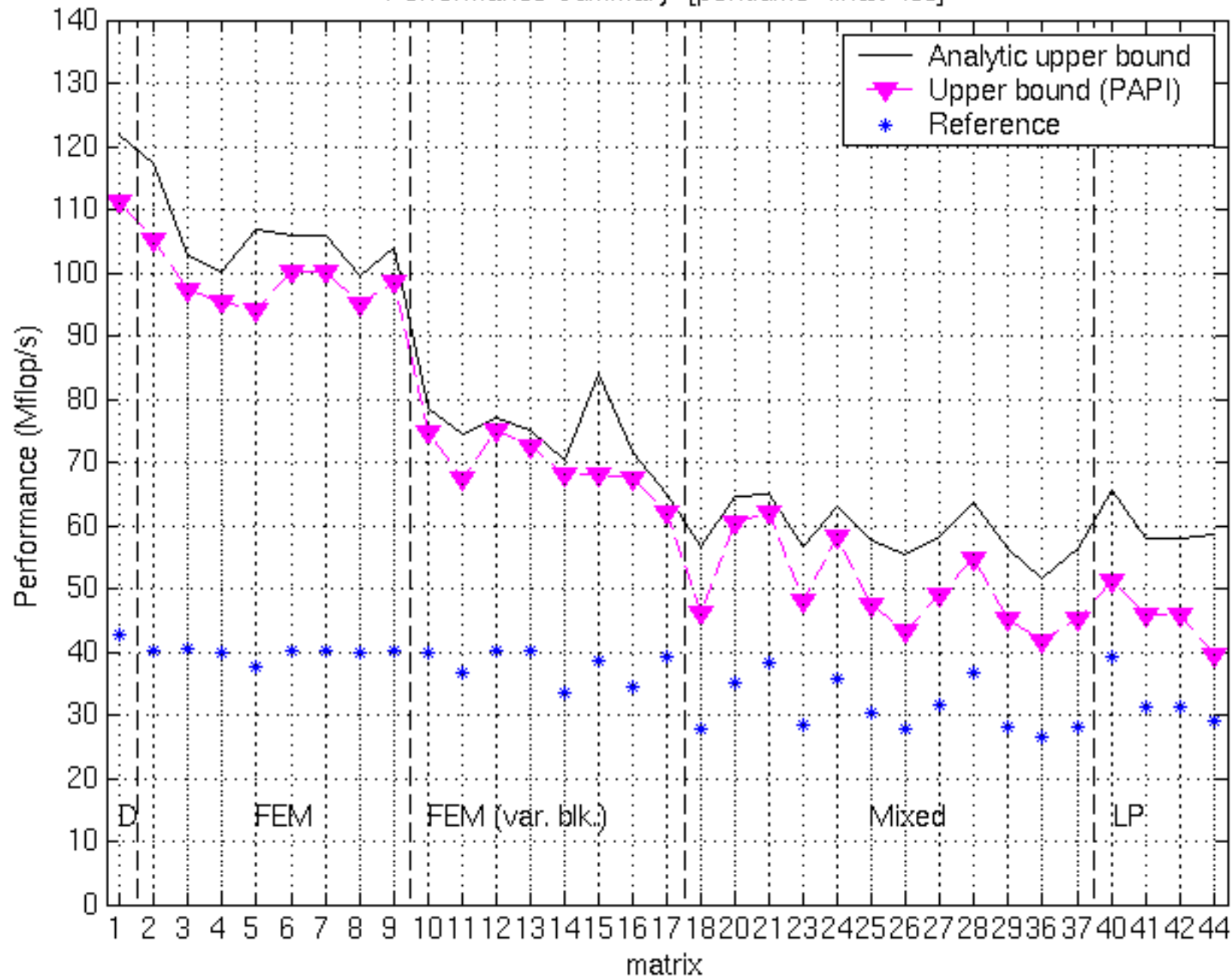


- More complicated non-zero structure in general
- One SPARSITY technique: uniform register-level blocking
- Example: 3x3 blocking
 - Logical 3x3 grid
 - Fill-in explicit zeros
 - “Fill ratio” = 1.5
- On Pentium III: **1.5x speedup!**

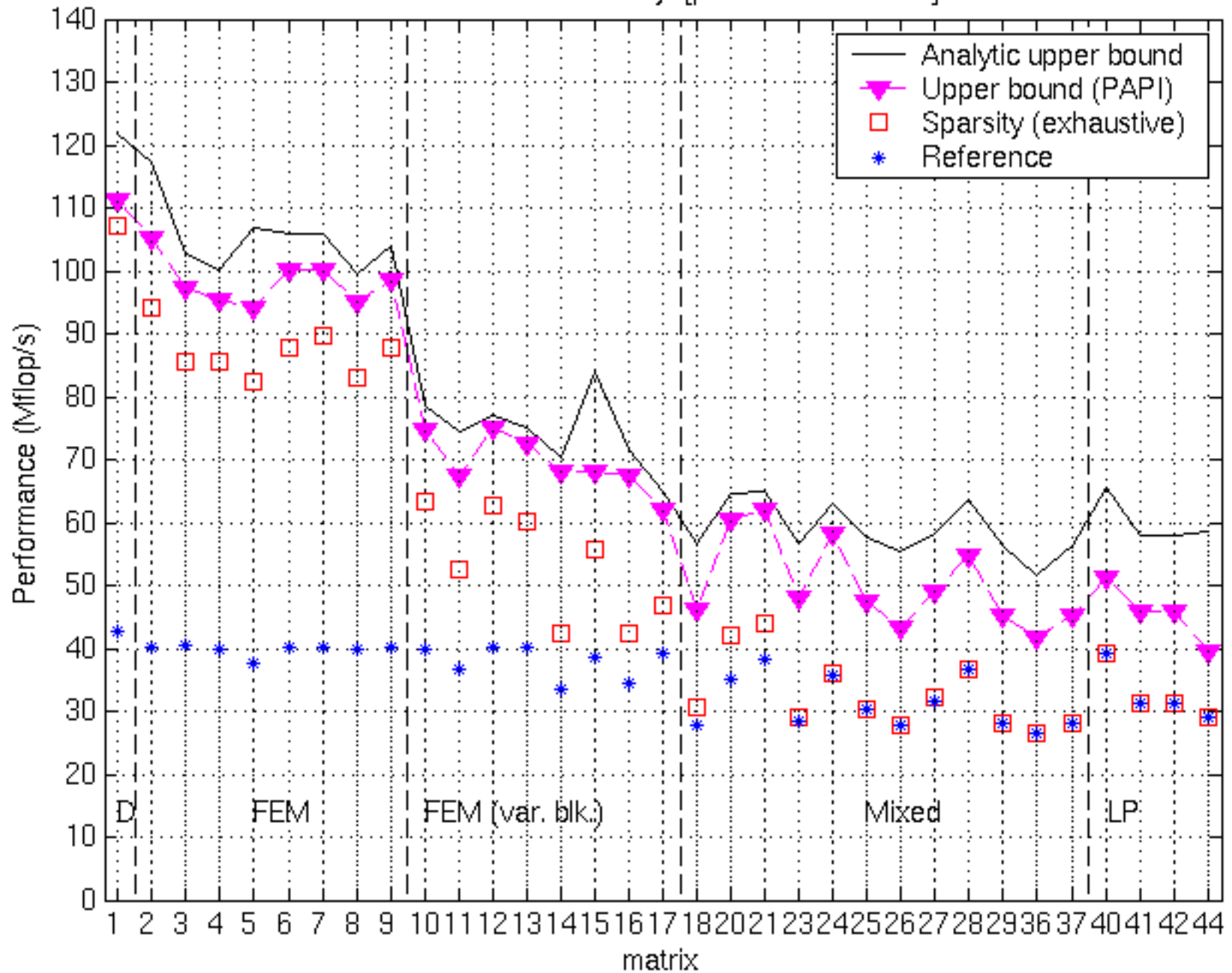
Approach to Automatic Tuning

- Recall: for each kernel,
 - Identify and generate implementation space
 - Search space to find fastest
- Selecting the $r \times c$ register block size
 - Off-line: Precompute performance **Mflops** of SpMV using dense A for various block sizes $r \times c$
 - Only once per architecture
 - Run-time: Given A , sample to estimate **Fill** for each $r \times c$
 - Choose r, c to maximize ratio **Mflops/Fill**

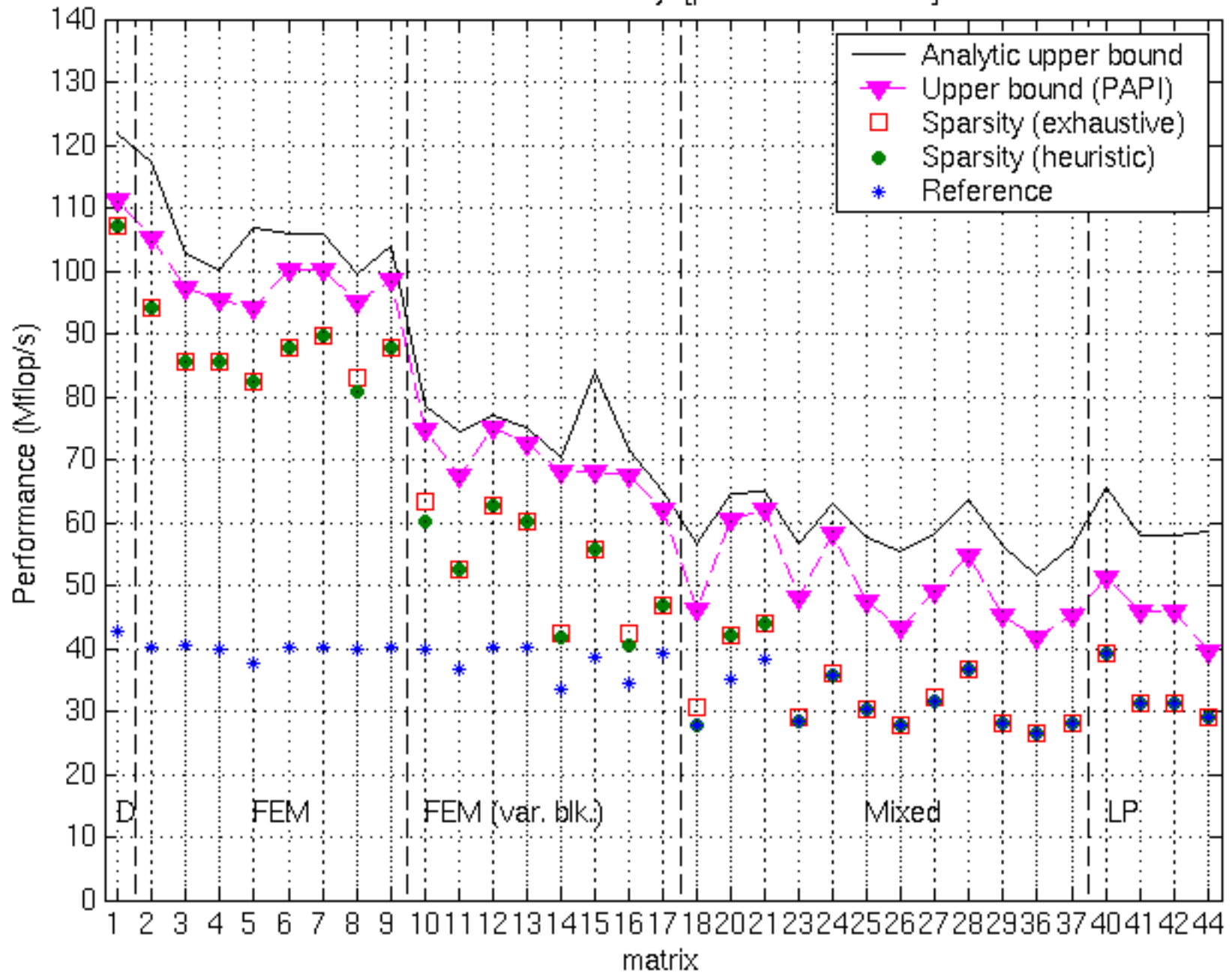
Performance Summary [pentium3-linux-icc]



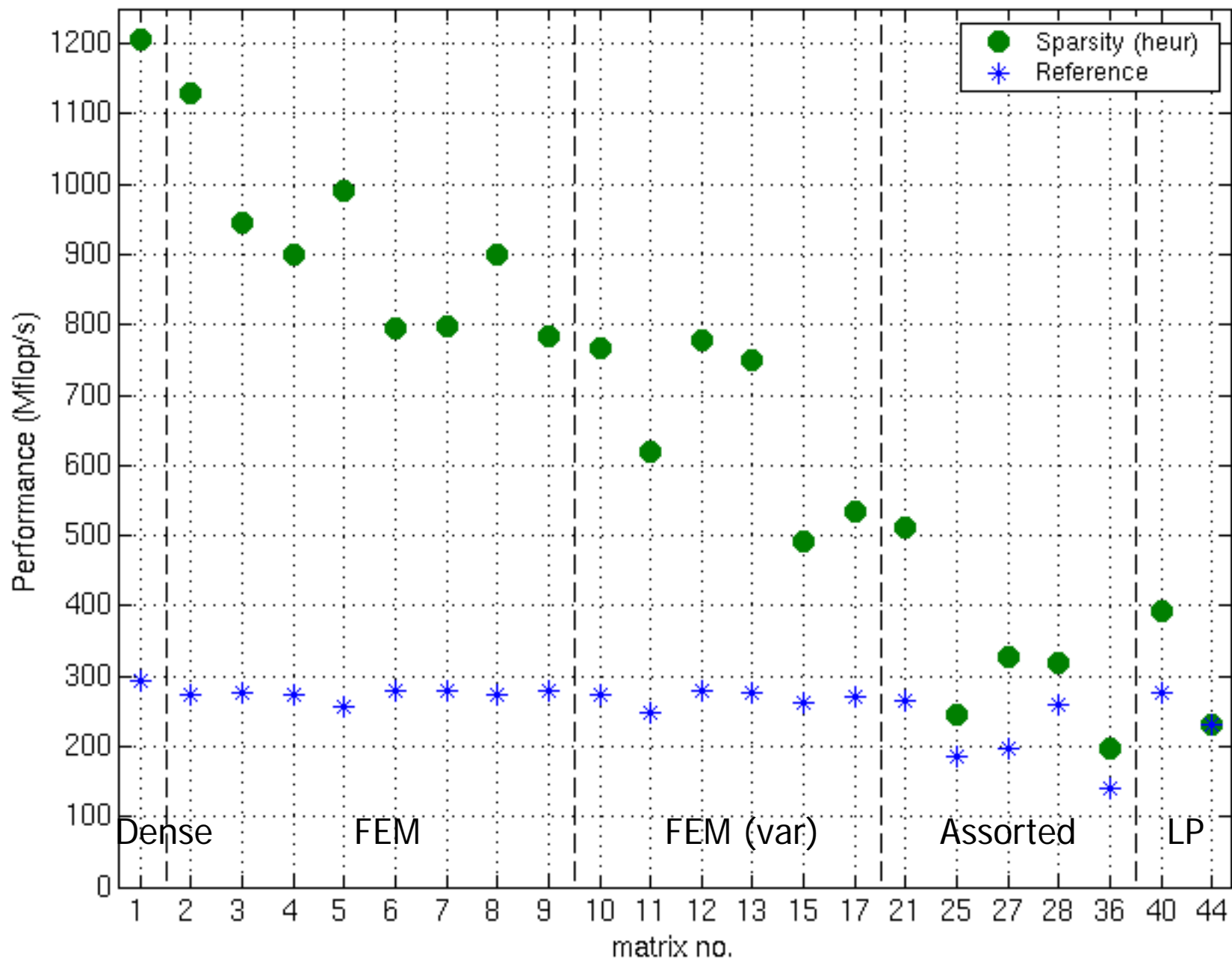
Performance Summary [pentium3-linux-icc]



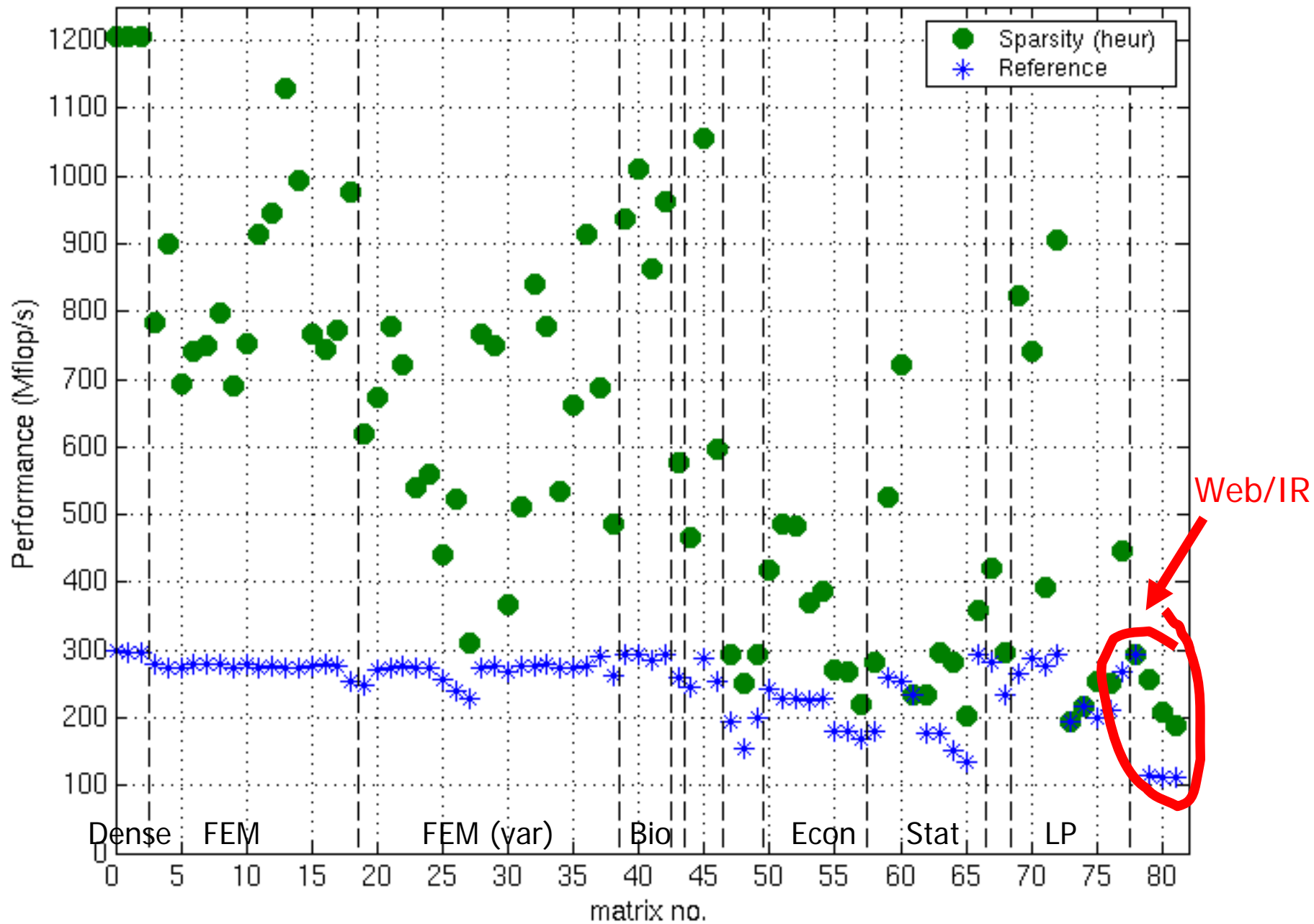
Performance Summary [pentium3-linux-icc]



Sparsity Register Blocking Performance [Itanium 2-900, Intel C v7.0]



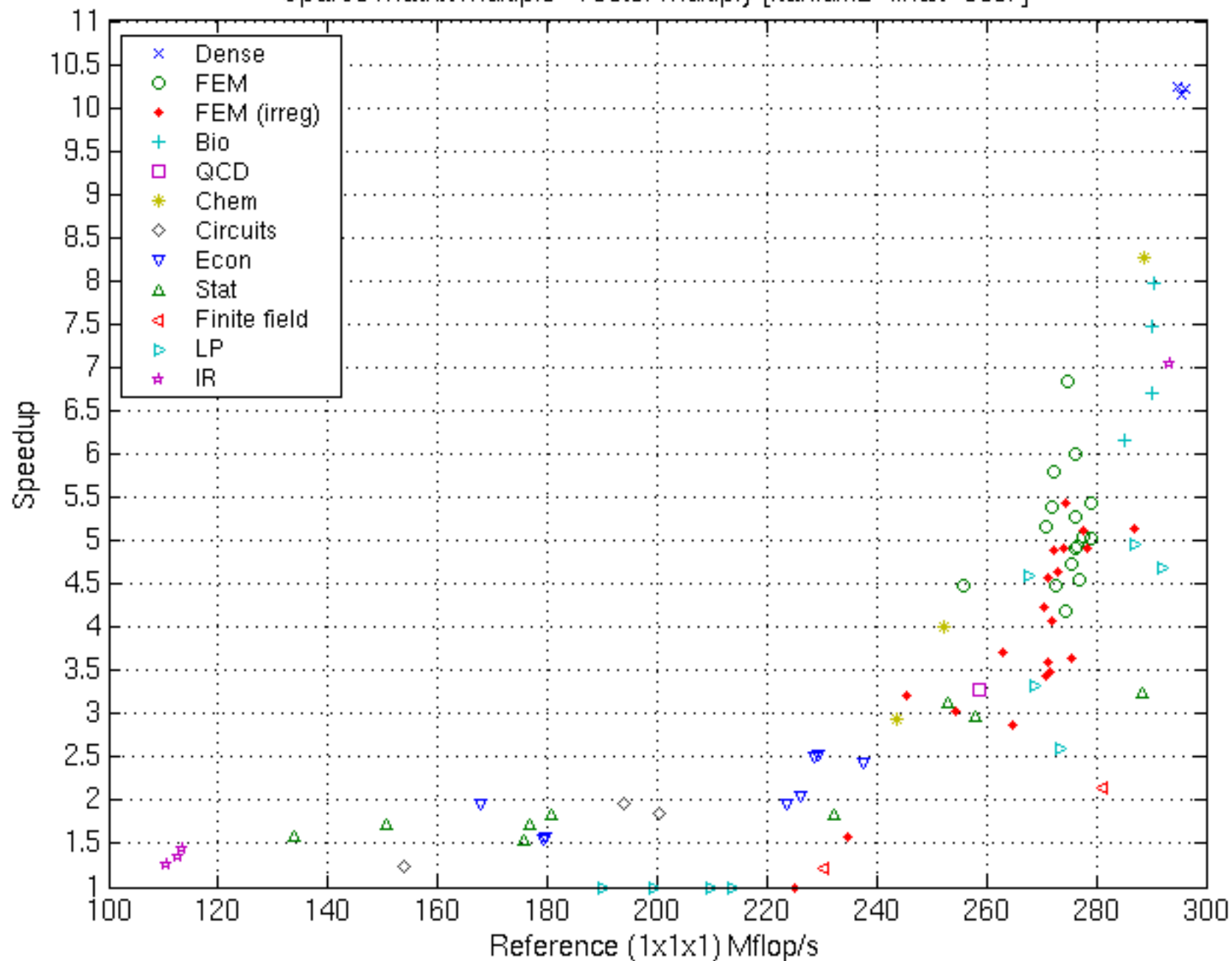
Sparsity Register Blocking Performance [Itanium 2-900, Intel C v7.0]



Exploiting Other Kinds of Structure

- Optimizations for SpMV
 - Symmetry (up to 2x speedup)
 - Diagonals, bands (up to 2.2x)
 - Splitting for variable block structure (1.3x—1.7x)
 - Reordering to create dense structure + splitting (up to 2x)
 - Cache blocking (1.5—4x)
 - Multiple vectors (2—7x)
 - And combinations...
- Sparse triangular solve
 - Hybrid sparse/dense data structure (1.2—1.8x)
- Higher-level kernels
 - $AA^T x$, $A^T A x$ (1.2—4.2x)
 - RAR^T , $A^k x$, ...

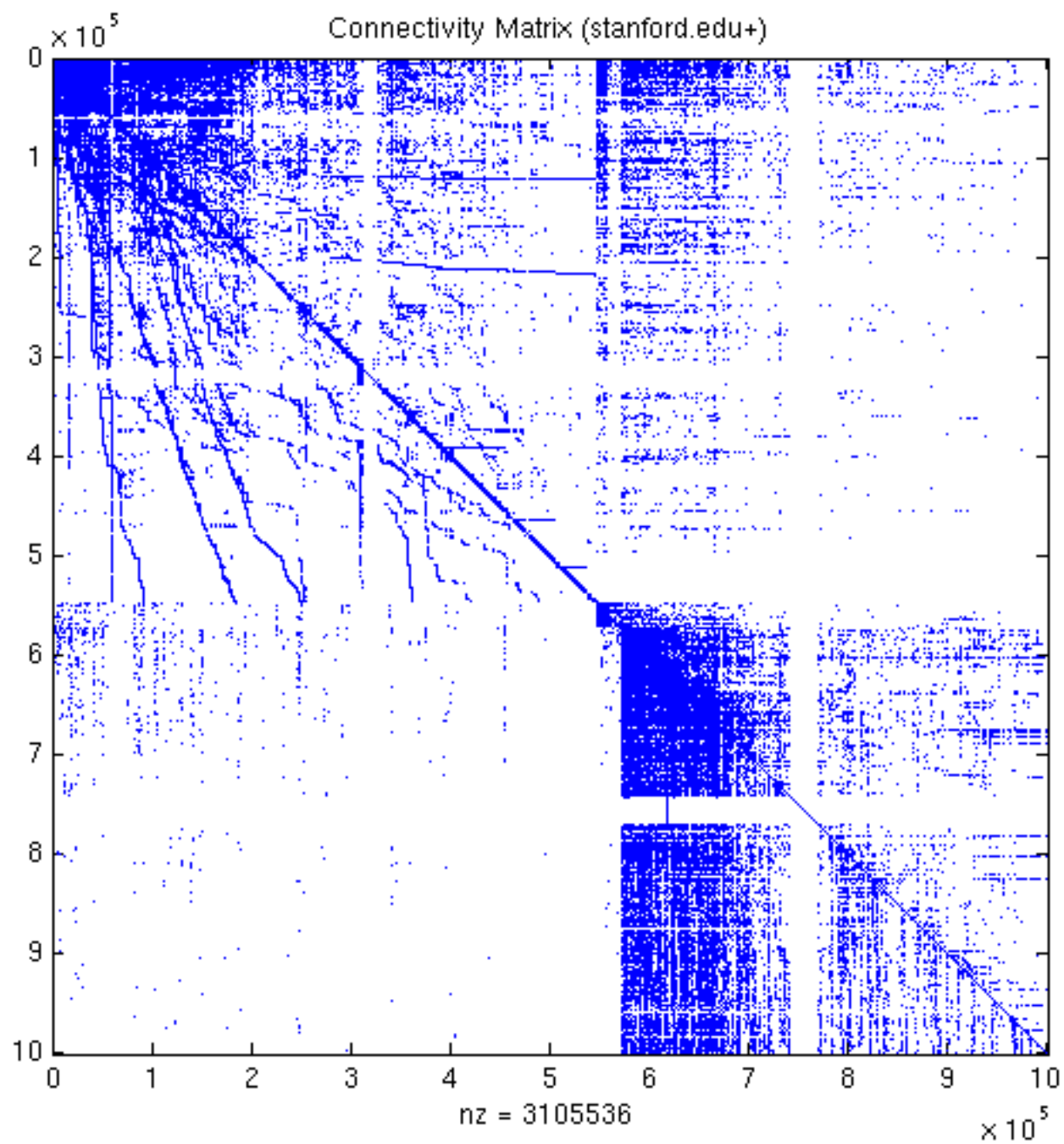
Sparse Matrix Multiple-Vector Multiply [itanium2-linux-ecc7]



What about the Google Matrix?

- Google approach
 - Approx. once a month: rank all pages using connectivity structure
 - Find dominant eigenvector of a matrix
 - At query-time: return list of pages ordered by rank
- Matrix: $A = \alpha G + (1-\alpha)(1/n)uu^T$
 - Markov model: Surfer follows link with probability α , jumps to a random page with probability $1-\alpha$
 - G is $n \times n$ connectivity matrix [$n \approx 3$ billion]
 - g_{ij} is non-zero if page i links to page j
 - Normalized so each column sums to 1
 - Very sparse: about 7–8 non-zeros per row (power law dist.)
 - u is a vector of all 1 values
 - Steady-state probability x_i of landing on page i is solution to $x = Ax$
- Approximate x by power method: $x = A^k x_0$
 - In practice, $k \approx 25$

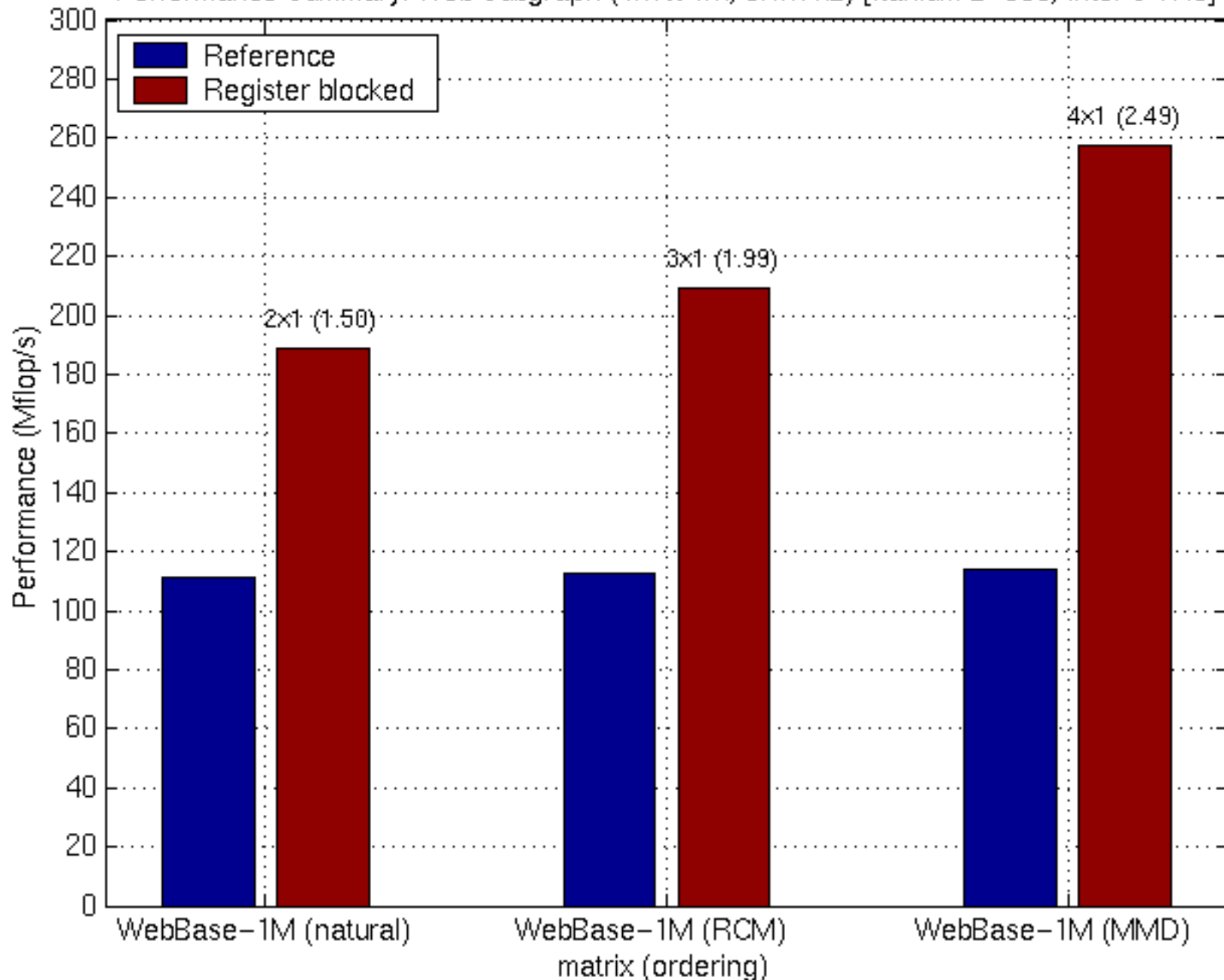
Connectivity Matrix (stanford.edu+)



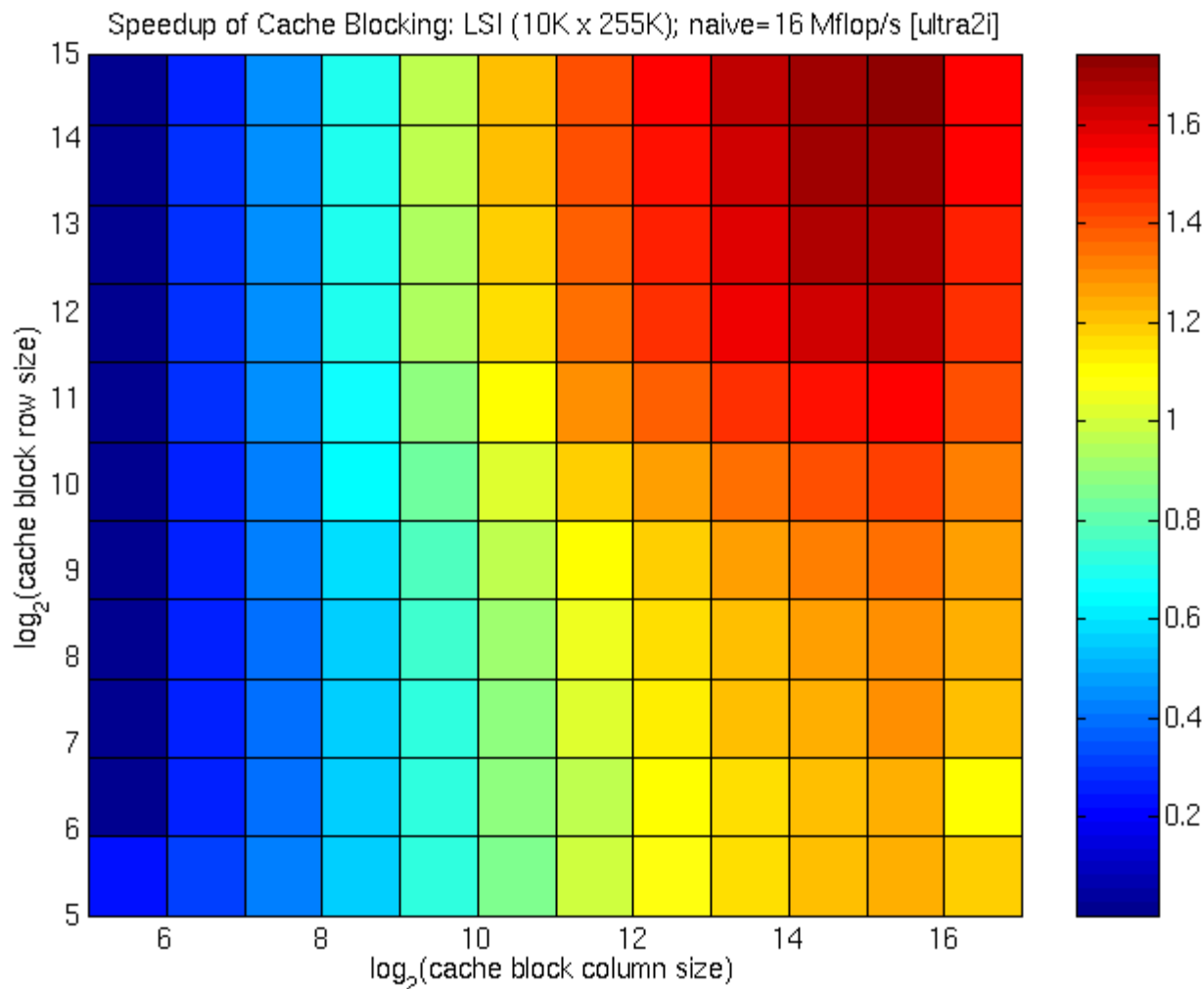
Possible Optimization Techniques

- Within an iteration, *i.e.*, computing $(G+uu^T)^*x$ once
 - Cache block $G*x$
 - On linear programming matrices and matrices with random structure (*e.g.*, LSI), 1.5—4x speedups
 - Best block size is matrix and machine dependent
 - Reordering and/or splitting of G to separate dense structure (rows, columns, blocks)
- Between iterations, *e.g.*, $(G+uu^T)^2x$
 - $(G+uu^T)^2x = G^2x + (Gu)u^Tx + u(u^TG)x + u(u^Tu)u^Tx$
 - Compute Gu , u^TG , u^Tu once for all iterations
 - G^2x : Inter-iteration tiling to read G only once

Performance Summary: Web Subgraph (1M x 1M, 3.1M nz) [Itanium 2-900, Intel C v7.0]



Cache Blocked SpMV on LSI Matrix: Ultra 2i



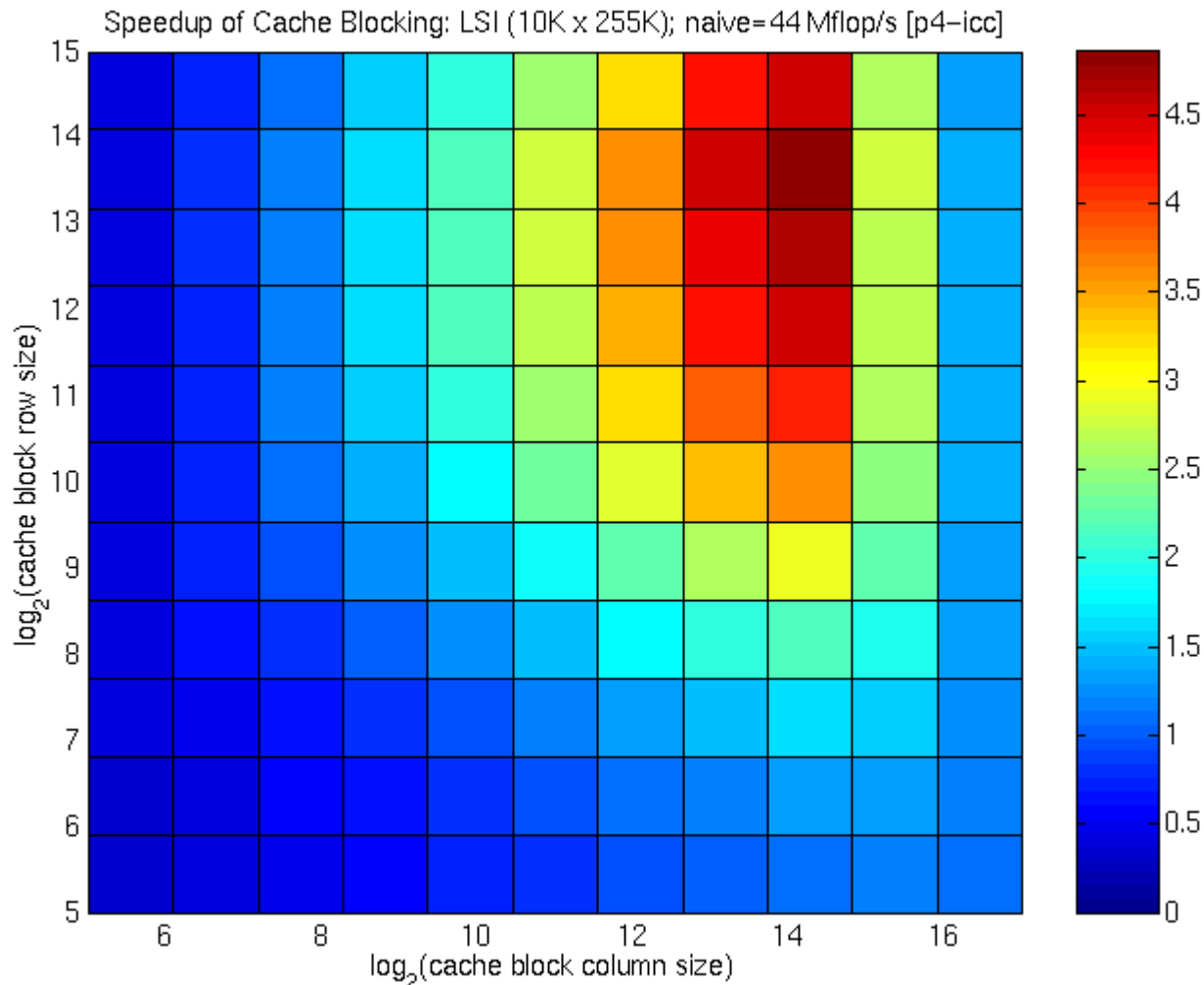
A

10k x 255k
3.7M non-zeros

Baseline:
16 Mflop/s

**Best block size
& performance:**
16k x 64k
28 Mflop/s

Cache Blocking on LSI Matrix: Pentium 4



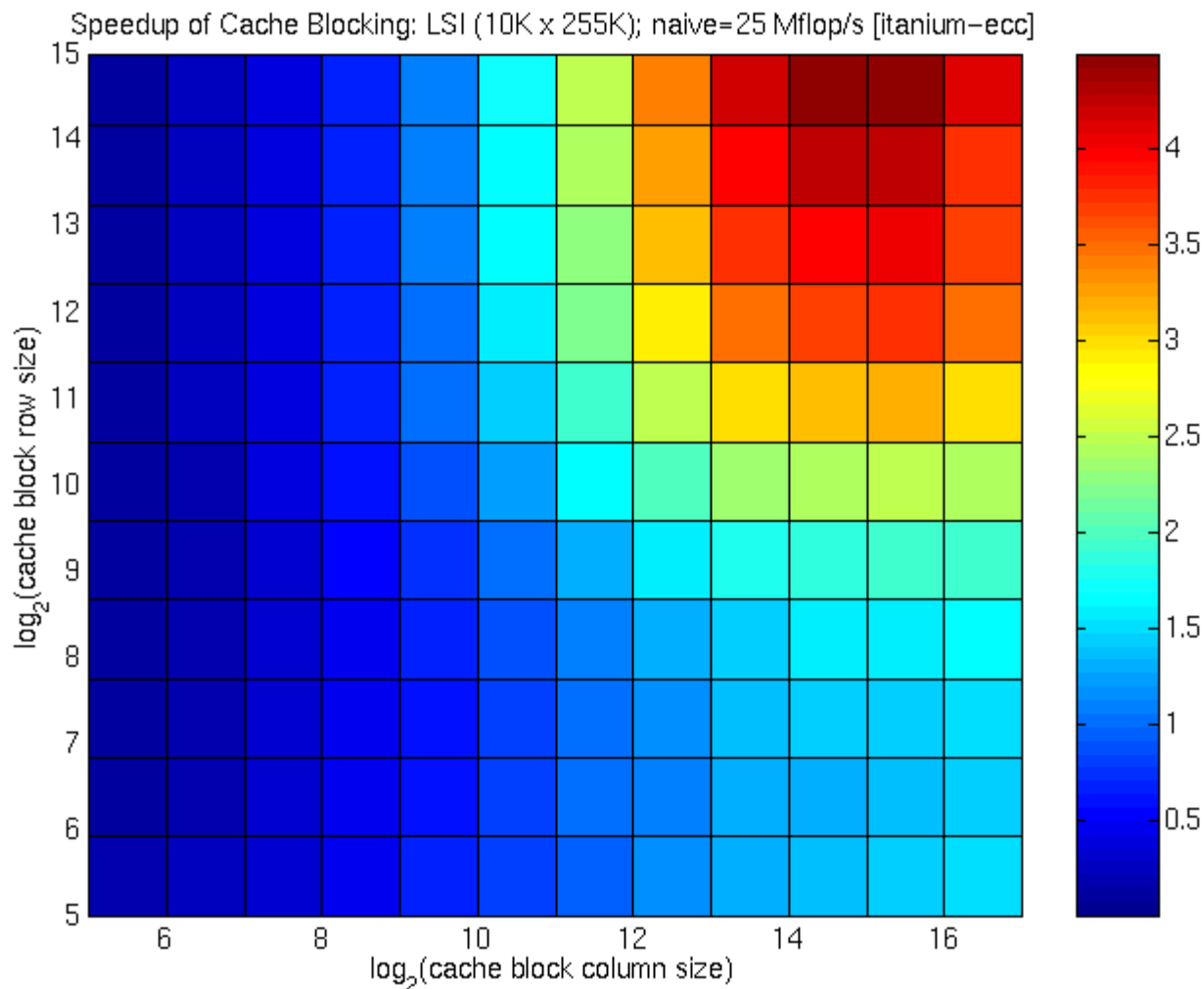
A

10k x 255k
3.7M non-zeros

Baseline:
44 Mflop/s

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

Cache Blocked SpMV on LSI Matrix: Itanium



A

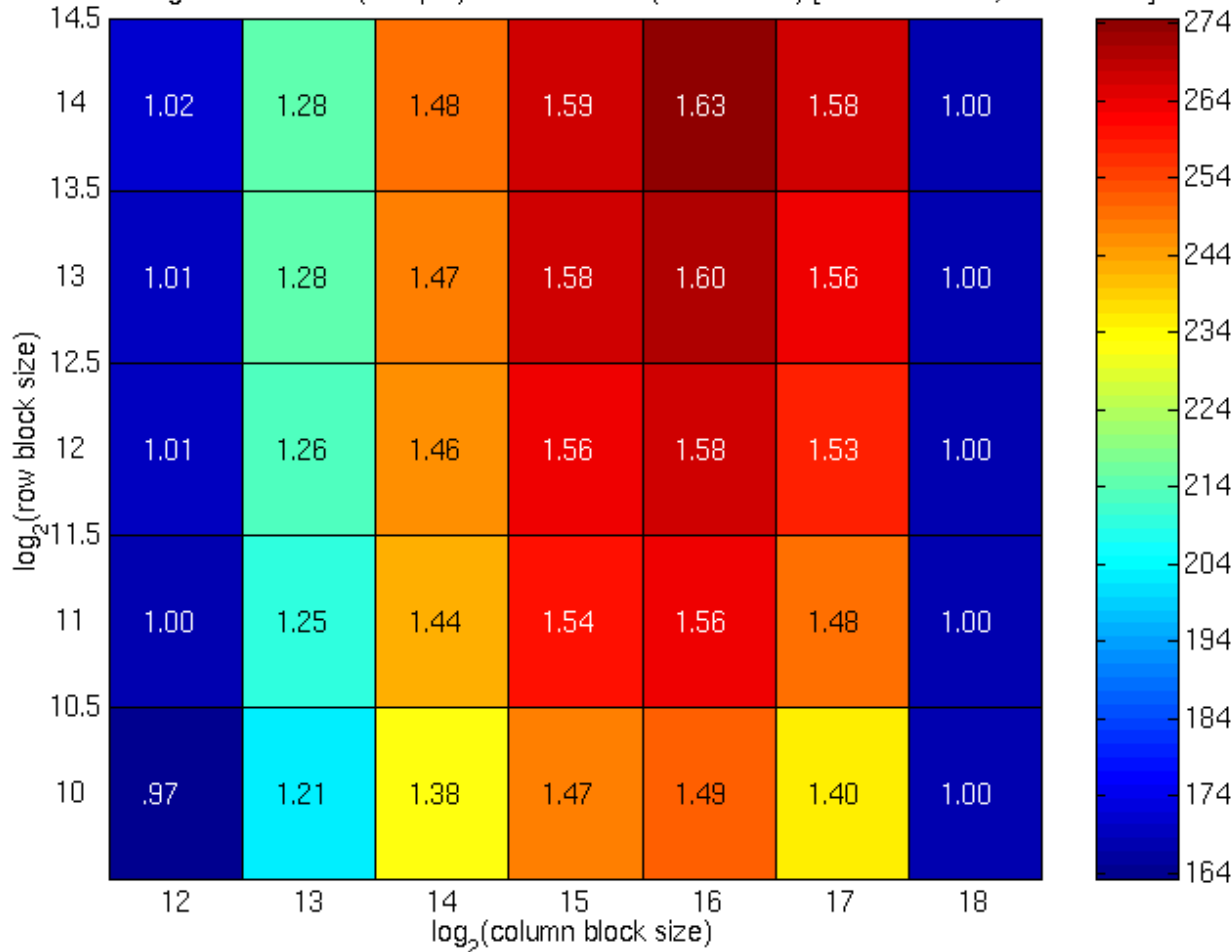
10k x 255k
3.7M non-zeros

Baseline:
25 Mflop/s

**Best block size
& performance:**
16k x 32k
72 Mflop/s

Cache Blocked SpMV on LSI Matrix: Itanium 2

Cache Blocking Performance (Mflop/s) -- LSI Matrix (10k x 255k) [Itanium 2-900, Intel C v7.0]



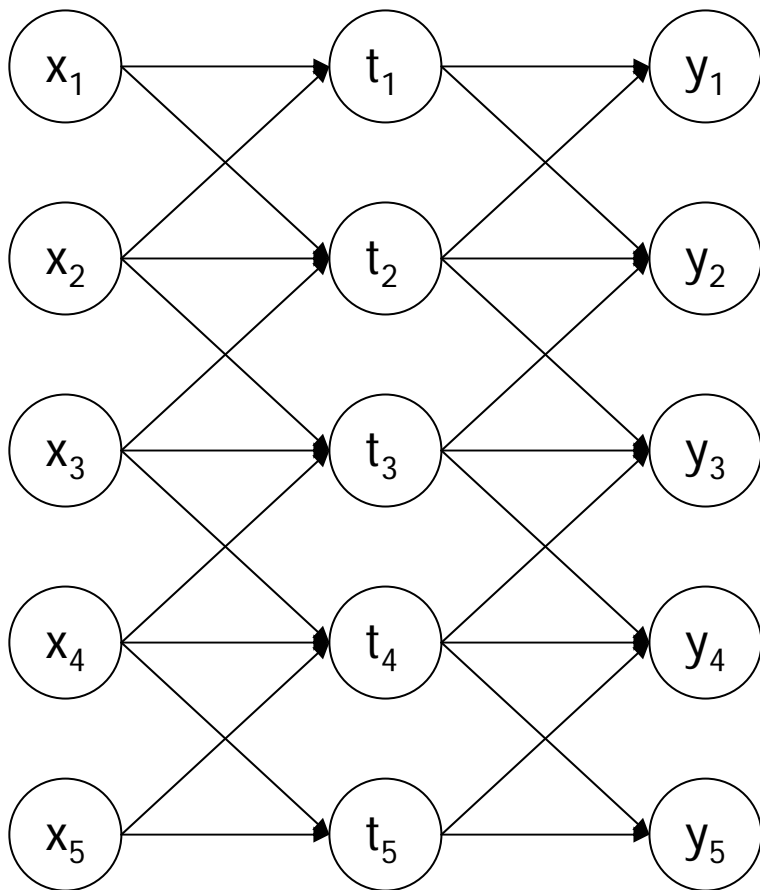
A

10k x 255k
3.7M non-zeros

Baseline:
170 Mflop/s

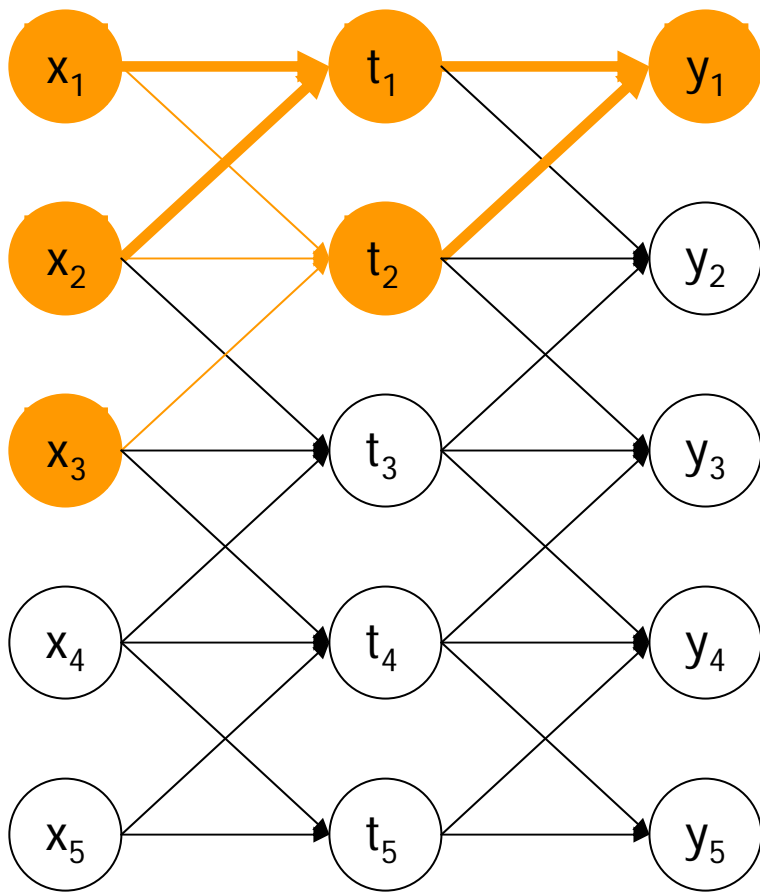
**Best block size
& performance:**
16k x 65k
275 Mflop/s

Inter-Iteration Sparse Tiling (1/3)



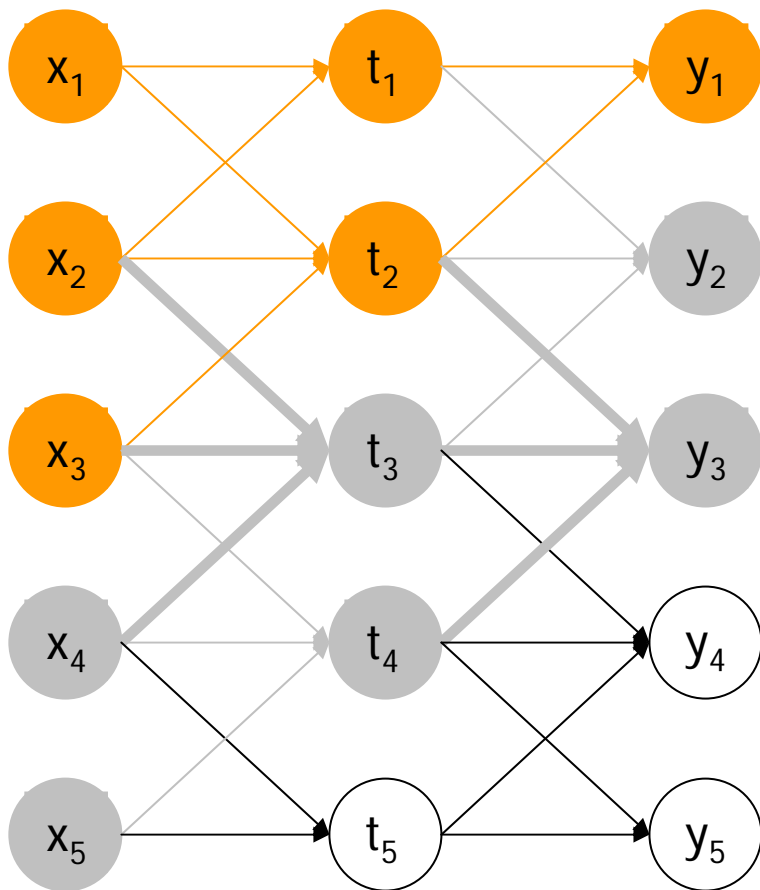
- Let A be 6×6 tridiagonal
- Consider $y = A^2 x$
 - $t = Ax, y = At$
- Nodes: vector elements
- Edges: matrix elements a_{ij}

Inter-Iteration Sparse Tiling (2/3)



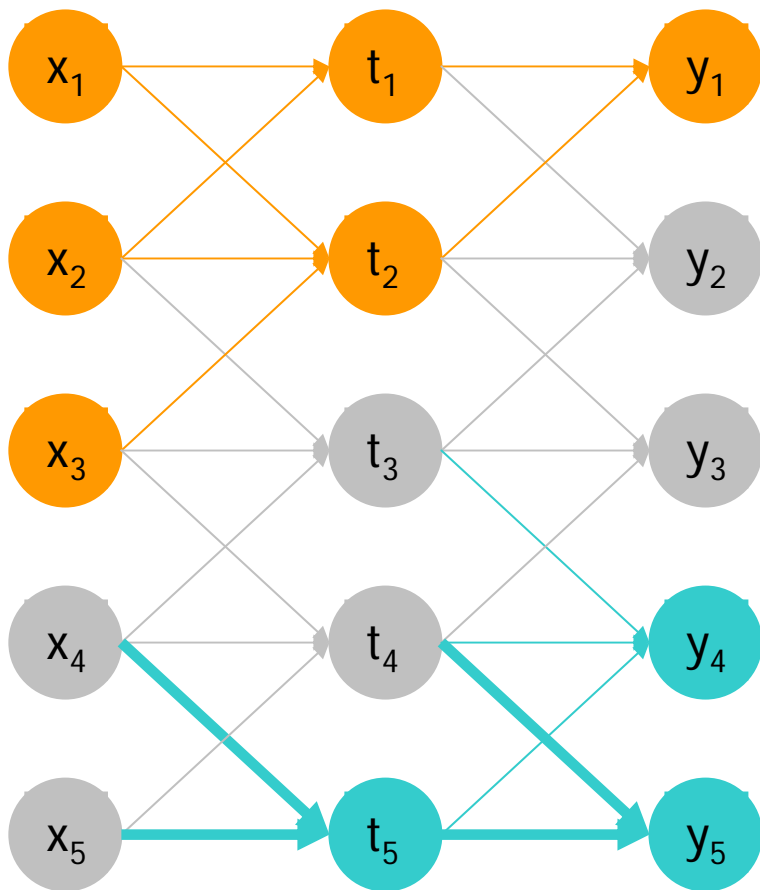
- Let A be 6×6 tridiagonal
- Consider $y = A^2 x$
 - $t = Ax, y = At$
- Nodes: vector elements
- Edges: matrix elements a_{ij}
- Orange = everything needed to compute y_1
 - Reuse a_{11}, a_{12}

Inter-Iteration Sparse Tiling (3/3)



- Let A be 6×6 tridiagonal
- Consider $y = A^2 x$
 - $t = Ax, y = At$
- Nodes: vector elements
- Edges: matrix elements a_{ij}
- Orange = everything needed to compute y_1
 - Reuse a_{11}, a_{12}
- Grey = y_2, y_3
 - Reuse a_{23}, a_{33}, a_{43}

Inter-Iteration Sparse Tiling: Issues



- Tile sizes (colored regions) grow with no. of iterations and increasing out-degree
 - G likely to have a few nodes with high out-degree (e.g., Yahoo)
- Mathematical tricks to limit tile size?
 - Judicious dropping of edges [Ng'01]

Summary and Questions

- Need to understand matrix structure and machine
 - BeBOP: suite of techniques to deal with different sparse structures and architectures
- Google matrix problem
 - Established techniques within an iteration
 - Ideas for inter-iteration optimizations
 - Mathematical structure of problem may help
- Questions
 - Structure of G ?
 - What are the computational bottlenecks?
 - Enabling future computations?
 - E.g., topic-sensitive PageRank \rightarrow multiple vector version [Haveliwala '02]
 - See www.cs.berkeley.edu/~richie/bebop/intel/google for more info, including more complete Itanium 2 results.

Extra slides

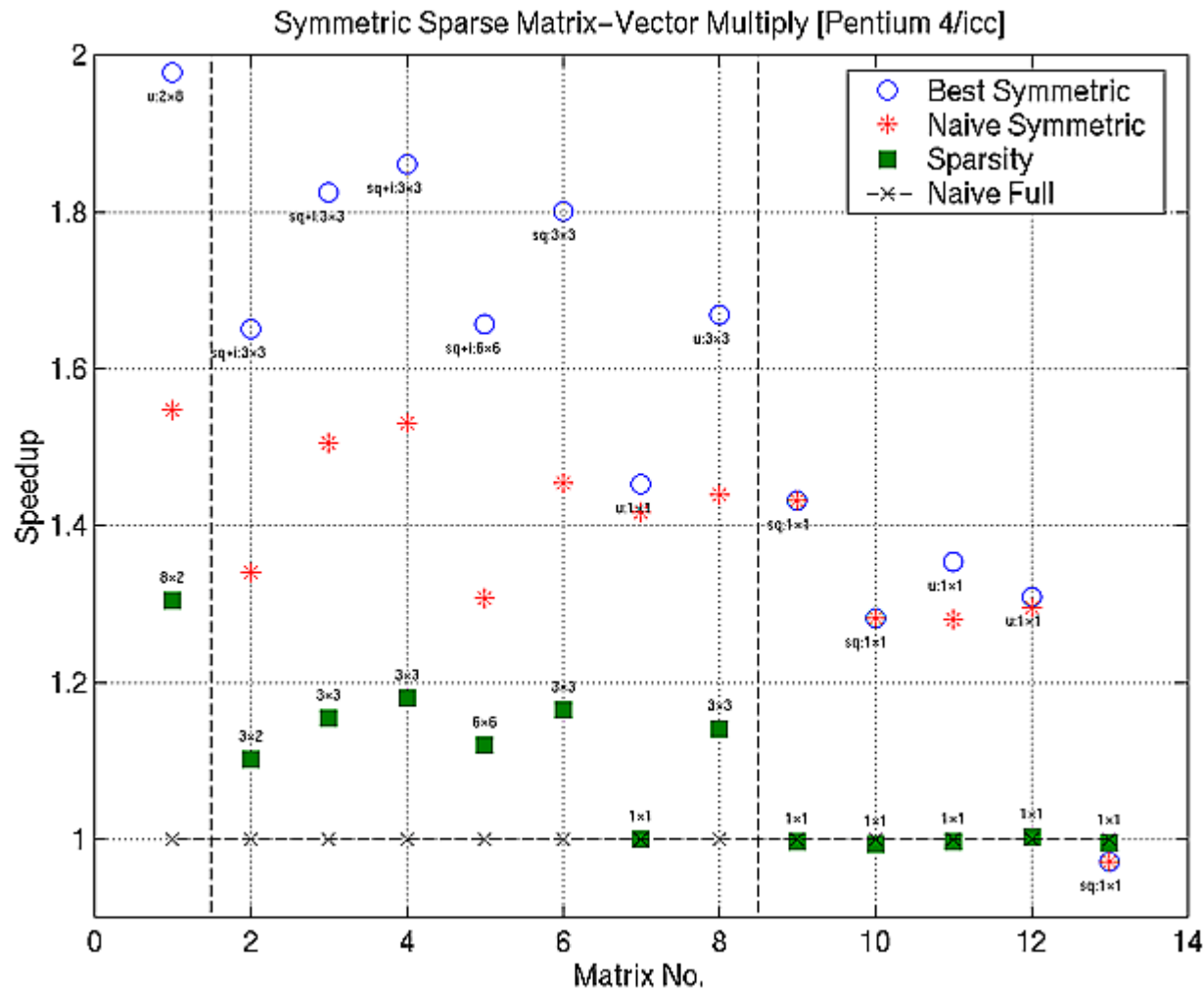
Sparse Kernels and Optimizations

- Kernels
 - **Sparse matrix-vector multiply (SpMV): $y=A * x$**
 - Sparse triangular solve (SpTS): $x=T^{-1} * b$
 - $y=AA^T * x, y=A^T A * x$
 - Powers ($y=A^k * x$), sparse triple-product ($R * A * R^T$), ...
- Optimization techniques (implementation space)
 - Register blocking
 - **Cache blocking**
 - Multiple dense vectors (x)
 - **A has special structure (e.g., symmetric, banded, ...)**
 - **Hybrid data structures (e.g., splitting, switch-to-dense, ...)**
 - Matrix reordering
- How and when do we search?
 - Off-line: Benchmark implementations
 - Run-time: Estimate matrix properties, evaluate performance models based on benchmark data

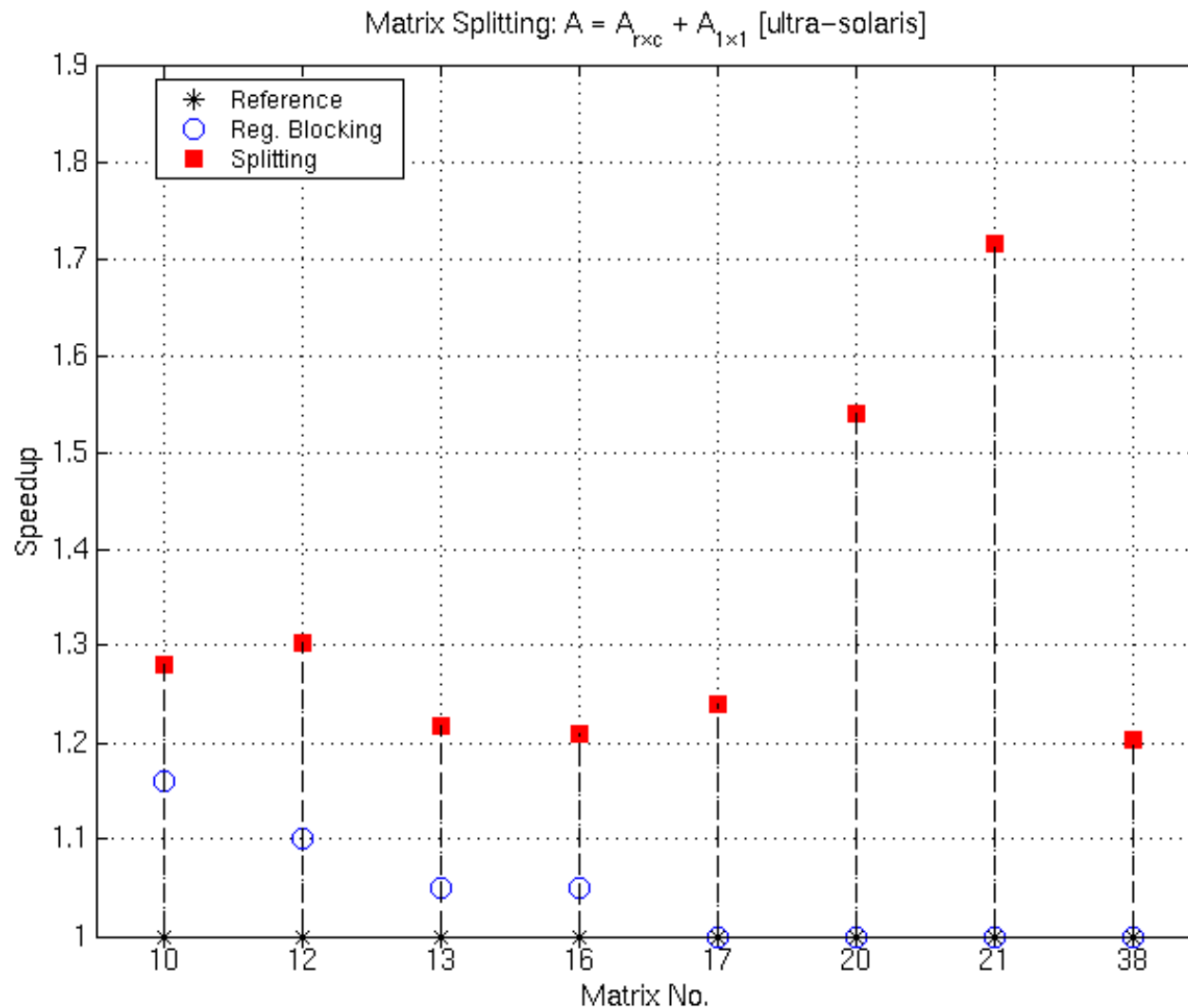
Exploiting Matrix Structure

- Symmetry (numerical or structural)
 - Reuse matrix entries
 - Can combine with register blocking, multiple vectors, ...
- Matrix splitting
 - Split the matrix, *e.g.*, into $r \times c$ and 1×1
 - No fill overhead
- Large matrices with random structure
 - *E.g.*, Latent Semantic Indexing (LSI) matrices
 - Technique: *cache blocking*
 - Store matrix as $2^i \times 2^j$ sparse submatrices
 - Effective when x vector is large
 - Currently, search to find fastest size

Symmetric SpMV Performance: Pentium 4

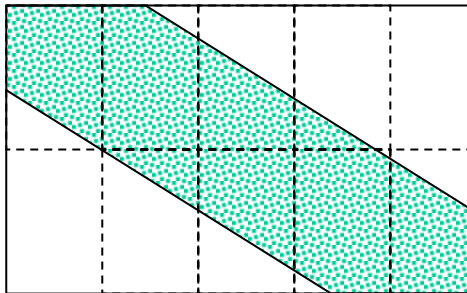


SpMV with Split Matrices: Ultra 2i

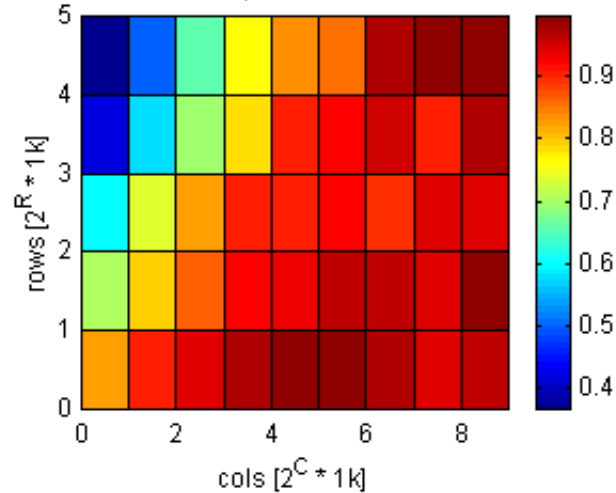


Cache Blocking on Random Matrices: Itanium

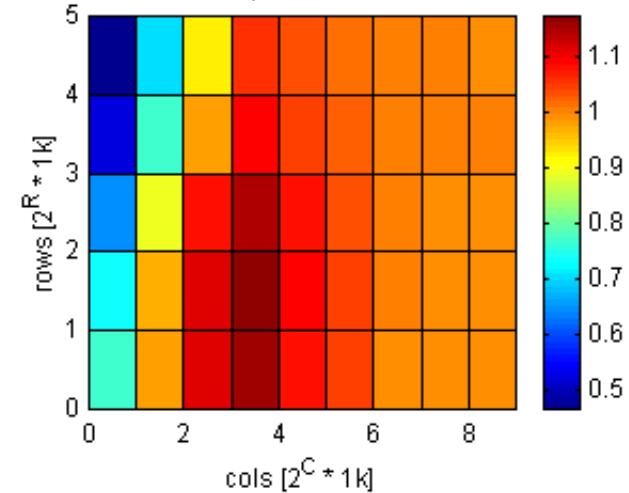
Speedup on four banded random matrices.



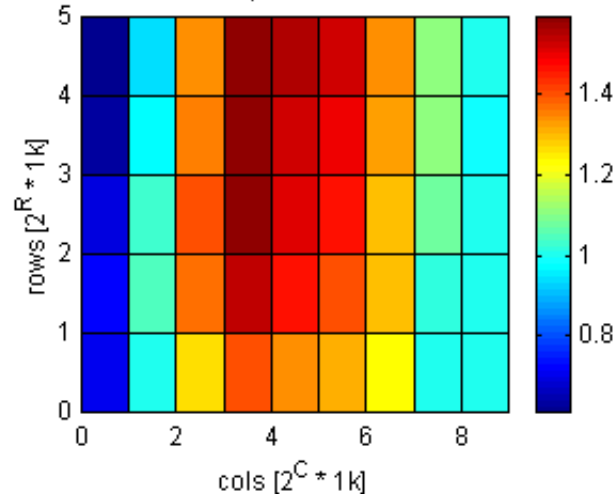
Z=0.015936 ; ref= 172.4 MFLOPS



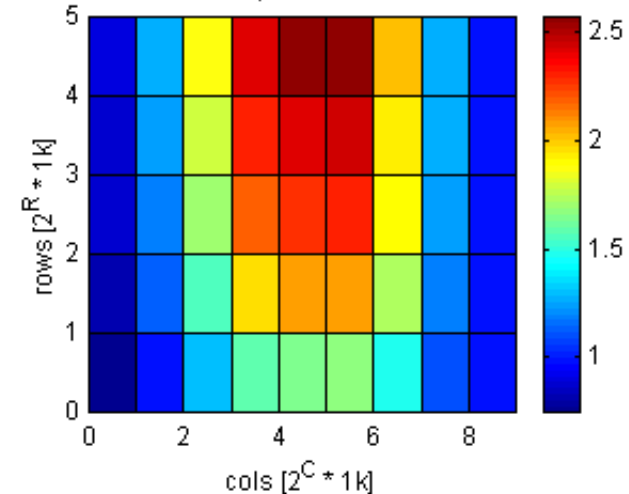
Z=0.126777 ; ref= 127.2 MFLOPS



Z=0.360000 ; ref= 79.6 MFLOPS



Z=1.000000 ; ref= 43.5 MFLOPS

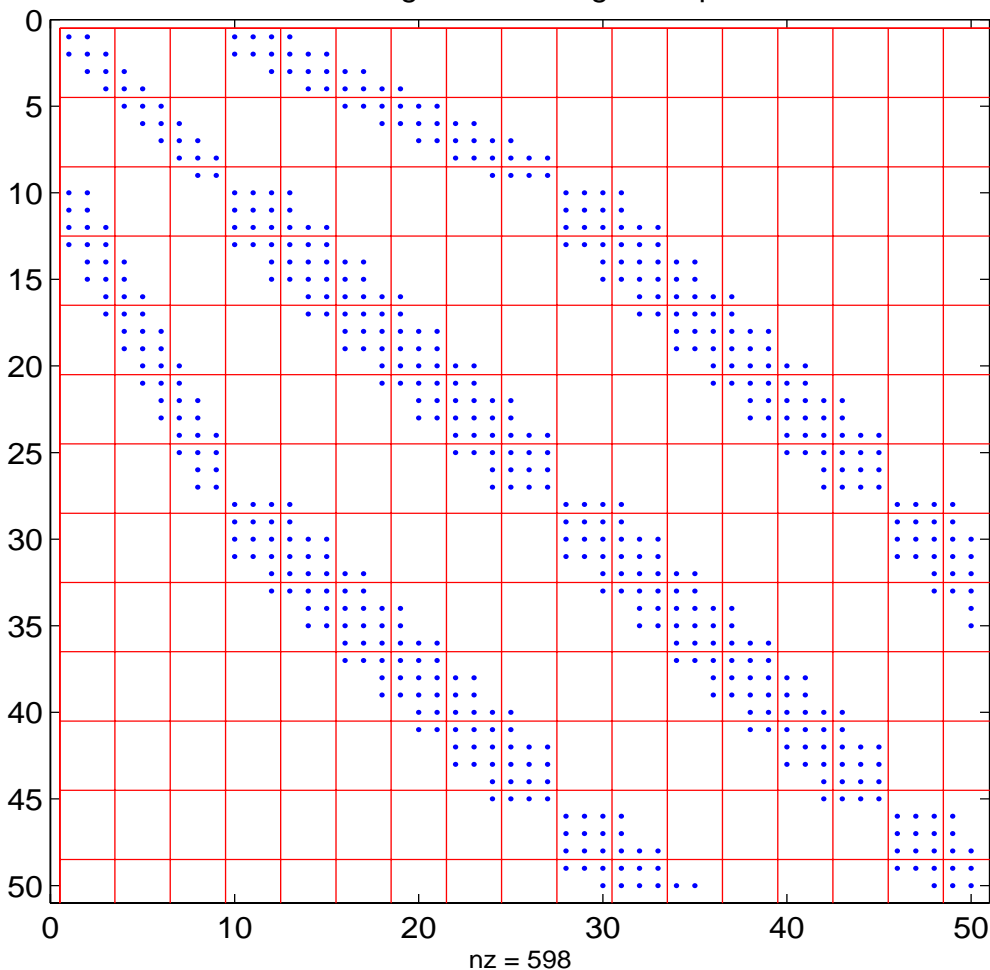


Sparse Kernels and Optimizations

- Kernels
 - **Sparse matrix-vector multiply (SpMV):** $y = A * x$
 - Sparse triangular solve (SpTS): $x = T^{-1} * b$
 - $y = AA^T * x$, $y = A^T A * x$
 - Powers ($y = A^k * x$), sparse triple-product ($R * A * R^T$), ...
- Optimization techniques (implementation space)
 - **Register blocking**
 - Cache blocking
 - **Multiple dense vectors (x)**
 - A has special structure (*e.g.*, symmetric, banded, ...)
 - Hybrid data structures (*e.g.*, splitting, switch-to-dense, ...)
 - Matrix reordering
- **How and when do we search?**
 - Off-line: Benchmark implementations
 - Run-time: Estimate matrix properties, evaluate performance models based on benchmark data

Example: Register Blocking for SpMV

4x3 Register Blocking Example



- Store dense $r \times c$ blocks
 - Reduces storage overhead and bandwidth requirements
- Fully unroll block multiplies
 - Improves register reuse
- Fill-in explicit zeros: trade-off extra computation for improved efficiency
 - 1.3-2.5x speedups on FEM matrices

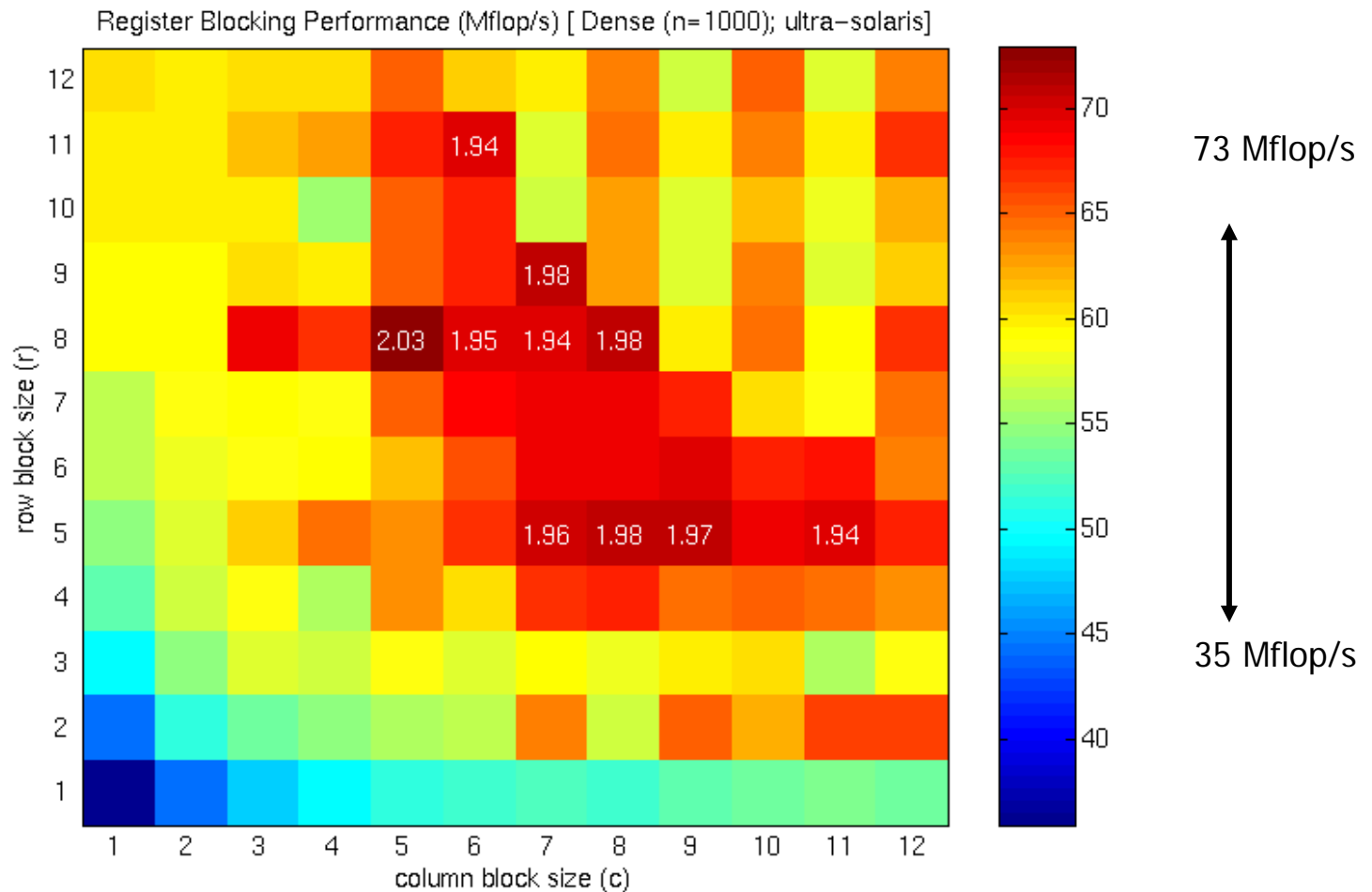
Tuning Sparse Matrix-Vector Multiply (SpMV)

- Sparsity [Im & Yelick '99]
 - Optimizes $y=A*x$ for sparse A , dense x, y
- Selecting the register block size
 - Precompute performance **Mflops** of of dense $A*x$ for various block sizes $r \times c$
 - Given A , sample to estimate **Fill** for each $r \times c$
 - Choose r, c to maximize ratio **Mflops/Fill**
- Multiplication by multiple dense vectors
 - Block across vectors (by vector block size, v)

Off-line Benchmarking: Register Profiles

Register blocking performance for a dense matrix in sparse format.

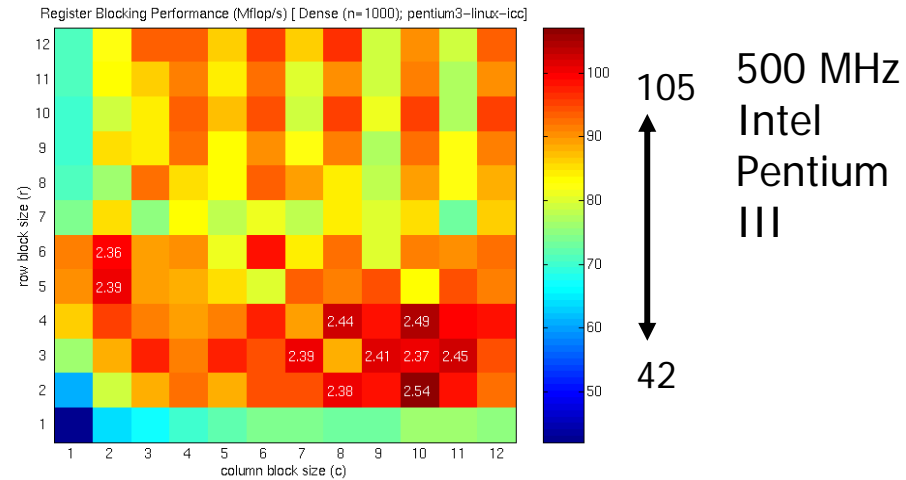
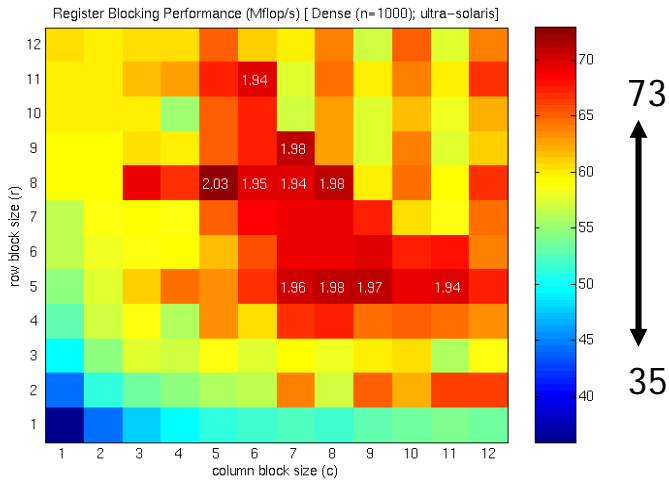
333 MHz
Sun
Ultra 2i



Off-line Benchmarking: Register Profiles

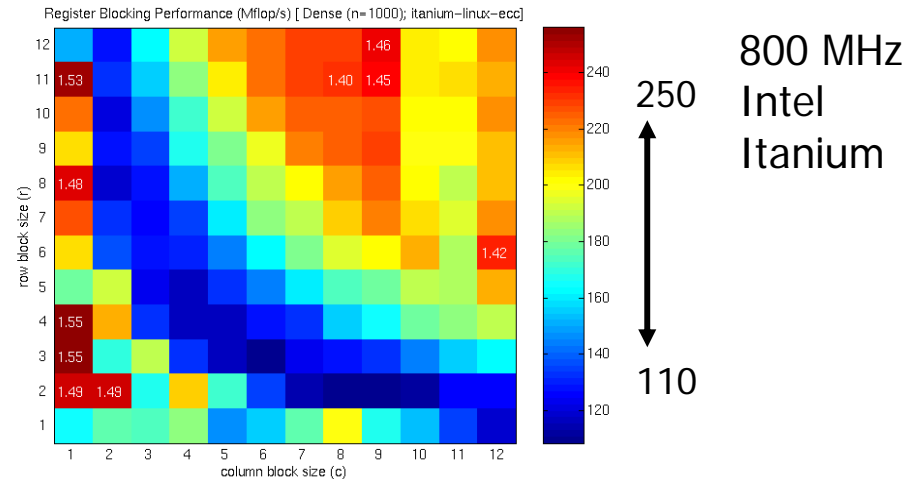
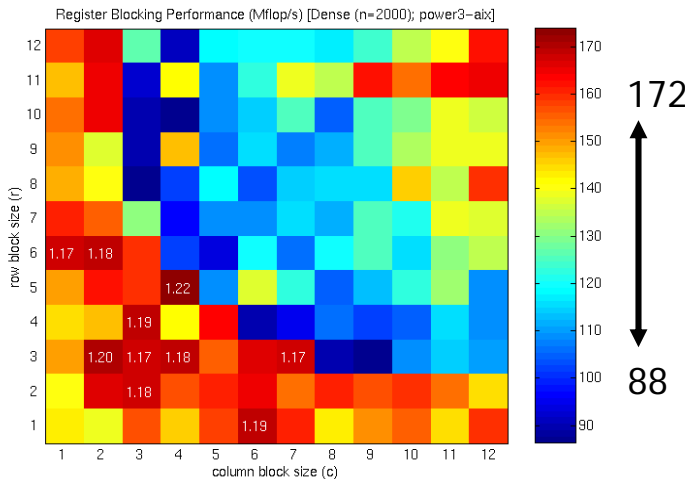
Register blocking performance for a dense matrix in sparse format.

333 MHz
Sun
Ultra 2i



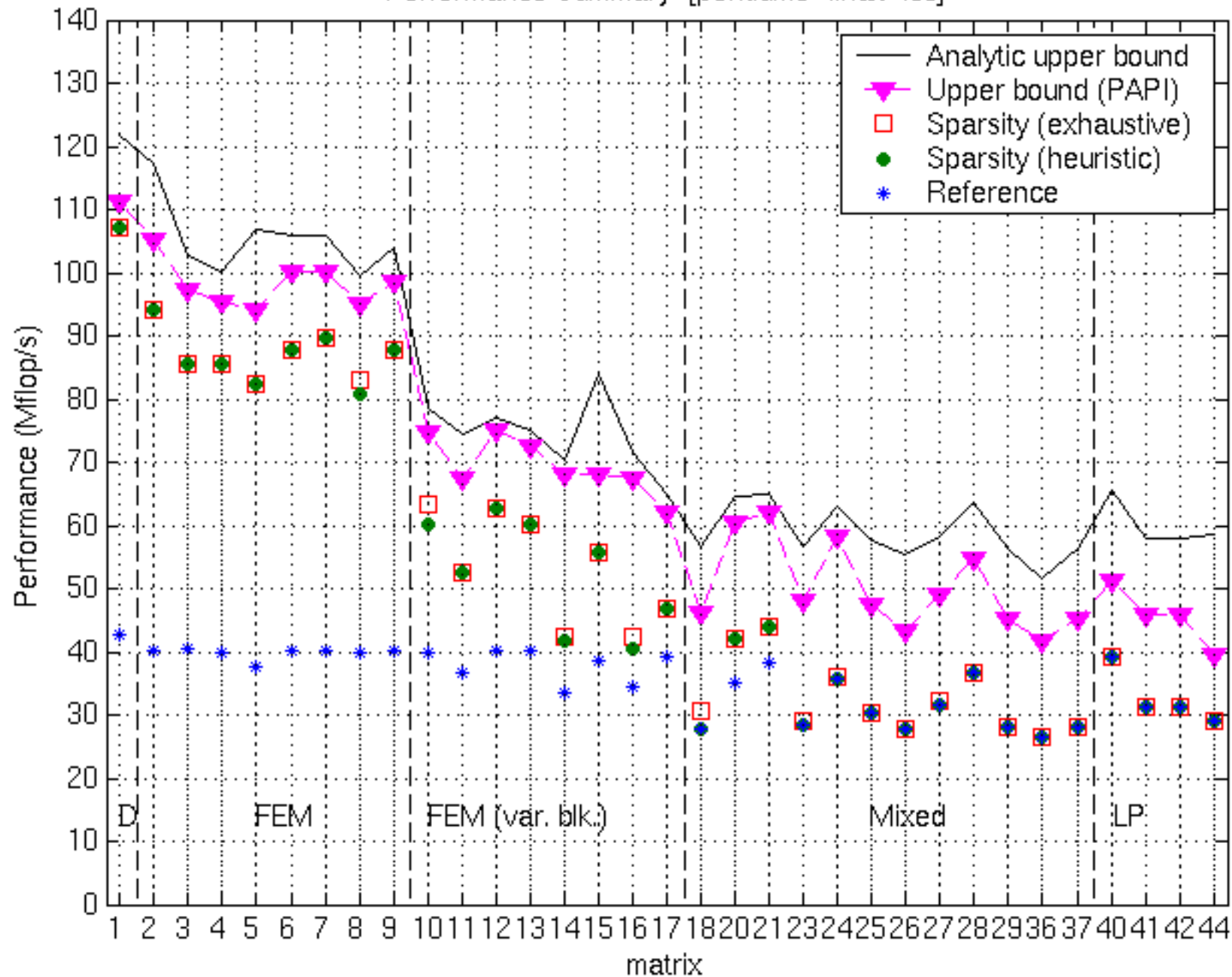
500 MHz
Intel
Pentium
III

375 MHz
IBM
Power3

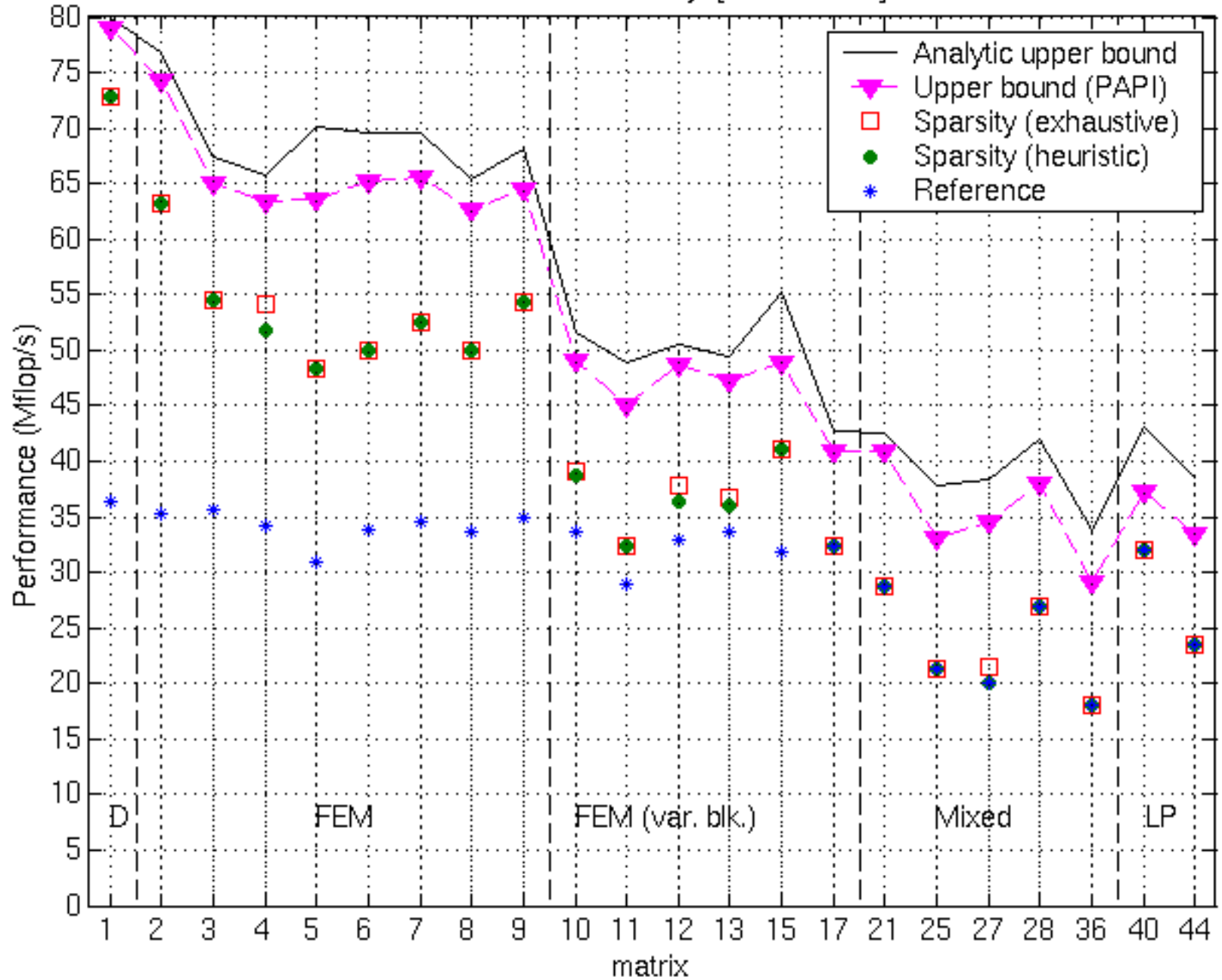


800 MHz
Intel
Itanium

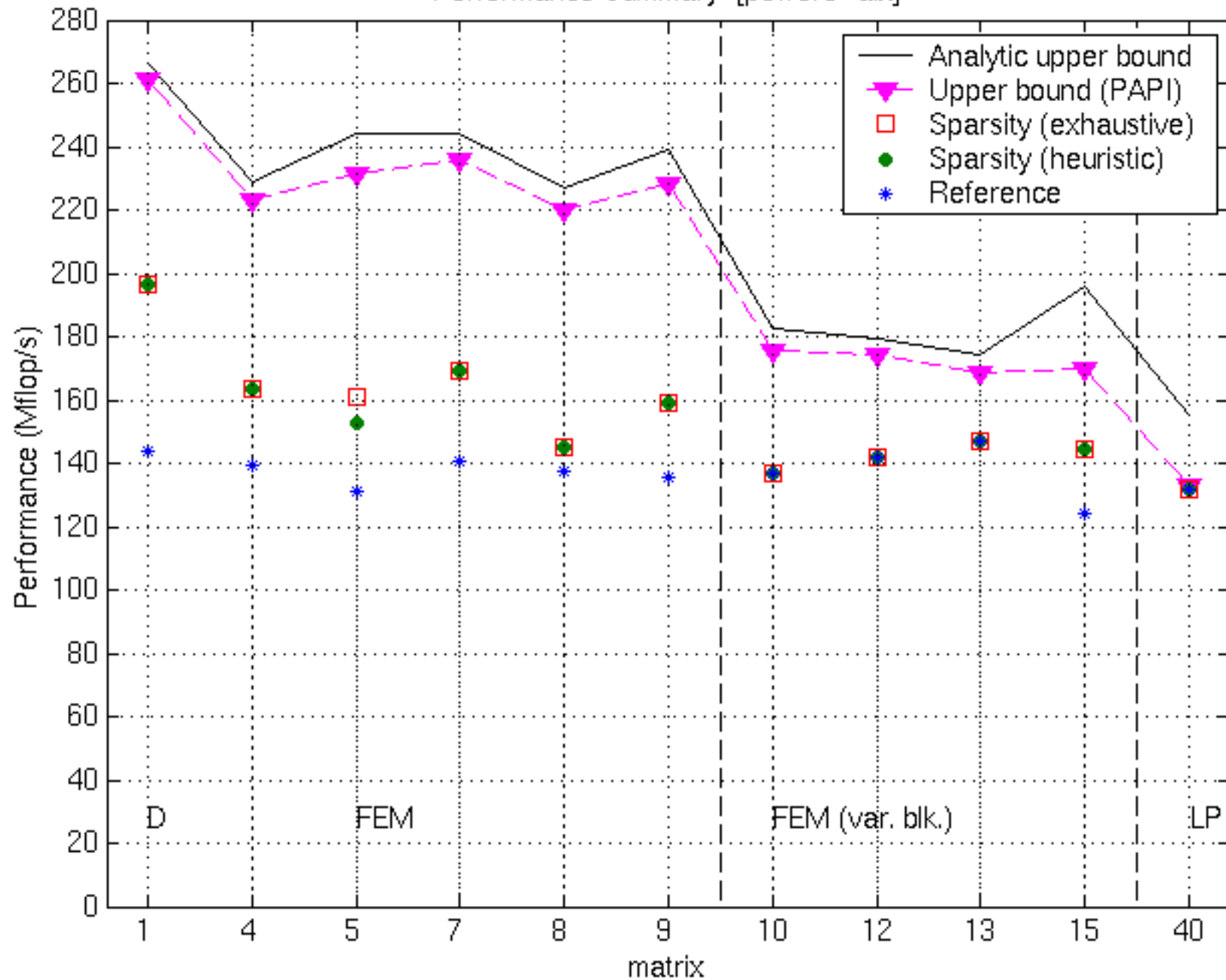
Performance Summary [pentium3-linux-icc]



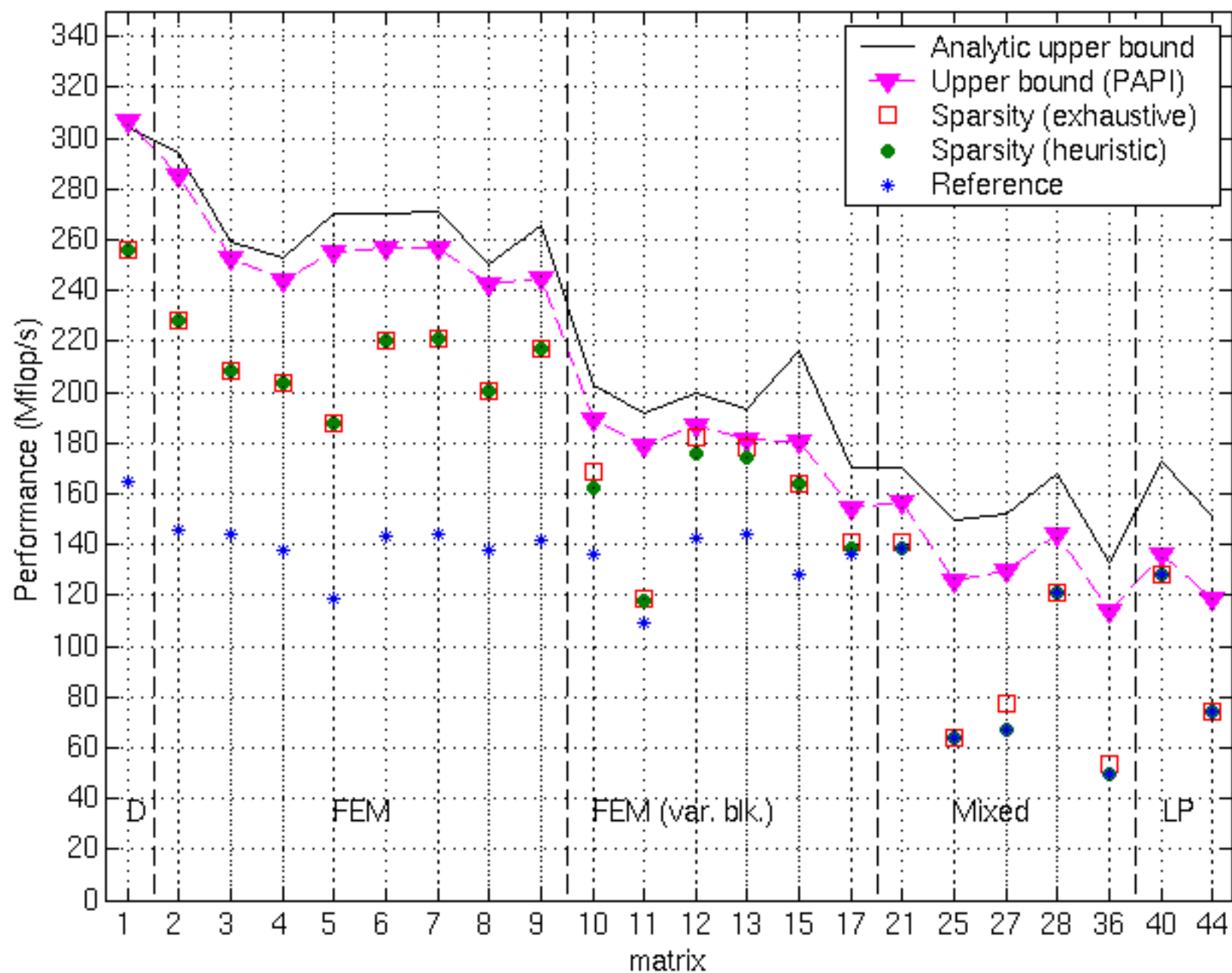
Performance Summary [ultra-solaris]



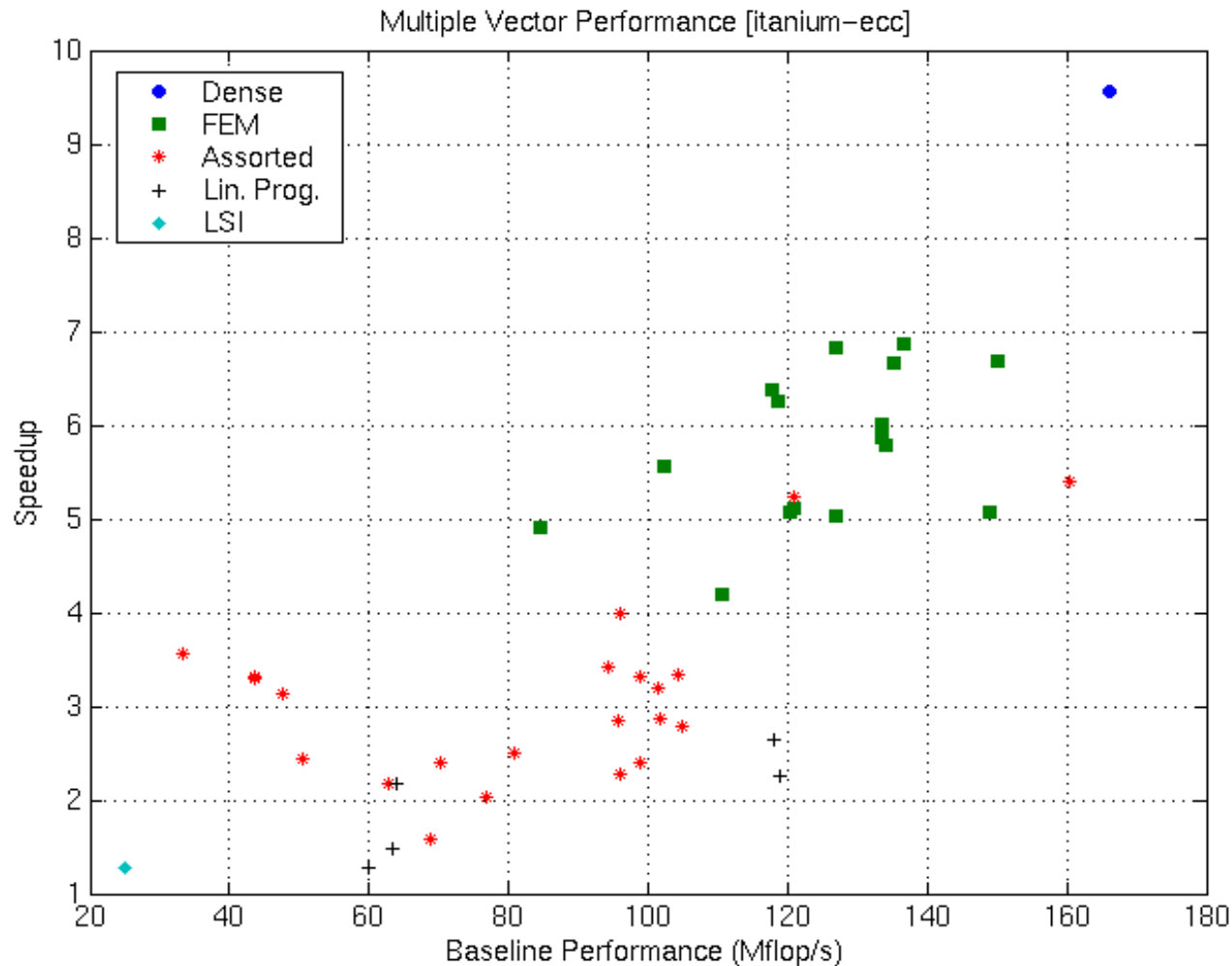
Performance Summary [power3-aix]



Performance Summary [itanium-linux-ecc]



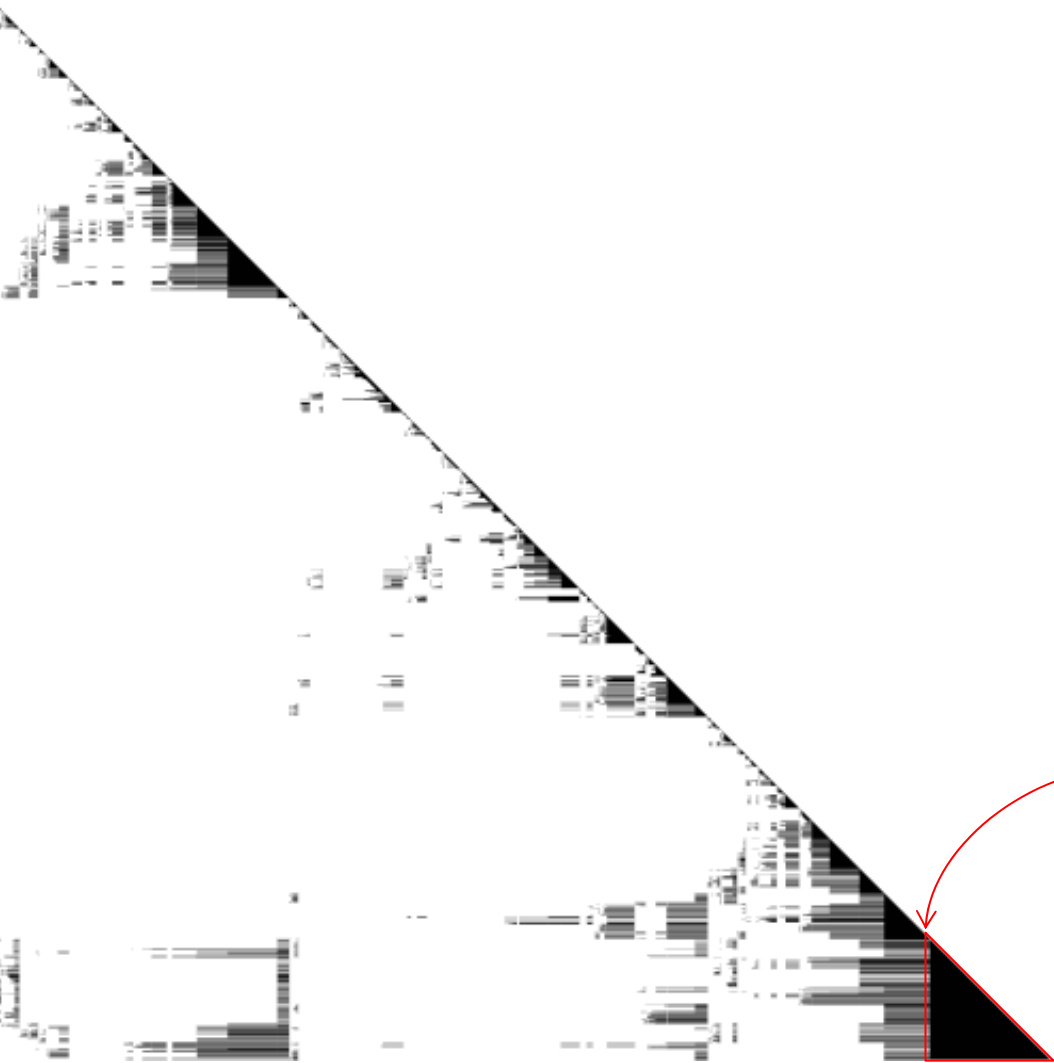
Multiple Vector Performance: Itanium



Sparse Kernels and Optimizations

- Kernels
 - Sparse matrix-vector multiply (SpMV): $y=A * x$
 - **Sparse triangular solve (SpTS): $x=T^{-1} * b$**
 - $y=AA^T * x, y=A^T A * x$
 - Powers ($y=A^k * x$), sparse triple-product ($R * A * R^T$), ...
- Optimization techniques (implementation space)
 - Register blocking
 - Cache blocking
 - Multiple dense vectors (x)
 - A has special structure (*e.g.*, symmetric, banded, ...)
 - **Hybrid data structures (*e.g.*, splitting, switch-to-dense, ...)**
 - Matrix reordering
- How and when do we search?
 - Off-line: Benchmark implementations
 - Run-time: Estimate matrix properties, evaluate performance models based on benchmark data

Example: Sparse Triangular Factor



- Raefsky4 (structural problem) + SuperLU + colmmd
- $N=19779$, $\text{nnz}=12.6 \text{ M}$

Dense trailing triangle:
dim=2268, 20% of
total nz

Tuning Sparse Triangular Solve (SpTS)

- Compute $x=L^{-1} * b$ where L sparse lower triangular, x & b dense
- L from sparse LU has rich dense substructure
 - Dense *trailing triangle* can account for 20—90% of matrix non-zeros
- SpTS optimizations
 - Split into sparse trapezoid and dense trailing triangle
 - Use tuned dense BLAS (DTRSV) on dense triangle
 - Use Sparsity register blocking on sparse part
- Tuning parameters
 - Size of dense trailing triangle
 - Register block size

Sparse/Dense Partitioning for SpTS

- Partition L into sparse (L_1, L_2) and dense L_D :

$$\begin{pmatrix} L_1 & \\ L_2 & L_D \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

- Perform SpTS in three steps:

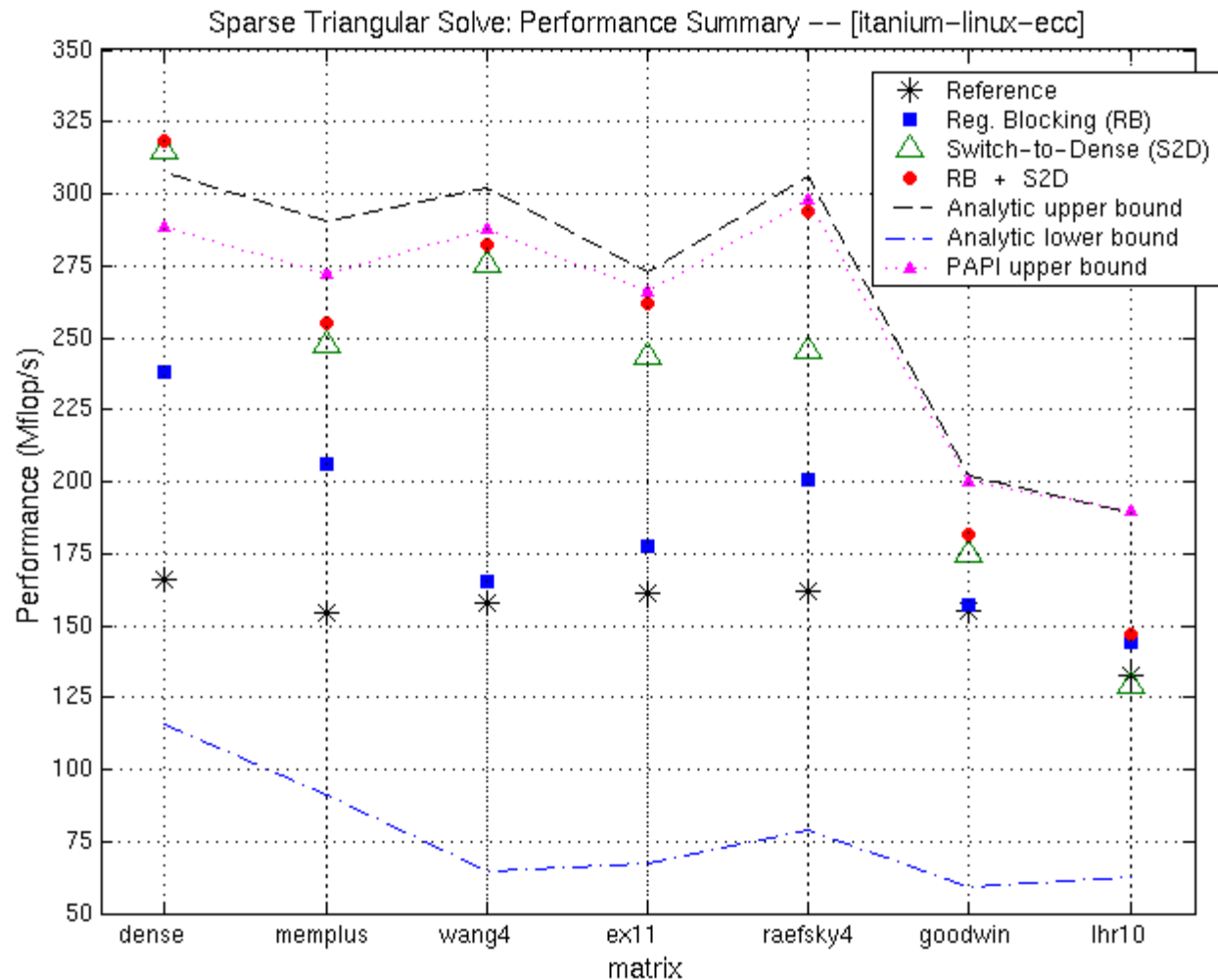
$$(1) \quad L_1 x_1 = b_1$$

$$(2) \quad \hat{b}_2 = b_2 - L_2 x_1$$

$$(3) \quad L_D x_2 = \hat{b}_2$$

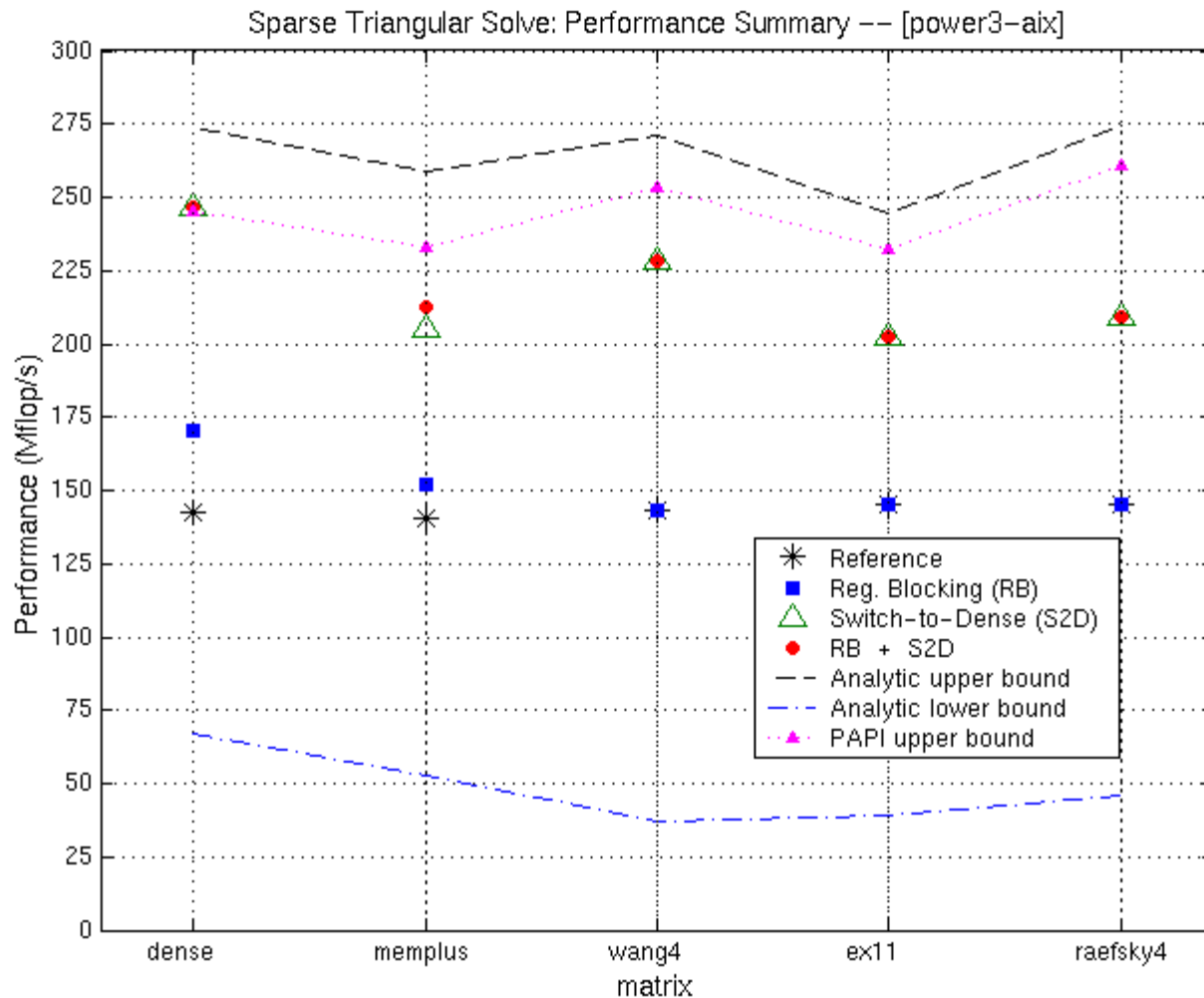
- Sparsity optimizations for (1)—(2); DTRSV for (3)

SpTS Performance: Itanium



(See POHLL '02 workshop paper, at ICS '02.)

SpTS Performance: Power3



Sparse Kernels and Optimizations

- Kernels
 - Sparse matrix-vector multiply (SpMV): $y=A * x$
 - Sparse triangular solve (SpTS): $x=T^{-1} * b$
 - $y=AA^T * x, y=A^T A * x$
 - Powers ($y=A^k * x$), sparse triple-product ($R * A * R^T$), ...
- Optimization techniques (implementation space)
 - **Register blocking**
 - Cache blocking
 - Multiple dense vectors (x)
 - A has special structure (*e.g.*, symmetric, banded, ...)
 - Hybrid data structures (*e.g.*, splitting, switch-to-dense, ...)
 - Matrix reordering
- How and when do we search?
 - Off-line: Benchmark implementations
 - Run-time: Estimate matrix properties, evaluate performance models based on benchmark data

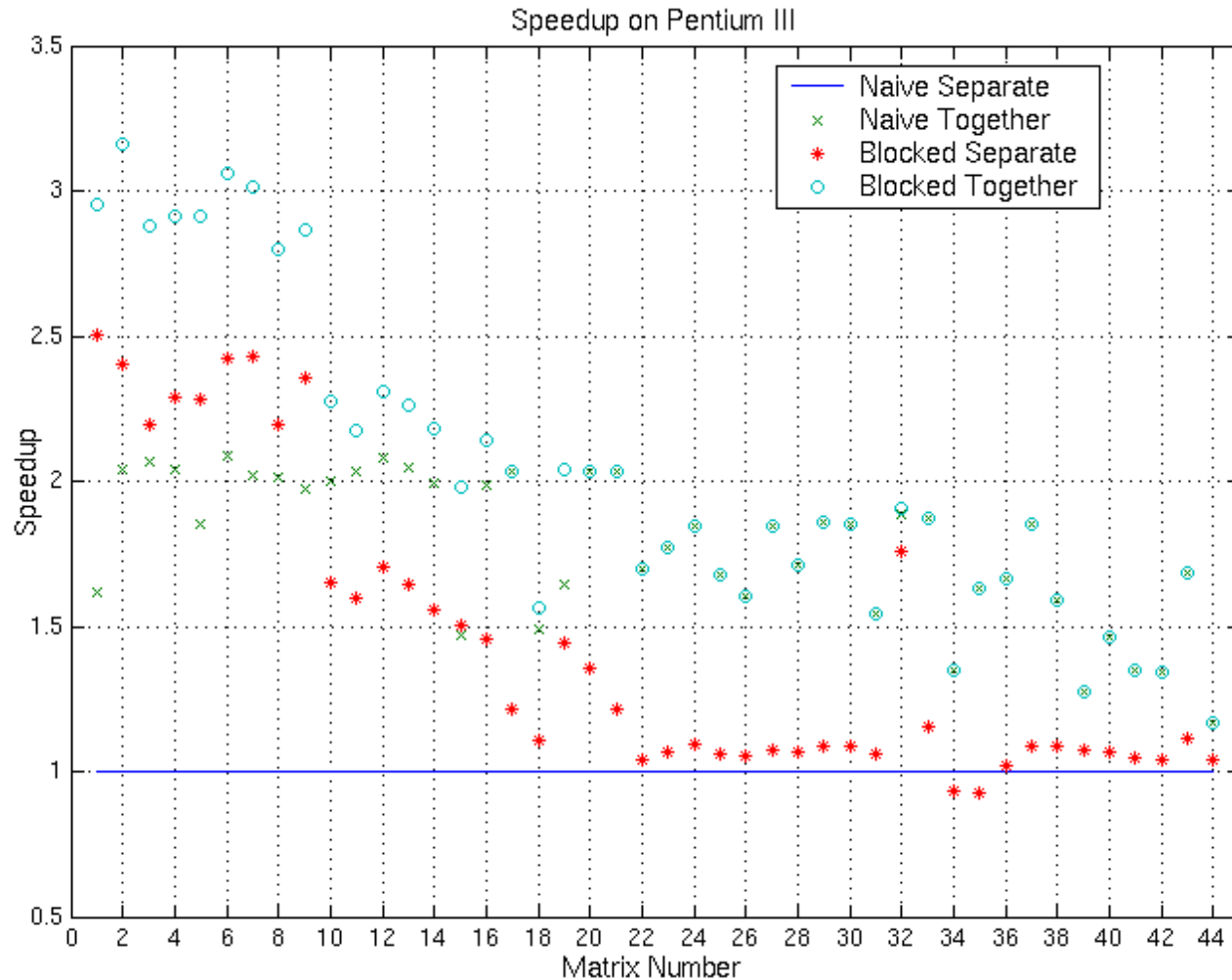
Optimizing $AA^T * x$

- Kernel: $y=AA^T * x$, where A is sparse, x & y dense
 - Arises in linear programming, computation of SVD
 - Conventional implementation: compute $z=A^T * x$, $y=A * z$
- Elements of A can be reused:

$$y = (a_1 \cdots a_n) \begin{pmatrix} a_1^T \\ \vdots \\ a_n^T \end{pmatrix} x = \sum_{k=1}^n a_k (a_k^T x)$$

- When a_k represent blocks of columns, can apply register blocking.

Optimized $AA^T * x$ Performance: Pentium III



Current Directions

- Applying new optimizations
 - Other split data structures (variable block, diagonal, ...)
 - Matrix reordering to create block structure
 - Structural symmetry
- New kernels (triple product RAR^T , powers A^k , ...)
- Tuning parameter selection
- Building an automatically tuned sparse matrix library
 - Extending the Sparse BLAS
 - Leverage existing sparse compilers as code generation infrastructure
 - More thoughts on this topic tomorrow

Related Work

- Automatic performance tuning systems
 - PHiPAC [Bilmes, *et al.*, '97], ATLAS [Whaley & Dongarra '98]
 - FFTW [Frigo & Johnson '98], SPIRAL [Pueschel, *et al.*, '00], UHFFT [Mirkovic and Johnsson '00]
 - MPI collective operations [Vadhiyar & Dongarra '01]
- Code generation
 - FLAME [Gunnels & van de Geijn, '01]
 - Sparse compilers: [Bik '99], Bernoulli [Pingali, *et al.*, '97]
 - Generic programming: Blitz++ [Veldhuizen '98], MTL [Siek & Lumsdaine '98], GMCL [Czarnecki, *et al.* '98], ...
- Sparse performance modeling
 - [Temam & Jalby '92], [White & Saddayappan '97], [Navarro, *et al.*, '96], [Heras, *et al.*, '99], [Fraguela, *et al.*, '99], ...

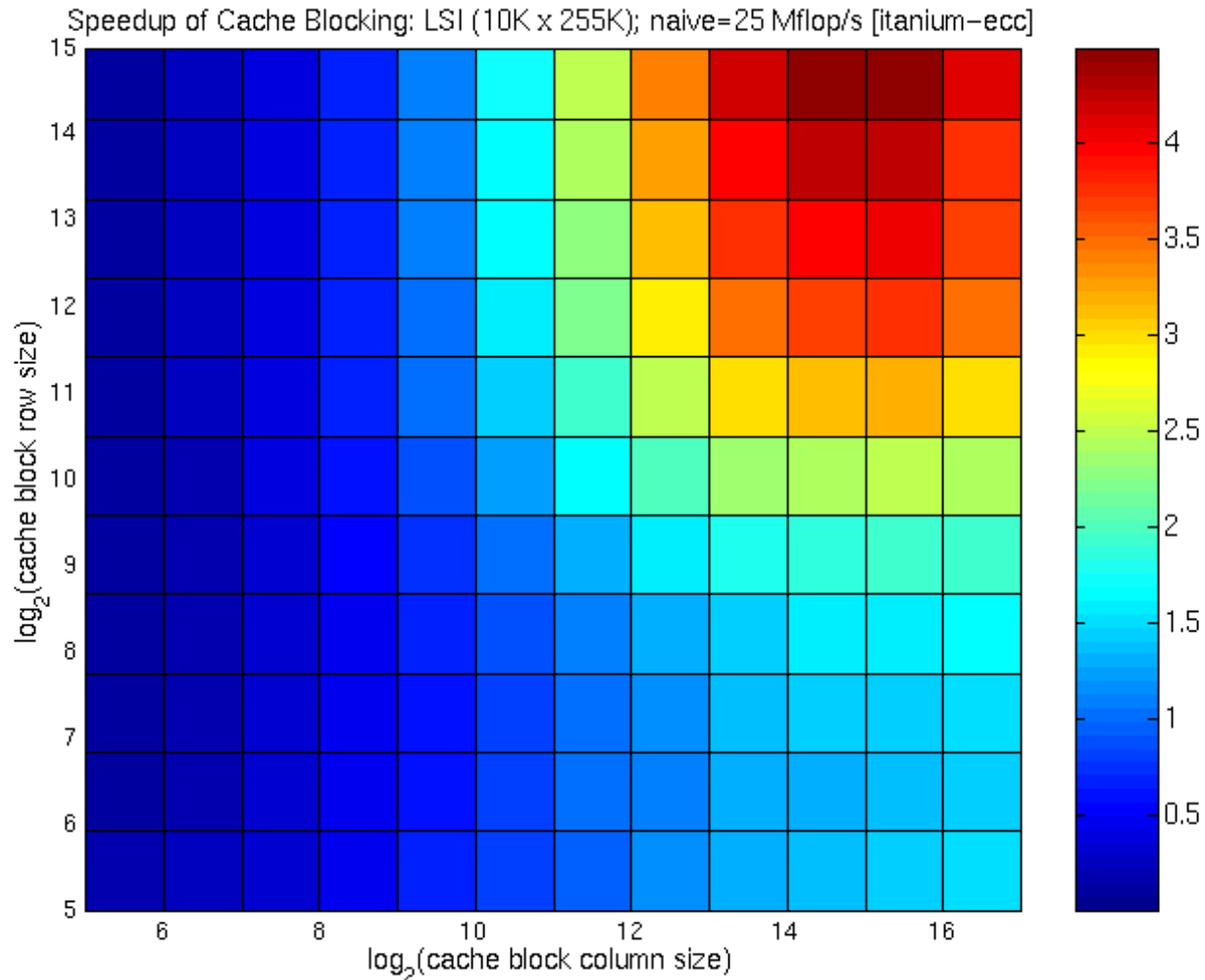
More Related Work

- Compiler analysis, models
 - CROPS [Carter, Ferrante, *et al.*]; Serial sparse tiling [Strout '01]
 - TUNE [Chatterjee, *et al.*]
 - Iterative compilation [O'Boyle, *et al.*, '98]
 - Broadway compiler [Guyer & Lin, '99]
 - [Brewer '95], ADAPT [Voss '00]
- Sparse BLAS interfaces
 - BLAST Forum (Chapter 3)
 - NIST Sparse BLAS [Remington & Pozo '94]; SparseLib++
 - SPARSKIT [Saad '94]
 - Parallel Sparse BLAS [Fillipone, *et al.* '96]

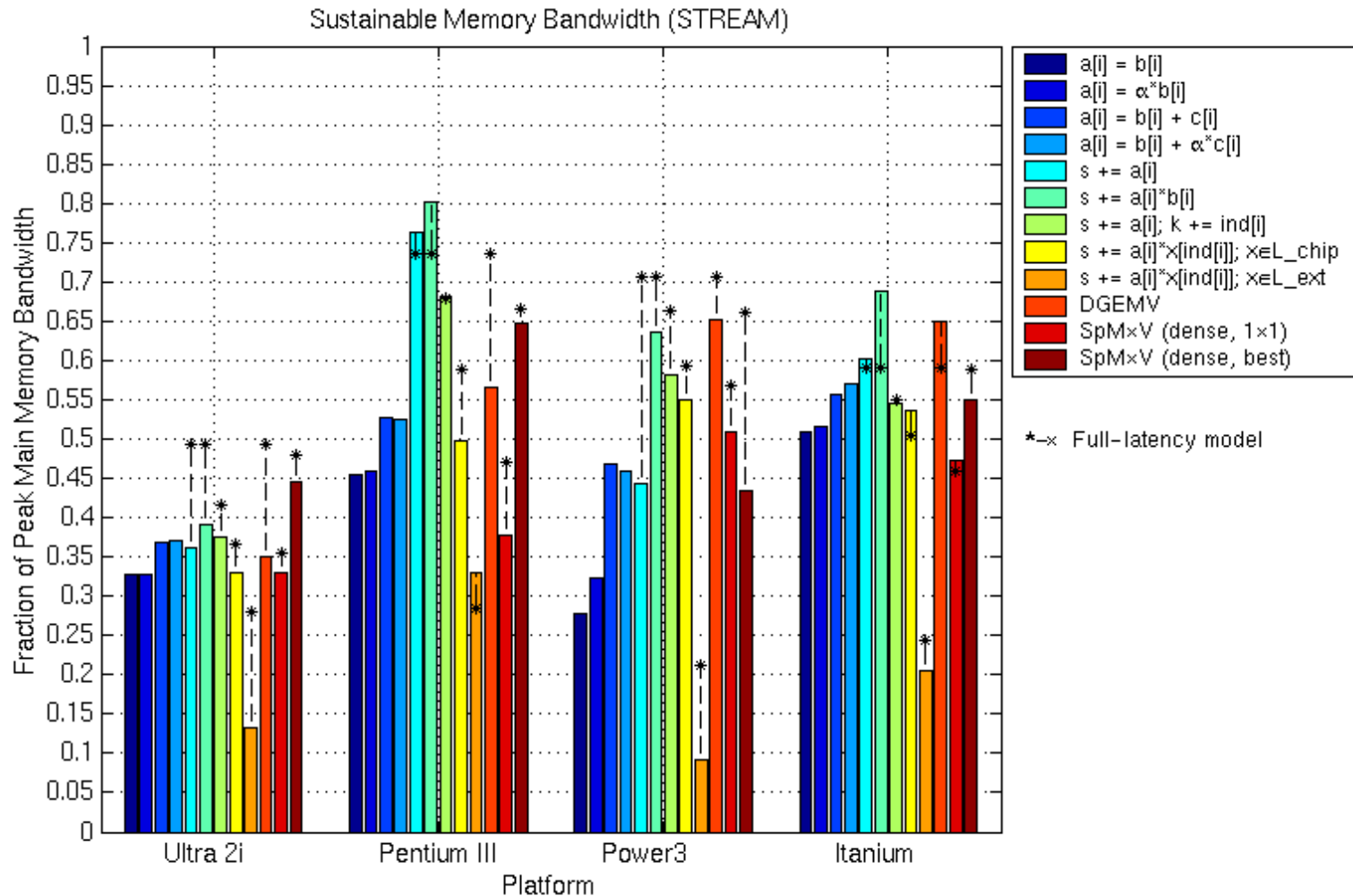
Context: Creating High-Performance Libraries

- Application performance dominated by a few *computational kernels*
- Today: Kernels hand-tuned by vendor or user
- Performance tuning challenges
 - Performance is a complicated function of kernel, architecture, compiler, and workload
 - Tedious and time-consuming
- Successful automated approaches
 - Dense linear algebra: ATLAS/PHiPAC
 - Signal processing: FFTW/SPIRAL/UHFFT

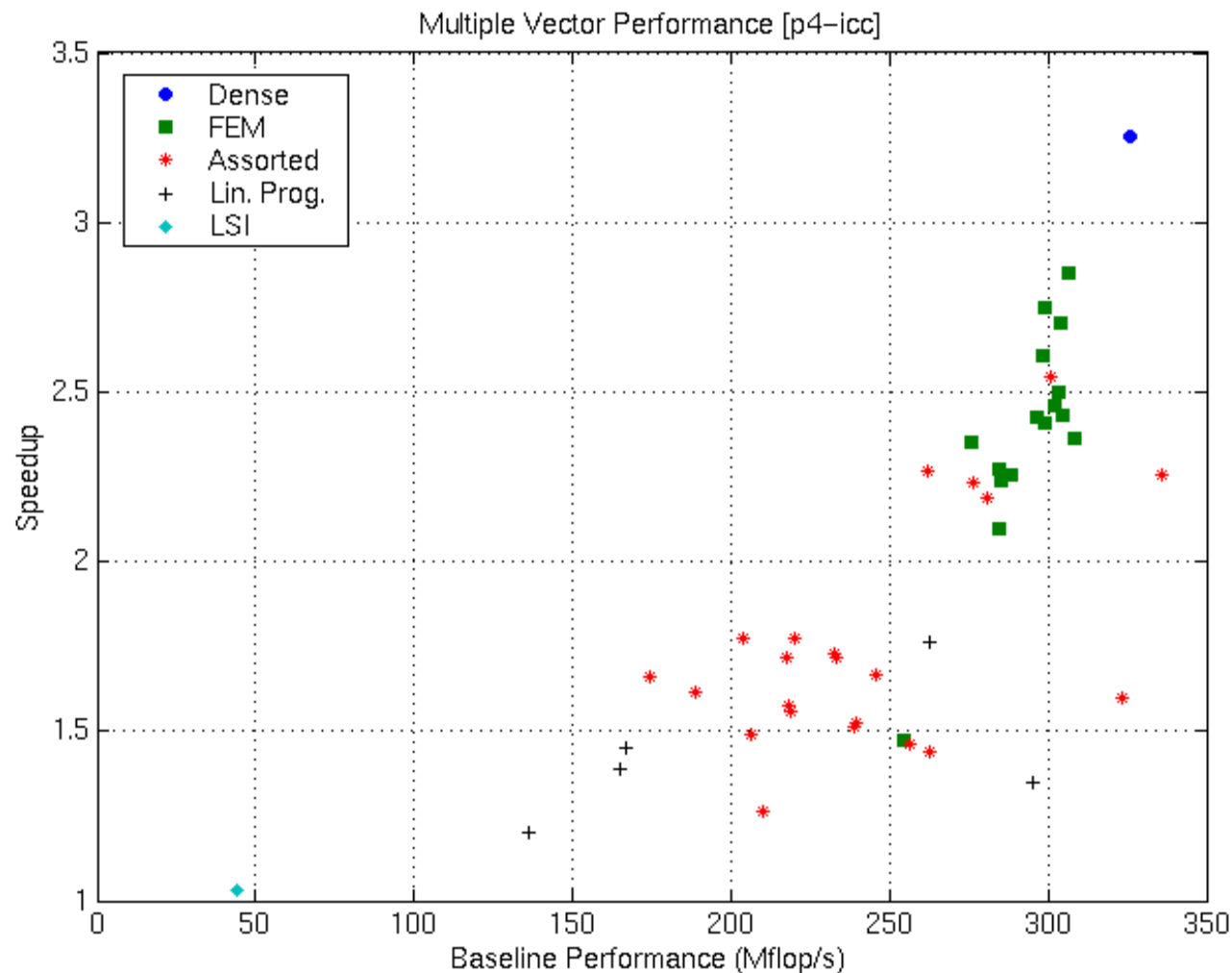
Cache Blocked SpMV on LSI Matrix: Itanium



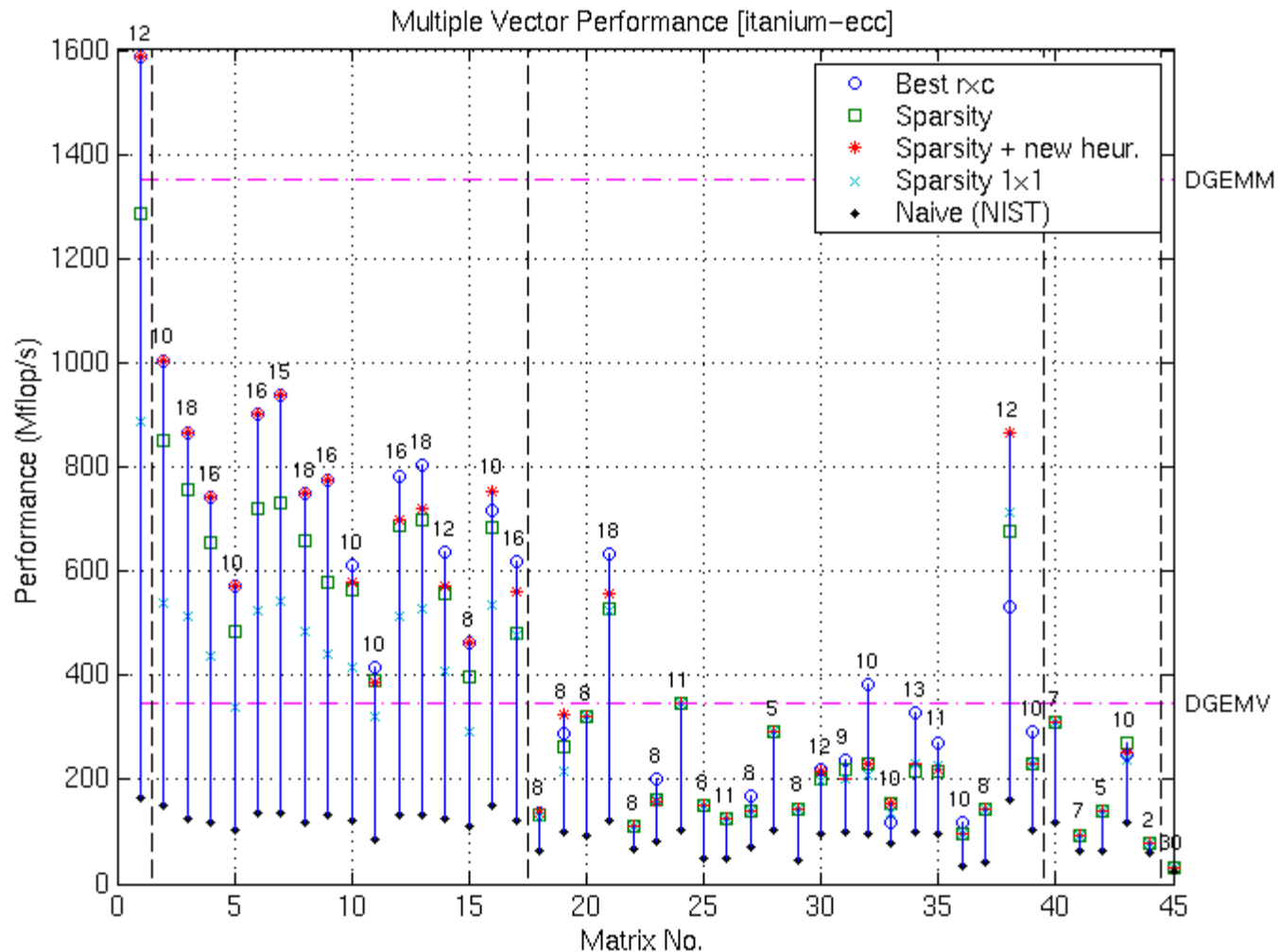
Sustainable Memory Bandwidth



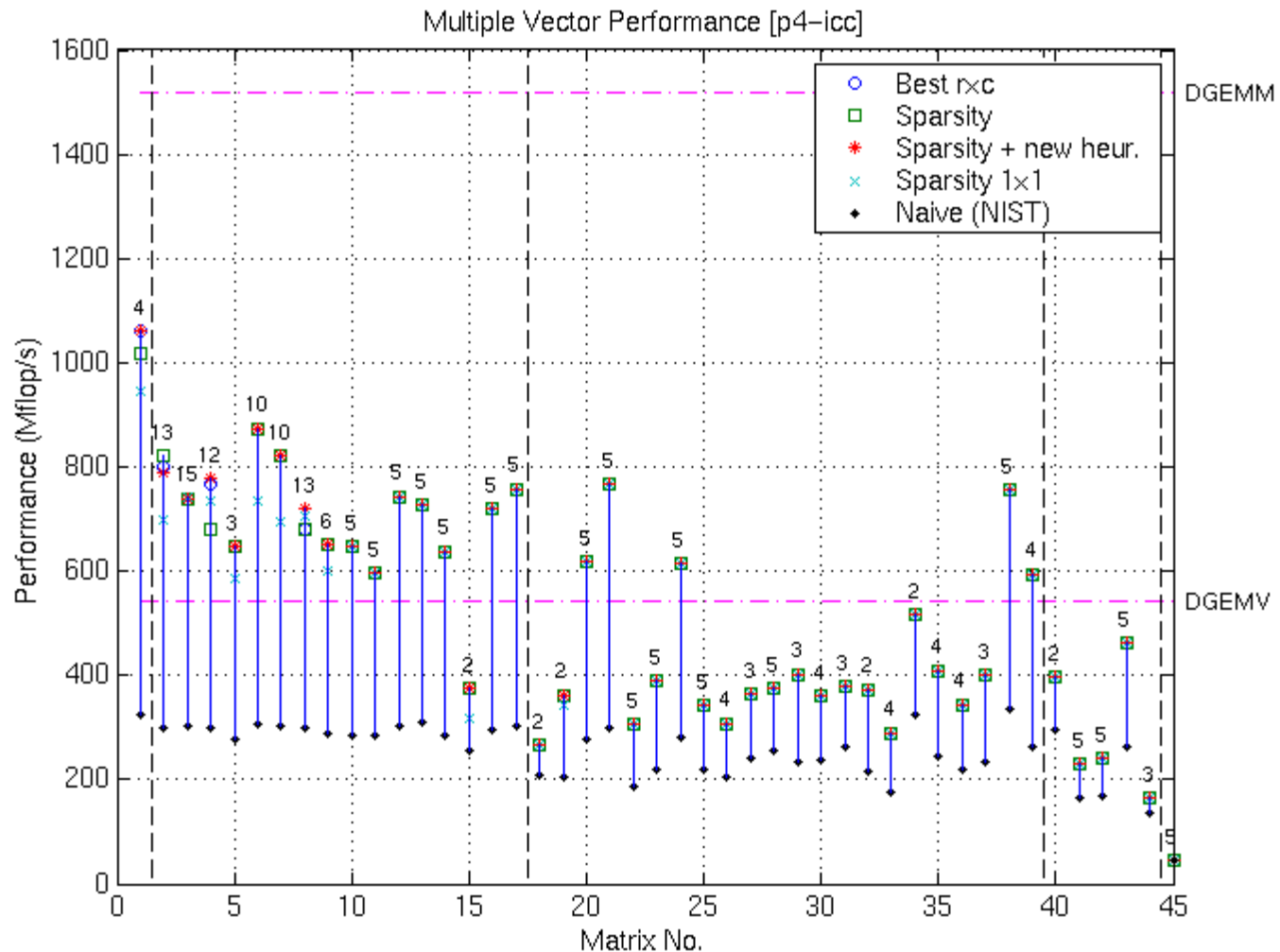
Multiple Vector Performance: Pentium 4



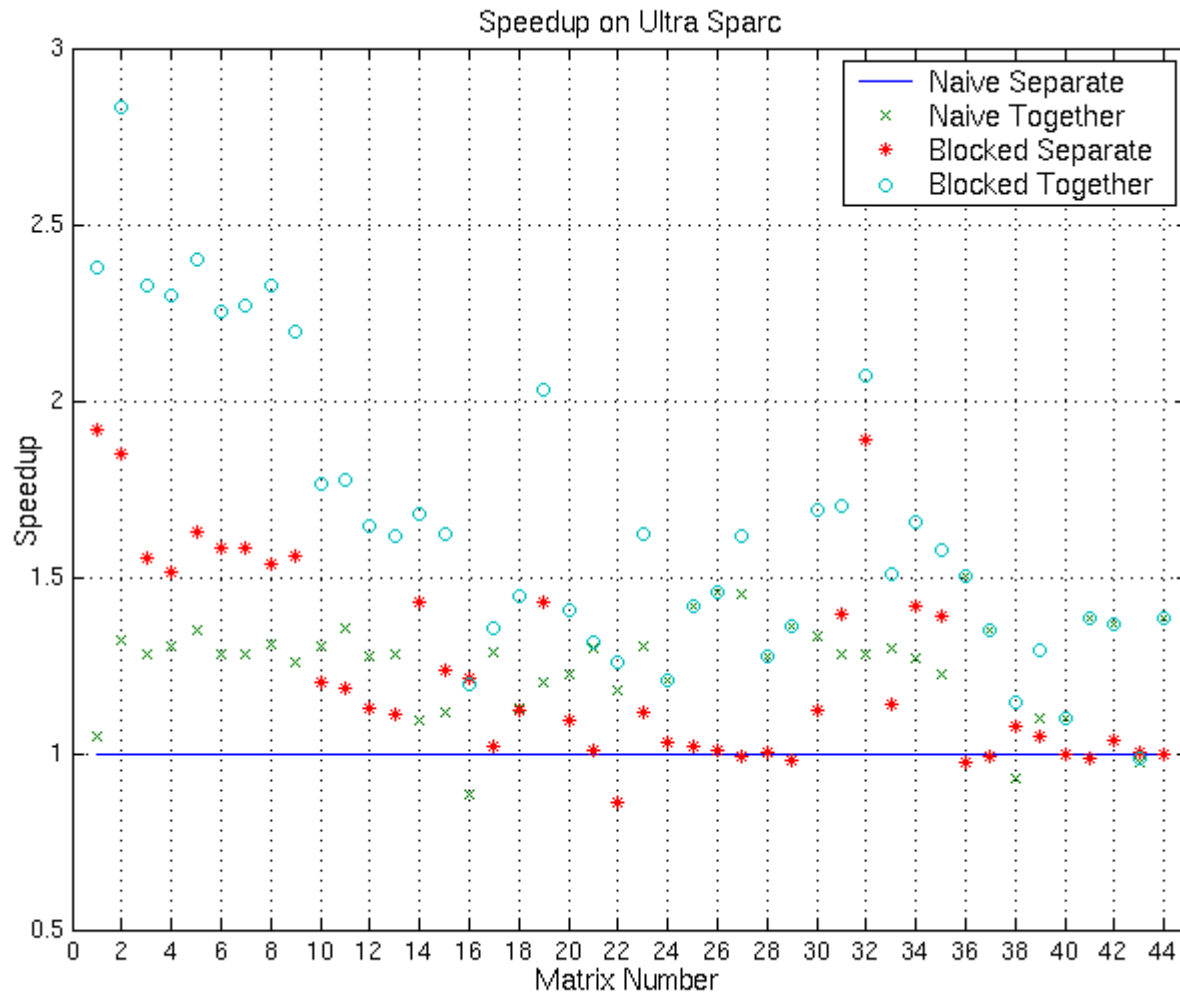
Multiple Vector Performance: Itanium



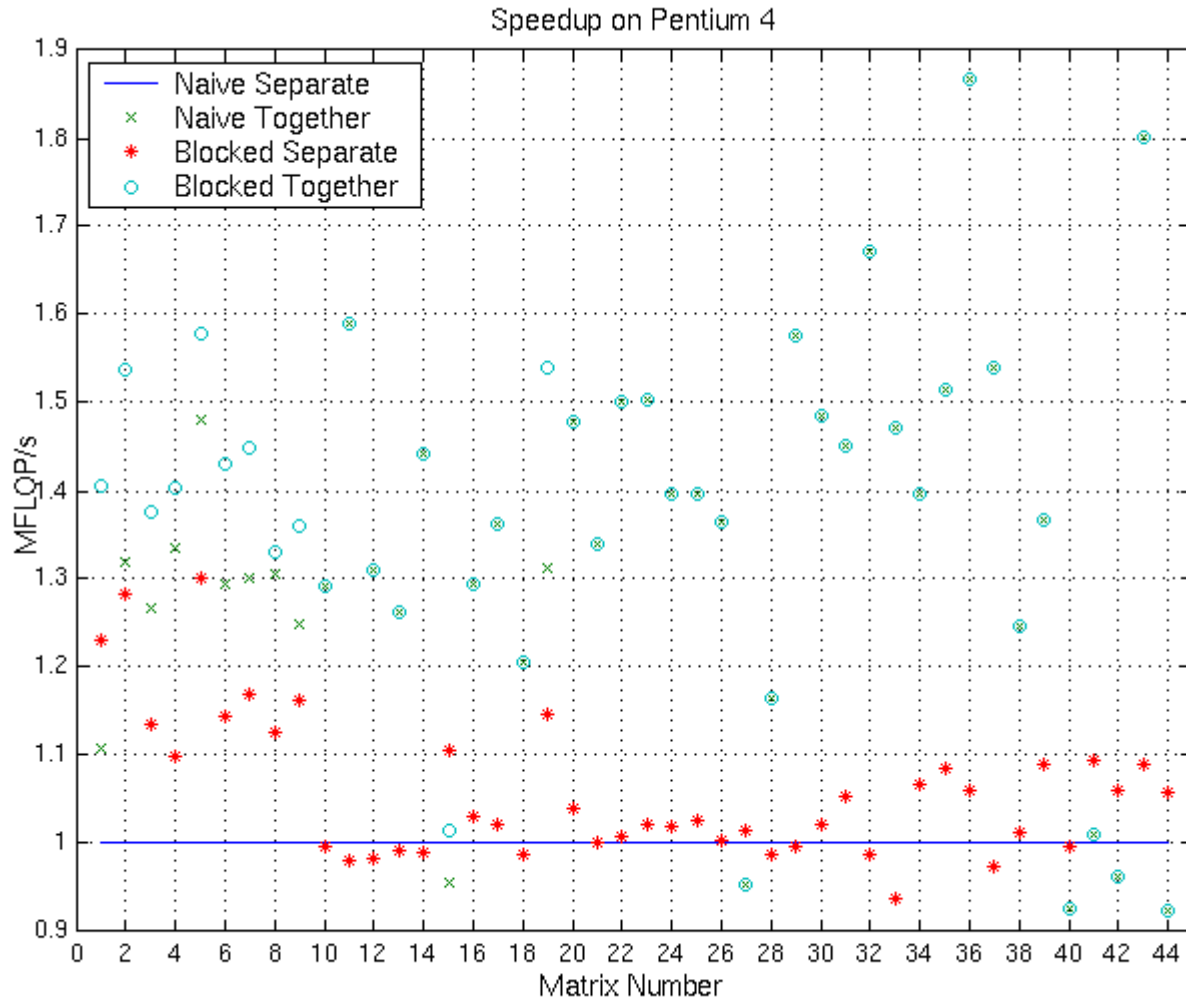
Multiple Vector Performance: Pentium 4



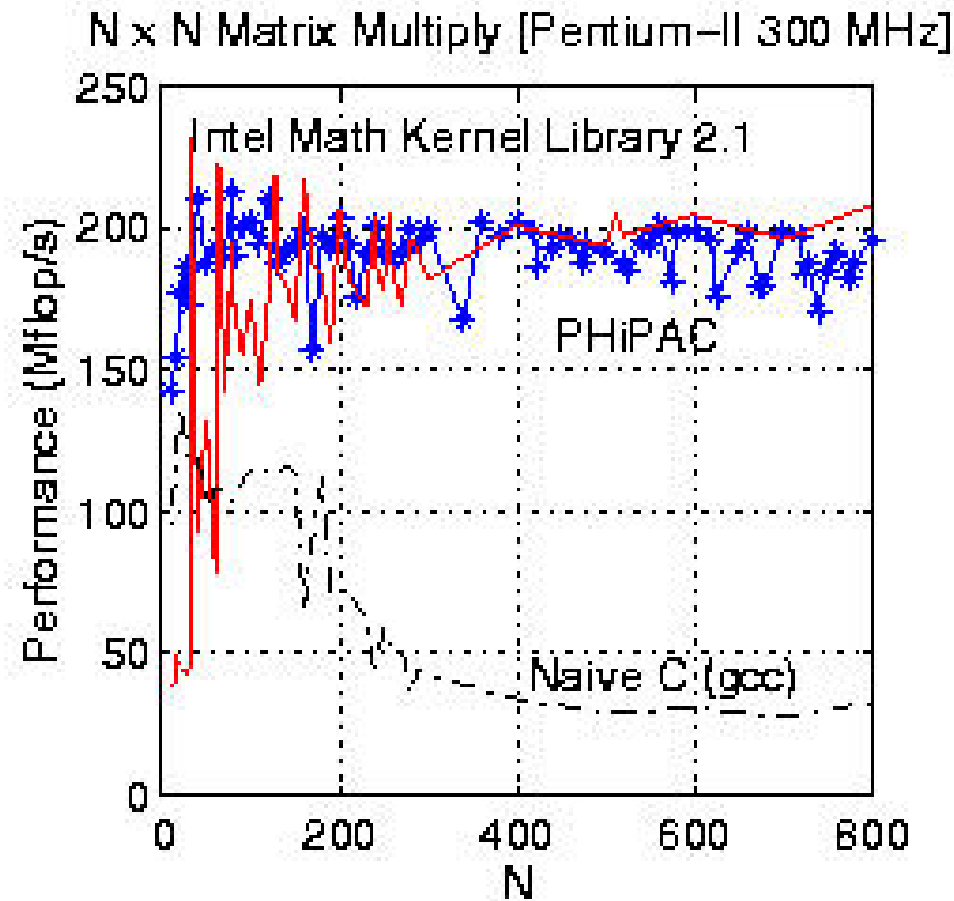
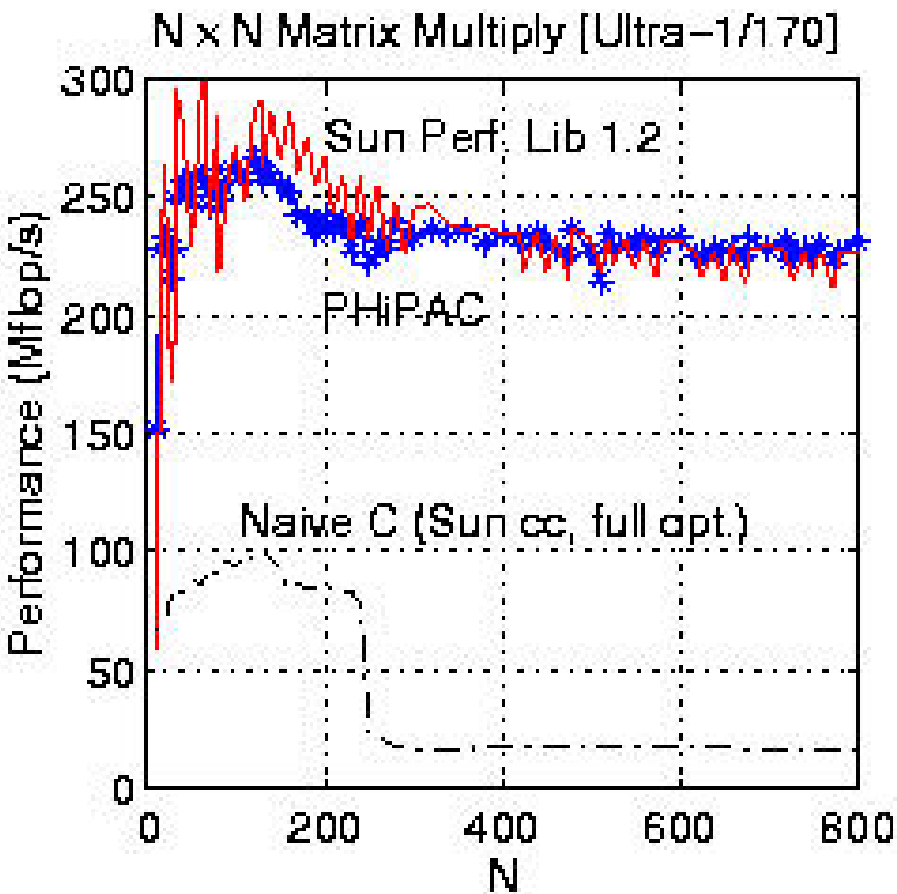
Optimized $AA^T * x$ Performance: Ultra 2i



Optimized $AA^T * x$ Performance: Pentium 4

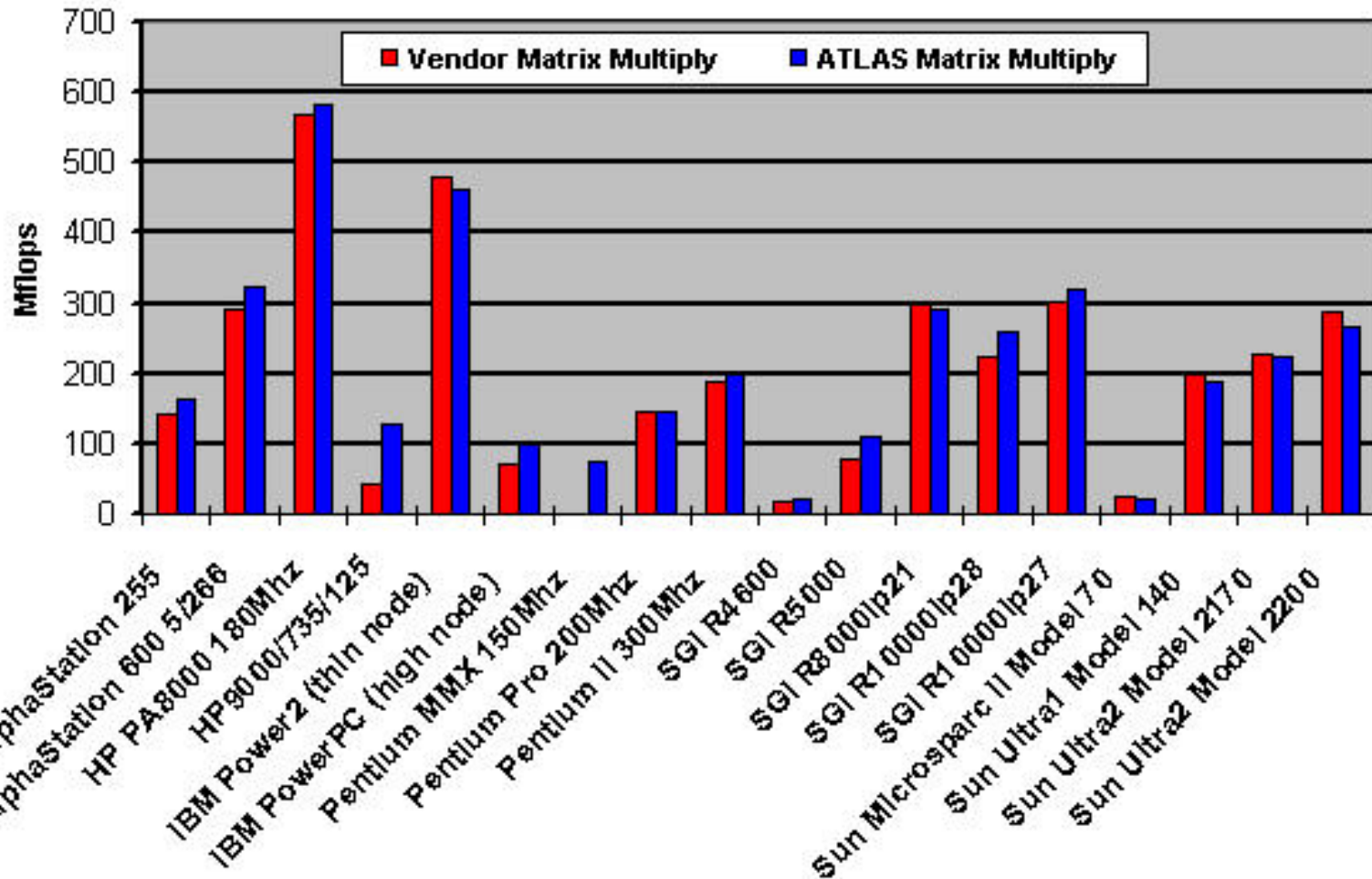


Tuning Pays Off—PHiPAC



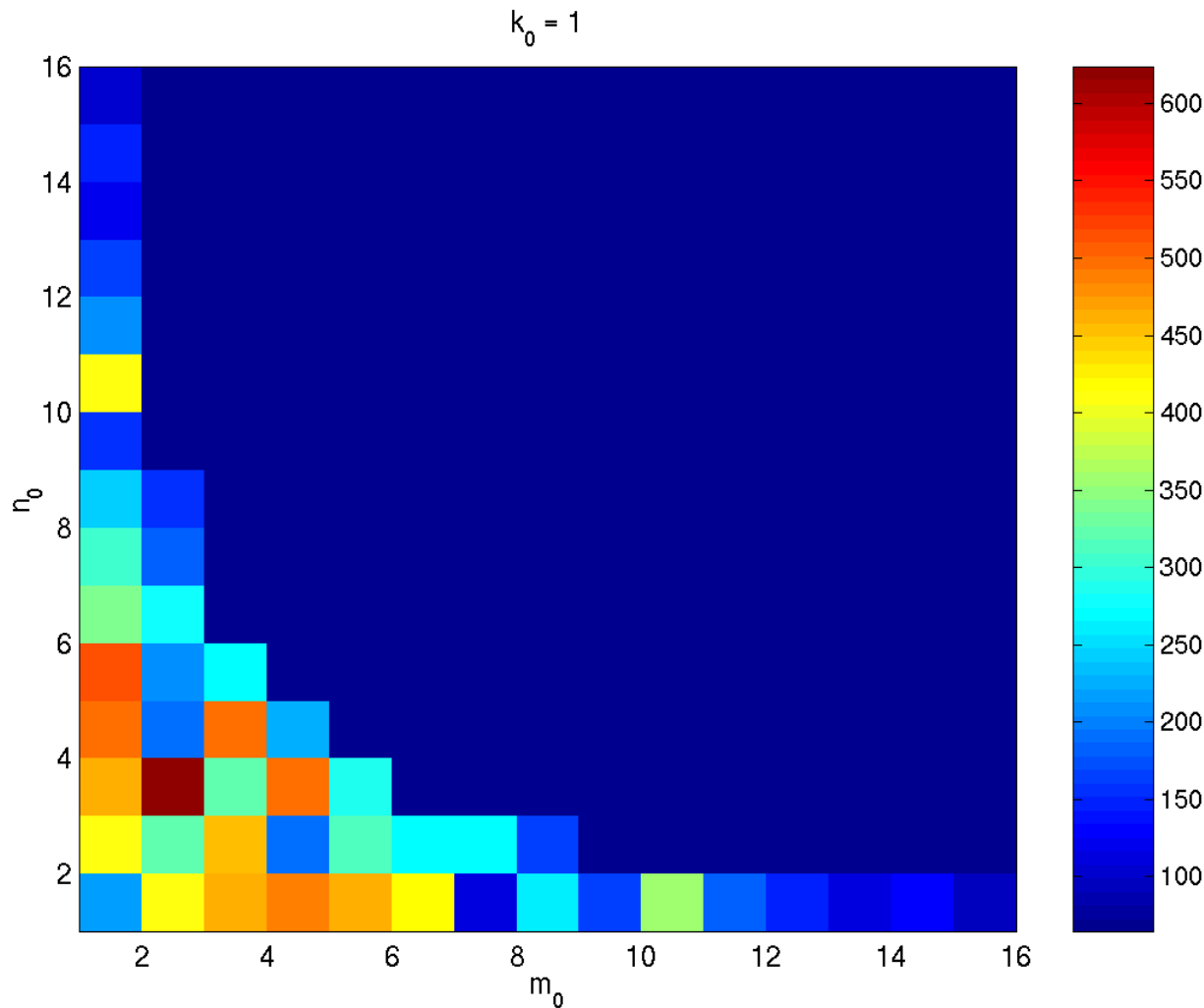
Tuning pays off – ATLAS

500x500 Double Precision Matrix-Matrix Multiply Across Multiple Architectures



Extends applicability of PHIPAC; Incorporated in Matlab (with rest of LAPACK)

Register Tile Sizes (Dense Matrix Multiply)

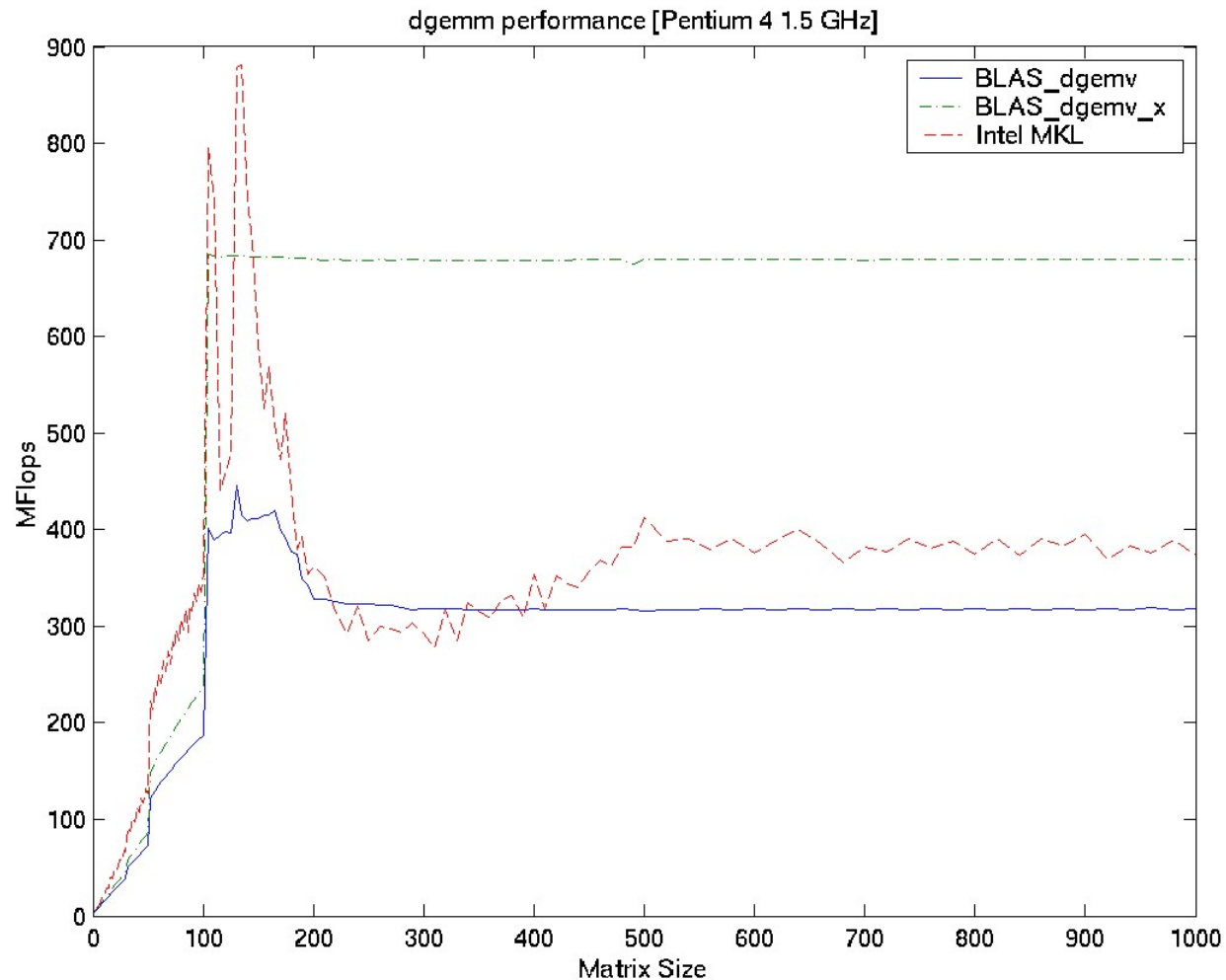


333 MHz Sun Ultra 2i

**2-D slice of 3-D space;
implementations color-
coded by performance
in Mflop/s**

**16 registers, but 2-by-3
tile size fastest**

High Precision GEMV (XBLAS)



High Precision Algorithms (XBLAS)

- Double-double (High precision word represented as pair of doubles)
 - Many variations on these algorithms; we currently use Bailey's
- Exploiting Extra-wide Registers
 - Suppose $s(1), \dots, s(n)$ have f -bit fractions, SUM has $F > f$ bit fraction
 - Consider following algorithm for $S = \sum_{i=1, n} s(i)$
 - Sort so that $|s(1)| \geq |s(2)| \geq \dots \geq |s(n)|$
 - $SUM = 0$, for $i = 1$ to n $SUM = SUM + s(i)$, end for, $sum = SUM$
 - Theorem (D., Hida) Suppose $F < 2f$ (less than double precision)
 - If $n \leq 2^{F-f} + 1$, then error ≤ 1.5 ulps
 - If $n = 2^{F-f} + 2$, then error $\leq 2^{2f-F}$ ulps (can be $\gg 1$)
 - If $n \geq 2^{F-f} + 3$, then error can be arbitrary ($S \neq 0$ but $sum = 0$)
 - Examples
 - $s(i)$ double ($f=53$), SUM double extended ($F=64$)
 - accurate if $n \leq 2^{11} + 1 = 2049$
 - Dot product of single precision $x(i)$ and $y(i)$
 - $s(i) = x(i)*y(i)$ ($f=2*24=48$), SUM double extended ($F=64$) \Rightarrow
 - accurate if $n \leq 2^{16} + 1 = 65537$