Code Synthesis for Automatic Tuning

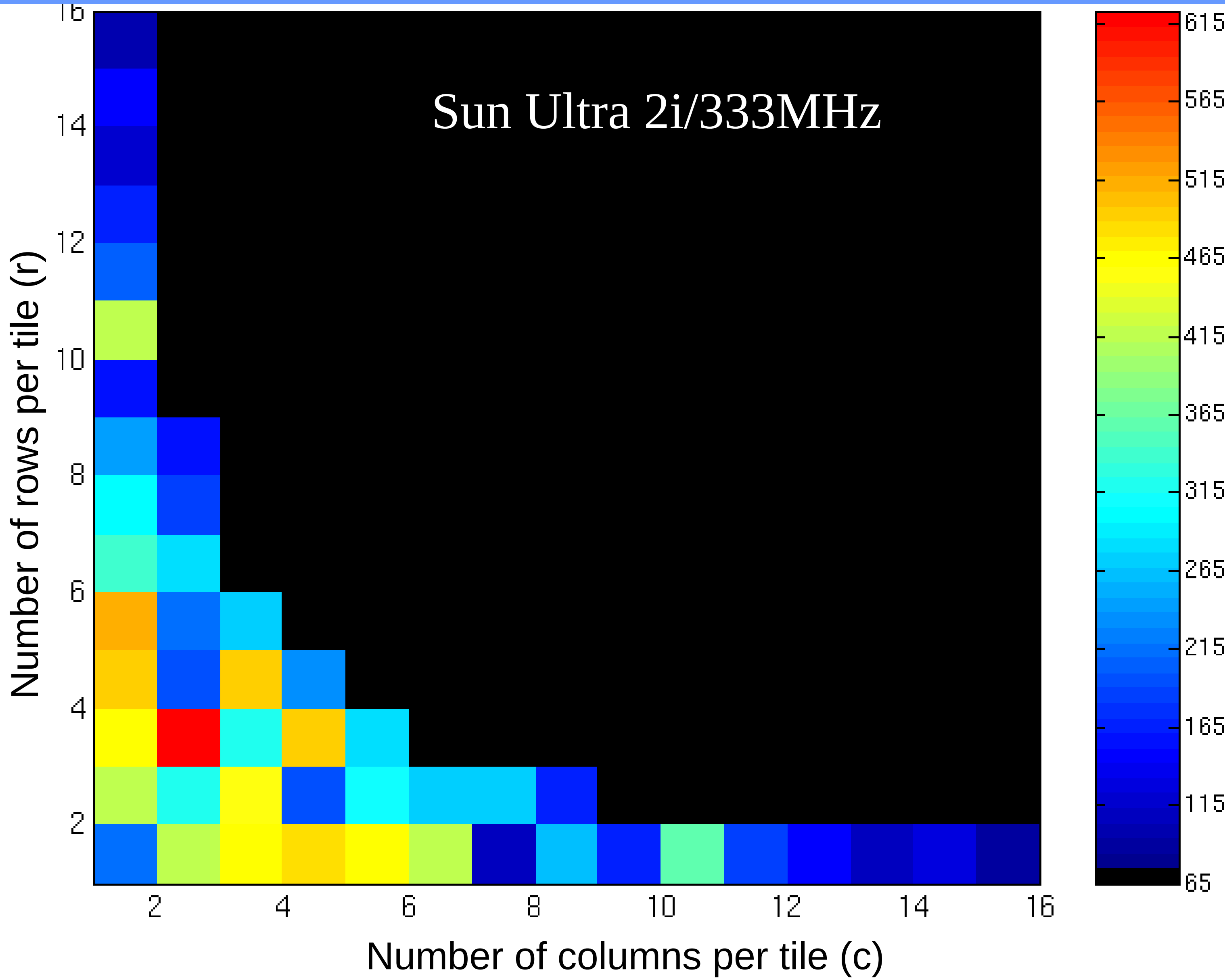
Kathy Yelick
U.C. Berkeley and Lawrence Berkeley National Laboratory

Richard Vuduc, Lawrence Livermore National Laboratory
James Demmel, U.C. Berkeley
Berkeley Benchmarking and OPtimization (BeBOP) Group
bebop.cs.berkeley.edu

Automatic Performance Tuning

- Motivation: replace hand tuning of computational kernels
  - Tedious and difficult
  - Too hard to keep up with new architectures, compilers, kernels
  - Sometimes tuning must be done at runtime
- Automatic performance tuning:
  - Approach
    - Generate “space” of candidate algorithms
    - Search space for best one
  - Examples
    - ATLAS – adopted by Matlab and elsewhere
    - PHiPAC - ATLAS predecessor
    - FFTW – 1999 Wilkinson Prize for Numerical Software
    - Spiral – signal processing
    - Sparsity/OSKI – sparse matrix-vector multiply

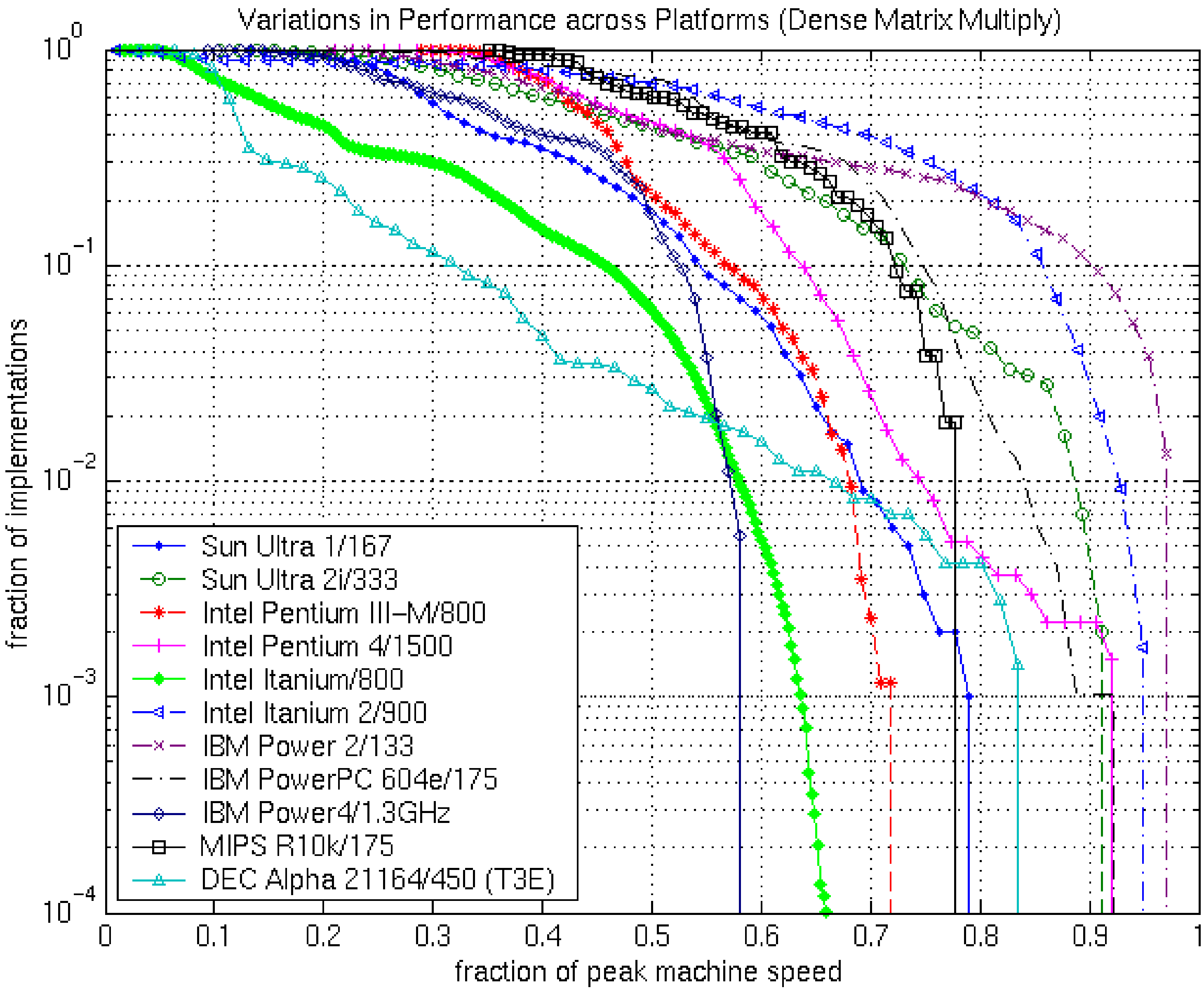
Dense Matrix-Matrix Multiplication



Finding the best block size is like finding a needle in a haystack!

Most Implementations are Not Good

Berkeley Benchmarking and OPtimization Group
bebop.cs.berkeley.edu



Phillip Colella's "Seven dwarfs"

7 numerical methods domain scientific computing

1. Structured Grids (including adaptive)
2. Unstructured Grids
3. Spectral methods (Fast Fourier Transform) ← FFTW
4. Dense Linear Algebra ← Atlas
5. Sparse Linear Algebra
6. Particle Methods
7. Monte Carlo

Well-defined targets from algorithmic, software, and architecture standpoint

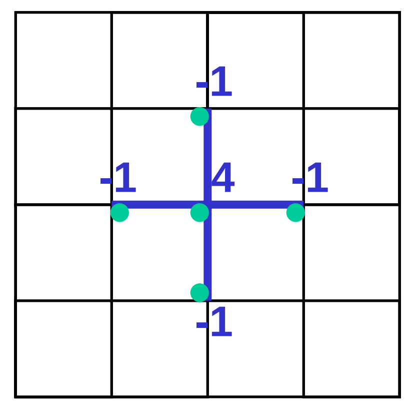
Slide from "Defining Software Requirements for Scientific Computing", Phillip Colella, 2004

Stencil on Grid → Matrix Vector Multiply on Matrix

- Shown for the 2D case, the matrix T is now
  – Grid points numbered left to right, top row to bottom row

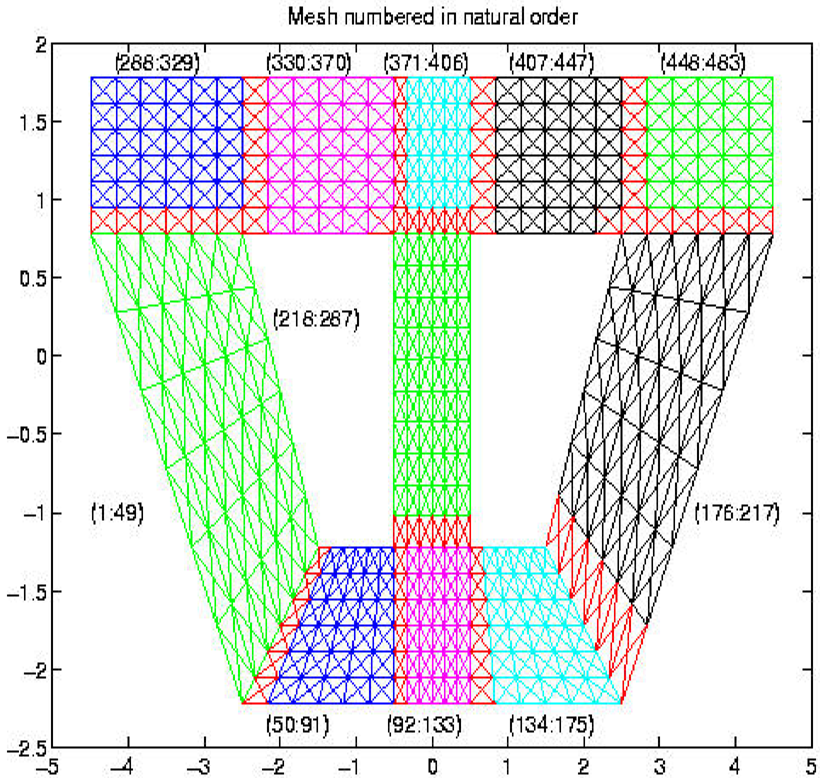
$$T = \left(\begin{array}{ccc|ccc|ccc} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ \hline -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ \hline & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{array}\right)$$

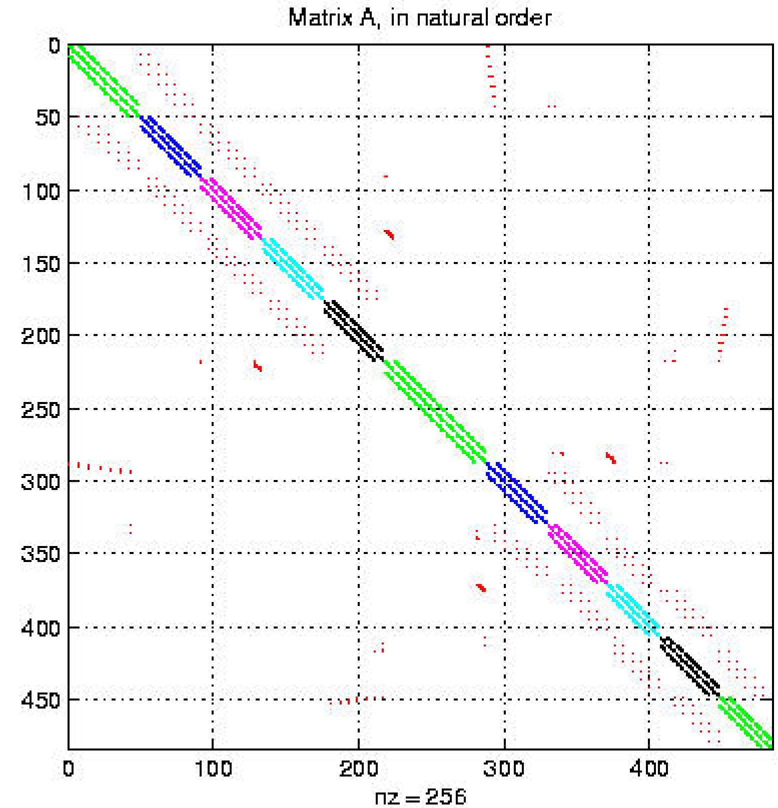
Graph and "stencil"



- Similar to "adjacency matrix" for arbitrary graph

Conversion between a mesh and matrix

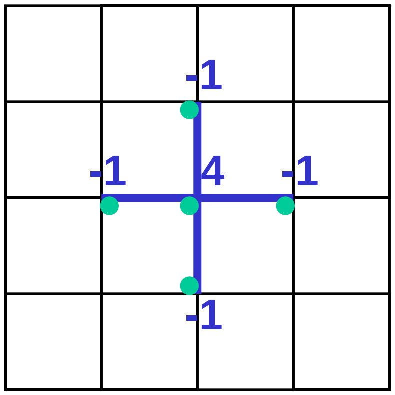




Hidden slide: shown in earlier lecture on sources of parallelism

Project Proposal: Stencil Generator

- Stencil operations on regular meshes are very common and have many variations
  - Dimension: 1D (e.g. 3pt), 2D (5pt or 9pt), 3D (7pt or 27pt)
  - Shape: 1D (e.g. 3pt), 2D (5pt or 9pt), 3D (7pt or 27pt), they need not be regular
  - Band: just your immediate neighbors (band=1), or their neighbor (band=2), or…
  - Balanced or unbalanced in various directions (isotropic, anisotropic)
  - coefficients (NAS MG)
    - constant, 1 point and all others
    - constant, 1 point and distance-based coefficients
    - variable, relative to each position
  - Update in place vs. 2nd grid
  - Colored algorithms (red-black in 2D)

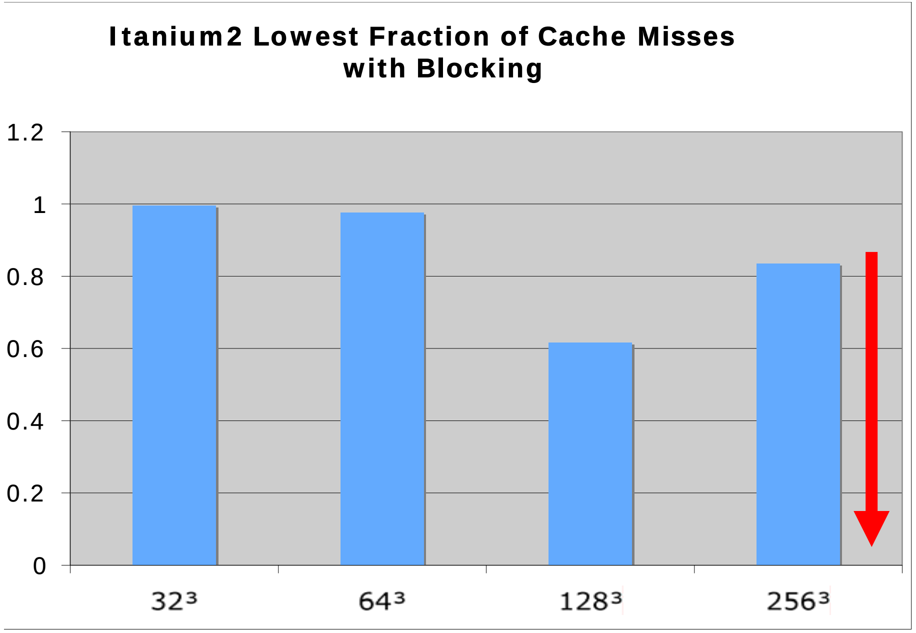


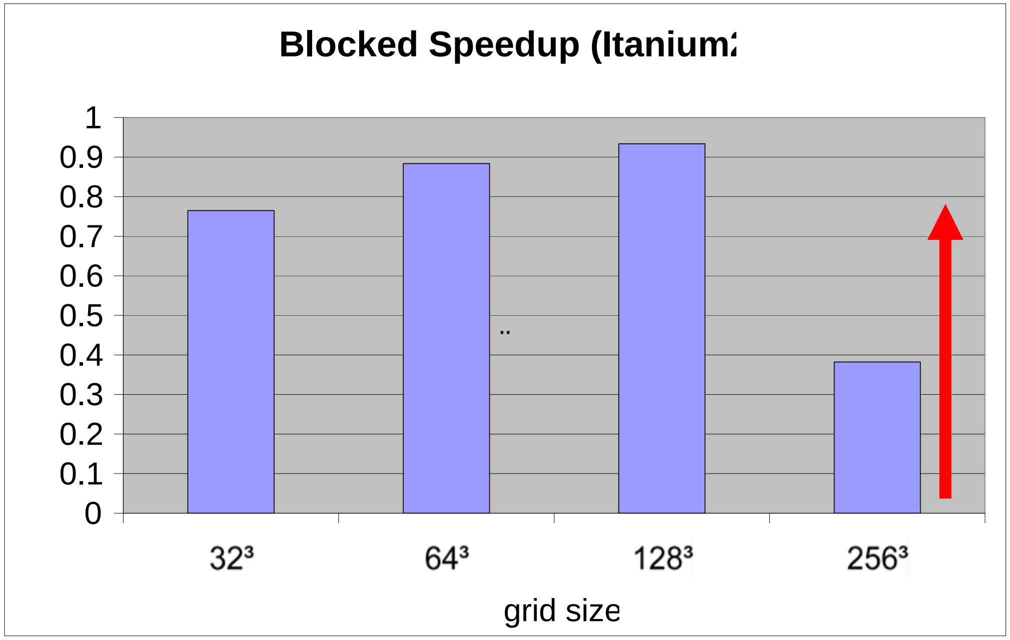
Optimizing Stencils

- Stencil operations have simple structure
  - Loop nest with single assignment in the simple case
  - Real applications use these and more complicated cases
- Low floating point rate:
  - Typically ~1 FLOP per load
  - Good spatial locality, but little temporal locality (re-use)
  - Run at small fraction of peak (<15%)!
- Optimizations:
  - Improve reuse within a sweep through the grid
  - Tile to improve chance that previous plane (or row) is still in cache when the neighboring one is processes

Tiling Stencil Computations

- Several papers on tiling stencil computations
  - E.g., Rivera and Tseng SC2002, …
- Old Conventional Wisdom
  - Cache misses are the most important factor





Cache misses in blocked/unblocked

Time for unblocked/blocked

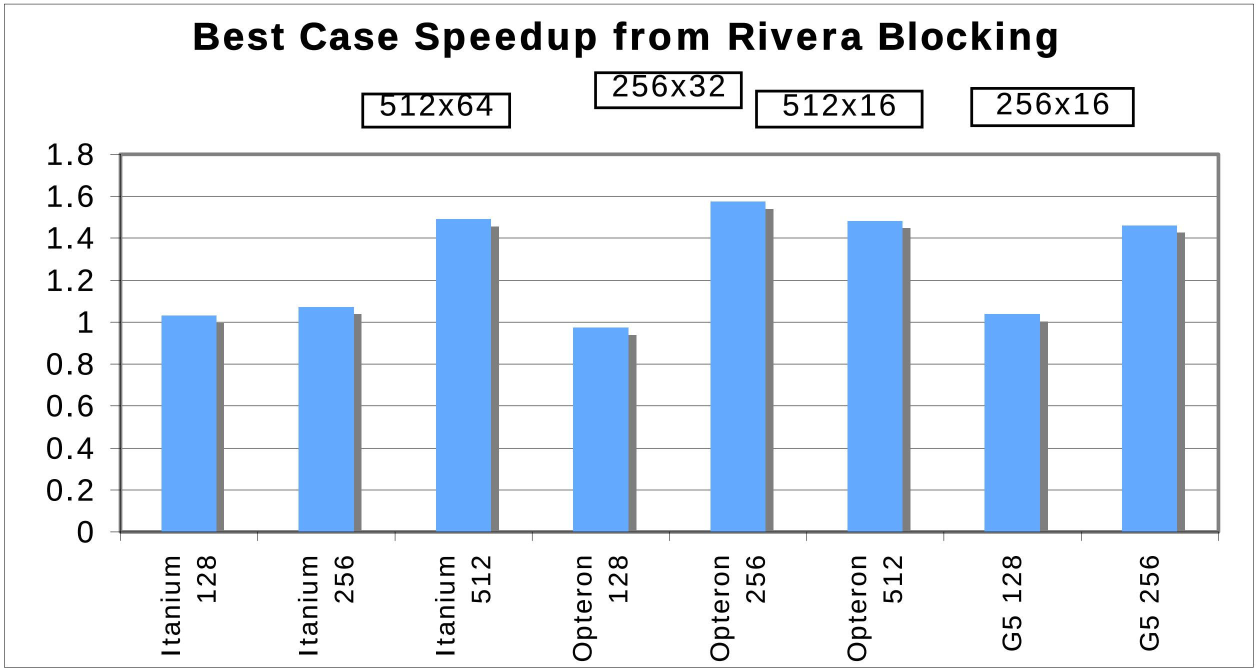
Stencil Probe Cache Blocking Revisited

New Conventional Wisdom: Prefetching is as important as caching

- Little’s Law (Bailey ‘97): need data in-flight = latency * bandwidth

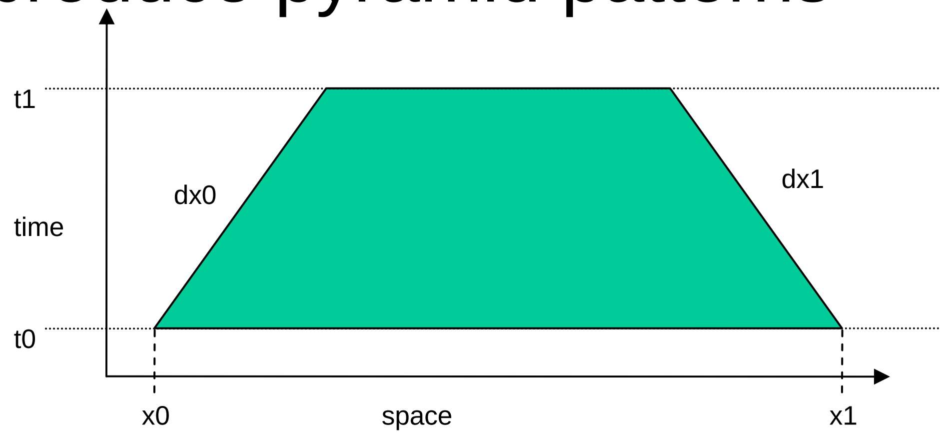
Cache blocking is useful for

1. large grid sizes: 3 planes do not fit in cache for 3D problem
2. do not cut/block the unit-stride dimension



Blocking Over Time / Iterations

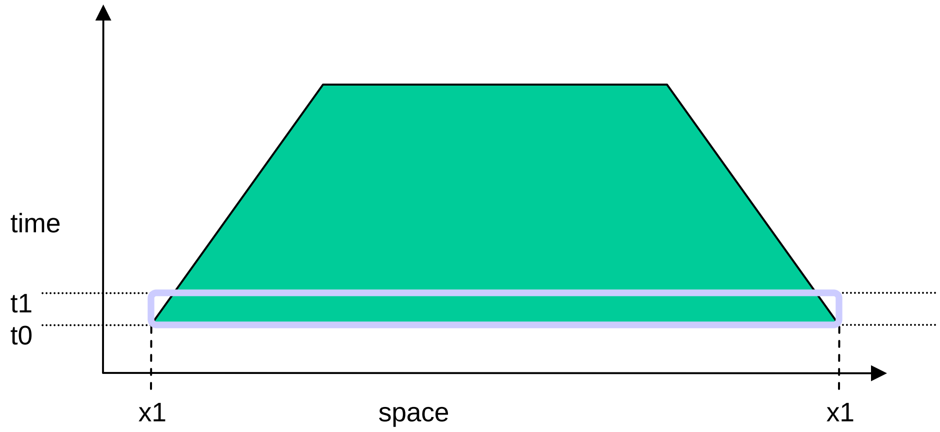
- Can we do better than this?
  - Code is still severely limited by memory bandwidth
- For some computations, you can merge across k sweeps over the grid
  - Re-use data k times (as well as re-use within a plane)
- Dependencies produce pyramid patterns



Frigo & Strumpen, *ICS05.*

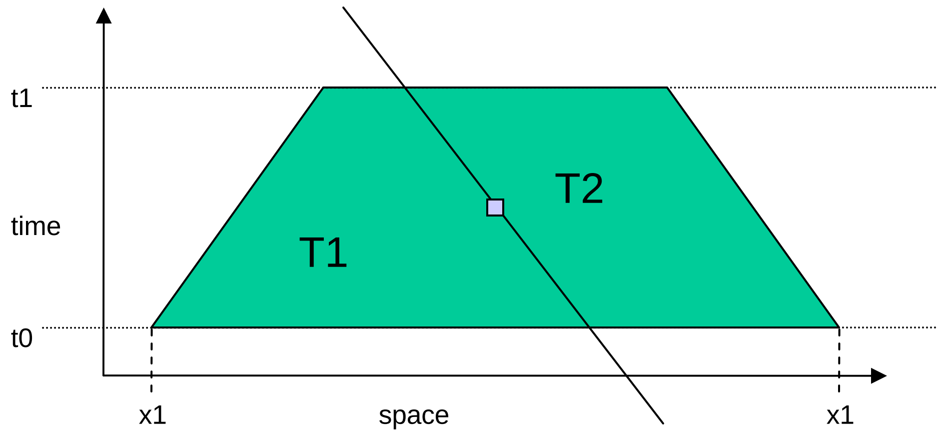
The Algorithm - Base Case

If the height is 1 (ie t1-t0=1) then we simply have a line of points (t0,x) where x0 <= x <= x. Do the kernel on this set of points. Order does not matter (no interdependencies).



The Algorithm - Space Cut

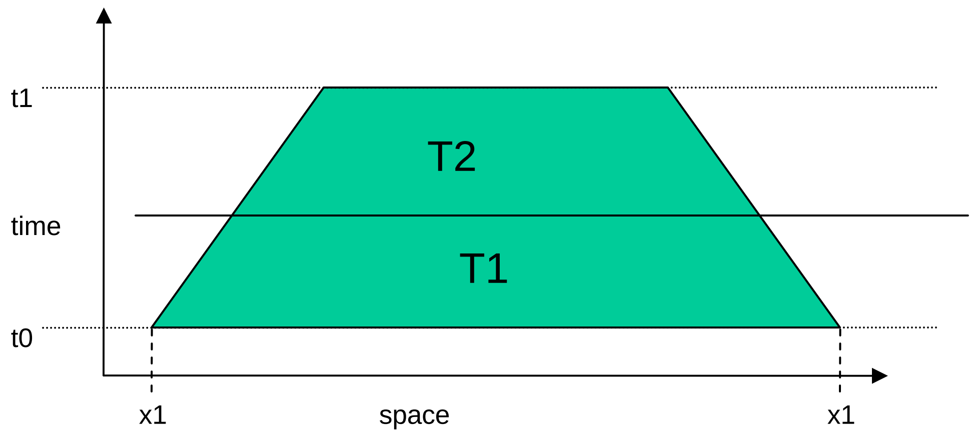
- If the width <= 2*height, then cut with slope=-1 through the center.



- Do T1, then T2. No point in T1 depends on values from T2.

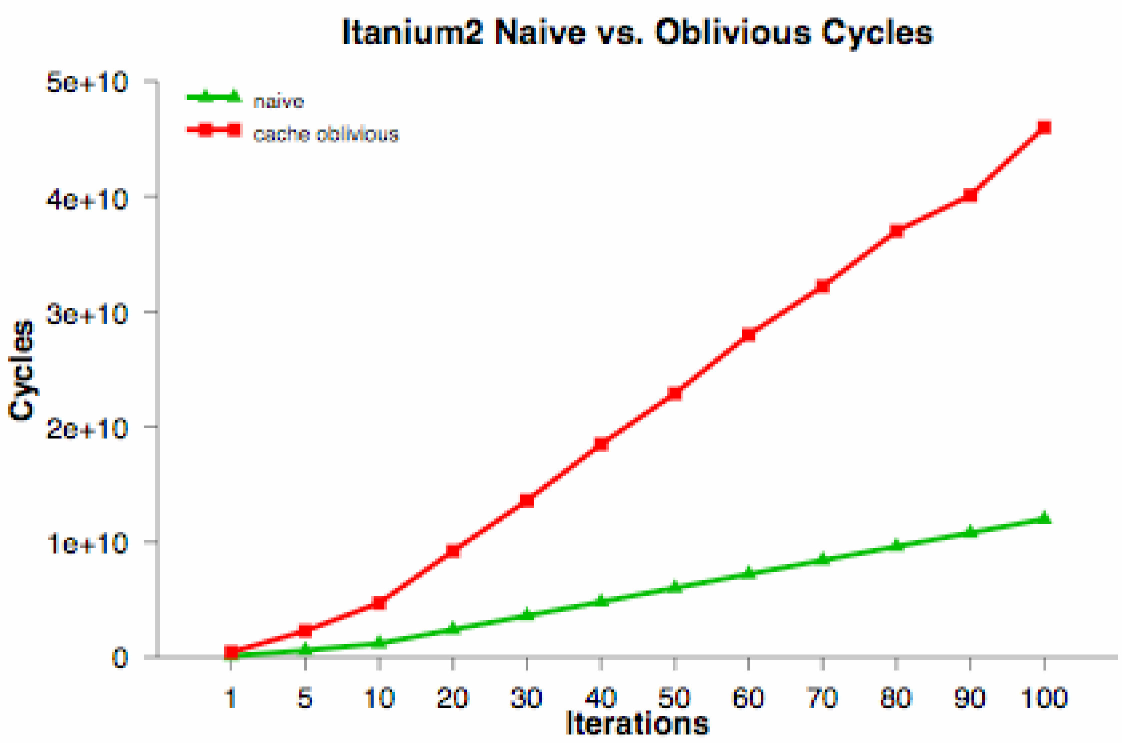
The Algorithm - Time Cut

- Otherwise, cut trapezoid in half in the time dimension.

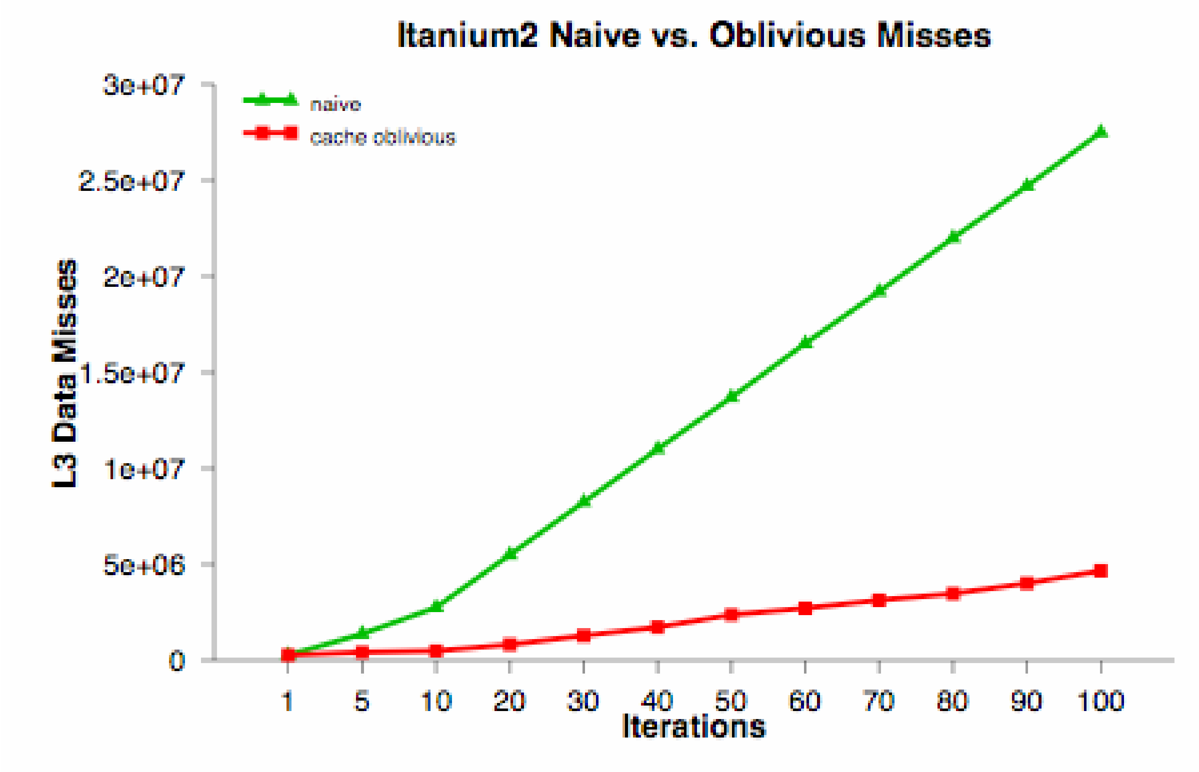


- Do T1, then T2. No point in T1 depends on values of T2.

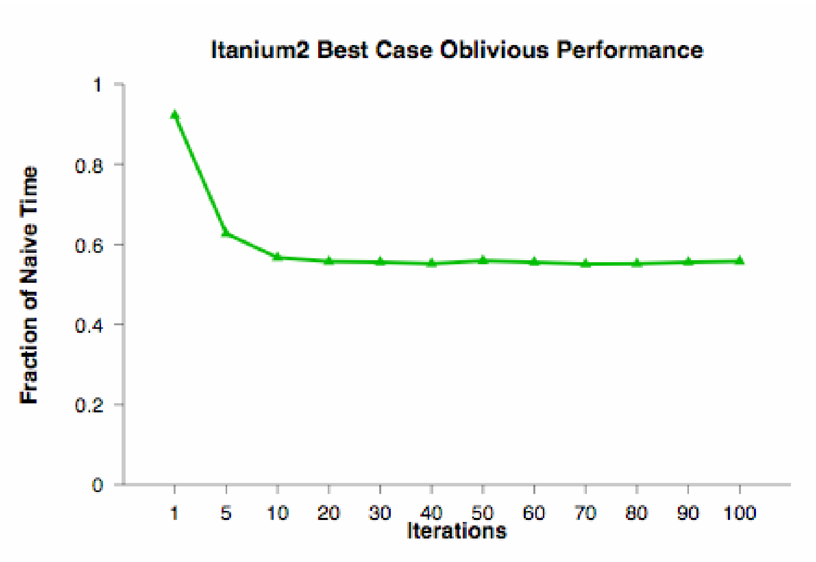
Initial Results - Itanium2

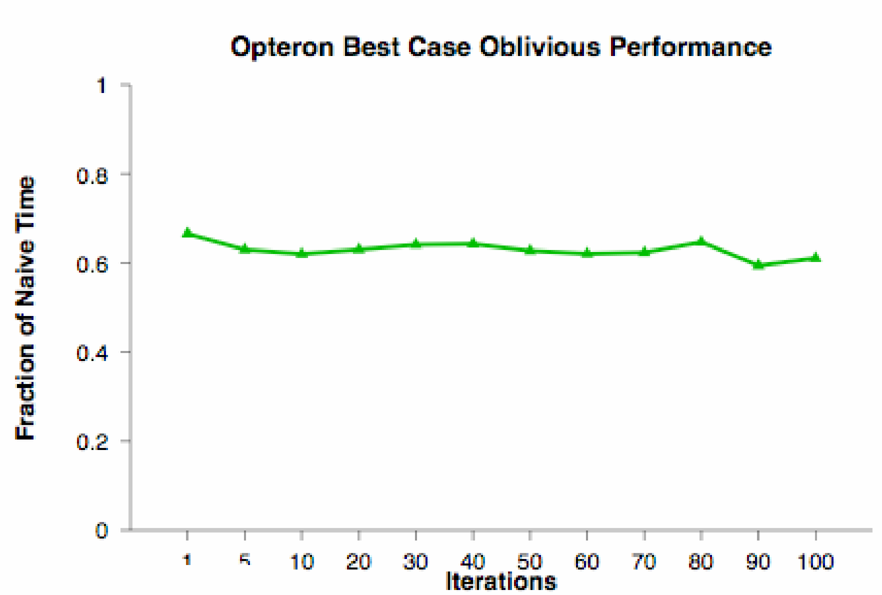


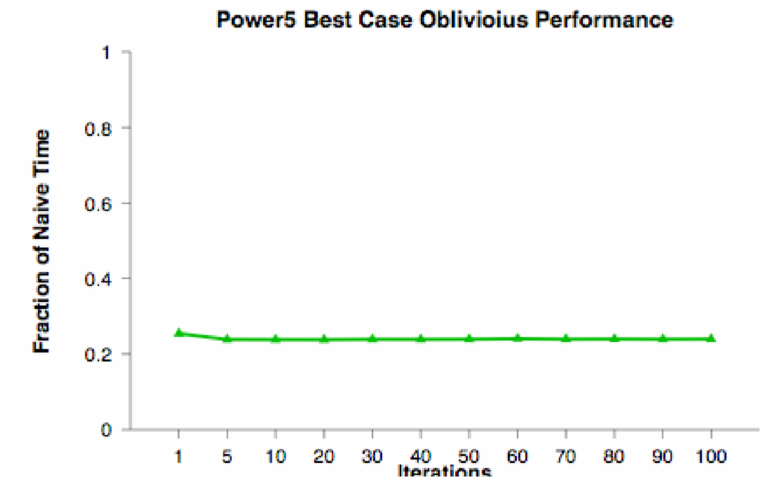
Initial Results - Itanium2



Best Performance







Project Idea Revisited

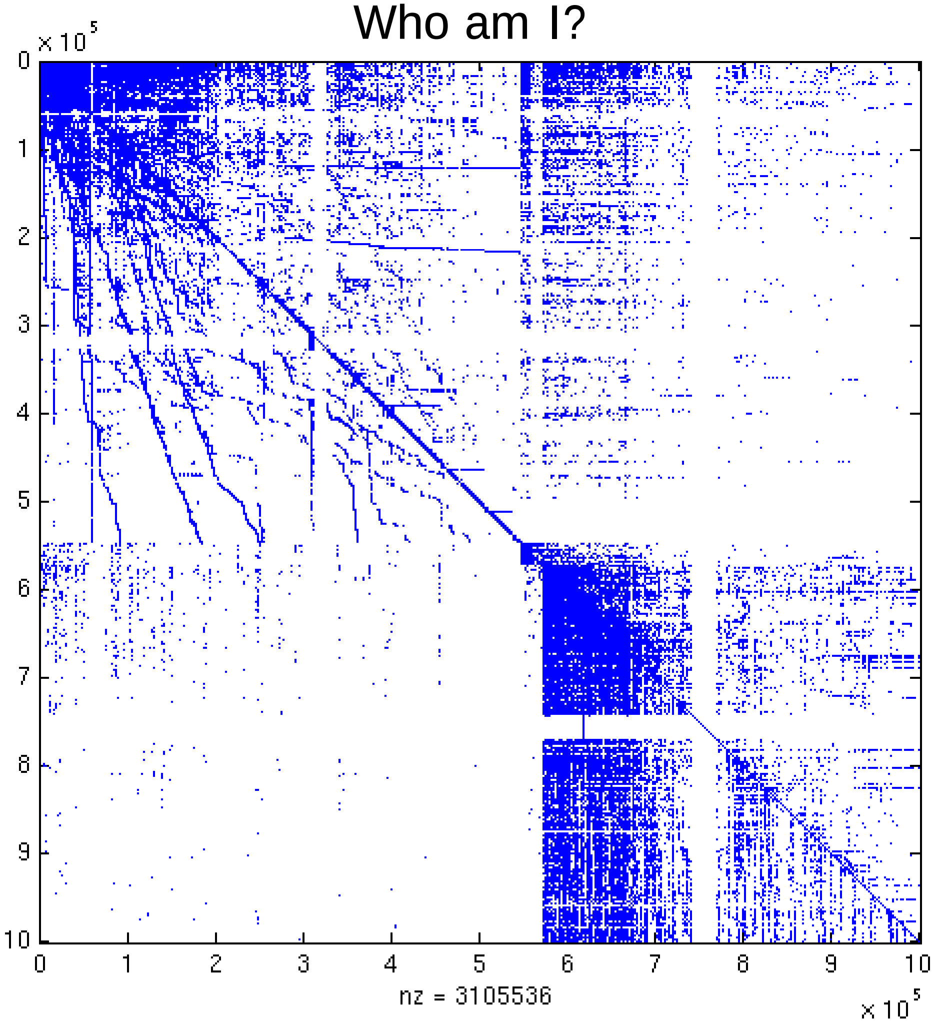
- Cache oblivious stencils not well tested
  - Only limited stencils (3D 7pt)
  - Applications use many different stencils
    - Requested work by apps folks
  - Paper by S. Kamil, Oliker, Shalf so that many optimizations are needed to make it really work
- Recursion is useful for understanding the algorithm
  - Can’t use recursion all the way to the bottom
  - A fixed tiling approach may work as well
  - Key inside is tile shapes: Pyramids and Parallelopipeds

General Sparse Matrix Case

- If this works for stencils, what about arbitrary matrices?
- Tuning arbitrary matrices
  - Project: code generator that is more flexible, maintainable, extensible than current approach
- The time-blocked approach extended to matrices
  - $A^k * x$
  - Intuition: most of cost in A*x is reading matrix A
  - Can we read A once and do k operations with it?
- Notes:
  - “Time” is used loosely; this is typically iterations in a solver
  - Many numerical “details” to make $A^k * x$ useful [Hoemmen]

A “Familiar” Sparse Matrix

Who am I?



I am a
Big Repository
Of useful
And useless
Facts alike.

Who am I?

(Hint: Not your e-mail inbox.)

Motivation for Tuning Sparse Matrices

- Sparse matrix kernels can dominate solver time
  - Sparse matrix-vector multiply (SpMV)
  - SpMV: **runs at < 10% of peak**
- Improving SpMV's performance is hard
  - Performance depends on machine, kernel, matrix
  - Matrix known only at **run-time**
  - Best data structure + implementation can be surprising
  - Tuning becoming **more difficult over time**
- Approach: Empirical modeling and search
  - Off-line benchmarking + run-time models
  - Up to **4x speedups** and **31% of peak** for SpMV
  - Other kernels: **1.8x** triangular solve, **4x** $A^T A \cdot x$, **2x** $A^2 \cdot x$

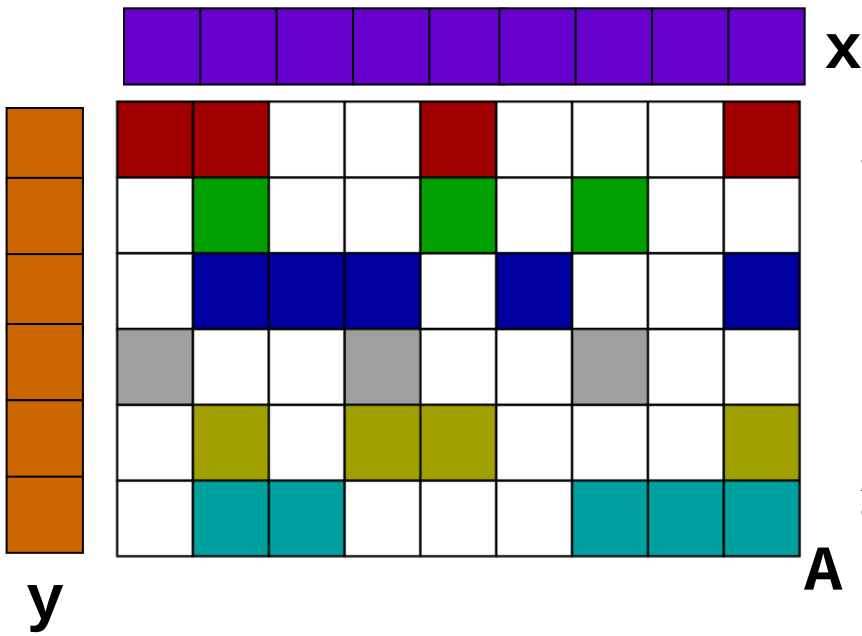
OSKI: Optimized Sparse Kernel Interface

- Sparse kernels tuned for user's matrix & machine
  - Hides complexity of run-time tuning
  - Low-level BLAS-style functionality
  - Includes fast locality-aware kernels: $A^T A \cdot x$, $A^k \cdot x$ …
  - Initial target: cache-based superscalar uniprocessors
- Target users: "advanced" users & solver library writers
- Current focus on uniprocessor tuning
  - Shared/distributed memory versions in progress
- Open-source (BSD) C library
  - 1.0 available: **`bebop.cs.berkeley.edu/oski`**
  - Recently integrated into PETSc

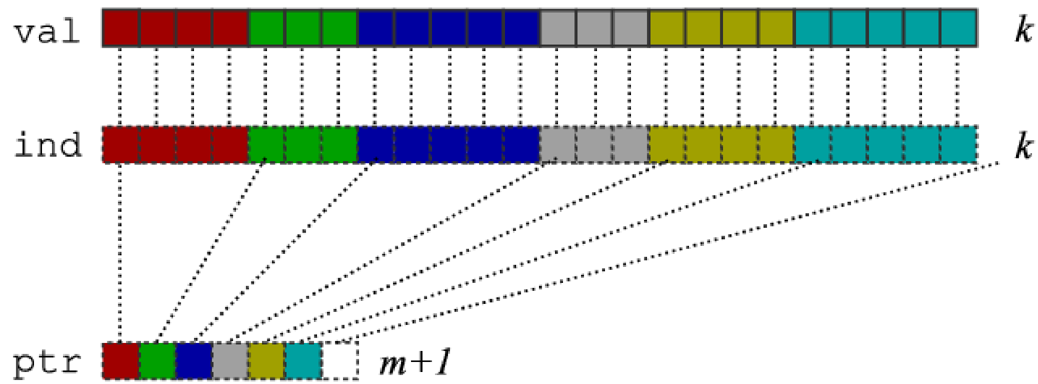
Road Map

- **Sparse matrix-vector multiply (SpMV) review**
  - **Why doesn’t my compiler solve the problem?**
- Historical trends
- Automatic tuning in OSKI
- Future work

Compressed Sparse Row (CSR) Storage

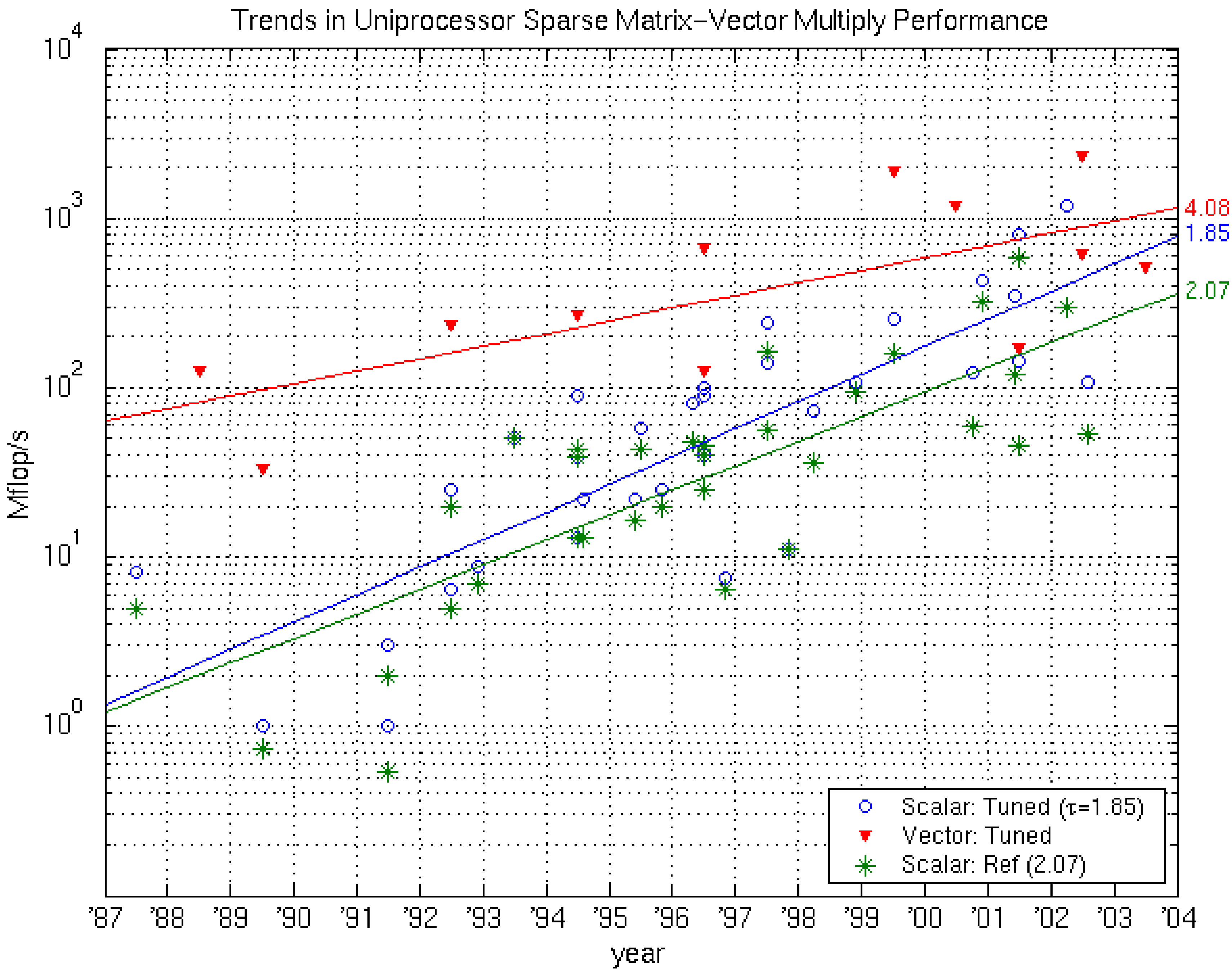


Representation of **A**

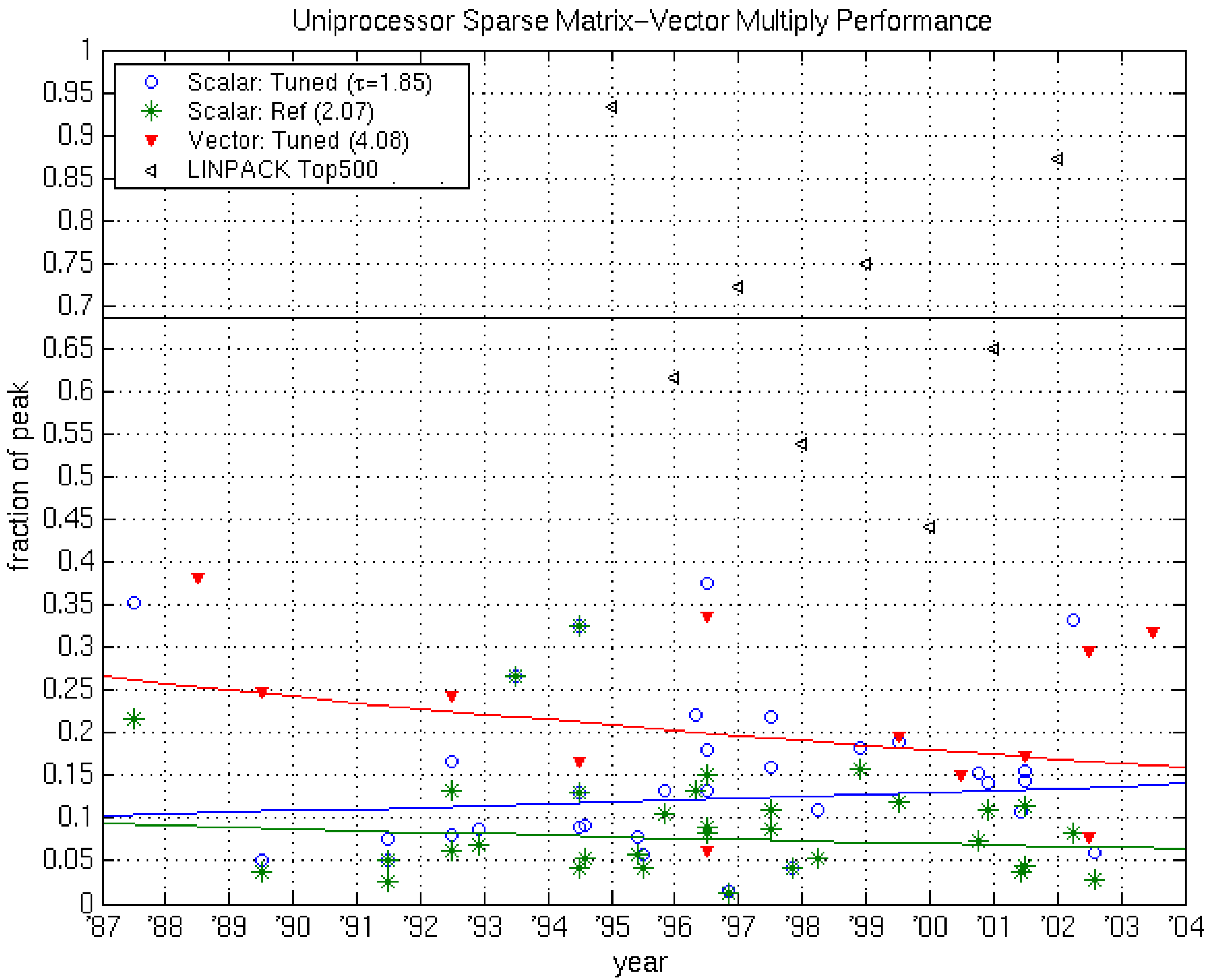


Matrix-vector multiply kernel: $y(i) \leftarrow y(i) + A(i,j) \cdot x(j)$

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

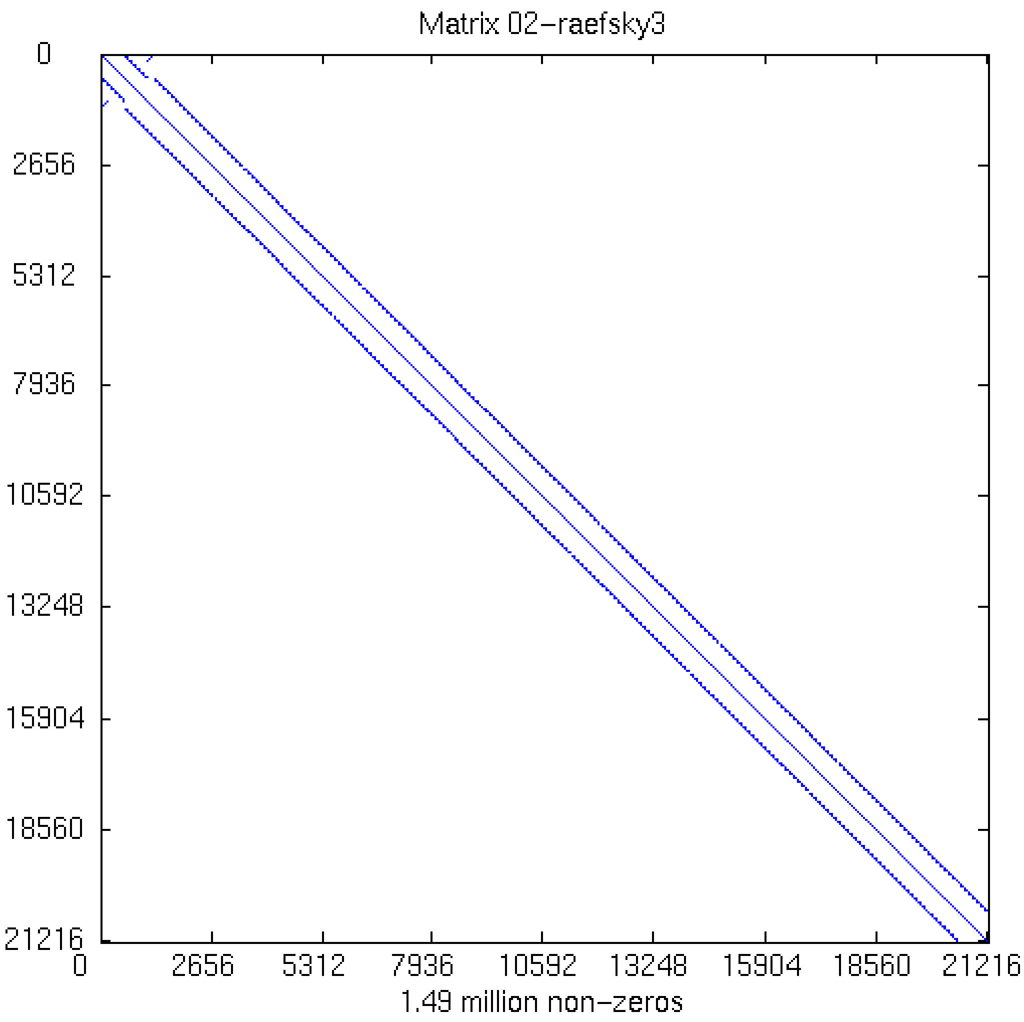


Trends in Uniprocessor Sparse Matrix−Vector Multiply Performance
Mflop/s
year
'87 '88 '89 '90 '91 '92 '93 '94 '95 '96 '97 '98 '99 '00 '01 '02 '03 '04
4.08
1.85
2.07
Scalar: Tuned (τ=1.85)
Vector: Tuned
Scalar: Ref (2.07)



Uniprocessor Sparse Matrix-Vector Multiply Performance
Scalar: Tuned (τ=1.85)
Scalar: Ref (2.07)
Vector: Tuned (4.08)
LINPACK Top500
fraction of peak
year
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55
0.6
0.65
0.7
0.75
0.8
0.85
0.9
0.95
1
'87
'88
'89
'90
'91
'92
'93
'94
'95
'96
'97
'98
'99
'00
'01
'02
'03
'04

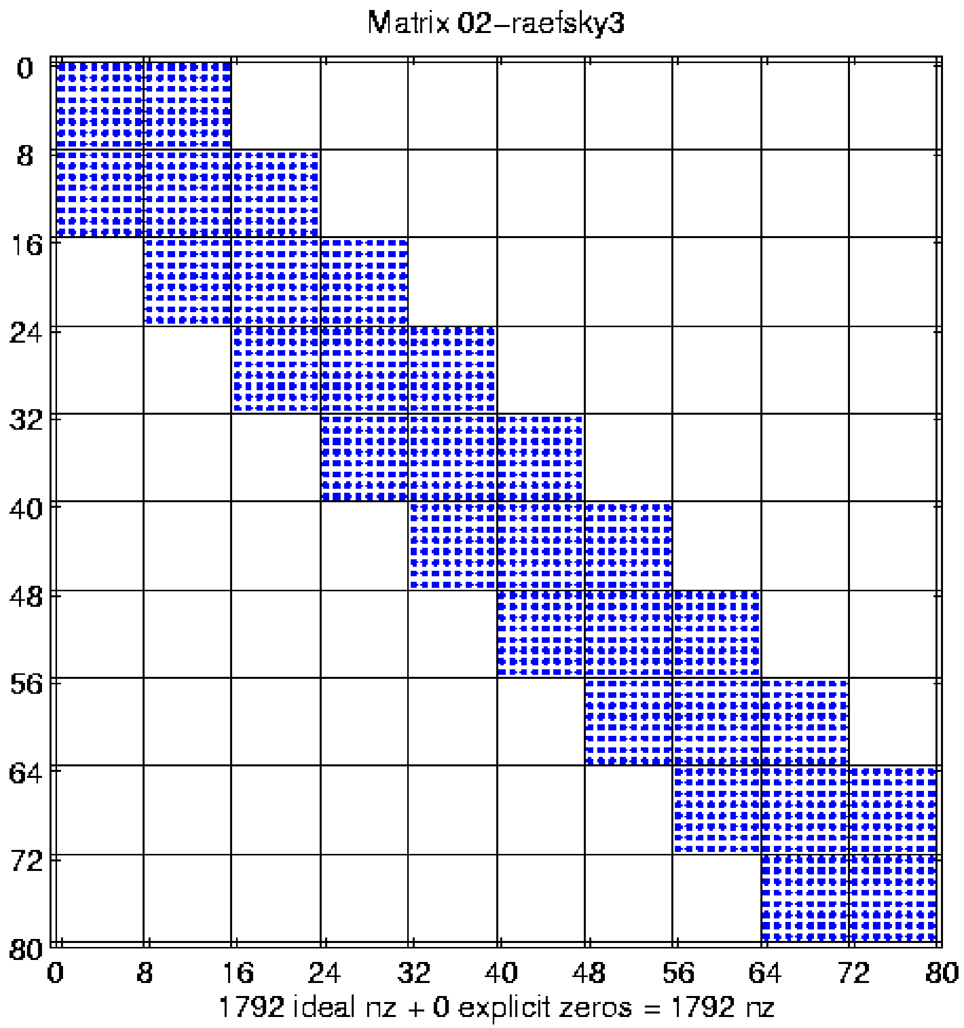
Example: The Difficulty of Tuning



- n = 21216
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem

Example: The Difficulty of Tuning



- n = 21216
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem

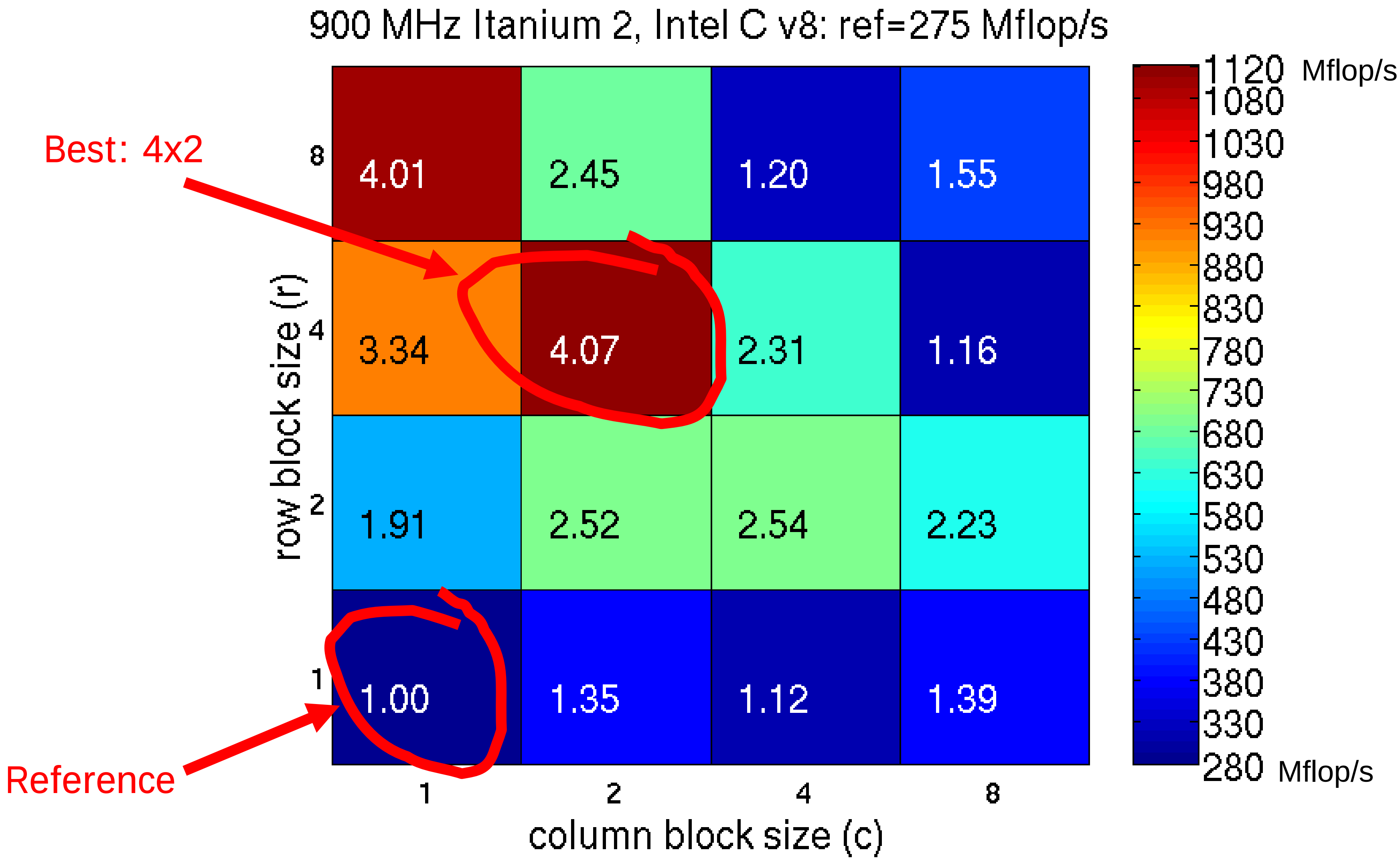
- **8x8** dense substructure

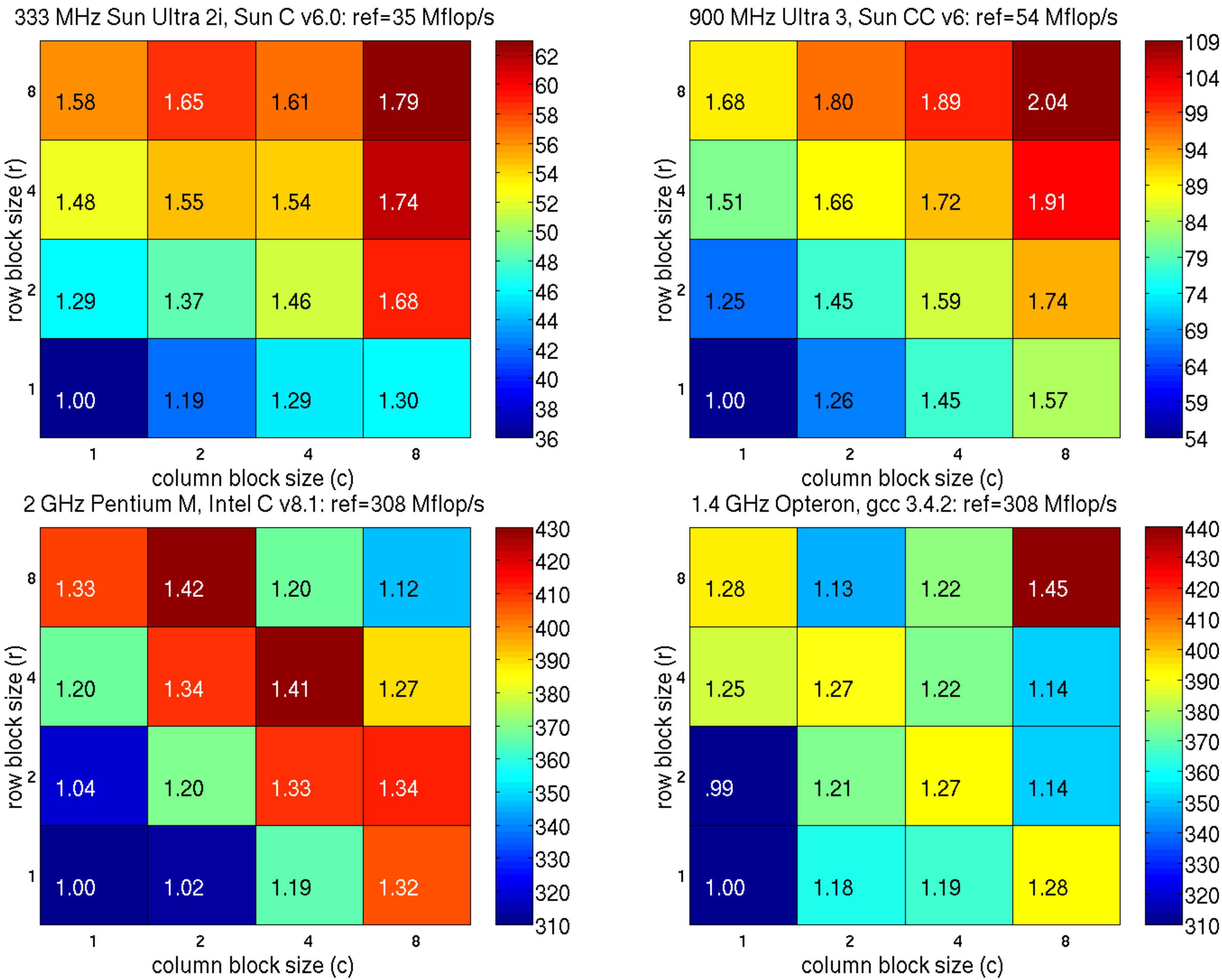
What We Expect

- Assume
  - Cost(SpMV) = time to read matrix
  - 1 double-word = 2 integers
  - r, c in {1, 2, 4, 8}
- CSR: 1 int / non-zero
- BCSR(r x c): 1 int / (r*c non-zeros)
- As r*c increases, speedup should
  - Increase smoothly
  - Approach 1.5

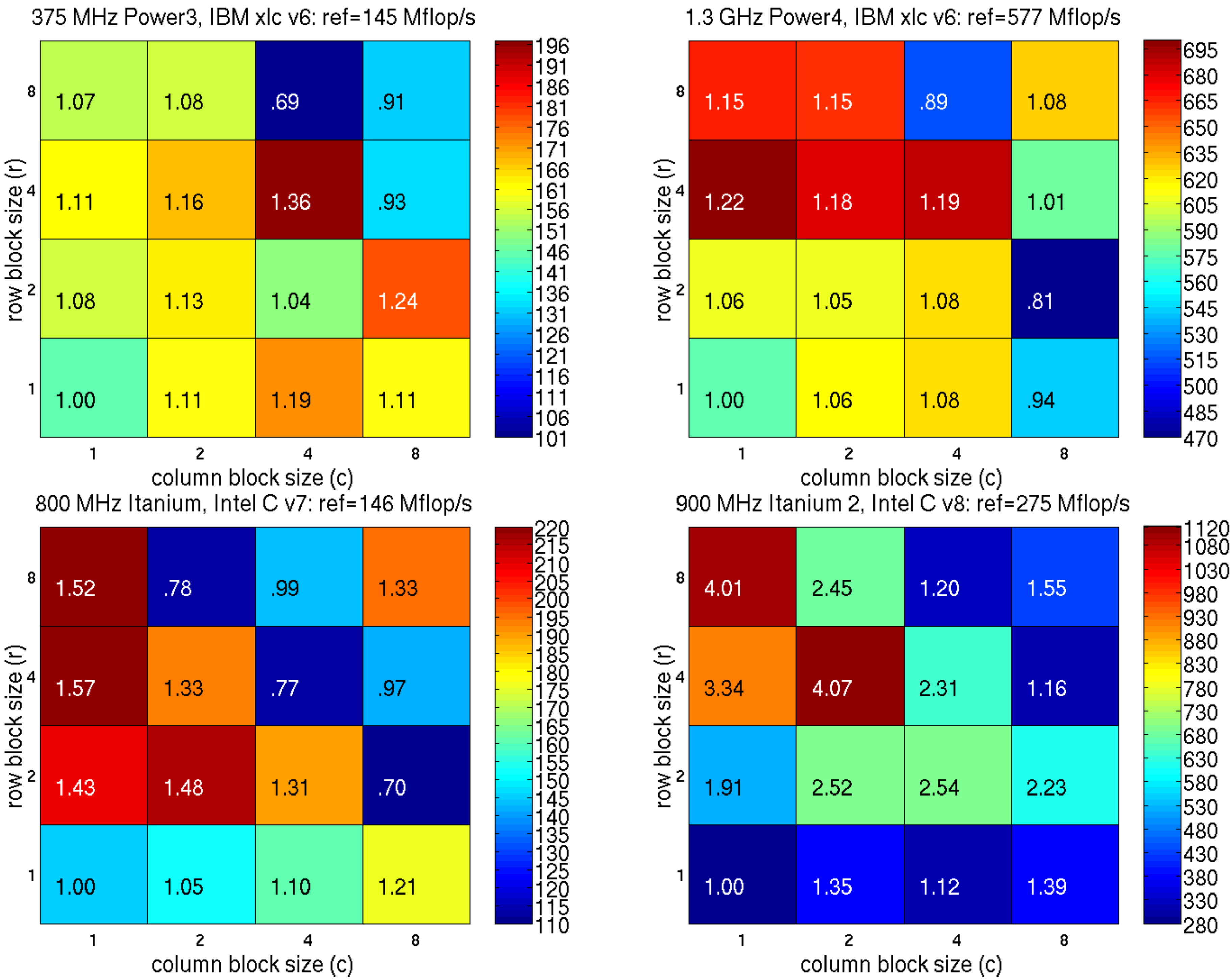
$$Speedup = \frac{T_{CSR}}{T_{BCSR}(r,c)} \approx \frac{1.5}{1+\frac{1}{rc}} \xrightarrow{r,c=\infty} 1.5$$

What We Get (The Need for Search)

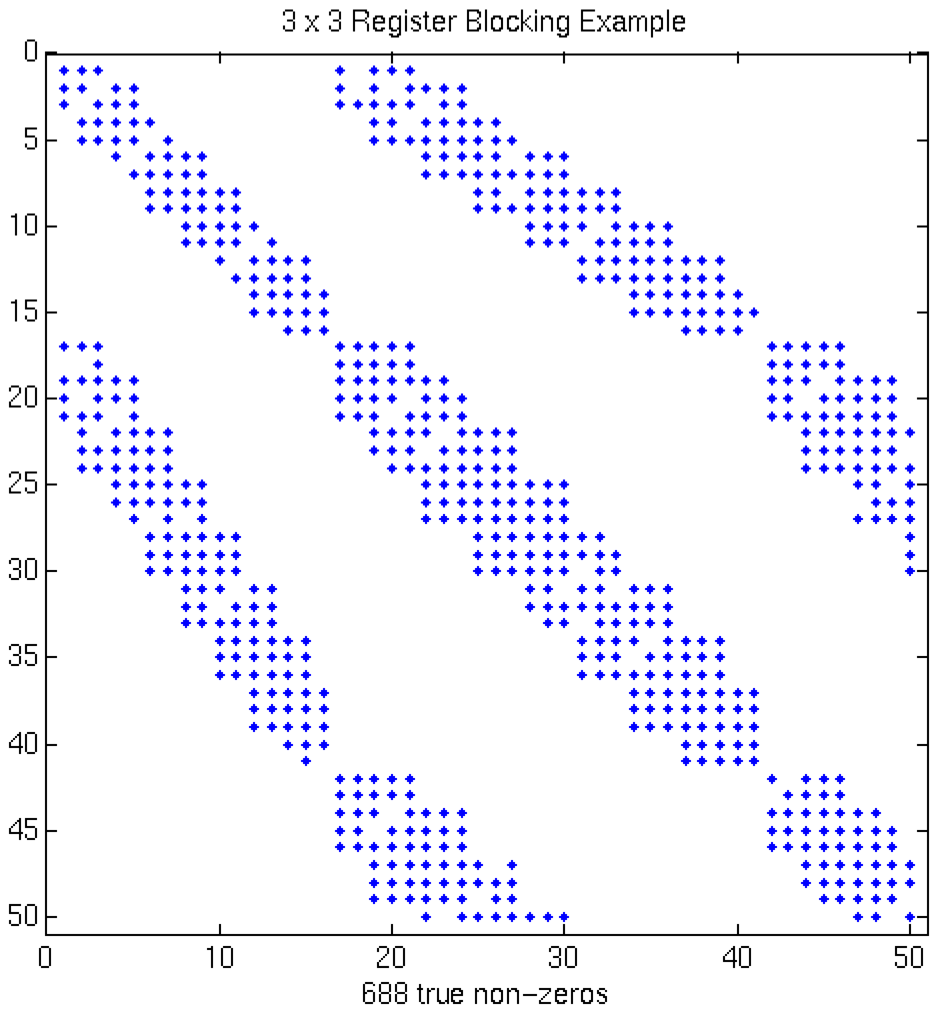




333 MHz Sun Ultra 2i, Sun C v6.0: ref=35 Mflop/s
1.58 1.65 1.61 1.79
1.48 1.55 1.54 1.74
1.29 1.37 1.46 1.68
1.00 1.19 1.29 1.30
row block size (r)
column block size (c)
900 MHz Ultra 3, Sun CC v6: ref=54 Mflop/s
1.68 1.80 1.89 2.04
1.51 1.66 1.72 1.91
1.25 1.45 1.59 1.74
1.00 1.26 1.45 1.57
2 GHz Pentium M, Intel C v8.1: ref=308 Mflop/s
1.33 1.42 1.20 1.12
1.20 1.34 1.41 1.27
1.04 1.20 1.33 1.34
1.00 1.02 1.19 1.32
1.4 GHz Opteron, gcc 3.4.2: ref=308 Mflop/s
1.28 1.13 1.22 1.45
1.25 1.27 1.22 1.14
.99 1.21 1.27 1.14
1.00 1.18 1.19 1.28

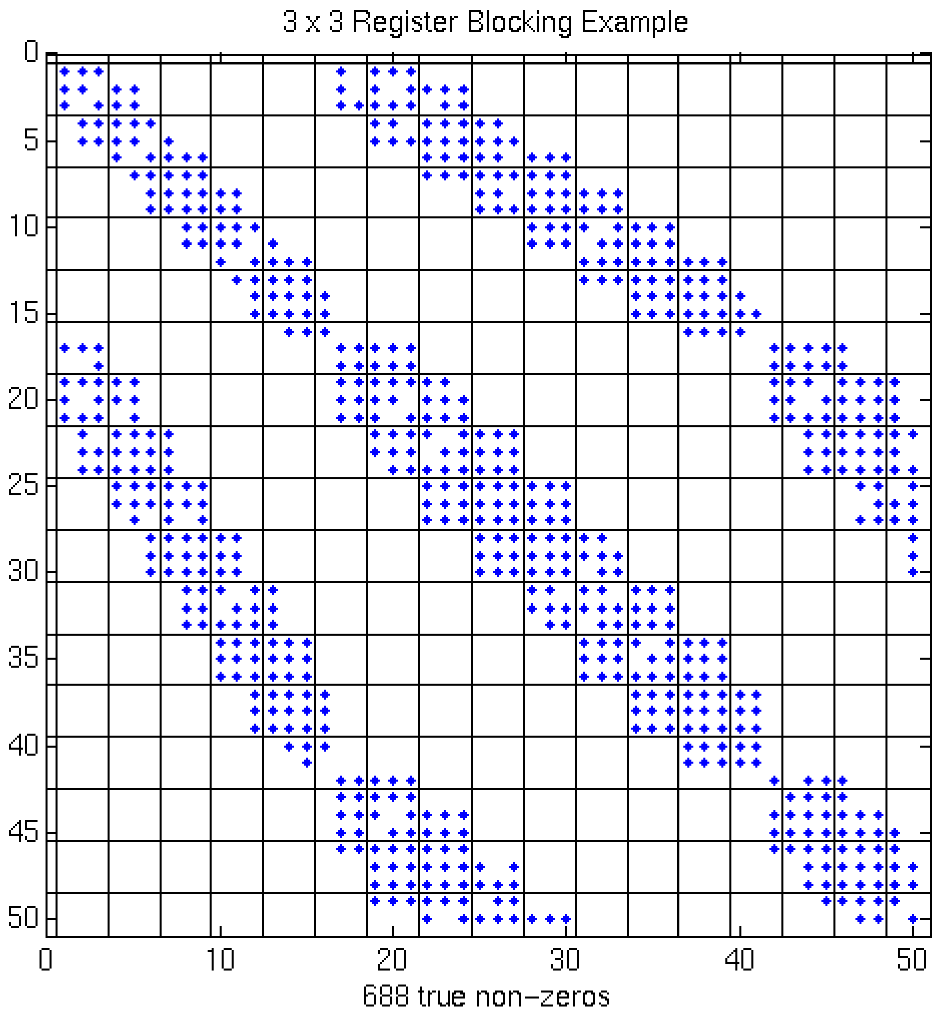

375 MHz Power3, IBM xlc v6: ref=145 Mflop/s
row block size (r)
column block size (c)
1.07
1.08
.69
.91
1.11
1.16
1.36
.93
1.08
1.13
1.04
1.24
1.00
1.11
1.19
1.11
1.3 GHz Power4, IBM xlc v6: ref=577 Mflop/s
1.15
1.15
.89
1.08
1.22
1.18
1.19
1.01
1.06
1.05
1.08
.81
1.00
1.06
1.08
.94
800 MHz Itanium, Intel C v7: ref=146 Mflop/s
1.52
.78
.99
1.33
1.57
1.33
.77
.97
1.43
1.48
1.31
.70
1.00
1.05
1.10
1.21
900 MHz Itanium 2, Intel C v8: ref=275 Mflop/s
4.01
2.45
1.20
1.55
3.34
4.07
2.31
1.16
1.91
2.52
2.54
2.23
1.00
1.35
1.12
1.39

Still More Surprises



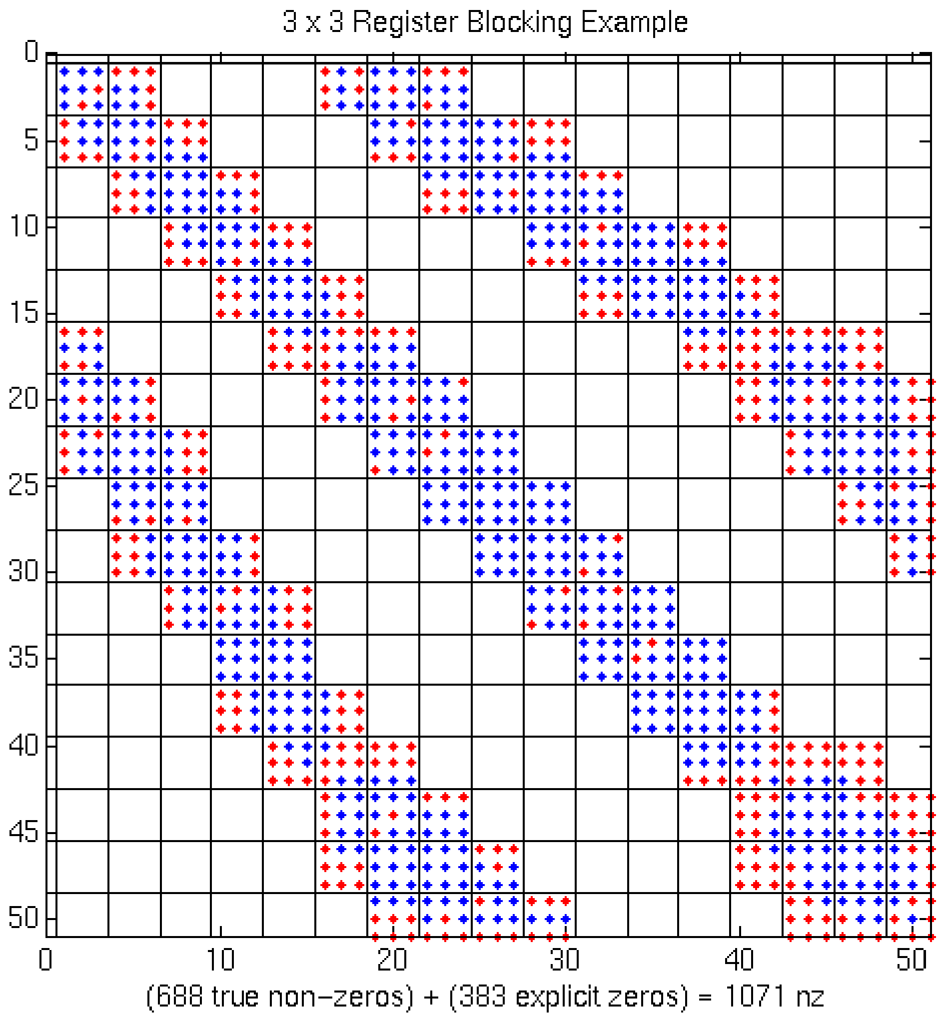
- More complicated non-zero structure in general

Still More Surprises



- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells

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
- On Pentium III: 1.5x speedup!

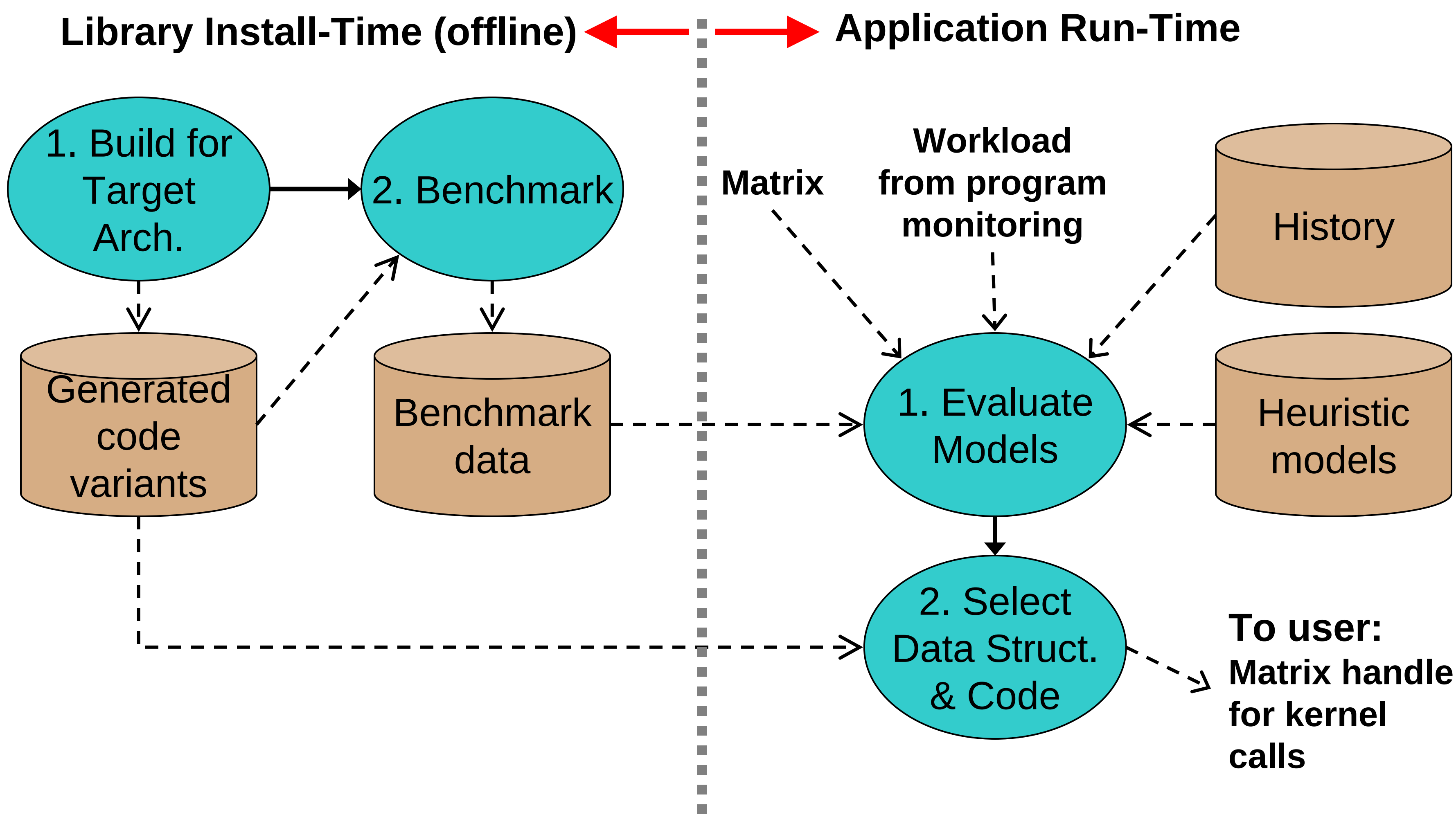
Historical Trends: Mixed News

- Observations
  - ++ Moore's law like behavior
  - **----** "Untuned" is 10% peak or less, worsening
  - ++ "Tuned" roughly 2x better today, and growing
  - **----** Tuning is complex
- LINPACK not representative of sparse apps

Road Map

- Sparse matrix-vector multiply (SpMV) in a nutshell
- Historical trends and the need for search
- **Automatic tuning in OSKI**
  - **How does OSKI work?**
- Current and future work

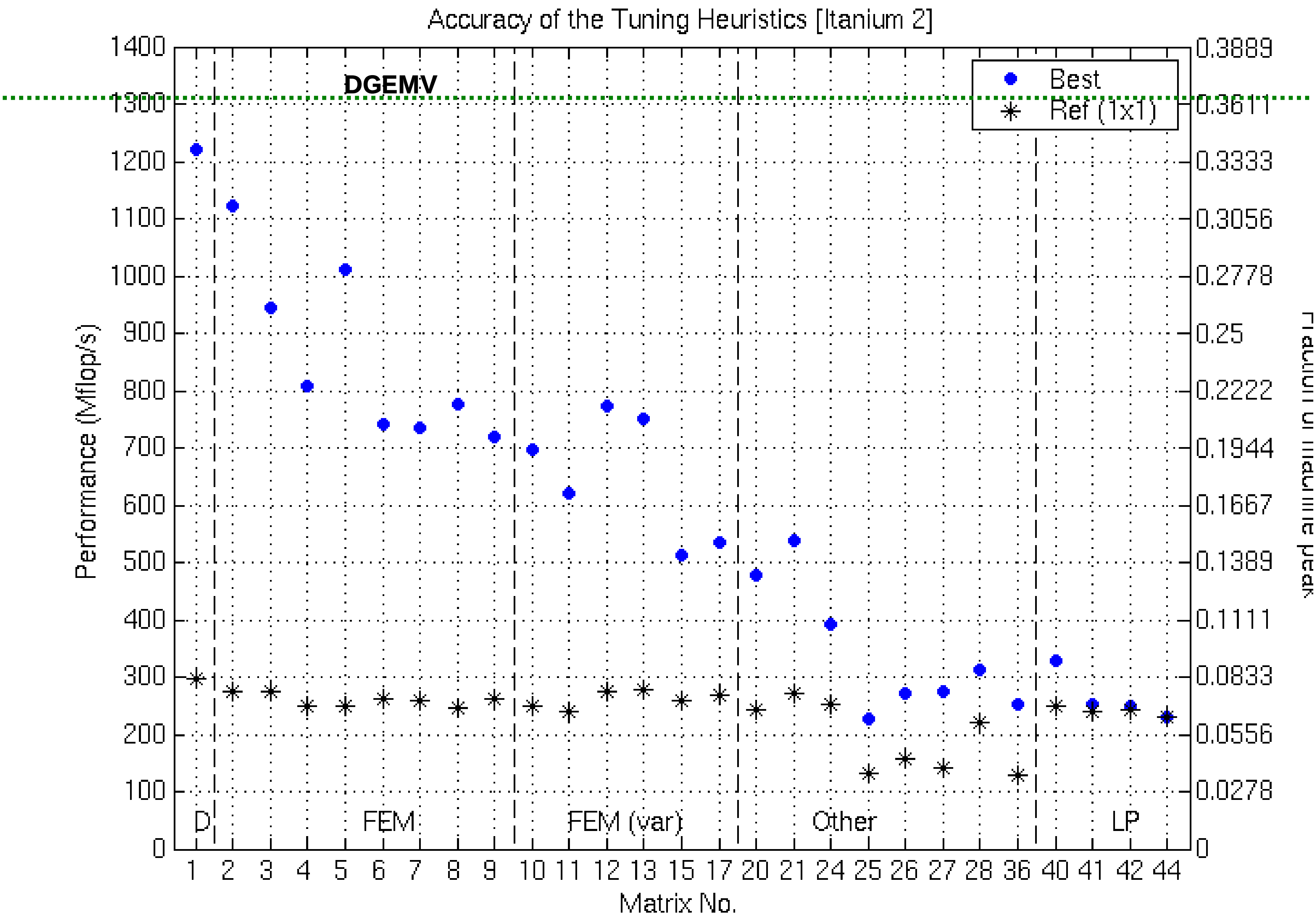
How OSKI Tunes (Overview)



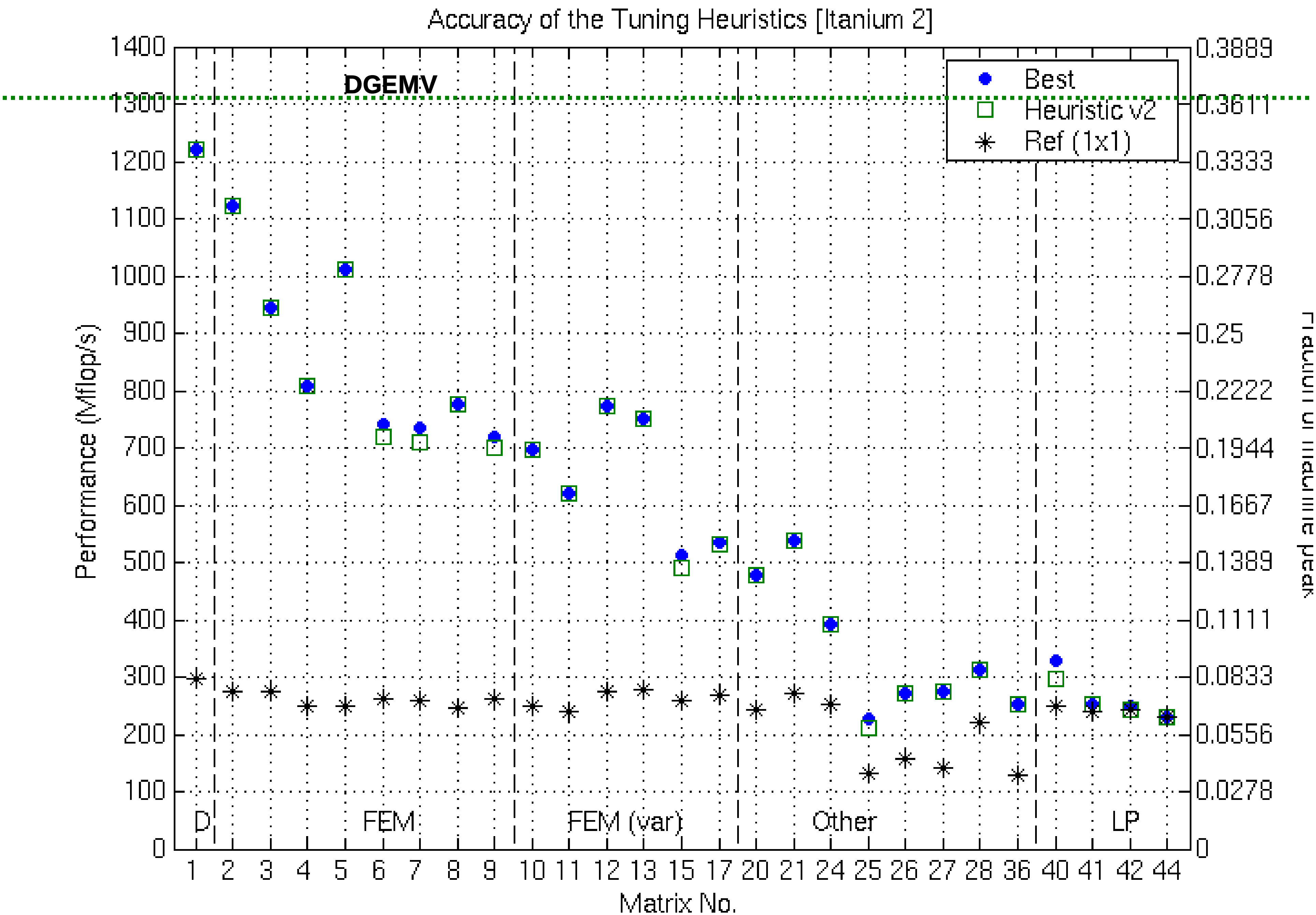
Extensibility: Advanced users may write & dynamically add "Code variants" and "Heuristic models" to system.

Example of a Tuning Heuristic

- Example: Selecting the r x c block size
  - **Off-line benchmark: characterize the machine**
    - Precompute **Mflops(r,c)** using dense matrix for each r x c
    - Once per machine/architecture
  - **Run-time "search": characterize the matrix**
    - Sample *A* to estimate **Fill(r,c)** for each r x c
  - **Run-time heuristic model**
    - Choose r, c to maximize **Mflops(r,c)** / **Fill(r,c)**
- Run-time costs
  - Up to ~40 SpMVs (empirical worst case)
  - Dominated by conversion
  - May be amortized if pattern fixed



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



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

Calling OSKI: Interface Design

- Support “legacy applications”
  - Gradual migration of apps to use OSKI
- Must call “tune” routine explicitly
  - Exposes cost of tuning and data structure reorganization
- May provide tuning hints
  - Structural: Hints about matrix
  - Workload: Hints about frequency of calls, to limit tuning time
- May save/restore tuning results
  - To apply on future runs with similar matrix
  - Stored in “human-readable” format

How to Call OSKI: Basic Usage

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

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

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

How to Call OSKI: Basic Usage

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

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
/* Step 1: Create OSKI wrappers around this data */
oski_matrix_t A_tunable = oski_CreateMatCSR(ptr, ind, val, num_rows,
   num_cols, SHARE_INPUTMAT, …);
oski_vecview_t x_view = oski_CreateVecView(x, num_cols, UNIT_STRIDE);
oski_vecview_t y_view = oski_CreateVecView(y, num_rows, UNIT_STRIDE);

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

How to Call OSKI: Basic Usage

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

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix, in CSR format */
double* x = …, *y = …; /* Let x and y be two dense vectors */
/* Step 1: Create OSKI wrappers around this data */
oski_matrix_t A_tunable = oski_CreateMatCSR(ptr, ind, val, num_rows,
   num_cols, SHARE_INPUTMAT, …);
oski_vecview_t x_view = oski_CreateVecView(x, num_cols, UNIT_STRIDE);
oski_vecview_t y_view = oski_CreateVecView(y, num_rows, UNIT_STRIDE);

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

How to Call OSKI: Tune with Explicit Hints

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

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

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

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

How the User Calls OSKI: Implicit Tuning

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

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

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

Saving and Restoring Tuning Transformations

- May selecting customized, complex transformations using embedded scripting language (OSKI-Lua)

```
/* 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_fast = P*A*P^T using
   Pinar's reordering algorithm
A_fast, P =
   reorder_TSP(InputMat);
# Split A_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;
```

Additional Features

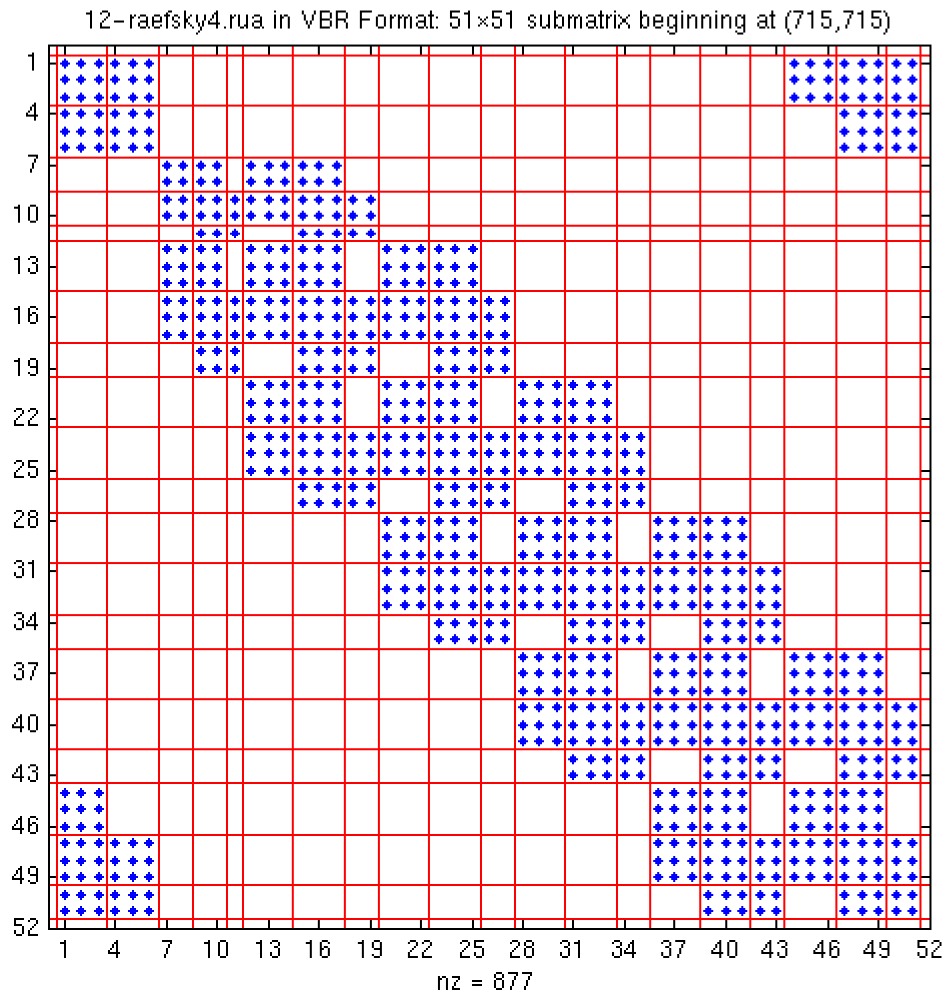
- Currently 5 tunable kernels
  - SpMV, triangular solve, $A \cdot x$ & $A^T \cdot w$, $A^T A \cdot x$, $A^k \cdot x$
- Support for several scalar type combinations
  - {32-bit, 64-bit int} x {single, double prec.} x {real, complex}
- "Plug-in" extensibility
  - Very advanced users may customize library (at run-time)
    - New heuristics (*e.g.*, Buttari, et al.)
    - Alternative data structures & code variants (*e.g.*, seg-scan for vector architectures)

Exploiting Problem-Specific Structure

- Optimizations for SpMV
  - **Register blocking (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 (2.2x over CSR)**
  - Multiple vectors (7x over CSR)
  - And combinations…
- Sparse triangular solve
  - **Hybrid sparse/dense data structure (1.8x over CSR)**
- Higher-level kernels
  - **$AA^T \cdot x$, $A^TA \cdot x$ (4x over CSR, 1.8x over RB)**
  - **$A^2 \cdot x$ (2x over CSR, 1.5x over RB)**

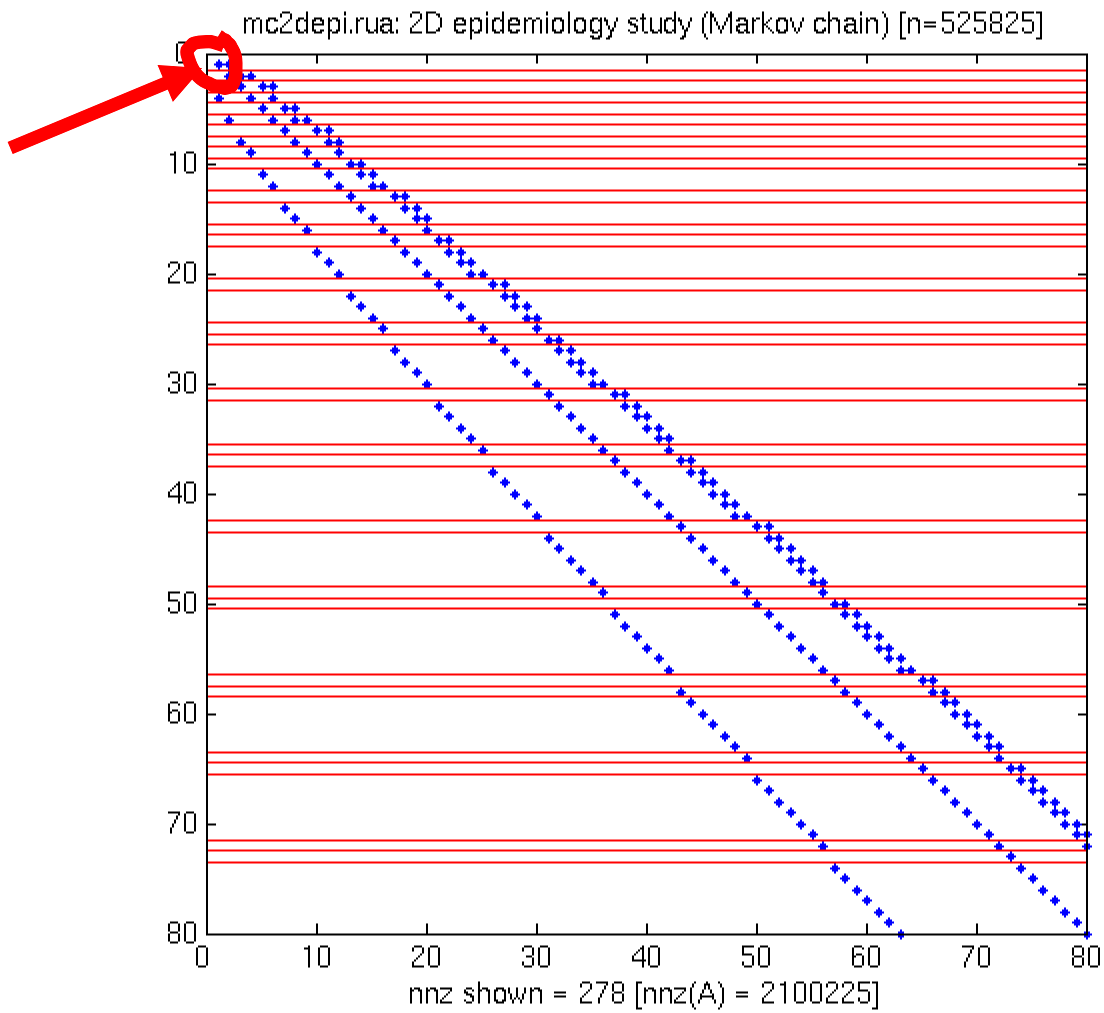
Berkeley Benchmarking and OPtimization Group
bebop.cs.berkeley.edu

Example: Variable Block Structure



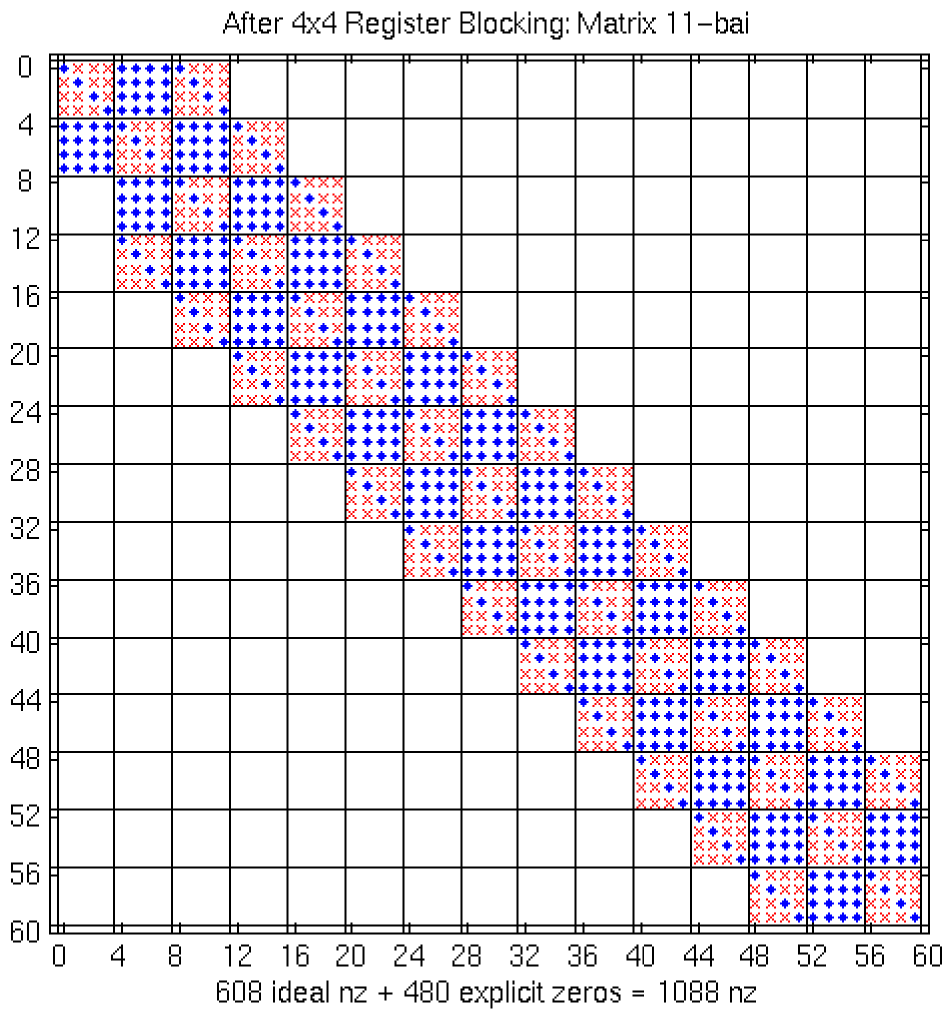
2.1x
over CSR
1.8x
over RB

Example: Row-Segmented Diagonals

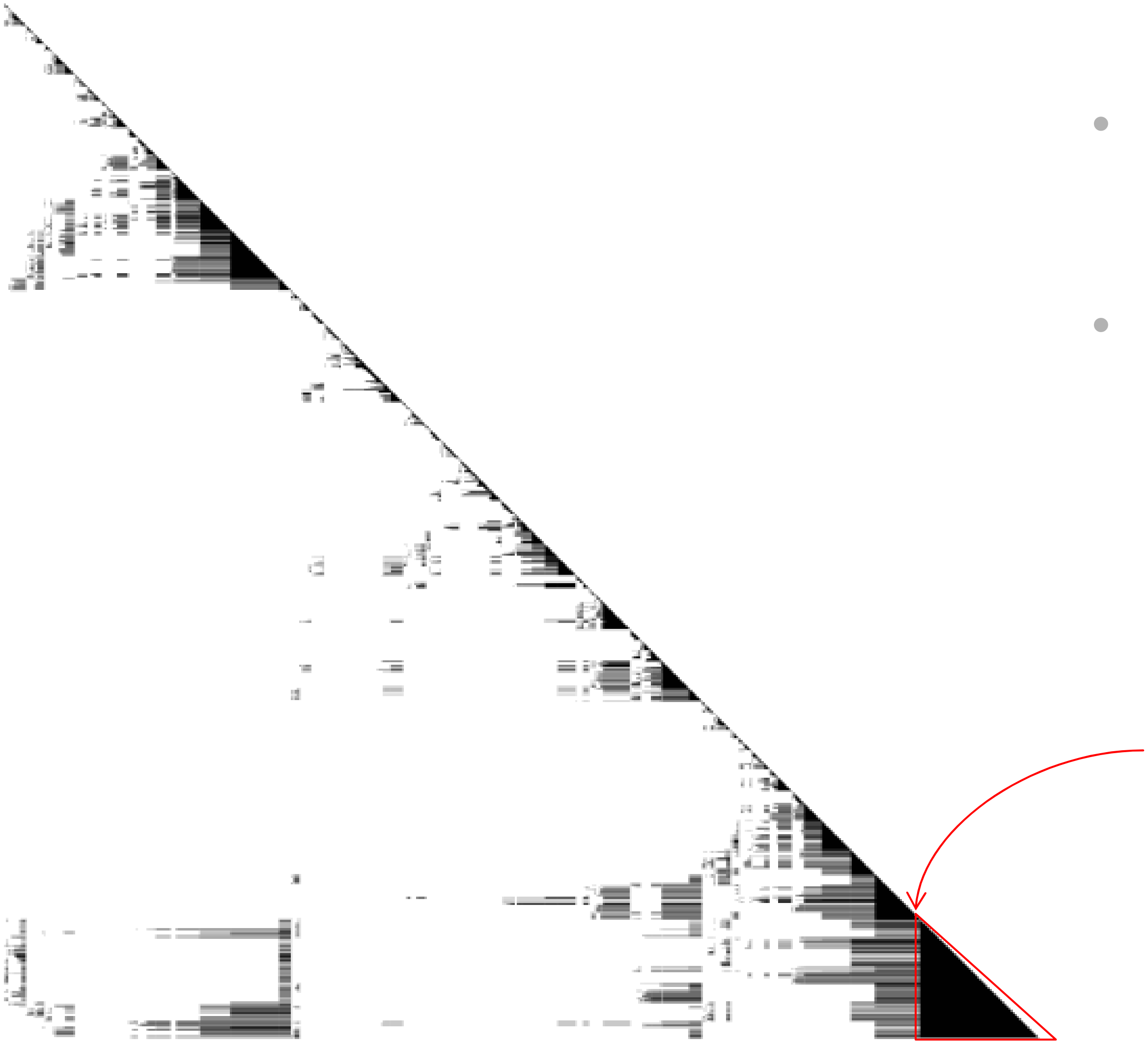


2x over CSR

Mixed Diagonal and Block Structure



Example: Sparse Triangular Factor



- Raefsky4 (structural problem) + SuperLU + colmmd
- N=19779, nnz=12.6 M

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

Can be as high as 90+%!

1.8x over CSR

Cache Optimizations for $AA^T{*}x$

- **Cache-level**: **Interleave** multiplication by A, A^T
- Sparse matrix-vector multiply (SpMV) in a nutshell
- Historical trends and the need for search
- Automatic tuning in OSKI
- **Current and future work**

$$AA^T \cdot x = (a_1 \, \Lambda \, a_n) \begin{pmatrix} a_1^T \\ \mathrm{M} \\ a_n^T \end{pmatrix} \cdot x = \sum_{i=1}^{n} a_i (a_i^T x)$$

"axpy"

dot product

- **Register-level**: a_i^T to be $r \times c$ block row, or diag row
- Algorithmic-level transformations for $A^2{*}x$, $A^3{*}x$, …

Example applications

- T3P – Accelerator Design – Ko
  - Register blocking, Symmetric Storage, Multiple vector
  - 1.68x faster on Itanium 2 for one vector
  - 4.4x faster for 8 vectors
- Omega3P – Accelerator Design – Ko
  - Register blocking, Symmetric storage, Reordering
  - 2.1x faster on Power4
- Semiconductor Industry:
  - 1.9x speedup over SPOOLES in CG at design firm
- Recent integration of OSKI into PETSc

Status and Future Work

- OSKI Release 1.0 and docs available

  `bebop.cs.berkeley.edu/oski`
- Performance bounds modeling (ongoing)
- Future OSKI work
  - Release of PETSc version with OSKI
  - Better “low-level” tuning, including vectors
  - Automatically tuned parallel sparse kernels
- Development of a new HPC Challenge Benchmark
  - Evaluate platforms based on tuned (blocked) SpMV performance
- Tuning higher level algorithms using $A^k x$
  - Models indicate large speedups possible

Current SPMV OSKI Code Generator

```
#!/bin/bash
#
# This script uses some bash extensions.
#

mattype=BCSR

if test x"$1" = x ; then
        echo ""
        echo "usage: $0 {full, source, makestub}"
        echo ""
        exit 1
fi

GENSOURCE="
GENMAKE="
case $1 in
[fF]*) GENSOURCE=yes ; GENMAKE=yes ;;
[sS]*) GENSOURCE=yes ; GENMAKE=no ;;
[mM]*) GENSOURCE=no ; GENMAKE=yes ;;
*) echo "*** Unknown option, '$1' ***" ; exit 1 ;;
esac

CreateOutfile() {
#-------------------------------------------------------------
# args:  <R> <C> <outfile>
#

R=$1
C=$2
outfile=$3

echo "/**
 *  \\file ${mattype}_${R}x${C}.c
 *  \\brief ${mattype} ${R}x${C} SpMV implementation, for all transpose
        options.
 *  \\ingroup MATTYPE_${mattype}
 *
 *  Automatically generated by $0 on `date`.
 */
```

```
if test ${GENMAKE} = yes ; then
        makestub=Make.${mattype}
        echo "#
# Automatically generated by $USER@`hostname`
# on `date`, running $0
#
" > ${makestub}
fi

for R in 1 2 3 4 5 6 7 8 ; do # row block size
for C in 1 2 3 4 5 6 7 8 ; do # column block size

        echo "${MATTYPE} ${R}x${C}..."

        outfile=${R}x${C}.c

        if test ${GENSOURCE} = yes ; then
                CreateOutfile ${R} ${C} ${outfile}

                for OP in normal trans conj herm ; do # transpose option
                for S in 1 general ; do  # stride
                                WriteKernel ${R} ${C} ${OP} ${S} ${outfile}
                done # S

                WriteShell_v1 ${OP} ${outfile}
                WriteShell ${OP} ${R} ${C} ${outfile}

                done # OP

                WriteMatReprMult ${R} ${C} ${outfile}
                WriteFooter ${outfile}
        fi


        if test ${GENMAKE} = yes ; then
                WriteMakeStub ${R} ${C} ${makestub}
        fi

done # C
done # R
exit 0
# eof
```

750 lines total

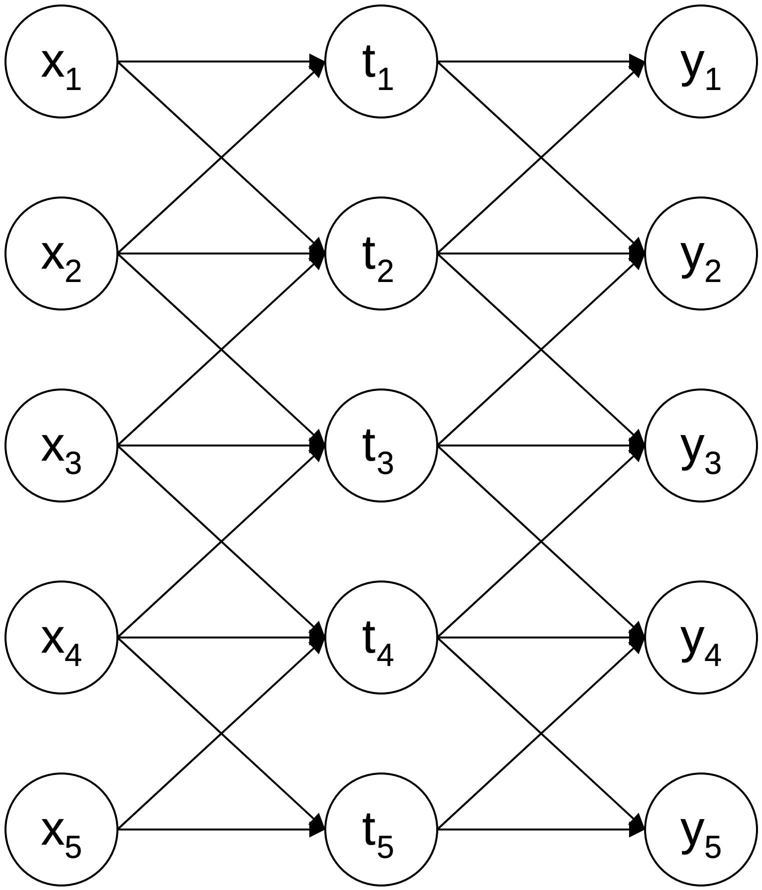
Project: Improved Code Generation

- Consider common kernels:
  - Matrix-vector multiply, triangular solve, etc.
- Different emphasis than Bernoulli
  - These are simpler kernels than they were interested in
  - Generate code for many formats, not fixed by programmer
  - Select between them using
    - Performance models
    - Search
- Approach may still apply
  - Use high level language (Matlab?) to "specify" kernels
  - Separate language to specify matrix format

Project Idea: Inter Iteration Tiling

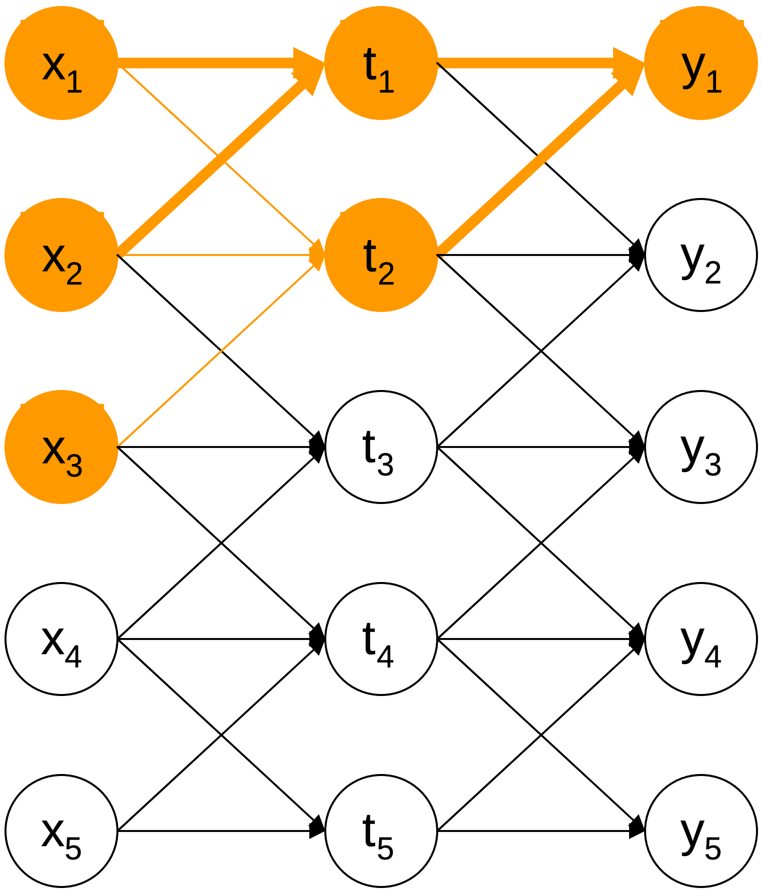
- $A^2 * x$ is done in Rich Vuduc's PhD thesis
- General case in Michelle Strout's thesis
- Code generation technology would be useful

Inter-Iteration Sparse Tiling (1/3)



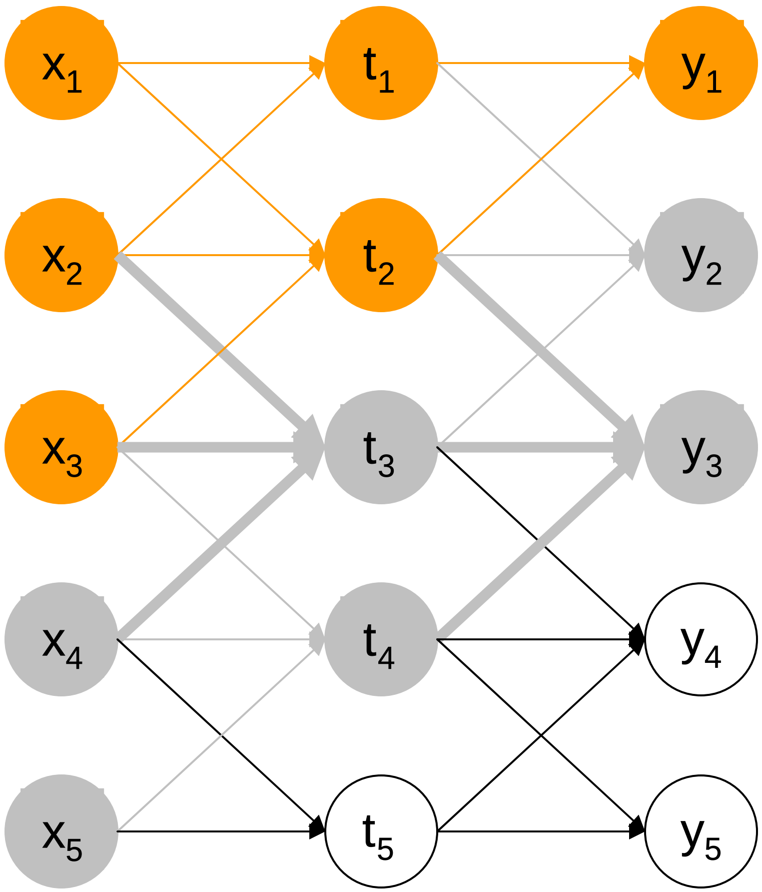
- Let A be 6x6 tridiagonal
- Consider $y=A^2x$
  - $t=Ax$, $y=At$
- Nodes: vector elements
- Edges: matrix elements a_{ij}

Inter-Iteration Sparse Tiling (2/3)



- Let A be 6x6 tridiagonal
- Consider $y=A^2x$
  - $t=Ax$, $y=At$
- 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)

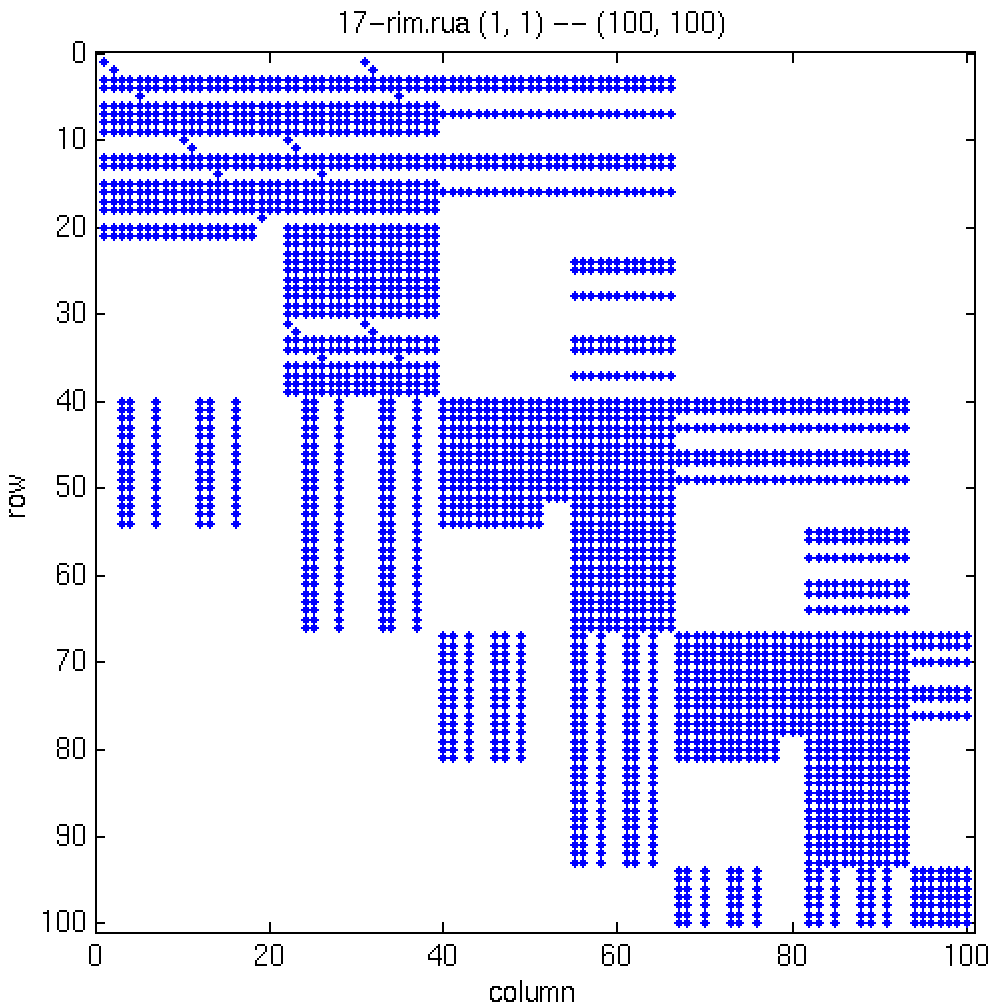


- Let A be 6x6 tridiagonal
- Consider $y=A^2x$
  - $t=Ax$, $y=At$
- Nodes: vector elements
- Edges: matrix elements a_{ij}

- Orange = everything needed to compute y_1
  - Reuse a_{11}, a_{12}
- Grey = y_2, y_3
  - Reuse a_{23}, a_{33}, a_{43}

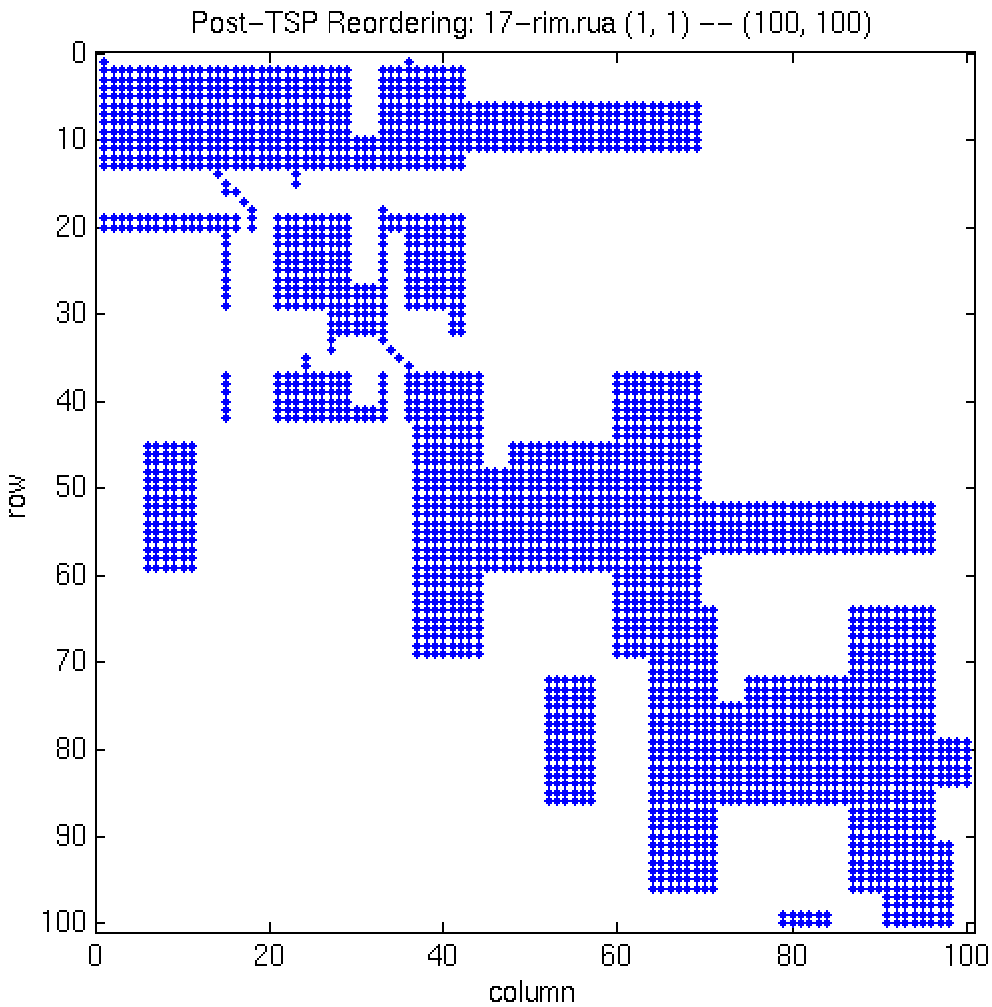
Extra slides

Creating Locality: TSP Reordering (Before)



(Pinar '97;
Moon, et al '04)

Creating Locality: TSP Reordering (After)



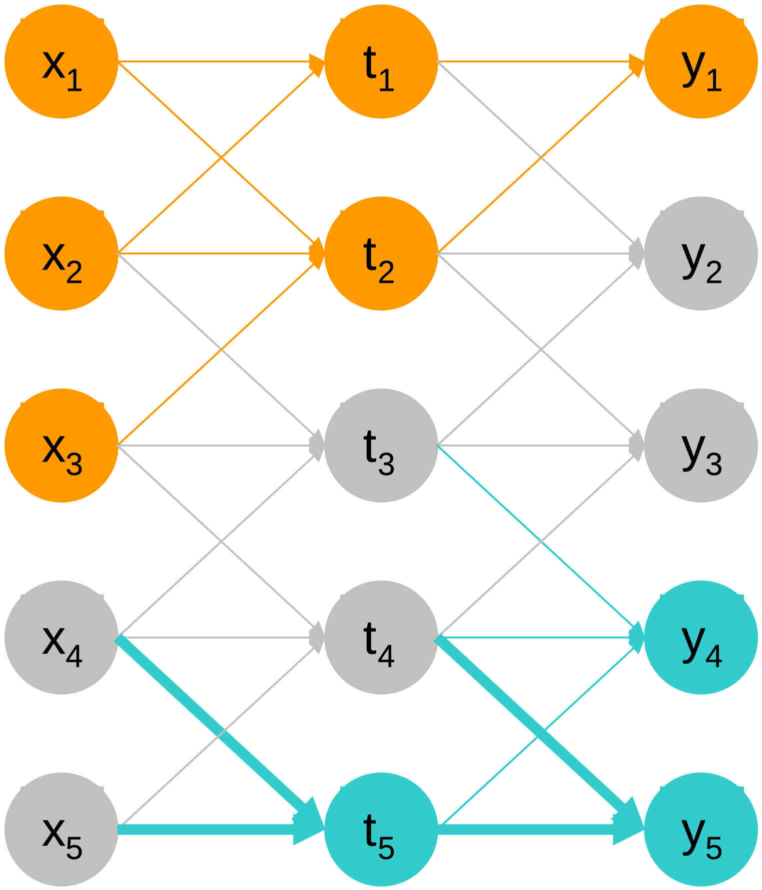
(Pinar '97; Moon, et al '04)

Up to 2x speedups over CSR

Road Map

- Sparse matrix-vector multiply (SpMV) in a nutshell
- Historical trends and the need for search
- Automatic tuning in OSKI
- **Current and future work**

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]

Splitting for Variable Blocks and Diagonals

- Decompose $A = A_1 + A_2 + \ldots A_t$
  - Detect "canonical" structures (sampling)
  - Split
  - Tune each A_i
  - Improve performance and **save storage**
- New data structures
  - Unaligned block CSR
    - Relax alignment in rows & columns
  - Row-segmented diagonals

Historical Trends in SpMV Performance

- The Data
  - Uniprocessor SpMV performance since 1987
  - “Untuned” and “Tuned” implementations
  - Cache-based superscalar micros; some vectors
  - LINPACK
    - Dense LU factorization
    - Top 500 List

Features

- Explicit Hints
  - Can suggest particular tuning technique
- Implicit Tuning: Ask library to infer workload
  - Library profiles all kernel calls
  - May periodically re-tune
- Scripting language for selecting customized transformations
  - Mechanism to save/restore transformations
- “Plug-in” extensibility
  - Very advanced users may customize library (at run-time)

Summary of High-Level Themes

- “Kernel-centric” optimization
  - Vs. basic block, trace, path optimization, for instance
  - Aggressive use of domain-specific knowledge
- Performance bounds modeling
  - Evaluating software quality
  - Architectural characterizations and consequences
- Empirical search
  - Hybrid off-line/run-time models
- Statistical performance models
  - Exploit information from sampling, measuring

Related Work

- My bibliography: 337 entries so far
- Sample area 1: Code generation
  - Generative & generic programming
  - Sparse compilers
  - Domain-specific generators
- Sample area 2: Empirical search-based tuning
  - Kernel-centric
    - linear algebra, signal processing, sorting, MPI, …
  - Compiler-centric
    - profiling + FDO, iterative compilation, superoptimizers, self-tuning compilers, continuous program optimization

Next Steps

- BeBOP Current Work
  - Public software release
  - Impact on library designs: Sparse BLAS, Trilinos, PETSc, …
  - Integration in large-scale applications
    - Accelerator design, plasma physics (DOE)
    - Geophysical simulation based on Block Lanczos (A^TA*X; LBL)
  - Systematic heuristics for data structure selection?
  - Evaluation of emerging architectures
    - Revisiting vector micros
  - Other sparse kernels
    - Matrix triple products, A^k*x
  - Parallelism

Future Directions (A Bag of Flaky Ideas)

- Composable code generators and search spaces
- New application domains
  - PageRank: multilevel block algorithms for topic-sensitive search?
- New kernels: cryptokernels
  - rich mathematical structure germane to performance; lots of hardware
- New tuning environments
  - Parallel, Grid, "whole systems"
- Statistical models of application performance
  - Statistical learning of concise parametric models from traces for architectural evaluation
  - Compiler/automatic derivation of parametric models

Acknowledgements

- Super-advisors: Jim and Kathy
- Undergraduate R.A.s: Attila, Ben, Jen, Jin, Michael, Rajesh, Shoaib, Sriram, Tuyet-Linh
- See pages *xvi—xvii* of dissertation.

Road Map

- Sparse matrix-vector multiply (SpMV) in a nutshell
- Historical trends and the need for search
- Automatic tuning techniques
- **Upper bounds on performance**
  - **SC'02**
- Statistical models of performance

Motivation for Upper Bounds Model

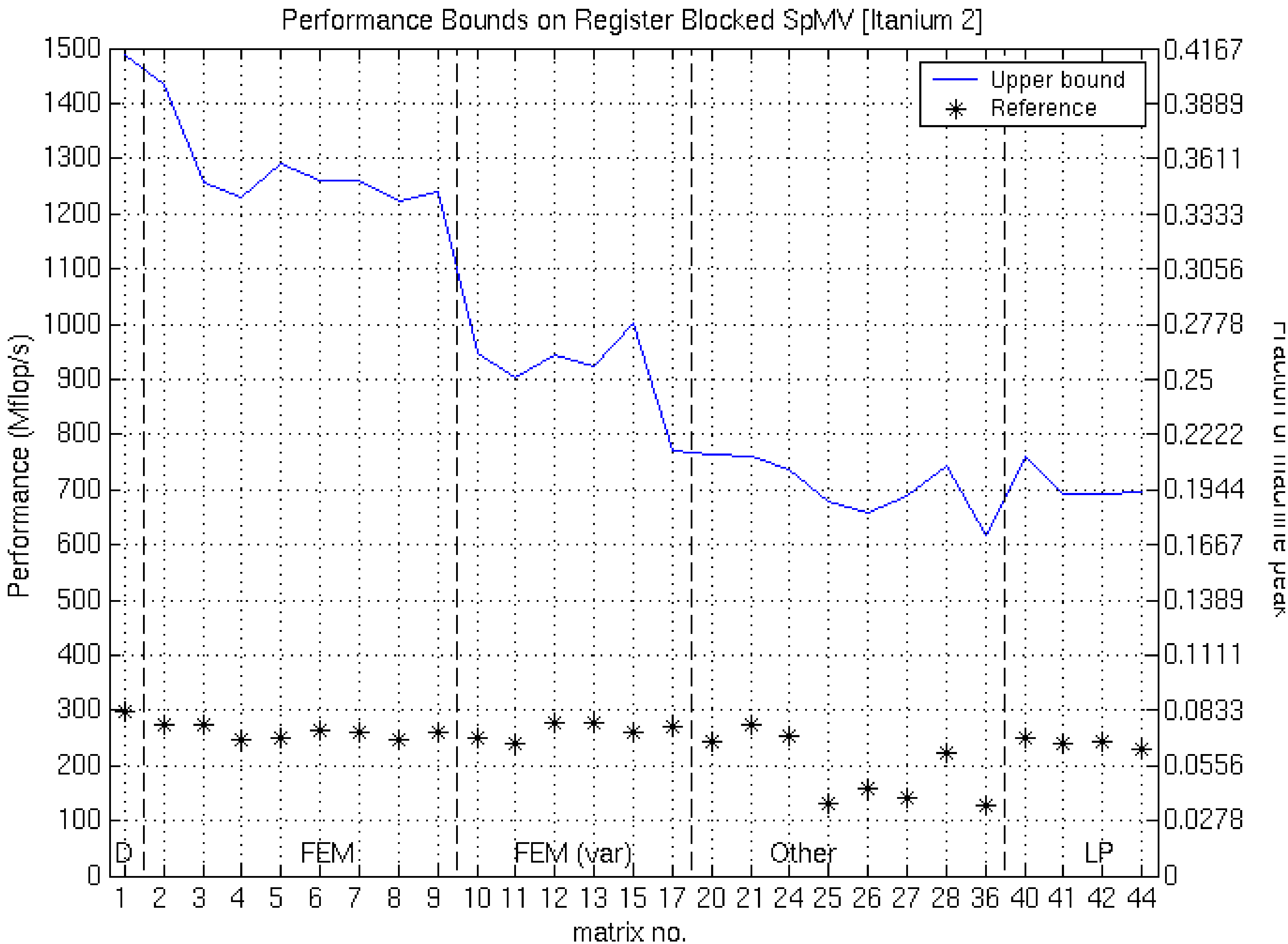
- Questions
  - Speedups are good, but what is the speed limit?
    - Independent of instruction scheduling, selection
  - What machines are “good” for SpMV?

Upper Bounds on Performance: Blocked SpMV

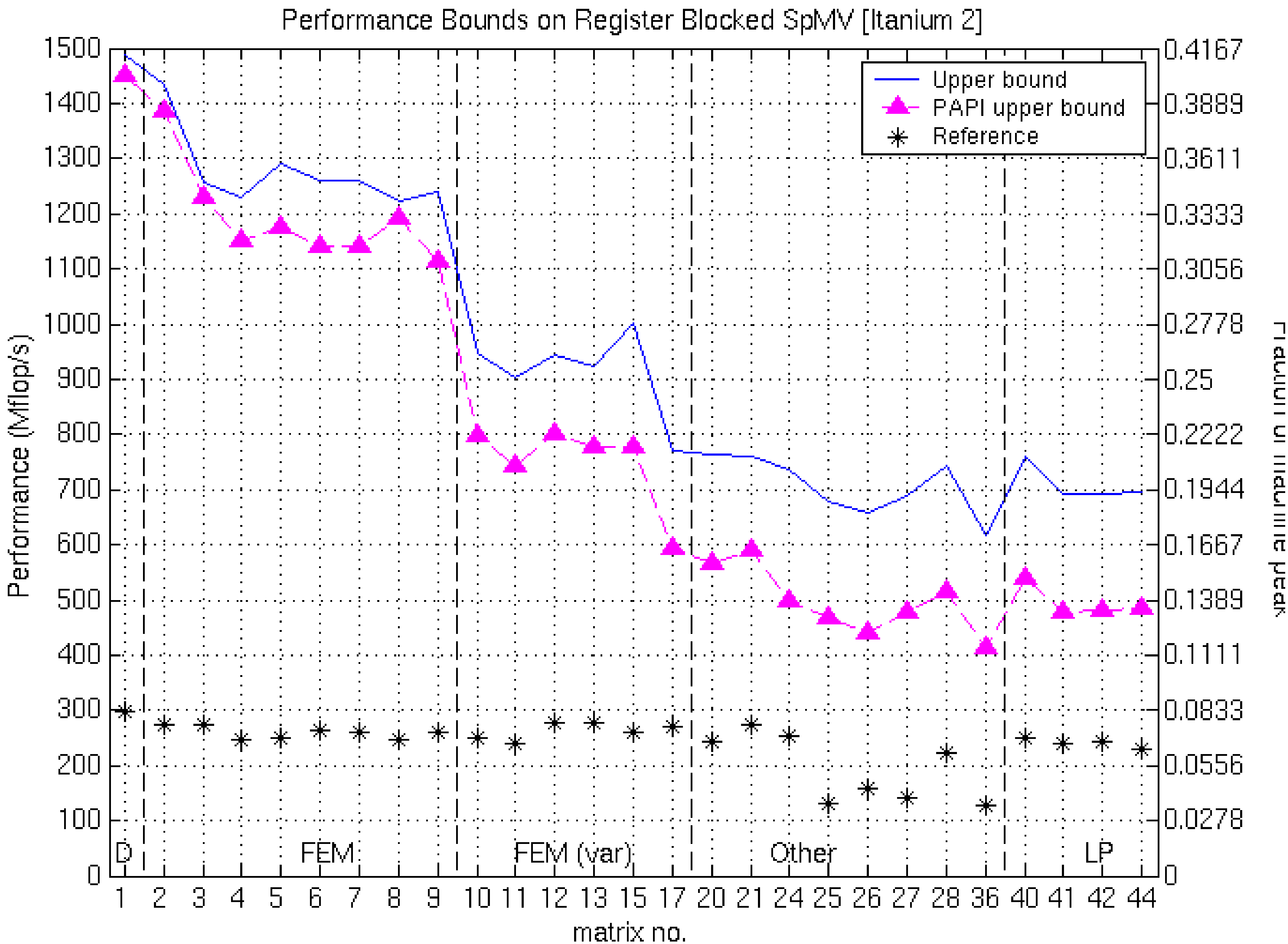
- P = (flops) / (time)
  - Flops = 2 * nnz(A)
- Lower bound on time: Two main assumptions
  - 1. Count **memory ops only** (streaming)
  - 2. Count only compulsory, capacity misses: **ignore conflicts**
    - Account for line sizes
    - Account for matrix size and nnz
- Charge min access "latency" α_i at L_i cache & α_{mem}
  - *e.g.*, Saavedra-Barrera and PMaC MAPS benchmarks

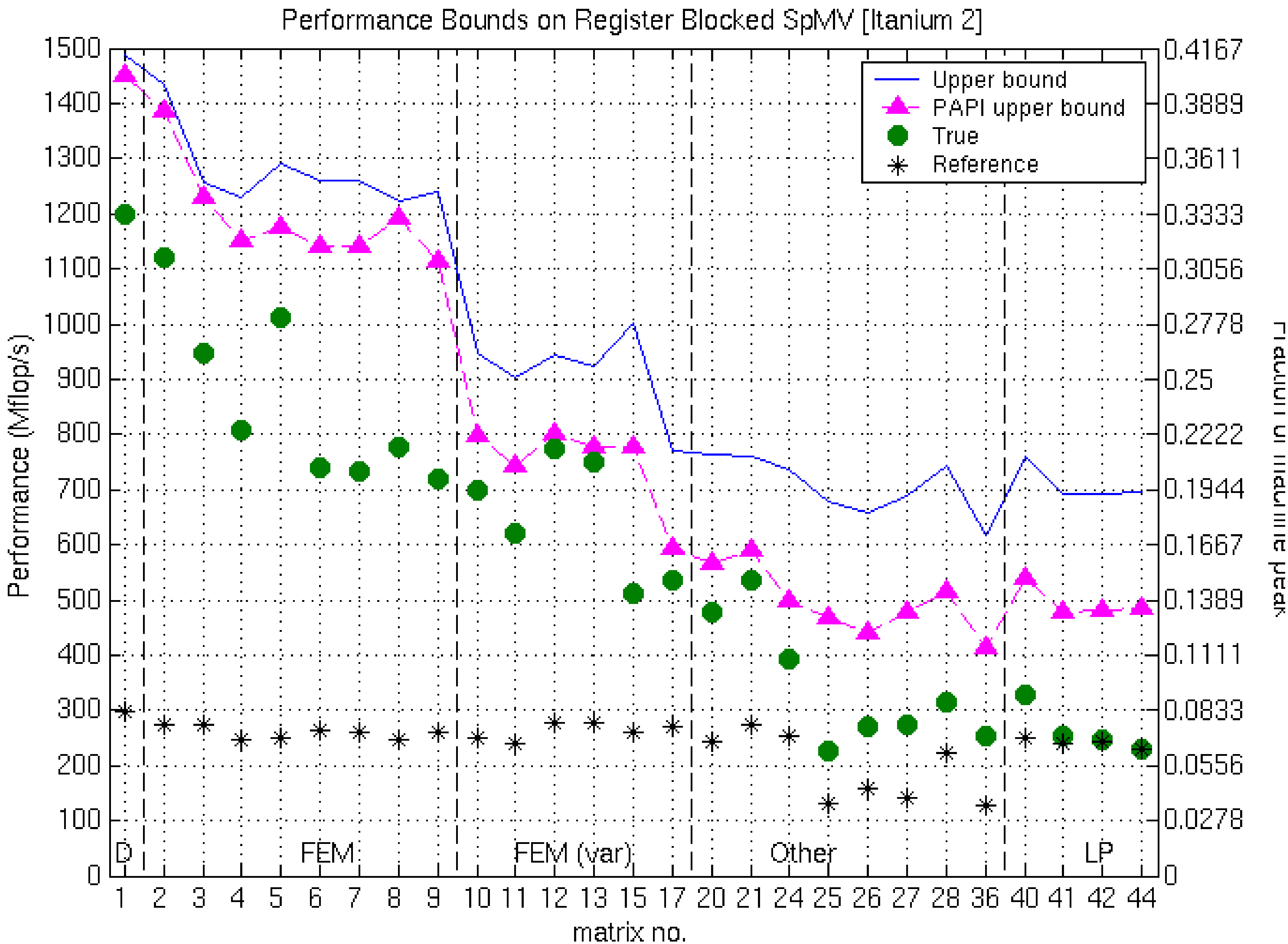
$$\text{Time} \geq \sum_{i=1}^{\kappa} \alpha_i \cdot \text{Hits}_i + \alpha_{\text{mem}} \cdot \text{Hits}_{\text{mem}}$$

$$= \alpha_1 \cdot \text{Loads} + \sum_{i=1}^{\kappa} (\alpha_{i+1} - \alpha_i) \cdot \text{Misses}_i + (\alpha_{\text{mem}} - \alpha_\kappa) \cdot \text{Misses}_\kappa$$

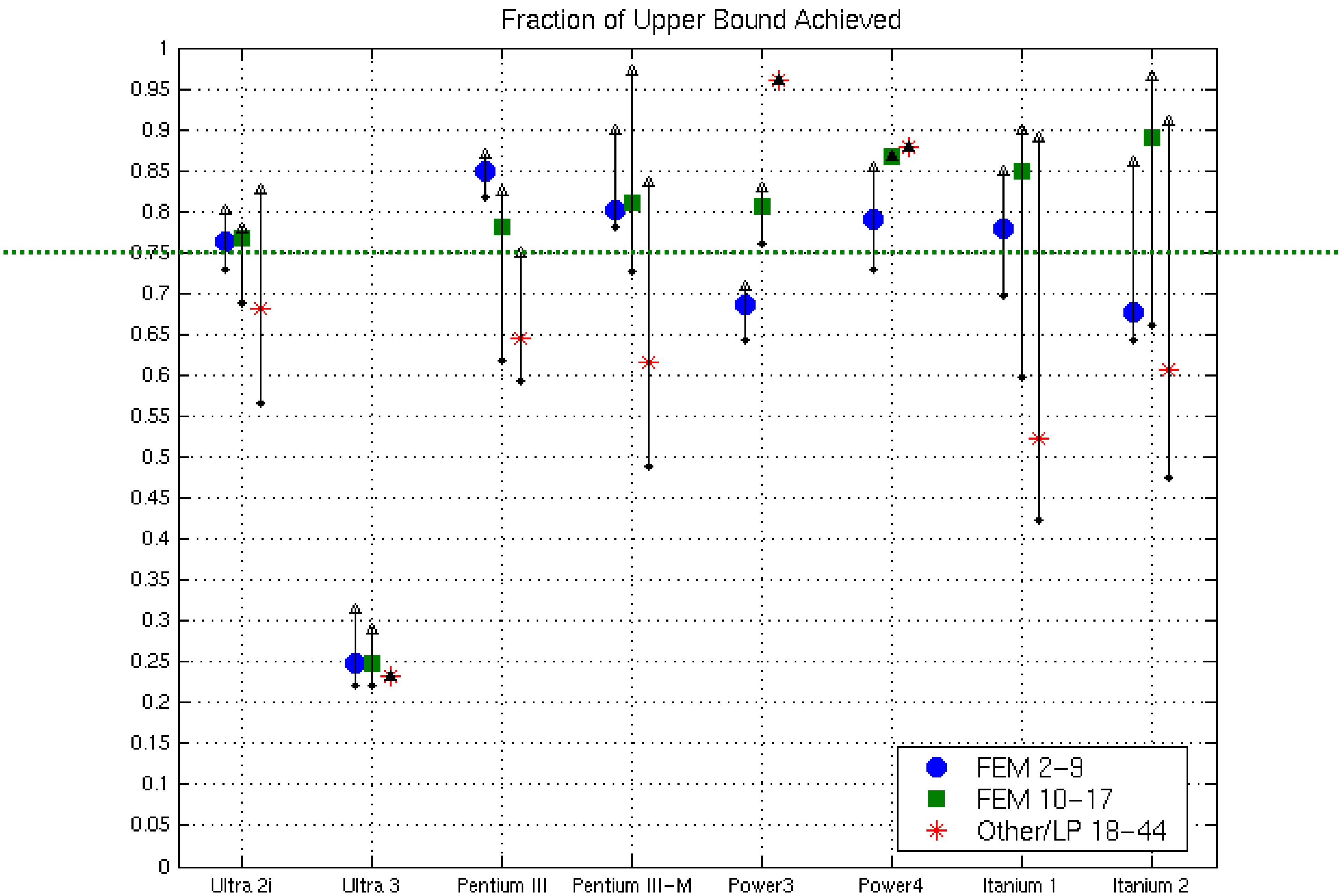


Performance Bounds on Register Blocked SpMV [Itanium 2]
Upper bound
Reference
Performance (Mflop/s)
matrix no.
D
FEM
FEM (var)
Other
LP


Performance Bounds on Register Blocked SpMV [Itanium 2]
Upper bound
PAPI upper bound
Reference
Performance (Mflop/s)
matrix no.
D
FEM
FEM (var)
Other
LP
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
0.4167
0.3889
0.3611
0.3333
0.3056
0.2778
0.25
0.2222
0.1944
0.1667
0.1389
0.1111
0.0833
0.0556
0.0278
0
1 2 3 4 5 6 7 8 9 10 11 12 13 15 17 20 21 24 25 26 27 28 36 40 41 42 44



Performance Bounds on Register Blocked SpMV [Itanium 2]
Upper bound
PAPI upper bound
True
Reference
Performance (Mflop/s)
matrix no.
D
FEM
FEM (var)
Other
LP

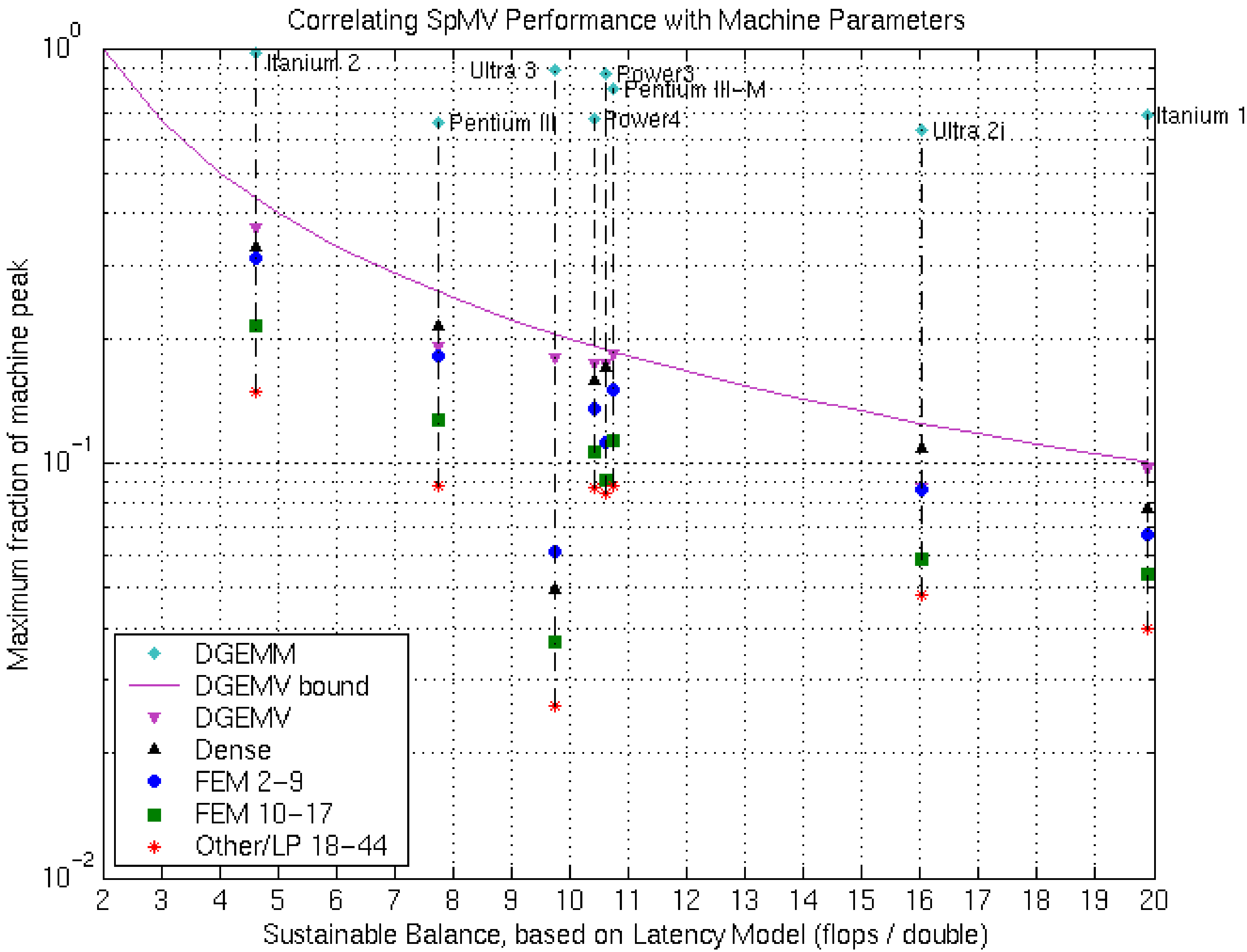

Fraction of Upper Bound Achieved
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Ultra 2i
Ultra 3
Pentium III
Pentium III−M
Power3
Power4
Itanium 1
Itanium 2
FEM 2−9
FEM 10−17
Other/LP 18−44

Achieved Performance and Machine Balance

- Machine balance [Callahan '88; McCalpin '95]
  - Balance = Peak Flop Rate / Bandwidth (flops / double)
- Ideal balance for mat-vec: ≤ 2 flops / double
  - For SpMV, even less

$$\text{Time} \geq \alpha_1 \cdot \text{Loads} + \sum_i (\alpha_{i+1} - \alpha_i) \cdot \text{Misses}_i + (\alpha_{\text{mem}} - \alpha_\kappa) \cdot \text{Misses}_\kappa$$

- SpMV ~ streaming
  - 1 / (avg load time to stream 1 array) ~ (bandwidth)
  - "Sustained" balance = peak flops / model bandwidth

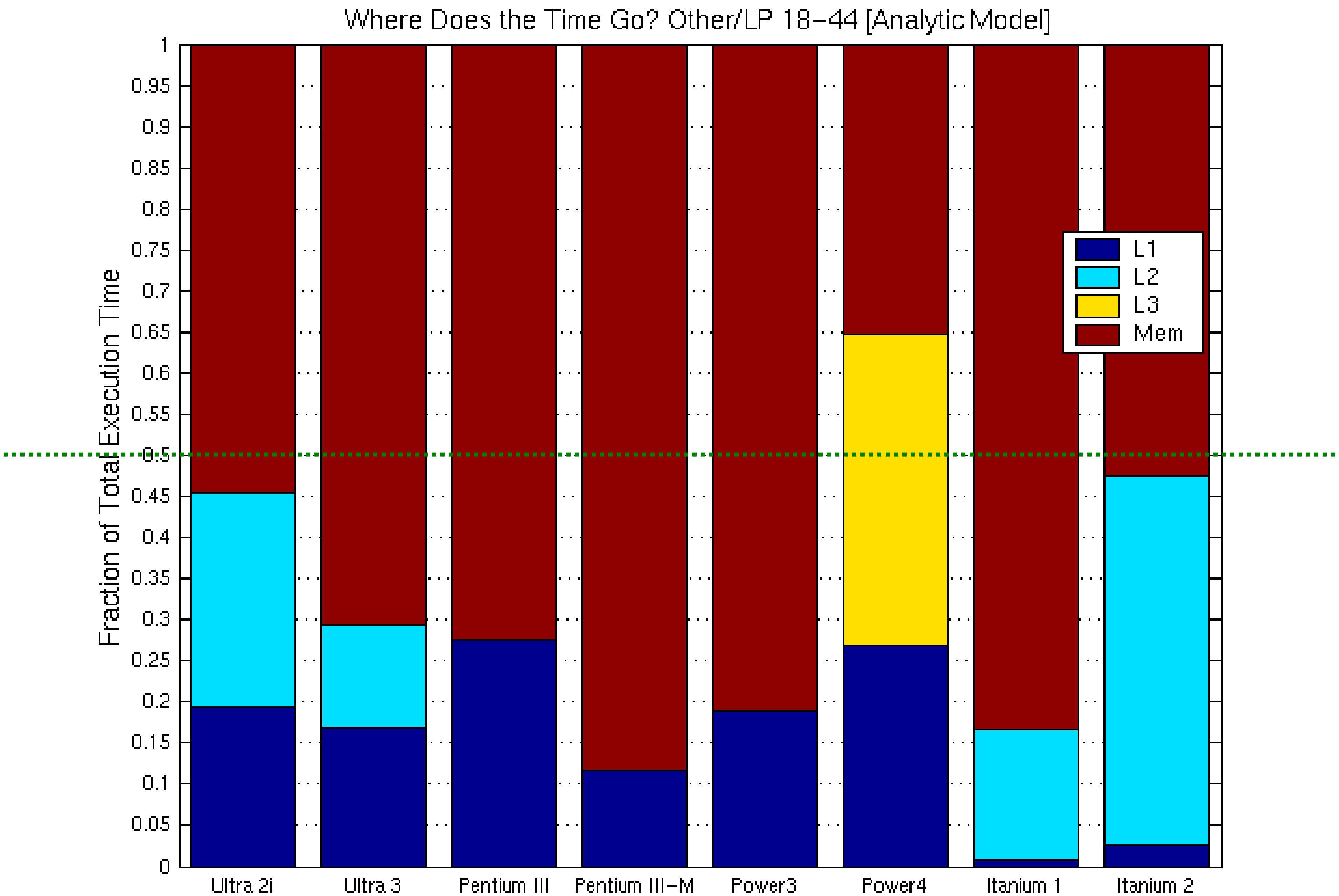


Correlating SpMV Performance with Machine Parameters
Itanium 2
Ultra 3
Power3
Pentium III-M
Pentium III
Power4
Ultra 2i
Itanium 1
DGEMM
DGEMV bound
DGEMV
Dense
FEM 2-9
FEM 10-17
Other/LP 18-44
Maximum fraction of machine peak
Sustainable Balance, based on Latency Model (flops / double)

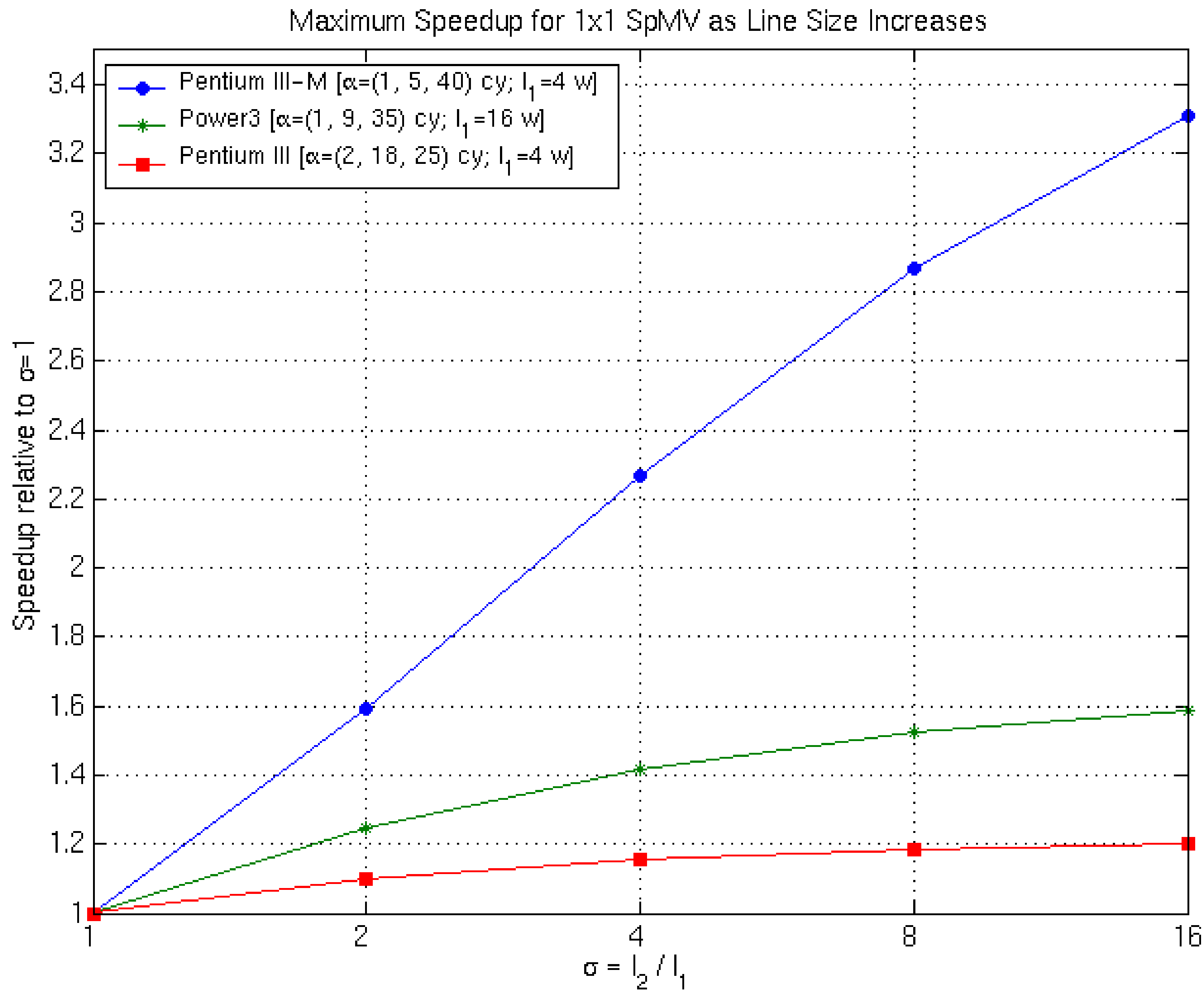
Where Does the Time Go?

$$\text{Time} \geq \sum_{i=1}^{\kappa} \alpha_i \cdot \text{Hits}_i + \alpha_{\text{mem}} \cdot \text{Hits}_{\text{mem}}$$

- Most time assigned to memory
- Caches “disappear” when line sizes are equal
  - Strictly increasing line sizes



Where Does the Time Go? Other/LP 18−44 [Analytic Model]
Fraction of Total Execution Time
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
L1
L2
L3
Mem
Ultra 2i
Ultra 3
Pentium III
Pentium III−M
Power3
Power4
Itanium 1
Itanium 2



Maximum Speedup for 1x1 SpMV as Line Size Increases
Pentium III-M [α=(1, 5, 40) cy; l1=4 w]
Power3 [α=(1, 9, 35) cy; l1=16 w]
Pentium III [α=(2, 18, 25) cy; l1=4 w]
Speedup relative to σ=1
σ = l2 / l1
1
2
4
8
16
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3
3.2
3.4

Summary: Performance Upper Bounds

- What is the best we can do for SpMV?
  - Limits to low-level tuning of blocked implementations
  - Refinements?
- What machines are good for SpMV?
  - Partial answer: balance characterization
- Architectural consequences?
  - Example: Strictly increasing line sizes

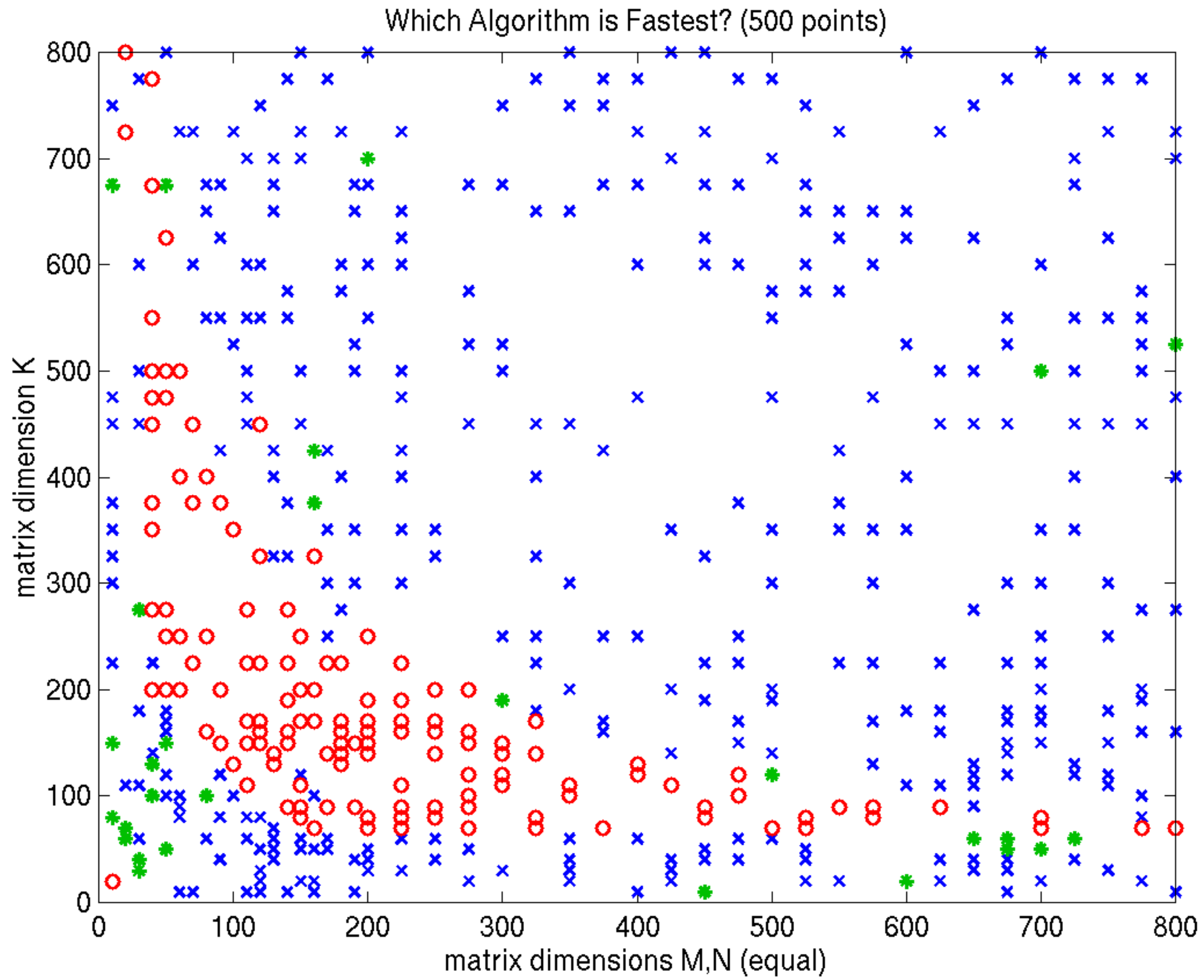
Road Map

- Sparse matrix-vector multiply (SpMV) in a nutshell
- Historical trends and the need for search
- Automatic tuning techniques
- Upper bounds on performance
- Tuning other sparse kernels
- **Statistical models of performance**
  - **FDO ’00; IJHPCA ’04a**

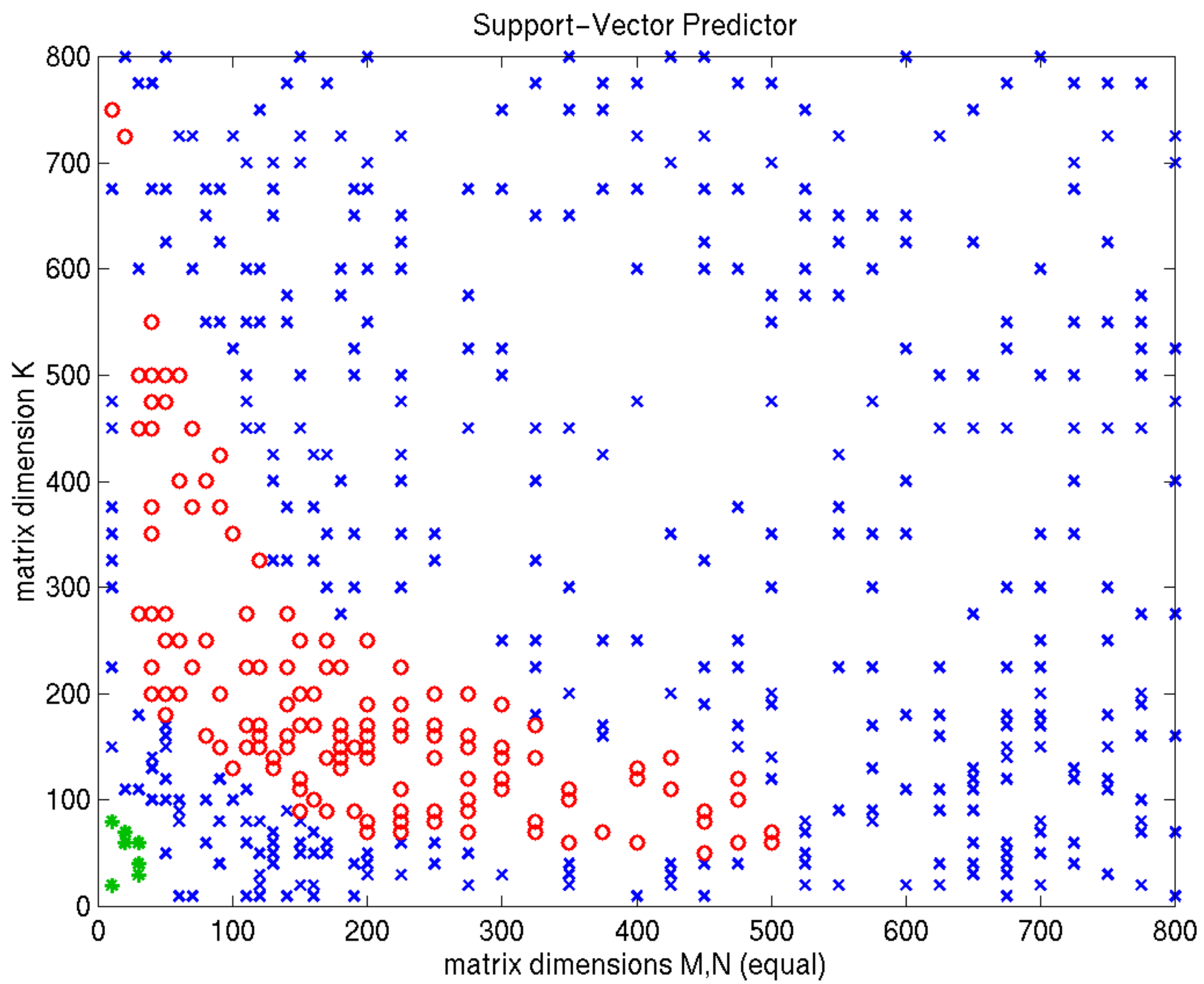
Statistical Models for Automatic Tuning

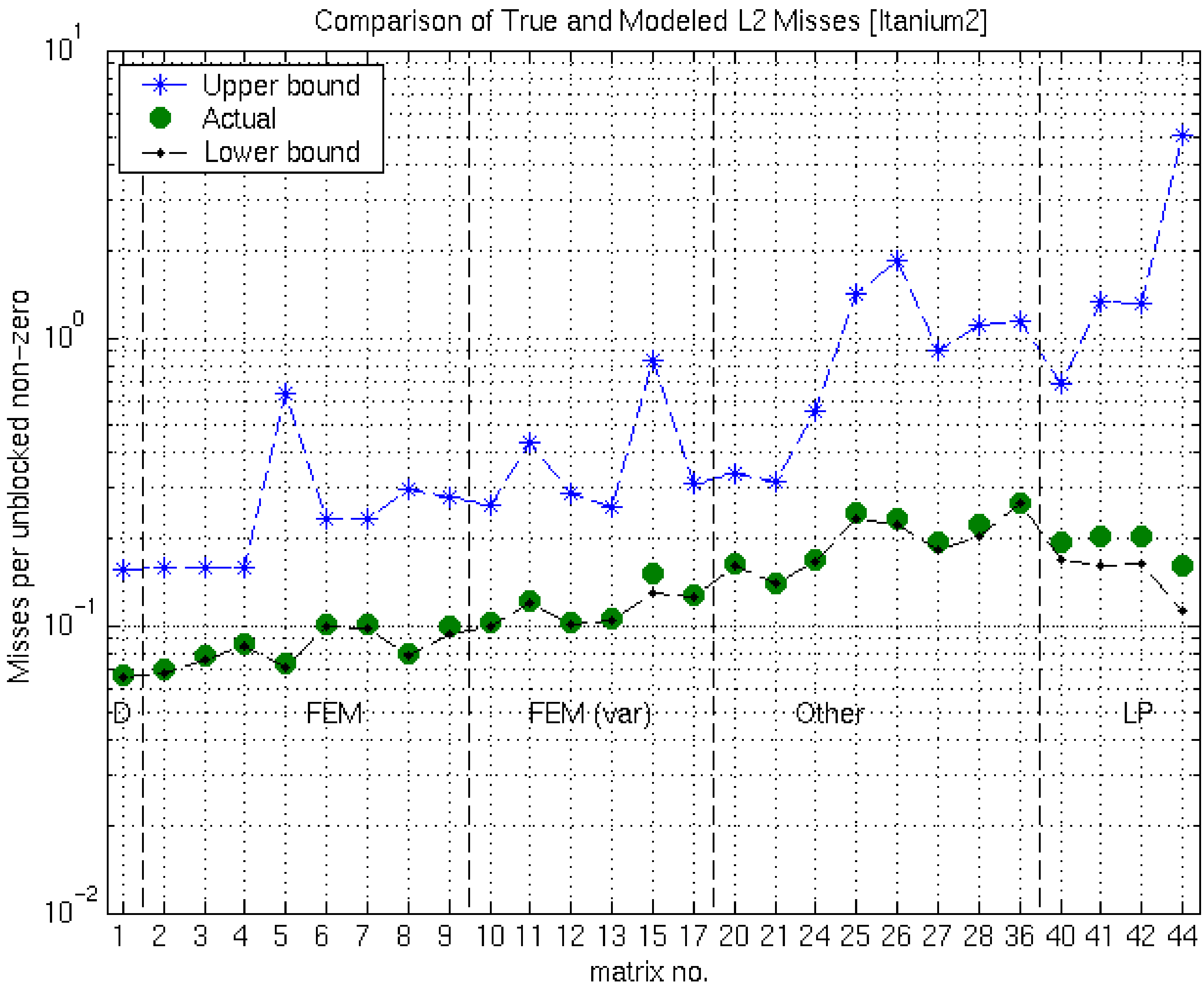
- Idea 1: Statistical criterion for stopping a search
  - A general search model
    - Generate implementation
    - Measure performance
    - Repeat
  - Stop when probability of being within ε of optimal falls below threshold
    - Can estimate distribution on-line
- Idea 2: Statistical performance models
  - Problem: Choose 1 among m implementations at run-time
  - Sample performance off-line, build statistical model

Example: Select a Matmul Implementation

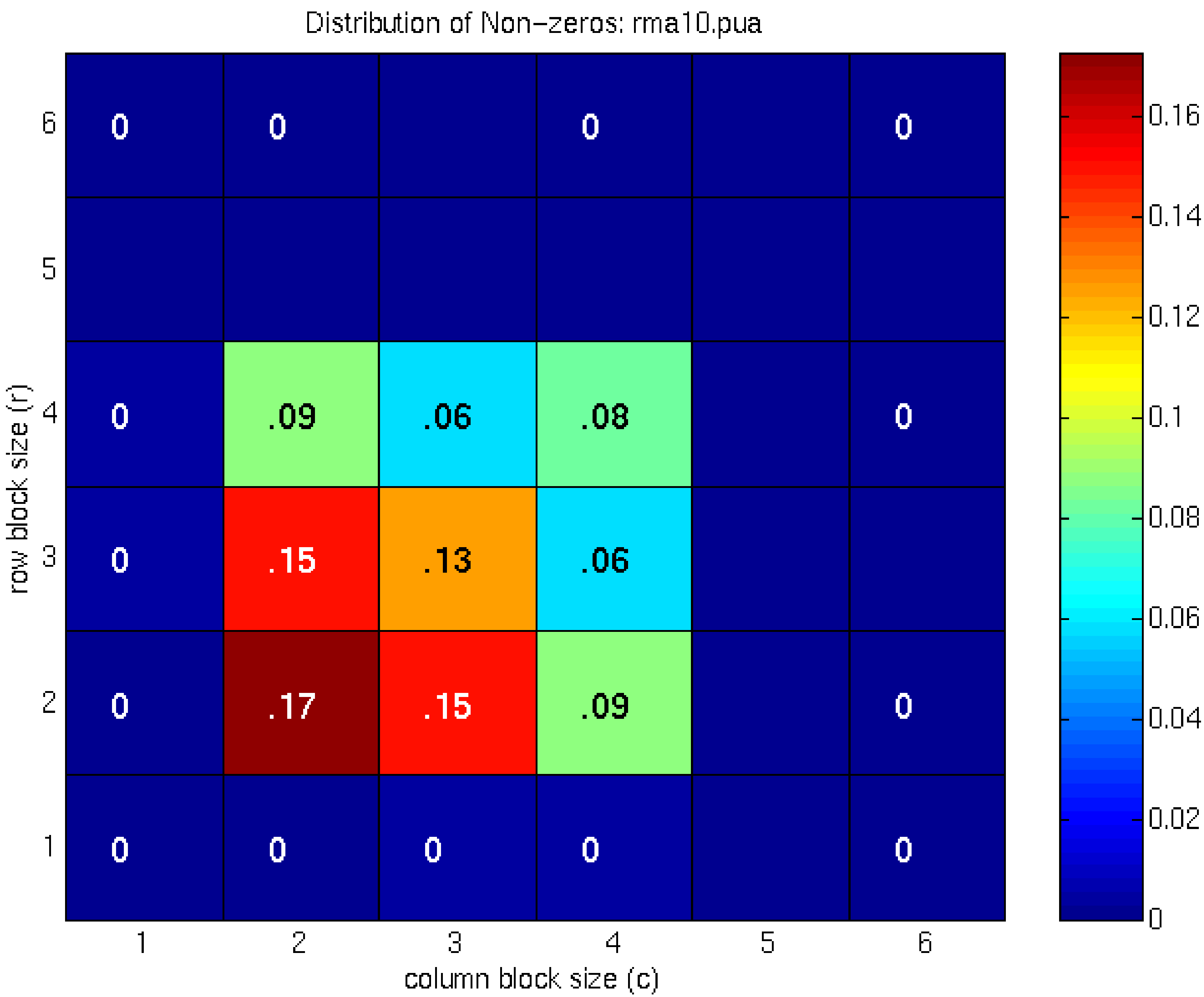


Example: Support Vector Classification



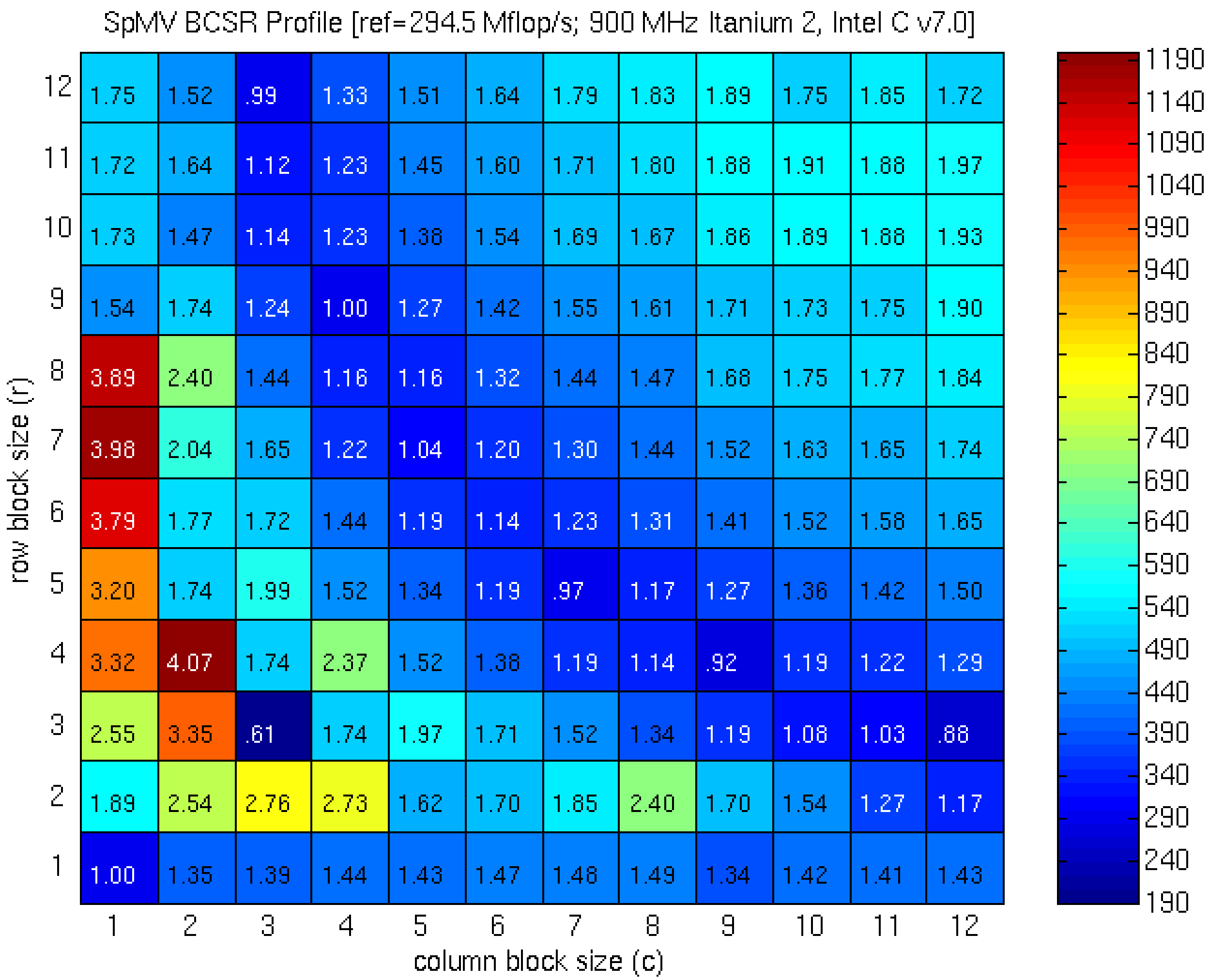


Misses measured using PAPI [Browne '00]



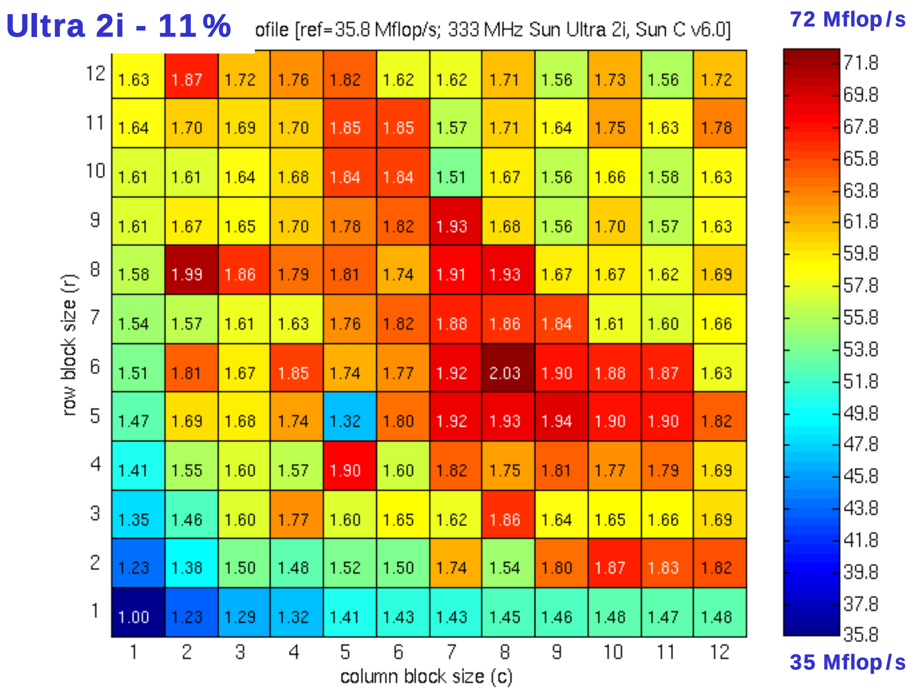
Distribution of Non−zeros: rma10.pua
row block size (r)
column block size (c)
1
2
3
4
5
6
0
.09
.06
.08
.15
.13
.17
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02

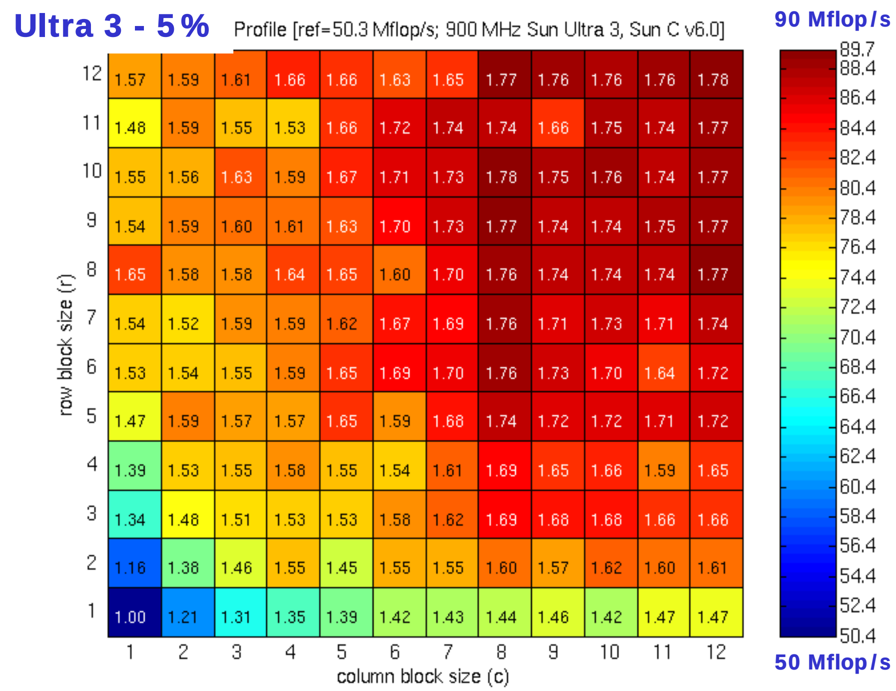
Register Profile: Itanium 2

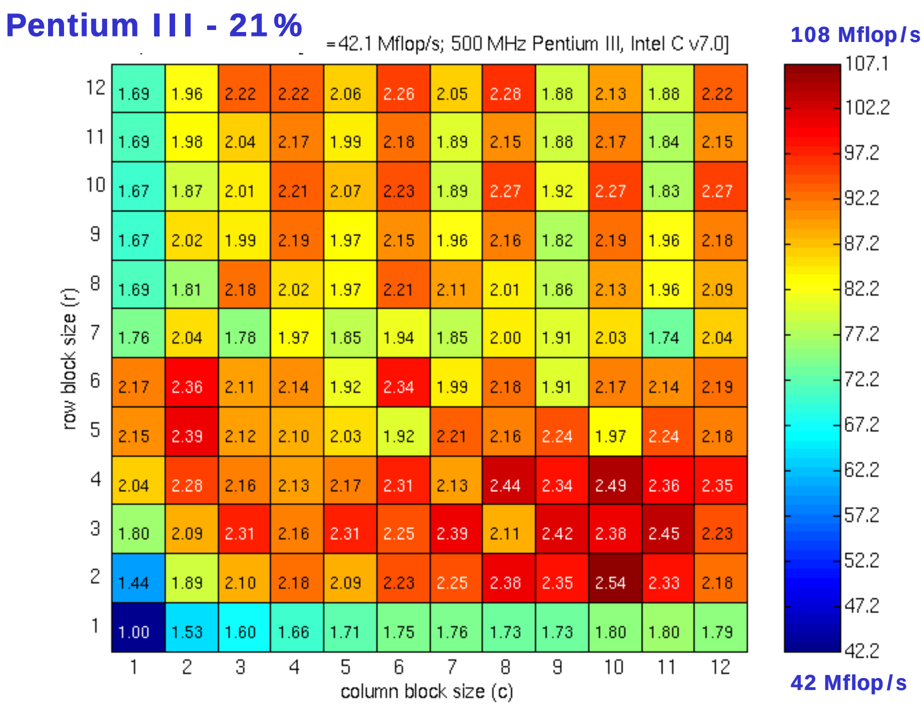


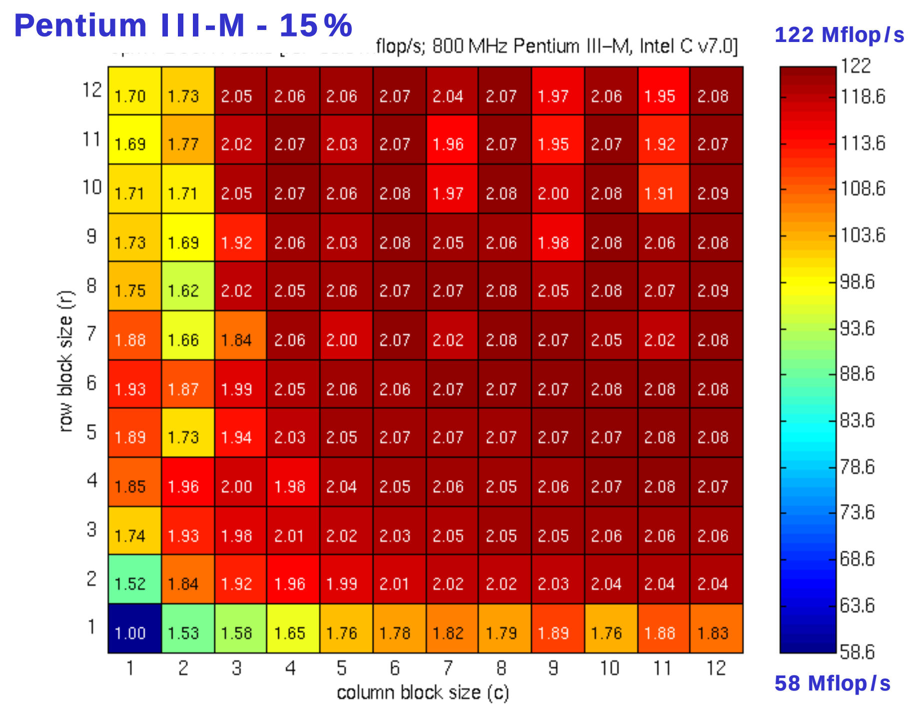
1190 Mflop/s

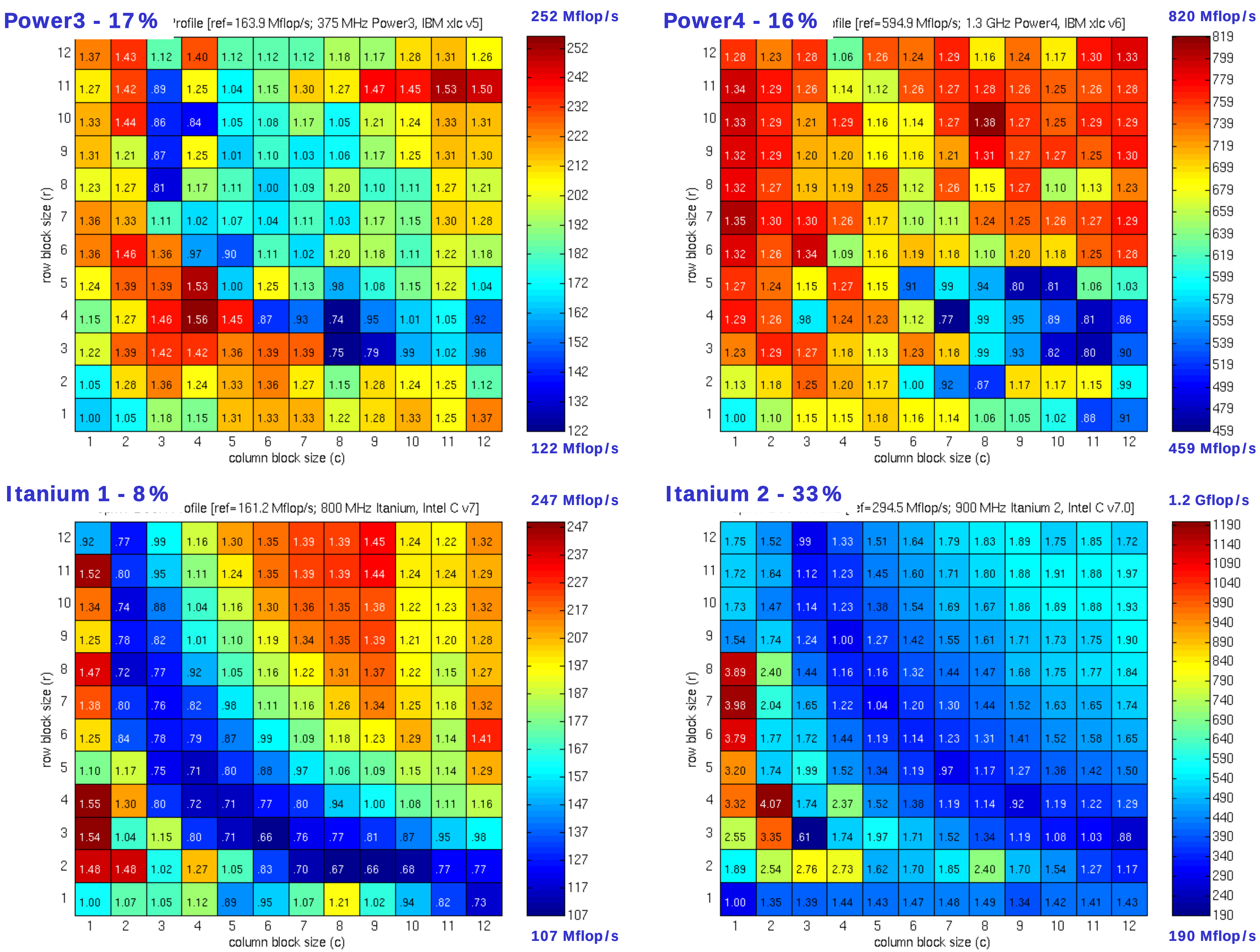
190 Mflop/s


Ultra 2i - 11%
ofile [ref=35.8 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]
72 Mflop/s
35 Mflop/s
71.8 69.8 67.8 65.8 63.8 61.8 59.8 57.8 55.8 53.8 51.8 49.8 47.8 45.8 43.8 41.8 39.8 37.8 35.8
row block size (r)
column block size (c)
12 1.63 1.87 1.72 1.76 1.82 1.62 1.62 1.71 1.56 1.73 1.56 1.72
11 1.64 1.70 1.69 1.70 1.85 1.85 1.57 1.71 1.64 1.75 1.63 1.78
10 1.61 1.61 1.64 1.68 1.84 1.84 1.51 1.67 1.56 1.66 1.58 1.63
9 1.61 1.67 1.65 1.70 1.78 1.82 1.93 1.68 1.56 1.70 1.57 1.63
8 1.58 1.99 1.86 1.79 1.81 1.74 1.91 1.93 1.67 1.67 1.62 1.69
7 1.54 1.57 1.61 1.63 1.76 1.82 1.88 1.86 1.84 1.61 1.60 1.66
6 1.51 1.81 1.67 1.85 1.74 1.77 1.92 2.03 1.90 1.88 1.87 1.63
5 1.47 1.69 1.68 1.74 1.32 1.80 1.92 1.93 1.94 1.90 1.90 1.82
4 1.41 1.55 1.60 1.57 1.90 1.60 1.82 1.75 1.81 1.77 1.79 1.69
3 1.35 1.46 1.60 1.77 1.60 1.65 1.62 1.86 1.64 1.65 1.66 1.69
2 1.23 1.38 1.50 1.48 1.52 1.50 1.74 1.54 1.80 1.87 1.83 1.82
1 1.00 1.23 1.29 1.32 1.41 1.43 1.43 1.45 1.46 1.48 1.47 1.48
1 2 3 4 5 6 7 8 9 10 11 12


Ultra 3 - 5%
Profile [ref=50.3 Mflop/s; 900 MHz Sun Ultra 3, Sun C v6.0]
90 Mflop/s
50 Mflop/s
89.7 88.4 86.4 84.4 82.4 80.4 78.4 76.4 74.4 72.4 70.4 68.4 66.4 64.4 62.4 60.4 58.4 56.4 54.4 52.4 50.4
row block size (r)
column block size (c)
12 1.57 1.59 1.61 1.66 1.66 1.63 1.65 1.77 1.76 1.76 1.76 1.78
11 1.48 1.59 1.55 1.53 1.66 1.72 1.74 1.74 1.66 1.75 1.74 1.77
10 1.55 1.56 1.63 1.59 1.67 1.71 1.73 1.78 1.75 1.76 1.74 1.77
9 1.54 1.59 1.60 1.61 1.63 1.70 1.73 1.77 1.74 1.74 1.75 1.77
8 1.65 1.58 1.58 1.64 1.65 1.60 1.70 1.76 1.74 1.74 1.74 1.77
7 1.54 1.52 1.59 1.59 1.62 1.67 1.69 1.76 1.71 1.73 1.71 1.74
6 1.53 1.54 1.55 1.59 1.65 1.69 1.70 1.76 1.73 1.70 1.64 1.72
5 1.47 1.59 1.57 1.57 1.65 1.59 1.68 1.74 1.72 1.72 1.71 1.72
4 1.39 1.53 1.55 1.58 1.55 1.54 1.61 1.69 1.65 1.66 1.59 1.65
3 1.34 1.48 1.51 1.53 1.53 1.58 1.62 1.69 1.68 1.68 1.66 1.66
2 1.16 1.38 1.46 1.55 1.45 1.55 1.55 1.60 1.57 1.62 1.60 1.61
1 1.00 1.21 1.31 1.35 1.39 1.42 1.43 1.44 1.46 1.42 1.47 1.47
1 2 3 4 5 6 7 8 9 10 11 12


Pentium III - 21%
=42.1 Mflop/s; 500 MHz Pentium III, Intel C v7.0]
108 Mflop/s
42 Mflop/s
107.1 102.2 97.2 92.2 87.2 82.2 77.2 72.2 67.2 62.2 57.2 52.2 47.2 42.2
row block size (r)
column block size (c)
12 1.69 1.96 2.22 2.22 2.06 2.26 2.05 2.28 1.88 2.13 1.88 2.22
11 1.69 1.98 2.04 2.17 1.99 2.18 1.89 2.15 1.88 2.17 1.84 2.15
10 1.67 1.87 2.01 2.21 2.07 2.23 1.89 2.27 1.92 2.27 1.83 2.27
9 1.67 2.02 1.99 2.19 1.97 2.15 1.96 2.16 1.82 2.19 1.96 2.18
8 1.69 1.81 2.18 2.02 1.97 2.21 2.11 2.01 1.86 2.13 1.96 2.09
7 1.76 2.04 1.78 1.97 1.85 1.94 1.85 2.00 1.91 2.03 1.74 2.04
6 2.17 2.36 2.11 2.14 1.92 2.34 1.99 2.18 1.91 2.17 2.14 2.19
5 2.15 2.39 2.12 2.10 2.03 1.92 2.21 2.16 2.24 1.97 2.24 2.18
4 2.04 2.28 2.16 2.13 2.17 2.31 2.13 2.44 2.34 2.49 2.36 2.35
3 1.80 2.09 2.31 2.16 2.31 2.25 2.39 2.11 2.42 2.38 2.45 2.23
2 1.44 1.89 2.10 2.18 2.09 2.23 2.25 2.38 2.35 2.54 2.33 2.18
1 1.00 1.53 1.60 1.66 1.71 1.75 1.76 1.73 1.73 1.80 1.80 1.79
1 2 3 4 5 6 7 8 9 10 11 12


Pentium III-M - 15%
flop/s; 800 MHz Pentium III–M, Intel C v7.0]
122 Mflop/s
58 Mflop/s
122 118.6 113.6 108.6 103.6 98.6 93.6 88.6 83.6 78.6 73.6 68.6 63.6 58.6
row block size (r)
column block size (c)
12 1.70 1.73 2.05 2.06 2.06 2.07 2.04 2.07 1.97 2.06 1.95 2.08
11 1.69 1.77 2.02 2.07 2.03 2.07 1.96 2.07 1.95 2.07 1.92 2.07
10 1.71 1.71 2.05 2.07 2.06 2.08 1.97 2.08 2.00 2.08 1.91 2.09
9 1.73 1.69 1.92 2.06 2.03 2.08 2.05 2.06 1.98 2.08 2.06 2.08
8 1.75 1.62 2.02 2.05 2.06 2.07 2.07 2.08 2.05 2.08 2.07 2.09
7 1.88 1.66 1.84 2.06 2.00 2.07 2.02 2.08 2.07 2.05 2.02 2.08
6 1.93 1.87 1.99 2.05 2.06 2.06 2.07 2.07 2.07 2.08 2.08 2.08
5 1.89 1.73 1.94 2.03 2.05 2.07 2.07 2.07 2.07 2.07 2.08 2.08
4 1.85 1.96 2.00 1.98 2.04 2.05 2.06 2.05 2.06 2.07 2.08 2.07
3 1.74 1.93 1.98 2.01 2.02 2.03 2.05 2.05 2.05 2.06 2.06 2.06
2 1.52 1.84 1.92 1.96 1.99 2.01 2.02 2.02 2.03 2.04 2.04 2.04
1 1.00 1.53 1.58 1.65 1.76 1.78 1.82 1.79 1.89 1.76 1.88 1.83
1 2 3 4 5 6 7 8 9 10 11 12


Power3 - 17%
rofile [ref=163.9 Mflop/s; 375 MHz Power3, IBM xlc v5]
252 Mflop/s
12: 1.37 1.43 1.12 1.40 1.12 1.12 1.12 1.18 1.17 1.28 1.31 1.26
11: 1.27 1.42 .89 1.25 1.04 1.15 1.30 1.27 1.47 1.45 1.53 1.50
10: 1.33 1.44 .86 .84 1.05 1.08 1.17 1.05 1.21 1.24 1.33 1.31
9: 1.31 1.21 .87 1.25 1.01 1.10 1.03 1.06 1.17 1.25 1.31 1.30
8: 1.23 1.27 .81 1.17 1.11 1.00 1.09 1.20 1.10 1.11 1.27 1.21
7: 1.36 1.33 1.11 1.02 1.07 1.04 1.11 1.03 1.17 1.15 1.30 1.28
6: 1.36 1.46 1.36 .97 .90 1.11 1.02 1.20 1.18 1.11 1.22 1.18
5: 1.24 1.39 1.39 1.53 1.00 1.25 1.13 .98 1.08 1.15 1.22 1.04
4: 1.15 1.27 1.46 1.56 1.45 .87 .93 .74 .95 1.01 1.05 .92
3: 1.22 1.39 1.42 1.42 1.36 1.39 1.39 .75 .79 .99 1.02 .96
2: 1.05 1.28 1.36 1.24 1.33 1.36 1.27 1.15 1.28 1.24 1.25 1.12
1: 1.00 1.05 1.18 1.15 1.31 1.33 1.33 1.22 1.28 1.33 1.25 1.37
row block size (r)
column block size (c)
1 2 3 4 5 6 7 8 9 10 11 12
252 242 232 222 212 202 192 182 172 162 152 142 132 122
122 Mflop/s
Power4 - 16%
file [ref=594.9 Mflop/s; 1.3 GHz Power4, IBM xlc v6]
820 Mflop/s
12: 1.28 1.23 1.28 1.06 1.26 1.24 1.29 1.16 1.24 1.17 1.30 1.33
11: 1.34 1.29 1.26 1.14 1.12 1.26 1.27 1.28 1.26 1.25 1.26 1.28
10: 1.33 1.29 1.21 1.29 1.16 1.14 1.27 1.38 1.27 1.25 1.29 1.29
9: 1.32 1.29 1.20 1.20 1.16 1.16 1.21 1.31 1.27 1.27 1.25 1.30
8: 1.32 1.27 1.19 1.19 1.25 1.12 1.26 1.15 1.27 1.10 1.13 1.23
7: 1.35 1.30 1.30 1.26 1.17 1.10 1.11 1.24 1.25 1.26 1.27 1.29
6: 1.32 1.26 1.34 1.09 1.16 1.19 1.18 1.10 1.20 1.18 1.25 1.28
5: 1.27 1.24 1.15 1.27 1.15 .91 .99 .94 .80 .81 1.06 1.03
4: 1.29 1.26 .98 1.24 1.23 1.12 .77 .99 .95 .89 .81 .86
3: 1.23 1.29 1.27 1.18 1.13 1.23 1.18 .99 .93 .82 .80 .90
2: 1.13 1.18 1.25 1.20 1.17 1.00 .92 .87 1.17 1.17 1.15 .99
1: 1.00 1.10 1.15 1.15 1.18 1.16 1.14 1.06 1.05 1.02 .88 .91
row block size (r)
column block size (c)
1 2 3 4 5 6 7 8 9 10 11 12
819 799 779 759 739 719 699 679 659 639 619 599 579 559 539 519 499 479 459
459 Mflop/s
Itanium 1 - 8%
ofile [ref=161.2 Mflop/s; 800 MHz Itanium, Intel C v7]
247 Mflop/s
12: .92 .77 .99 1.16 1.30 1.35 1.39 1.39 1.45 1.24 1.22 1.32
11: 1.52 .80 .95 1.11 1.24 1.35 1.39 1.39 1.44 1.24 1.24 1.29
10: 1.34 .74 .88 1.04 1.16 1.30 1.36 1.35 1.38 1.22 1.23 1.32
9: 1.25 .78 .82 1.01 1.10 1.19 1.34 1.35 1.39 1.21 1.20 1.28
8: 1.47 .72 .77 .92 1.05 1.16 1.22 1.31 1.37 1.22 1.15 1.27
7: 1.38 .80 .76 .82 .98 1.11 1.16 1.26 1.34 1.25 1.18 1.32
6: 1.25 .84 .78 .79 .87 .99 1.09 1.18 1.23 1.29 1.14 1.41
5: 1.10 1.17 .75 .71 .80 .88 .97 1.06 1.09 1.15 1.14 1.29
4: 1.55 1.30 .80 .72 .71 .77 .80 .94 1.00 1.08 1.11 1.16
3: 1.54 1.04 1.15 .80 .71 .66 .76 .77 .81 .87 .95 .98
2: 1.48 1.48 1.02 1.27 1.05 .83 .70 .67 .66 .68 .77 .77
1: 1.00 1.07 1.05 1.12 .89 .95 1.07 1.21 1.02 .94 .82 .73
row block size (r)
column block size (c)
1 2 3 4 5 6 7 8 9 10 11 12
247 237 227 217 207 197 187 177 167 157 147 137 127 117 107
107 Mflop/s
Itanium 2 - 33%
ef=294.5 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]
1.2 Gflop/s
12: 1.75 1.52 .99 1.33 1.51 1.64 1.79 1.83 1.89 1.75 1.85 1.72
11: 1.72 1.64 1.12 1.23 1.45 1.60 1.71 1.80 1.88 1.91 1.88 1.97
10: 1.73 1.47 1.14 1.23 1.38 1.54 1.69 1.67 1.86 1.89 1.88 1.93
9: 1.54 1.74 1.24 1.00 1.27 1.42 1.55 1.61 1.71 1.73 1.75 1.90
8: 3.89 2.40 1.44 1.16 1.16 1.32 1.44 1.47 1.68 1.75 1.77 1.84
7: 3.98 2.04 1.65 1.22 1.04 1.20 1.30 1.44 1.52 1.63 1.65 1.74
6: 3.79 1.77 1.72 1.44 1.19 1.14 1.23 1.31 1.41 1.52 1.58 1.65
5: 3.20 1.74 1.99 1.52 1.34 1.19 .97 1.17 1.27 1.36 1.42 1.50
4: 3.32 4.07 1.74 2.37 1.52 1.38 1.19 1.14 .92 1.19 1.22 1.29
3: 2.55 3.35 .61 1.74 1.97 1.71 1.52 1.34 1.19 1.08 1.03 .88
2: 1.89 2.54 2.76 2.73 1.62 1.70 1.85 2.40 1.70 1.54 1.27 1.17
1: 1.00 1.35 1.39 1.44 1.43 1.47 1.48 1.49 1.34 1.42 1.41 1.43
row block size (r)
column block size (c)
1 2 3 4 5 6 7 8 9 10 11 12
1190 1140 1090 1040 990 940 890 840 790 740 690 640 590 540 490 440 390 340 290 240 190
190 Mflop/s

Accurate and Efficient Adaptive Fill Estimation

- Idea: Sample matrix
  - Fraction of matrix to sample: $s \in [0,1]$
  - Cost ~ O(s * nnz)
  - Control cost by controlling s
    - Search at run-time: the constant matters!
- Control s automatically by computing statistical confidence intervals
  - Idea: Monitor variance
- Cost of tuning
  - Lower bound: convert matrix in 5 to 40 unblocked SpMVs
  - Heuristic: 1 to 11 SpMVs

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:

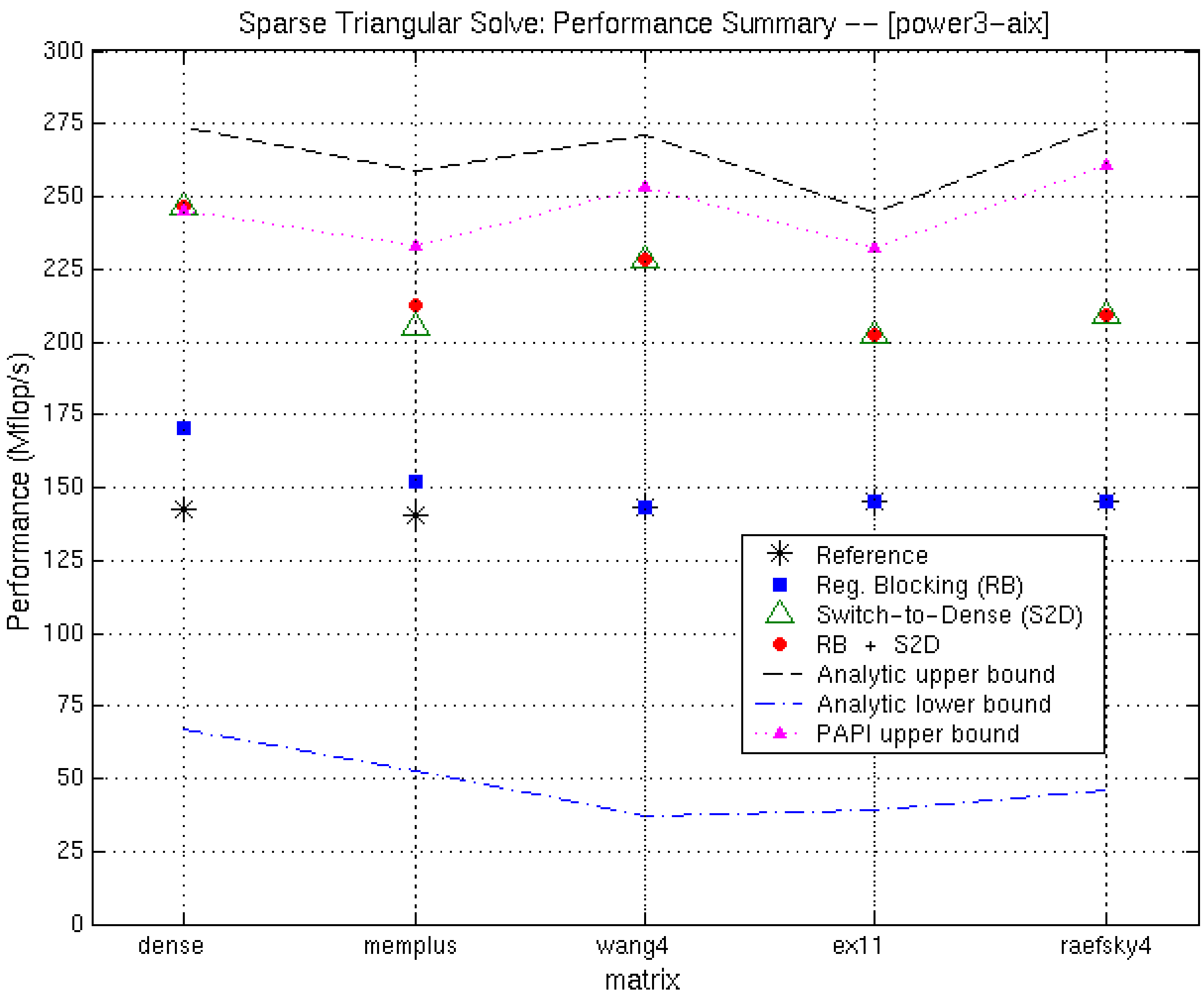
$$(1) \quad L_1 x_1 = b_1$$

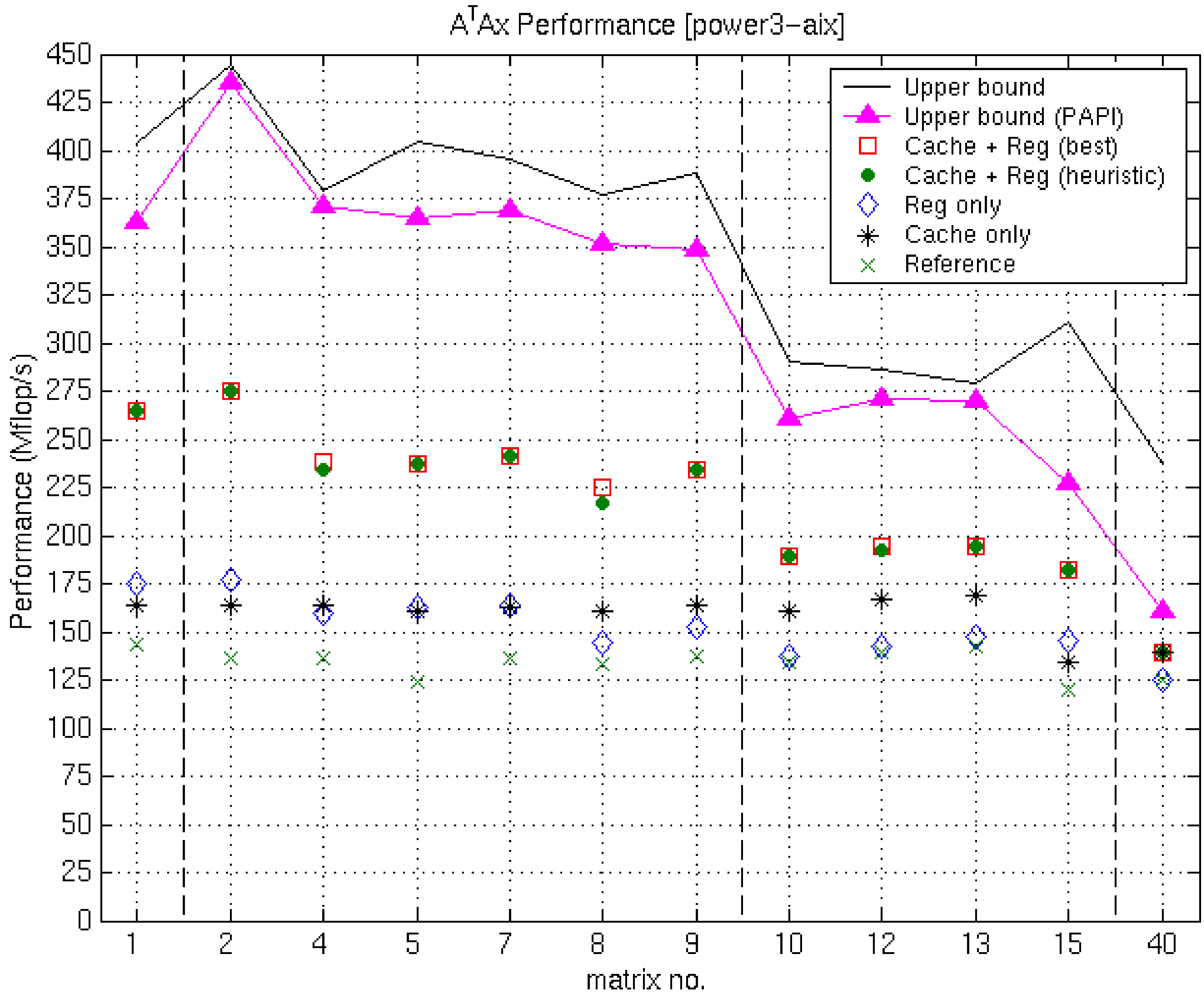
$$(2) \quad \hat{b}_2 = b_2 - L_2 x_1$$

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

- Sparsity optimizations for (1)—(2); DTRSV for (3)
- Tuning parameters: block size, size of dense triangle

SpTS Performance: Power3

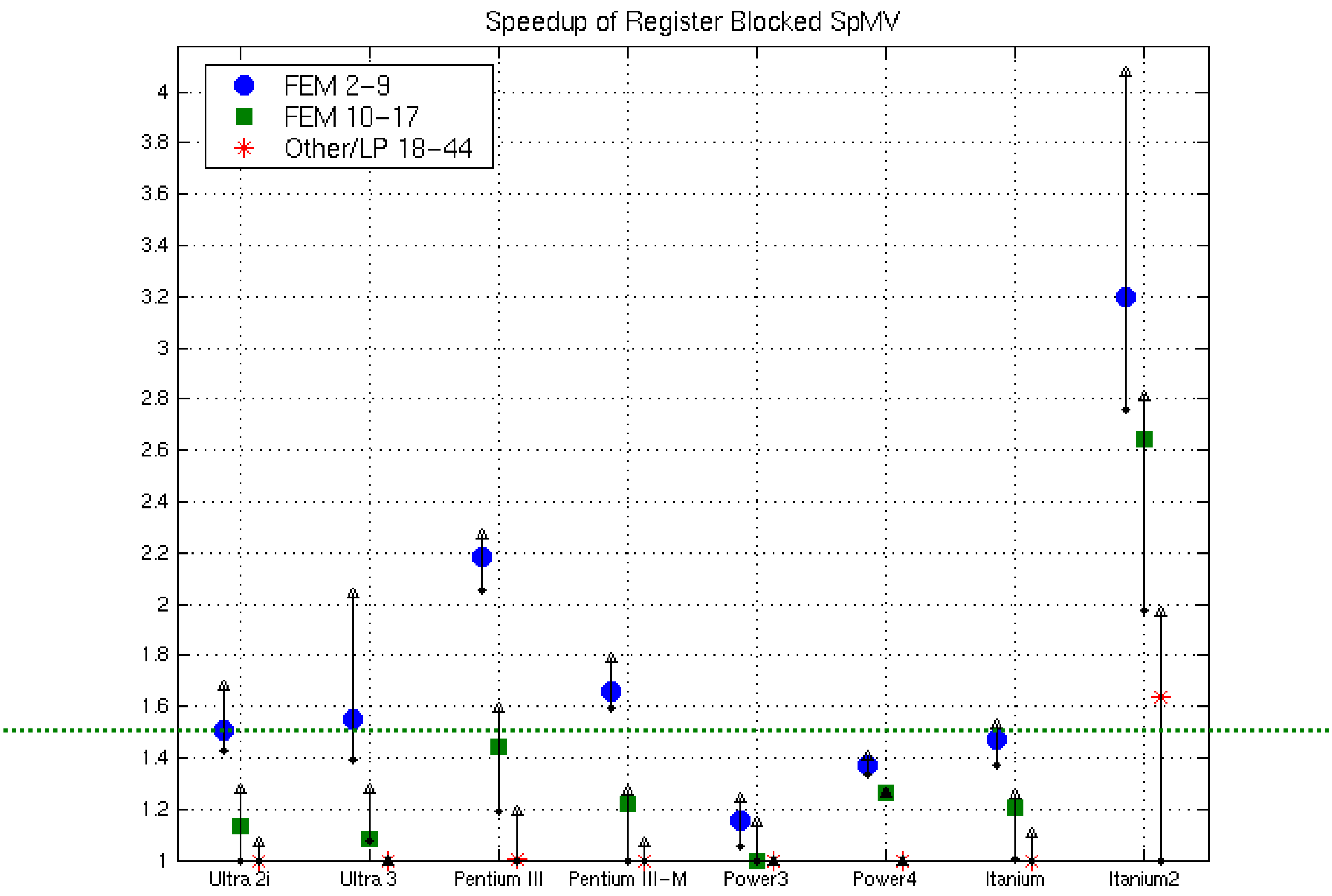




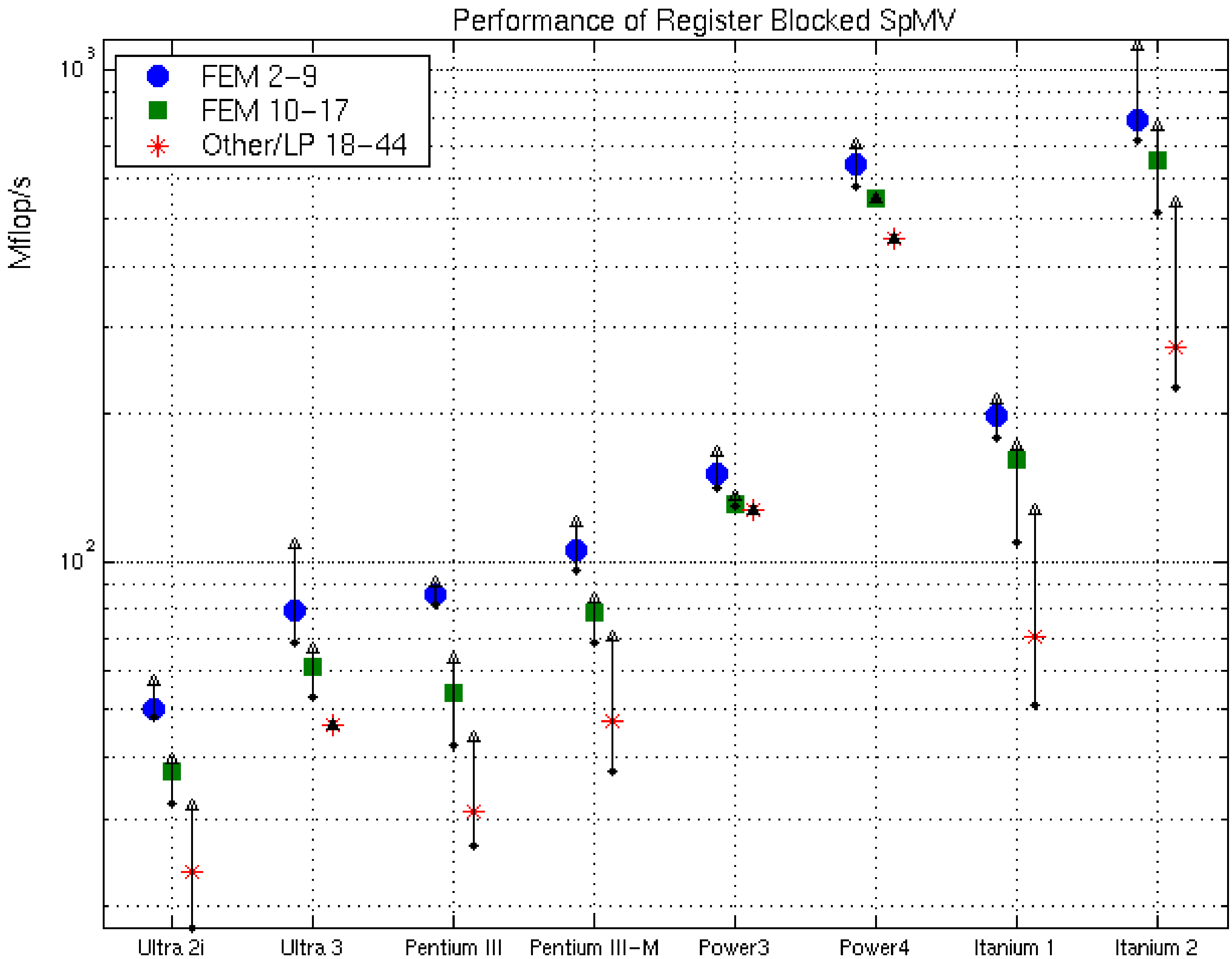
$A^T Ax$ Performance [power3−aix]
Upper bound
Upper bound (PAPI)
Cache + Reg (best)
Cache + Reg (heuristic)
Reg only
Cache only
Reference
Performance (Mflop/s)
matrix no.
450
425
400
375
350
325
300
275
250
225
200
175
150
125
100
75
50
25
0
1
2
4
5
7
8
9
10
12
13
15
40

Summary of SpTS and AA^T*x Results

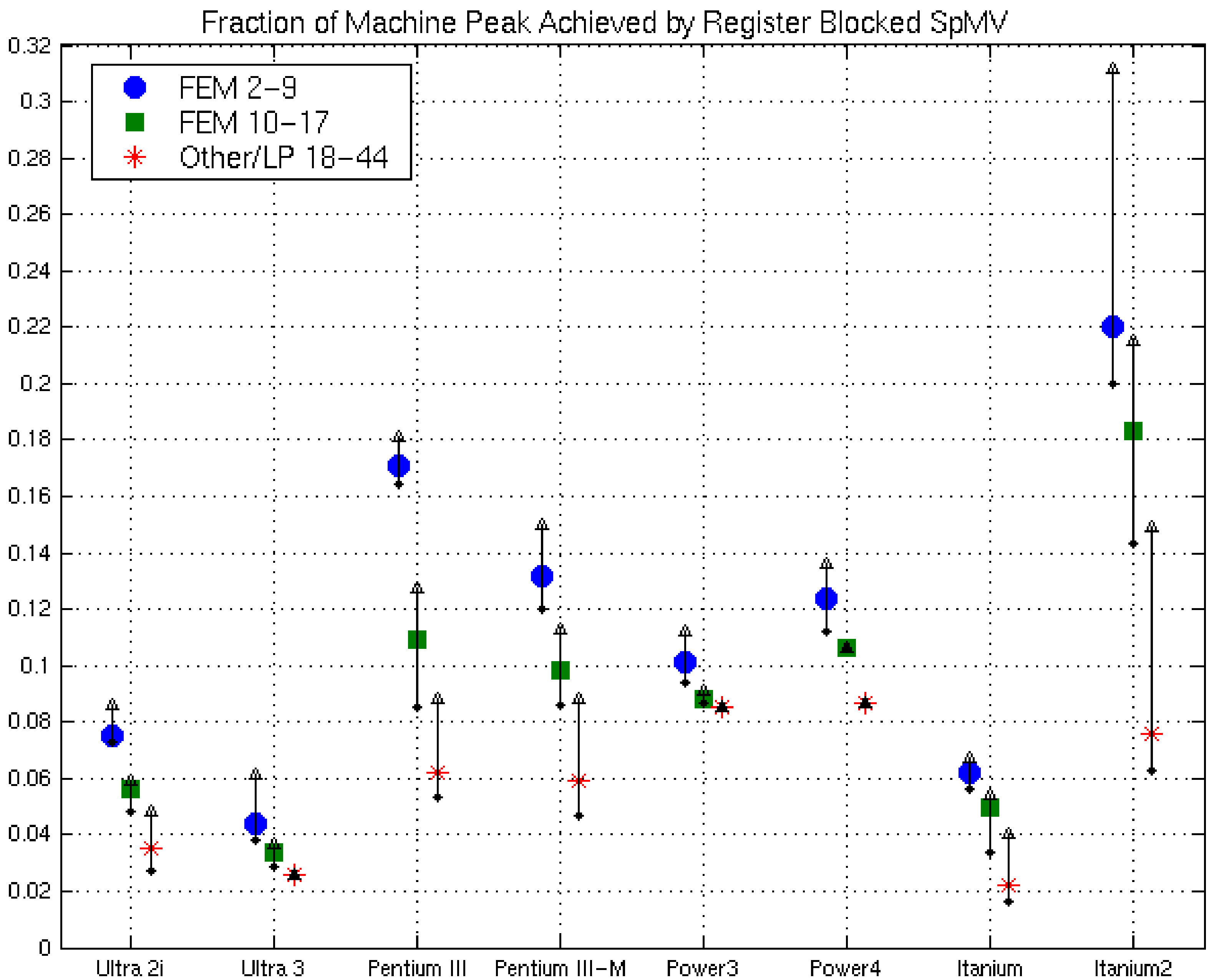
- SpTS — Similar to SpMV
  - 1.8x speedups; limited benefit from low-level tuning
- AA^Tx, A^TAx
  - Cache interleaving only: up to 1.6x speedups
  - Reg + cache: up to 4x speedups
    - 1.8x speedup over register only
  - Similar heuristic; same accuracy (~ 10% optimal)
  - Further from upper bounds: 60—80%
    - Opportunity for better low-level tuning *a la* PHiPAC/ATLAS
- Matrix triple products? A^k*x?
  - Preliminary work

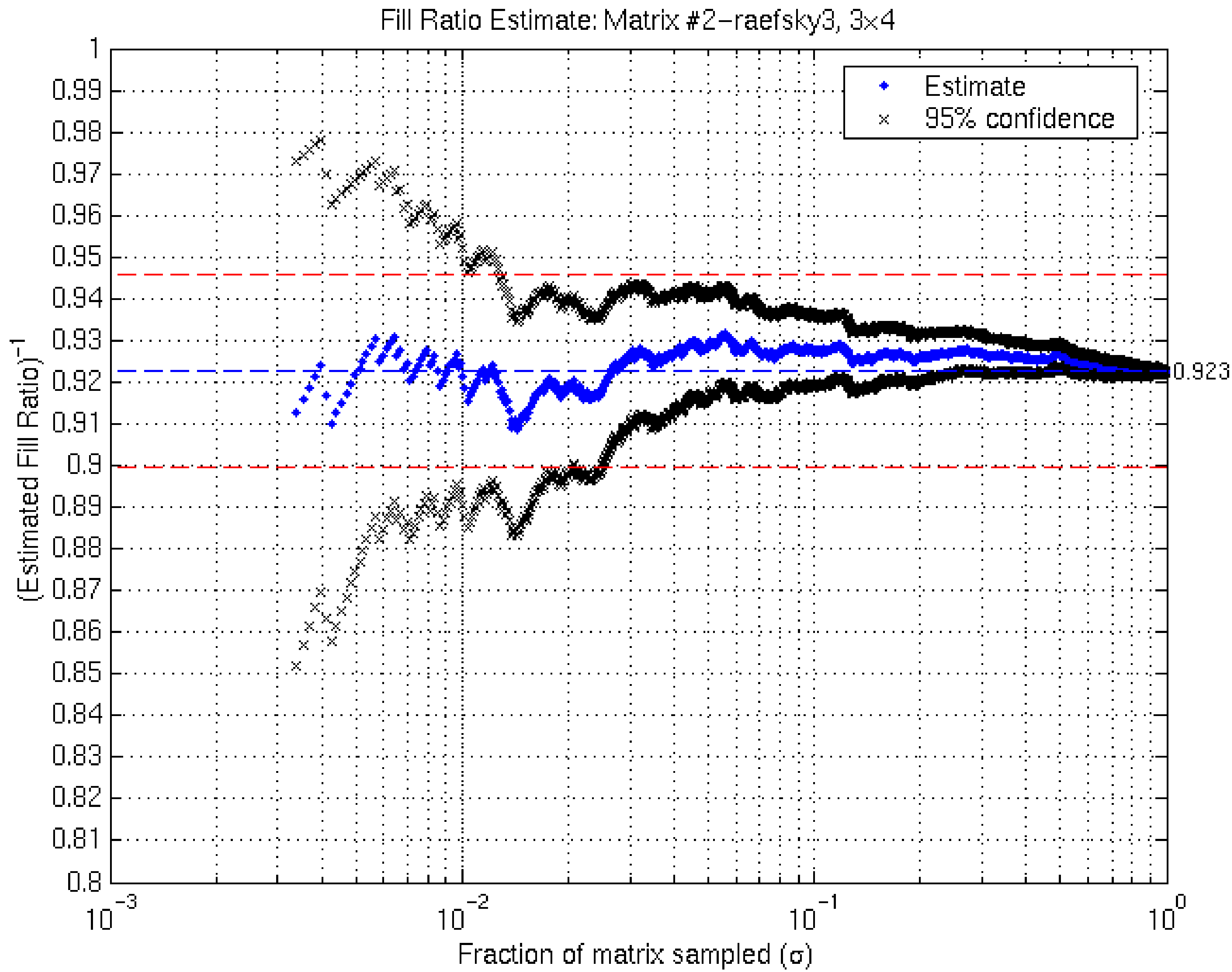


Speedup of Register Blocked SpMV
FEM 2−9
FEM 10−17
Other/LP 18−44
4
3.8
3.6
3.4
3.2
3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
Ultra 2i
Ultra 3
Pentium III
Pentium III−M
Power3
Power4
Itanium
Itanium2

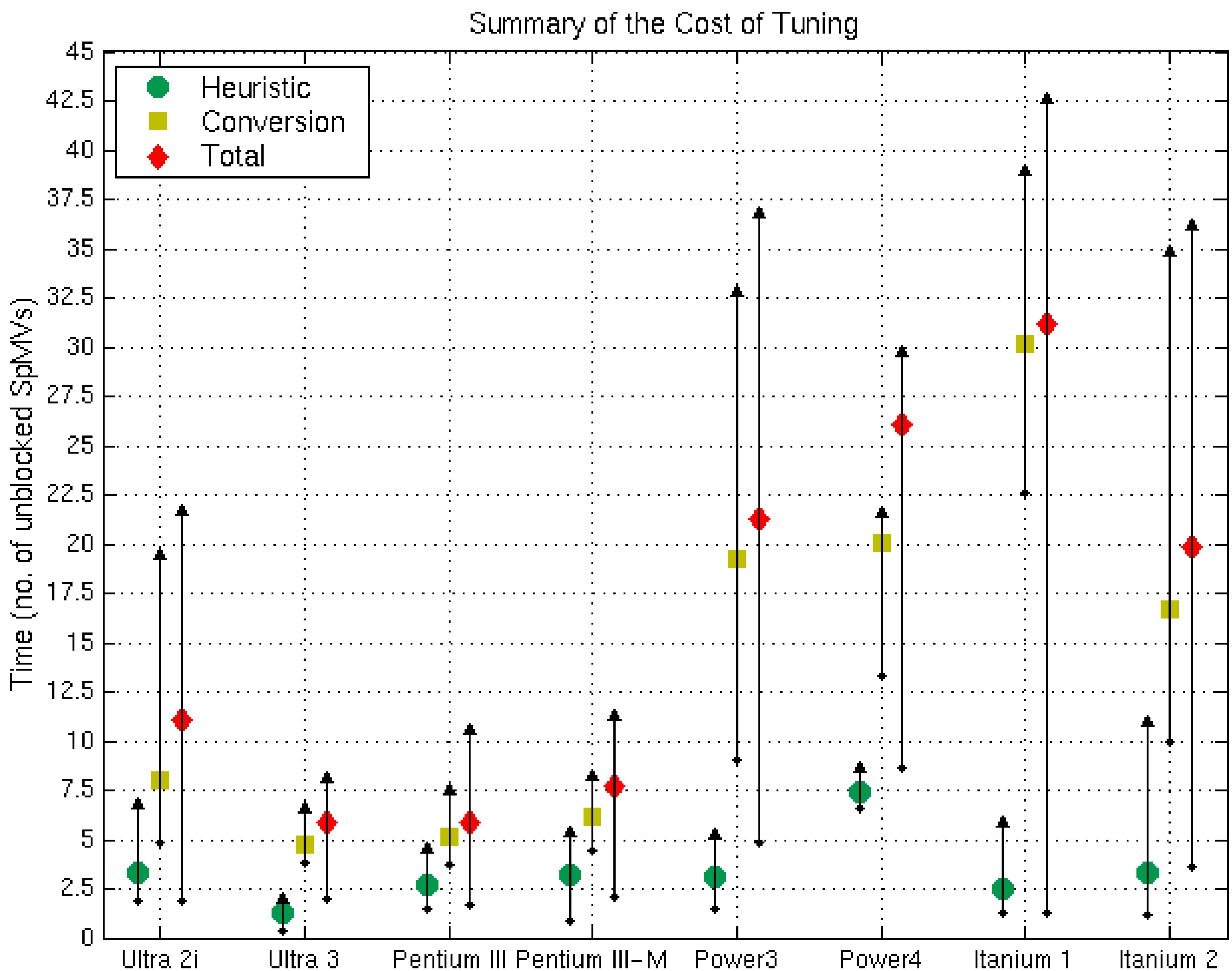


Performance of Register Blocked SpMV
Mflop/s
10^3
10^2
FEM 2–9
FEM 10–17
Other/LP 18–44
Ultra 2i
Ultra 3
Pentium III
Pentium III–M
Power3
Power4
Itanium 1
Itanium 2

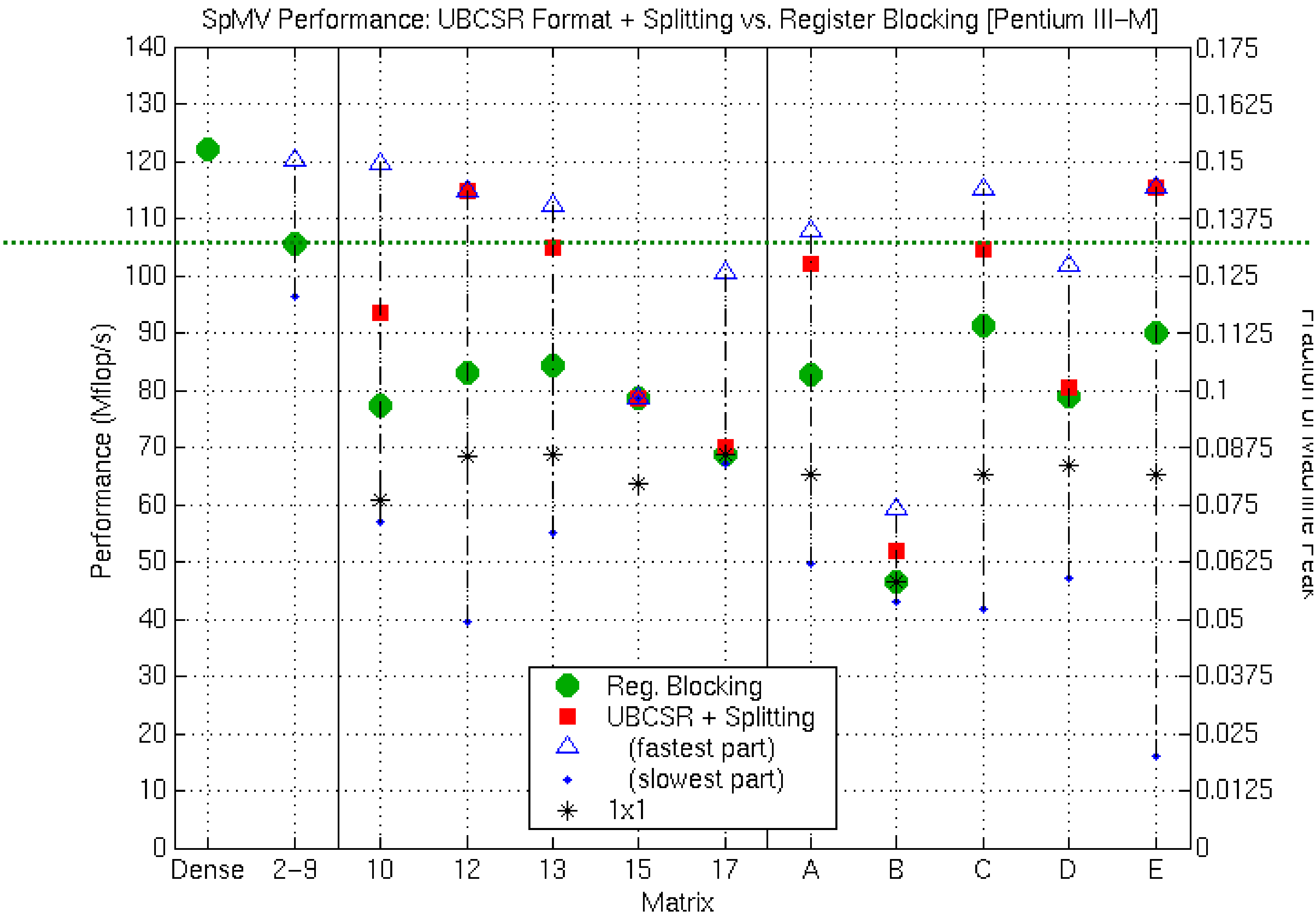

Fraction of Machine Peak Achieved by Register Blocked SpMV
0.32
0.3
0.28
0.26
0.24
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
FEM 2–9
FEM 10–17
Other/LP 18–44
Ultra 2i
Ultra 3
Pentium III
Pentium III–M
Power3
Power4
Itanium
Itanium2



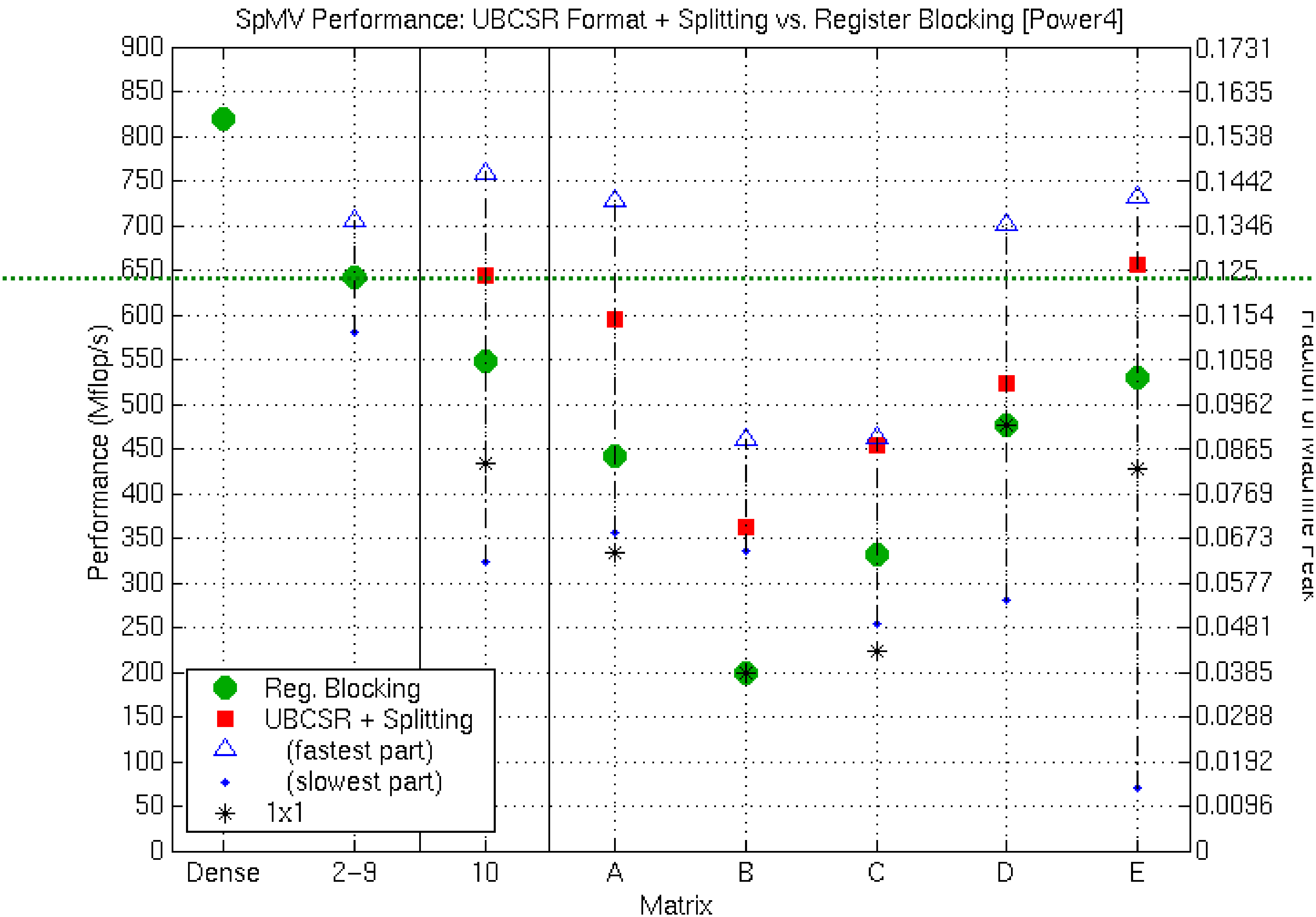
Fill Ratio Estimate: Matrix #2−raefsky3, 3×4
Estimate
95% confidence
0.923
(Estimated Fill Ratio)$^{-1}$
Fraction of matrix sampled (σ)
1
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.91
0.9
0.89
0.88
0.87
0.86
0.85
0.84
0.83
0.82
0.81
0.8
10^{-3}
10^{-2}
10^{-1}
10^{0}

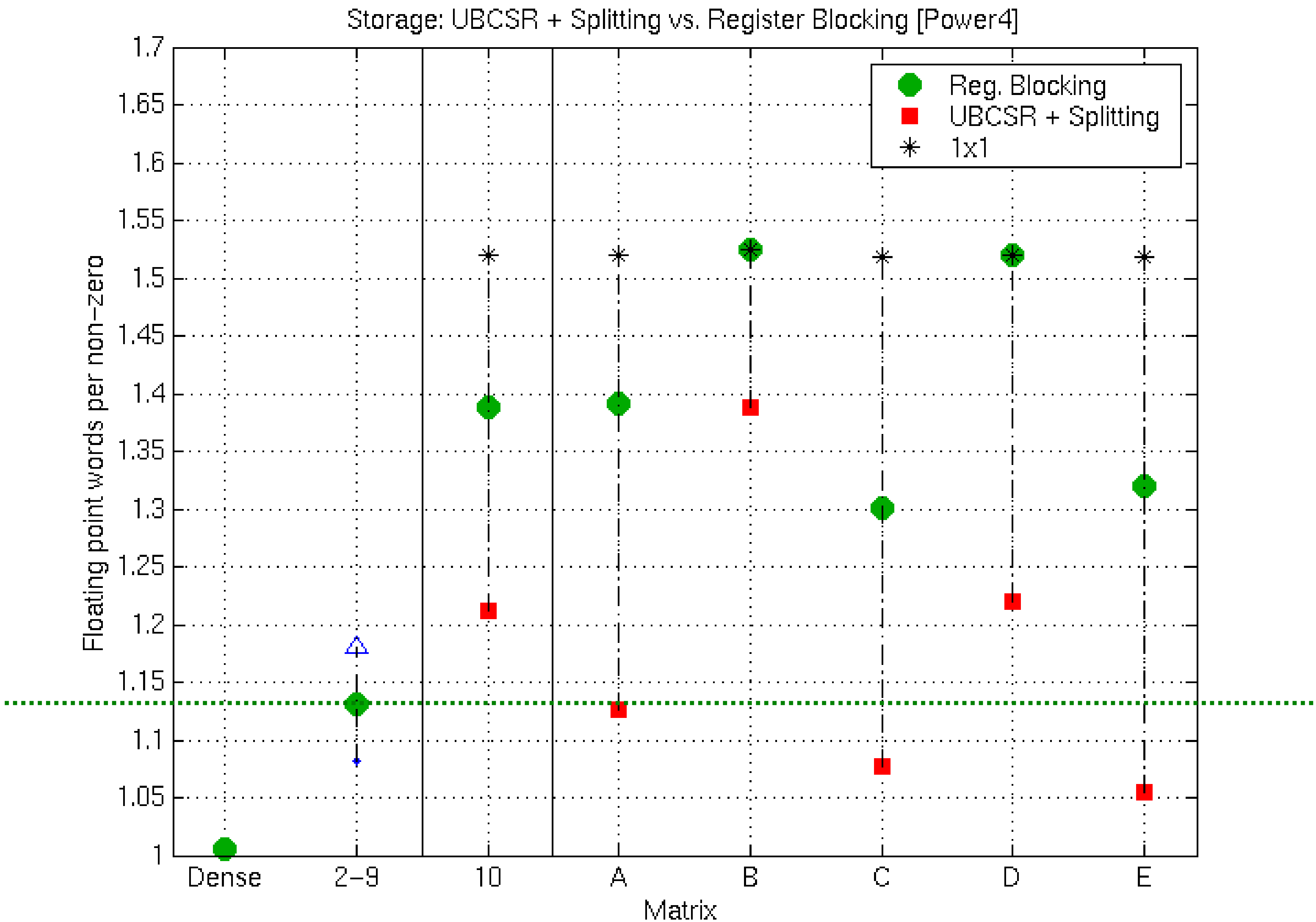


Summary of the Cost of Tuning
Heuristic
Conversion
Total
Time (no. of unblocked SpMVs)
0
2.5
5
7.5
10
12.5
15
17.5
20
22.5
25
27.5
30
32.5
35
37.5
40
42.5
45
Ultra 2i
Ultra 3
Pentium III
Pentium III-M
Power3
Power4
Itanium 1
Itanium 2

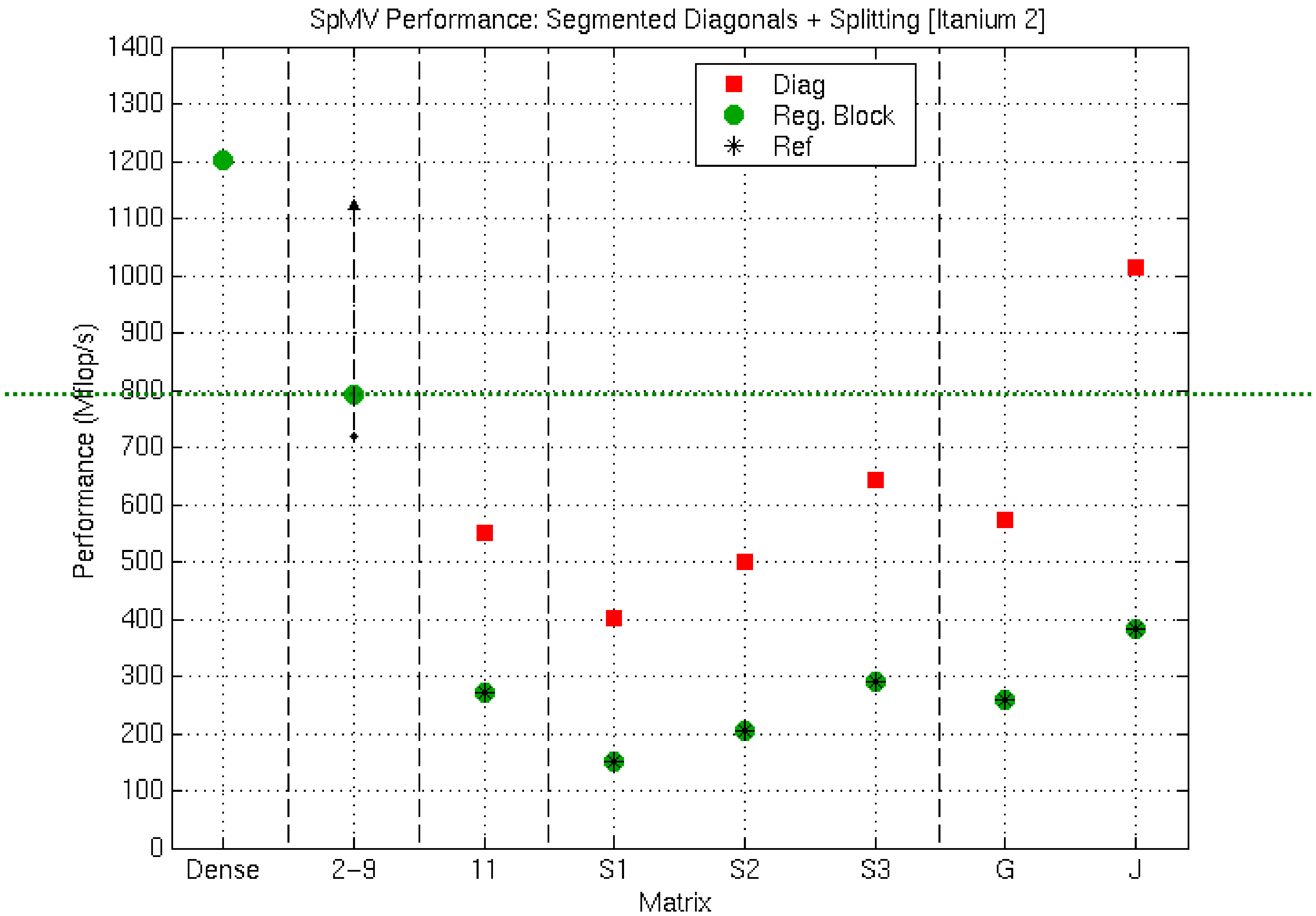


SpMV Performance: UBCSR Format + Splitting vs. Register Blocking [Pentium III–M]
Performance (Mflop/s)
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
0.175
0.1625
0.15
0.1375
0.125
0.1125
0.1
0.0875
0.075
0.0625
0.05
0.0375
0.025
0.0125
0
Dense
2–9
10
12
13
15
17
A
B
C
D
E
Matrix
Reg. Blocking
UBCSR + Splitting
(fastest part)
(slowest part)
1x1

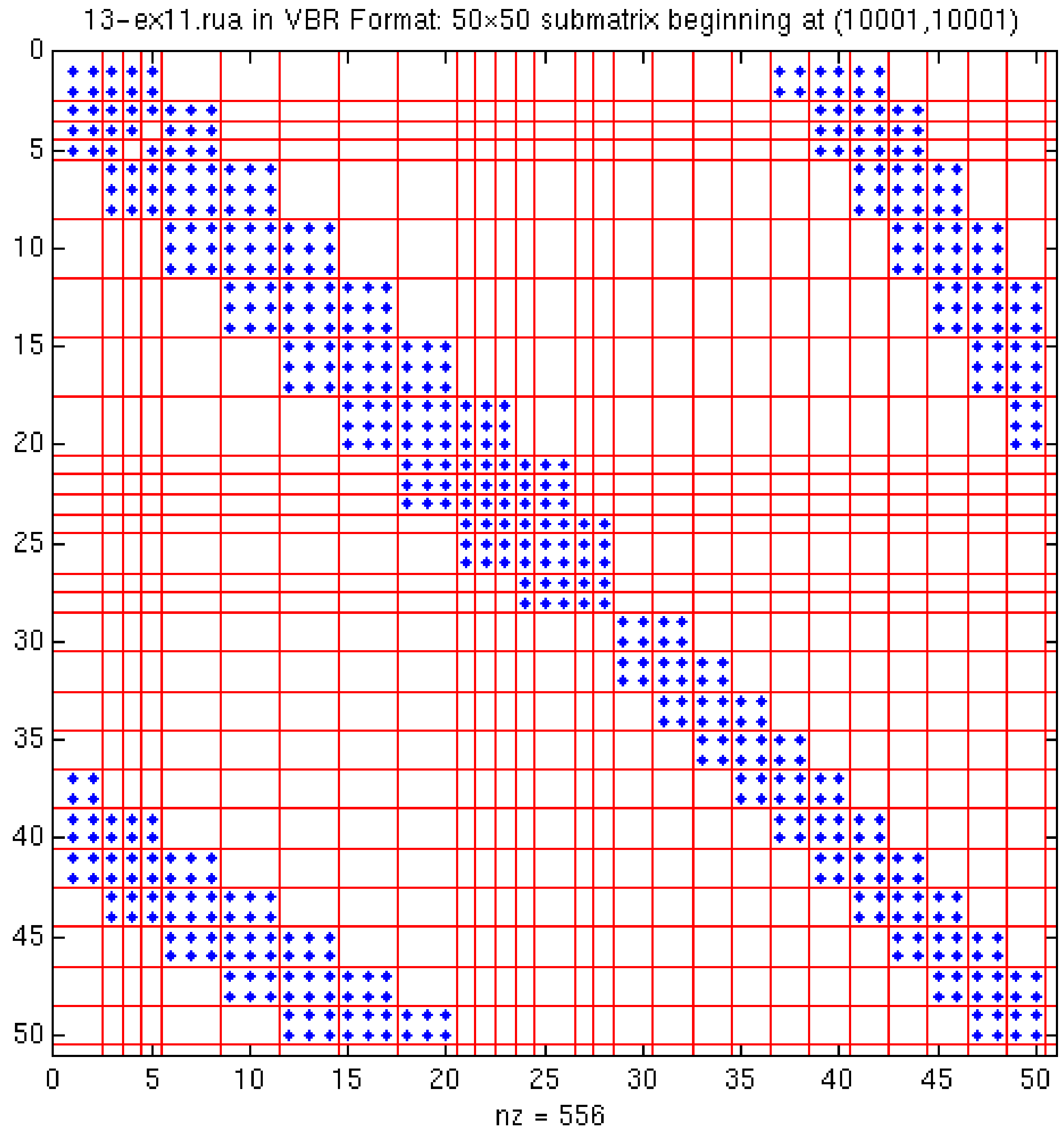

SpMV Performance: UBCSR Format + Splitting vs. Register Blocking [Power4]
Performance (Mflop/s)
900
850
800
750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
0.1731
0.1635
0.1538
0.1442
0.1346
0.125
0.1154
0.1058
0.0962
0.0865
0.0769
0.0673
0.0577
0.0481
0.0385
0.0288
0.0192
0.0096
0
Dense
2−9
10
A
B
C
D
E
Matrix
Reg. Blocking
UBCSR + Splitting
(fastest part)
(slowest part)
1x1



Storage: UBCSR + Splitting vs. Register Blocking [Power4]
Floating point words per non−zero
1.7
1.65
1.6
1.55
1.5
1.45
1.4
1.35
1.3
1.25
1.2
1.15
1.1
1.05
1
Dense
2−9
10
A
B
C
D
E
Matrix
Reg. Blocking
UBCSR + Splitting
1x1

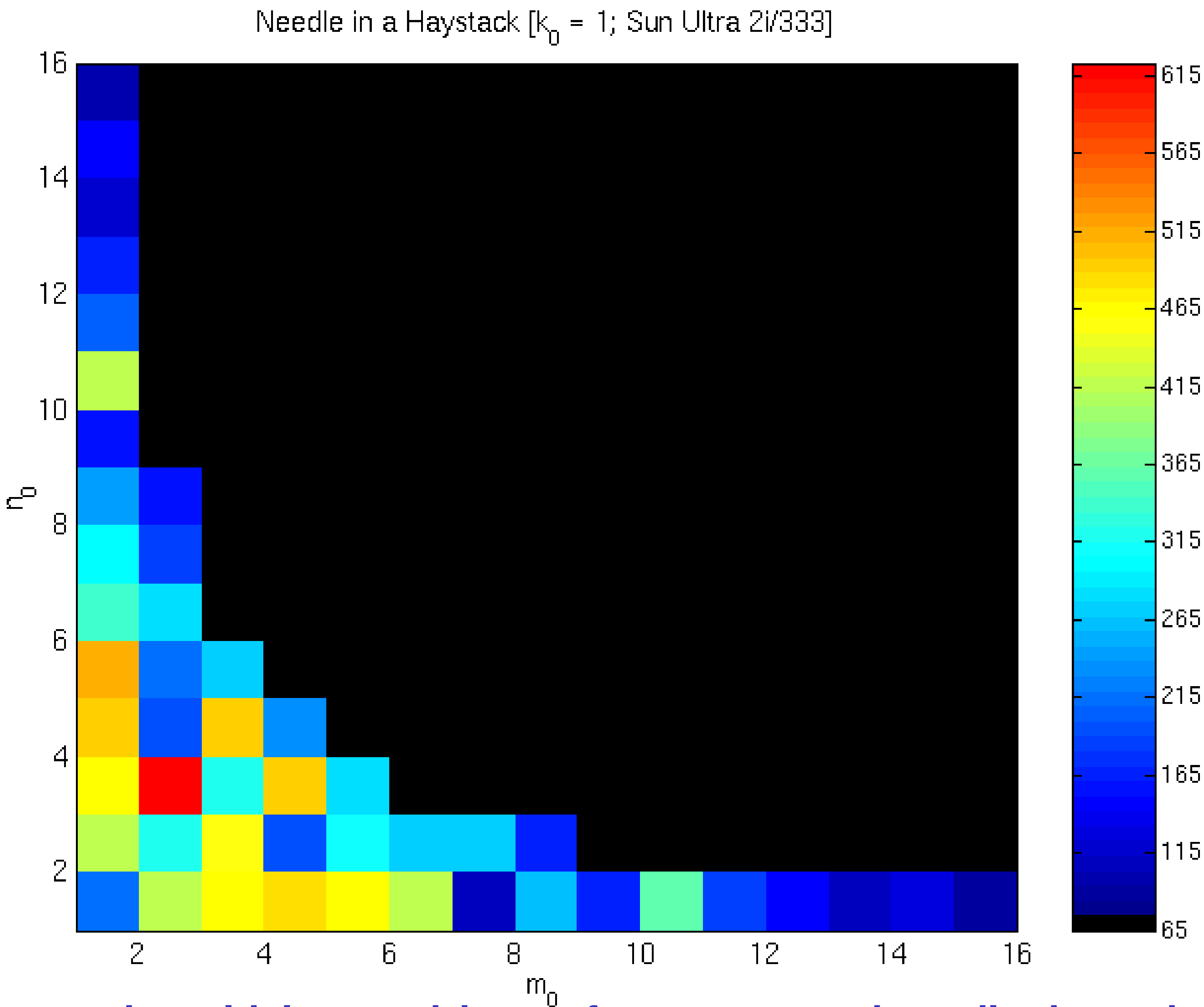


SpMV Performance: Segmented Diagonals + Splitting [Itanium 2]
Diag
Reg. Block
Ref
Performance (Mflop/s)
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
Dense
2−9
11
S1
S2
S3
G
J
Matrix

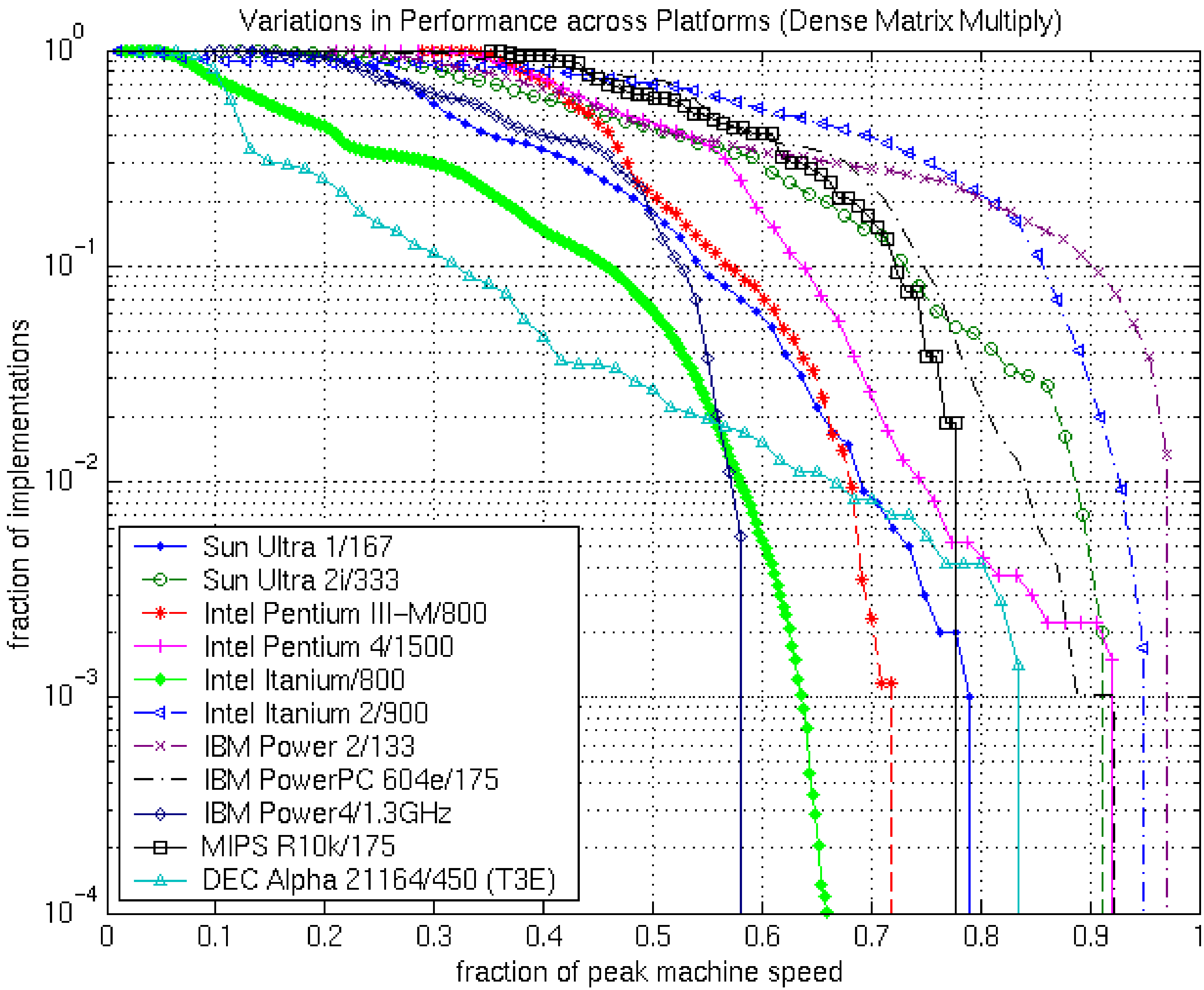

13-ex11.rua in VBR Format: 50×50 submatrix beginning at (10001,10001)
0
5
10
15
20
25
30
35
40
45
50
nz = 556

Dense Tuning is Hard, Too

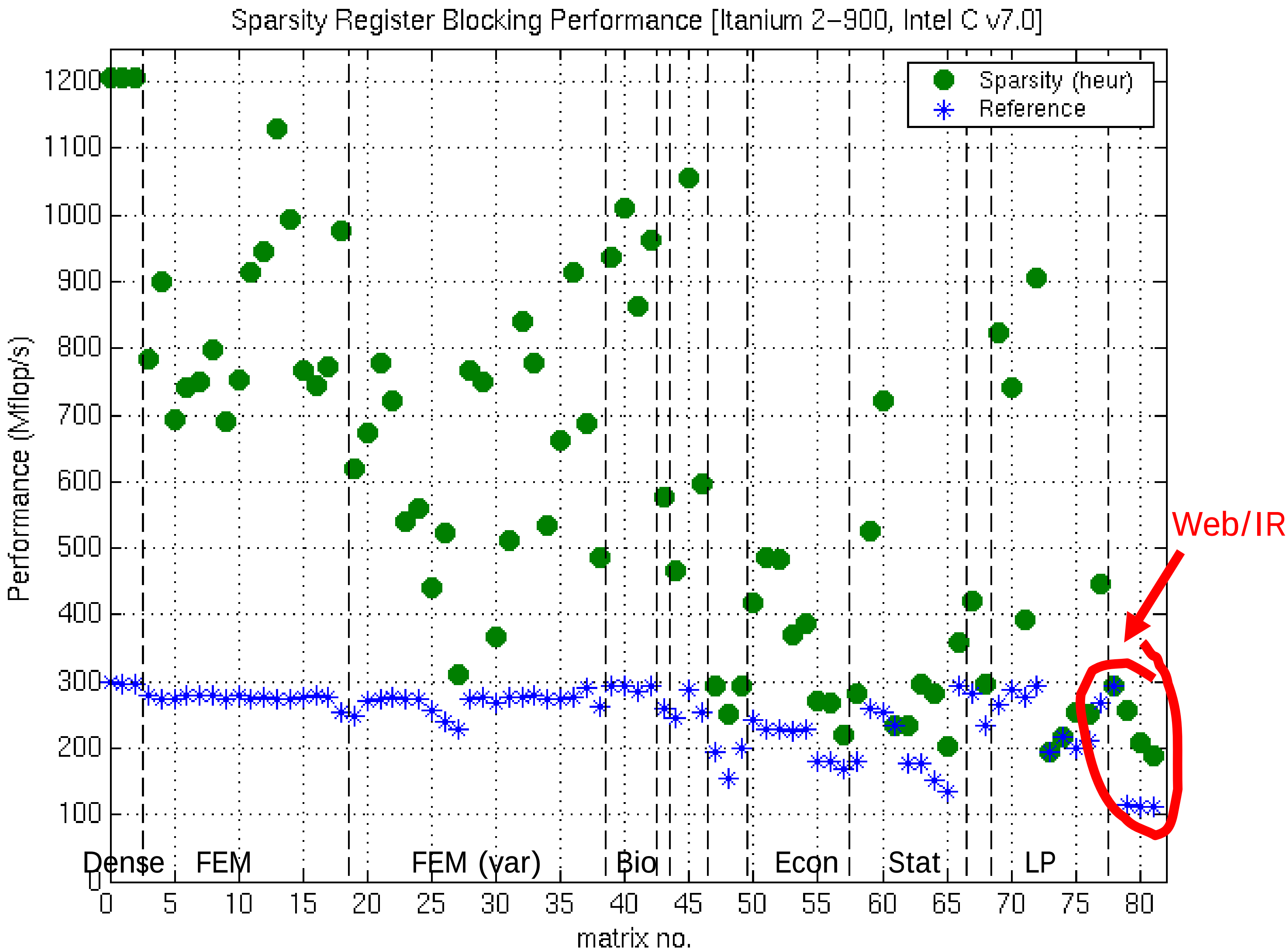
- Even dense matrix multiply can be notoriously difficult to tune



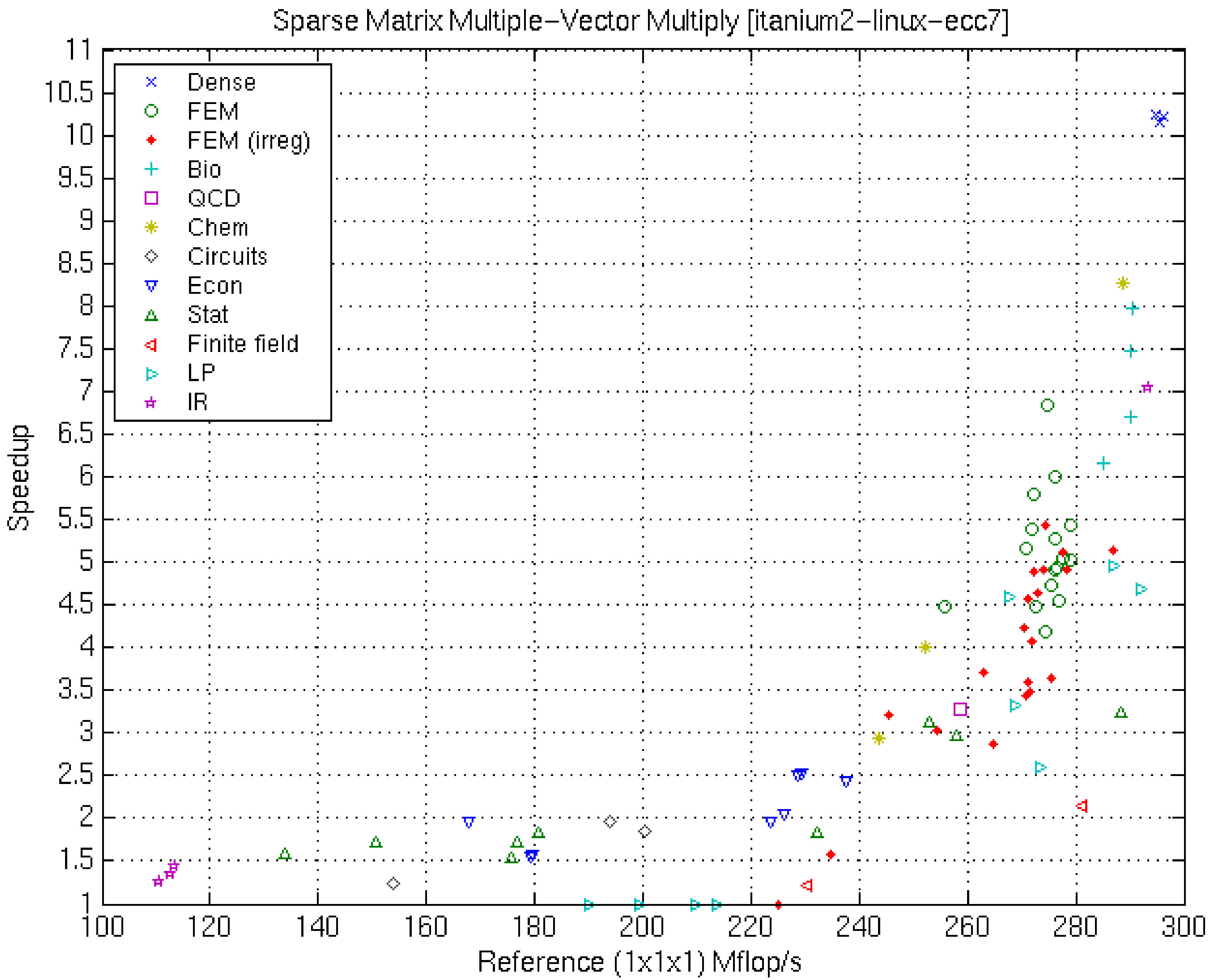
Dense matrix multiply: surprising performance as register tile size varies.

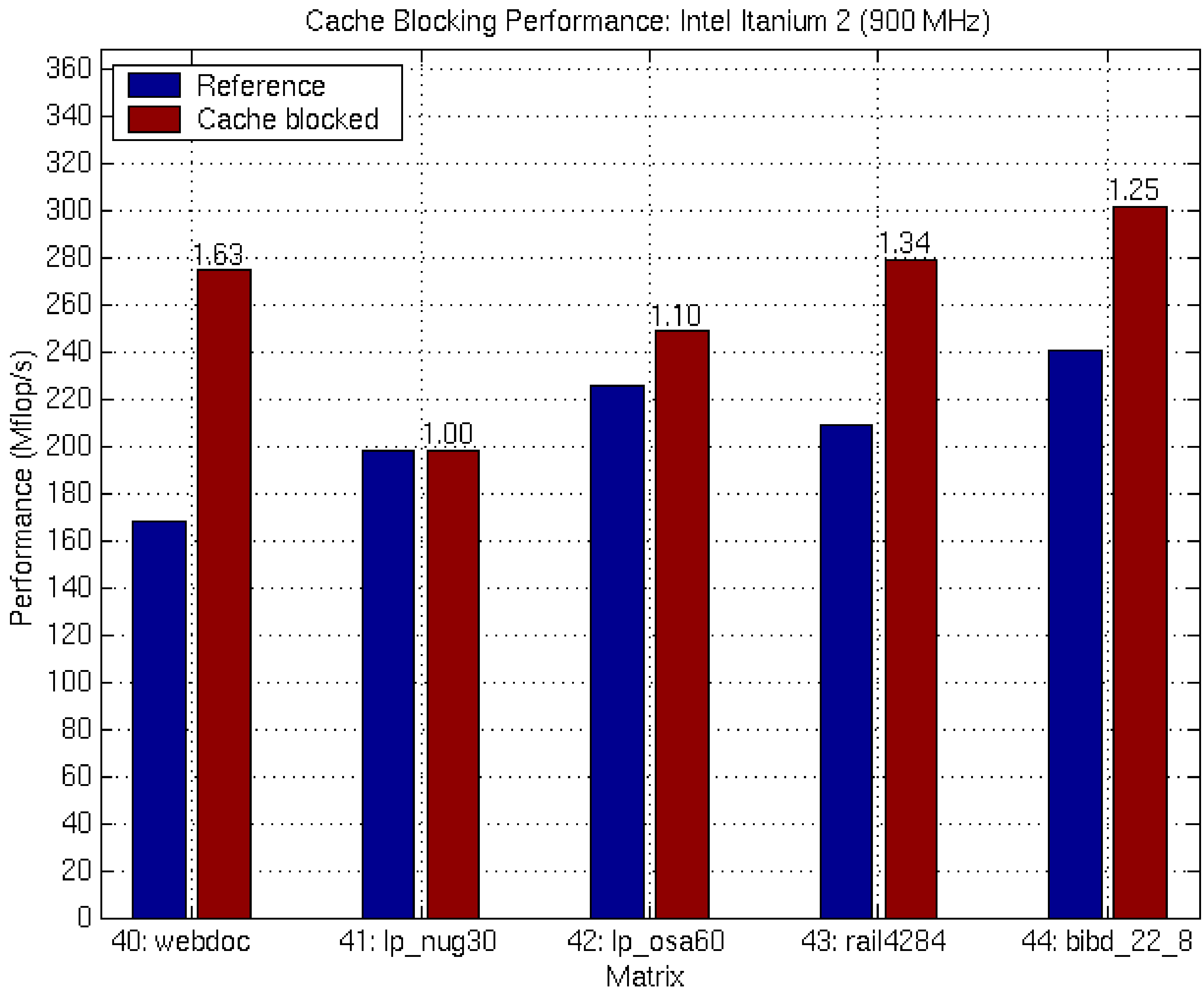


Variations in Performance across Platforms (Dense Matrix Multiply)
fraction of implementations
fraction of peak machine speed
10^0
10^-1
10^-2
10^-3
10^-4
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Sun Ultra 1/167
Sun Ultra 2i/333
Intel Pentium III−M/800
Intel Pentium 4/1500
Intel Itanium/800
Intel Itanium 2/900
IBM Power 2/133
IBM PowerPC 604e/175
IBM Power4/1.3GHz
MIPS R10k/175
DEC Alpha 21164/450 (T3E)



Sparsity Register Blocking Performance [Itanium 2−900, Intel C v7.0]
Sparsity (heur)
Reference
Performance (Mflop/s)
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
Dense
FEM
FEM (var)
Bio
Econ
Stat
LP
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
matrix no.
Web/IR

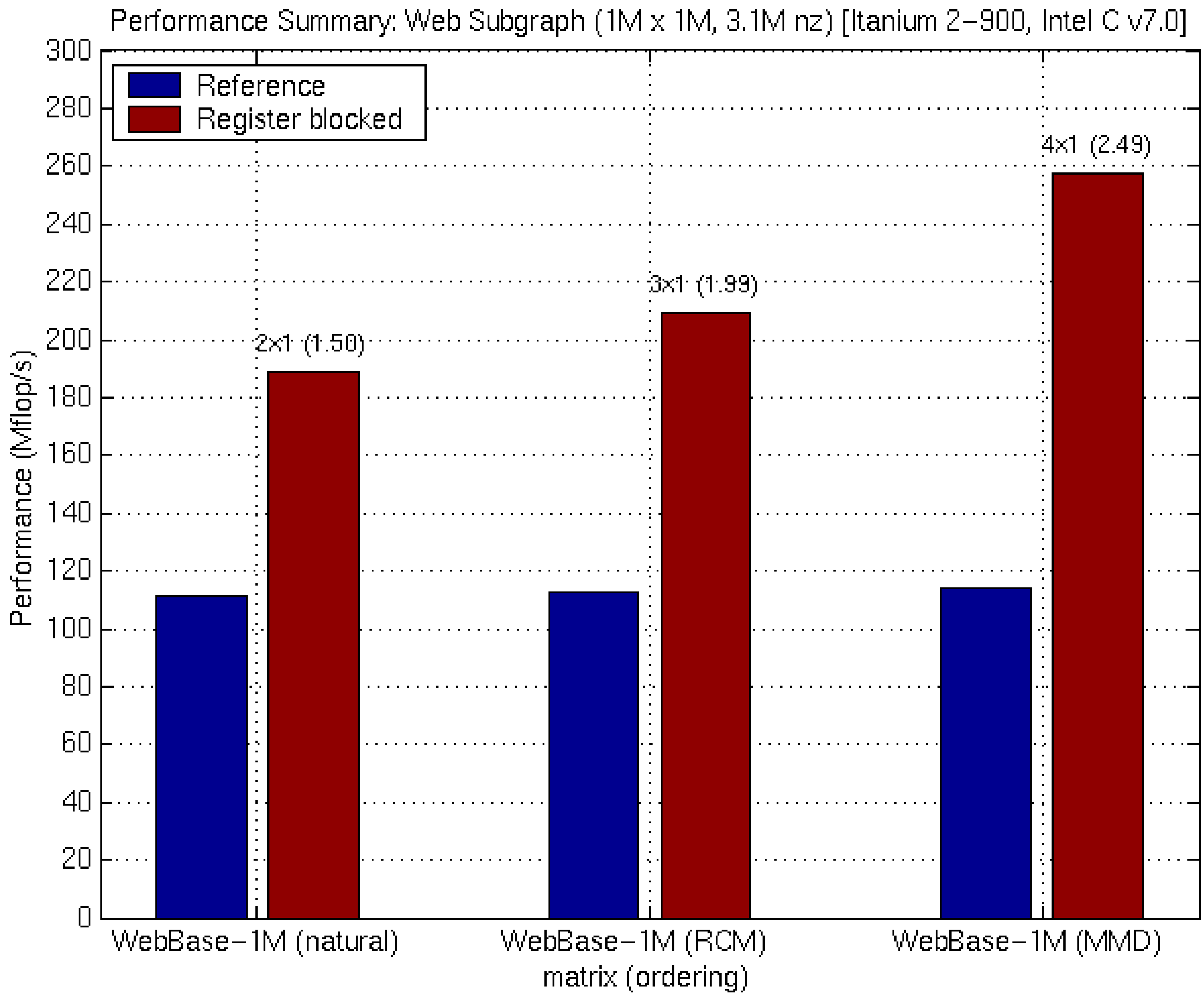

Sparse Matrix Multiple−Vector Multiply [itanium2−linux−ecc7]
Dense
FEM
FEM (irreg)
Bio
QCD
Chem
Circuits
Econ
Stat
Finite field
LP
IR
Speedup
Reference (1x1x1) Mflop/s
11
10.5
10
9.5
9
8.5
8
7.5
7
6.5
6
5.5
5
4.5
4
3.5
3
2.5
2
1.5
1
100
120
140
160
180
200
220
240
260
280
300



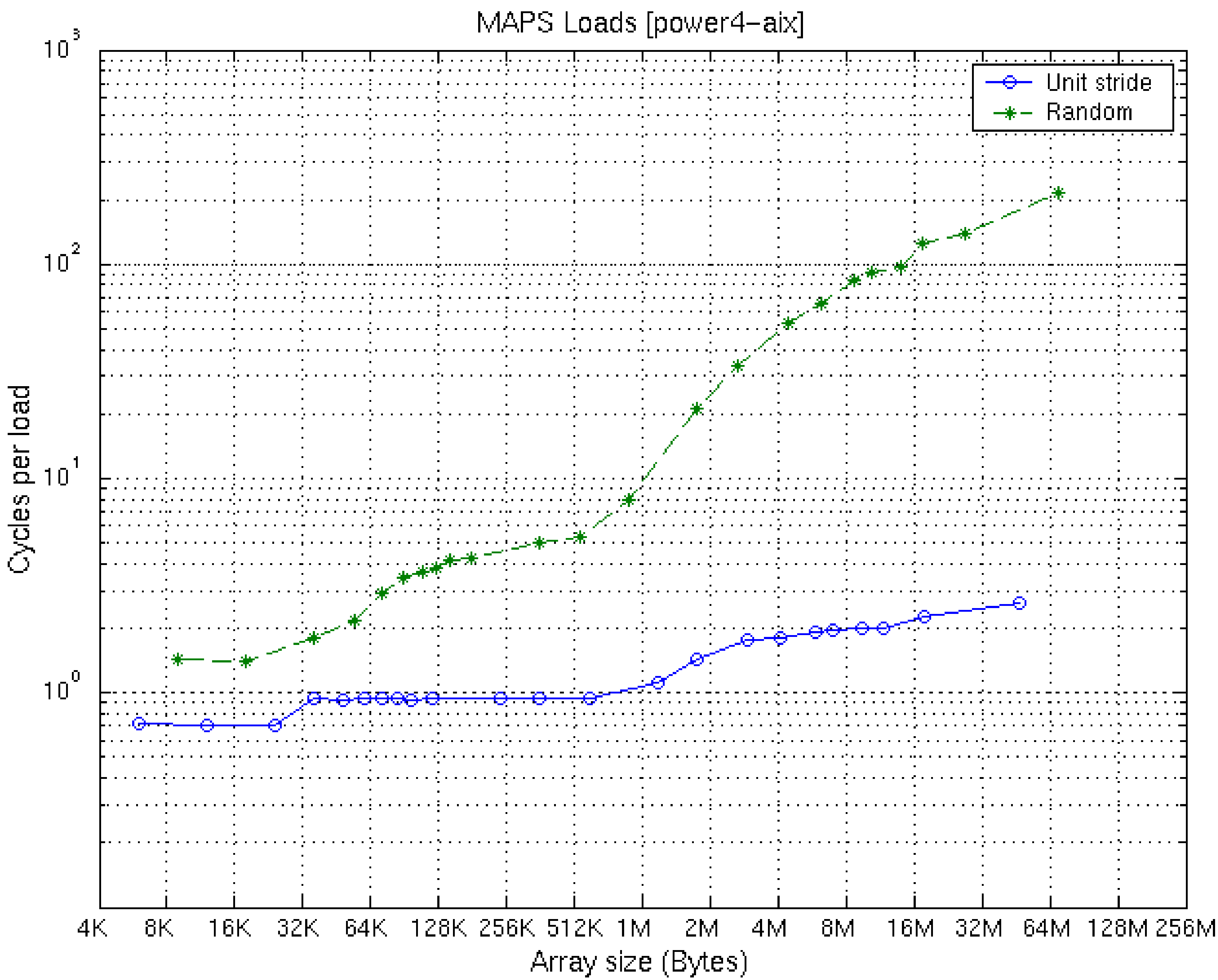
Cache Blocking Performance: Intel Itanium 2 (900 MHz)
Reference
Cache blocked
Performance (Mflop/s)
0
20
40
60
80
100
120
140
160
180
200
220
240
260
280
300
320
340
360
1.63
1.00
1.10
1.34
1.25
40: webdoc
41: lp_nug30
42: lp_osa60
43: rail4284
44: bibd_22_8
Matrix

What about the Google Matrix?

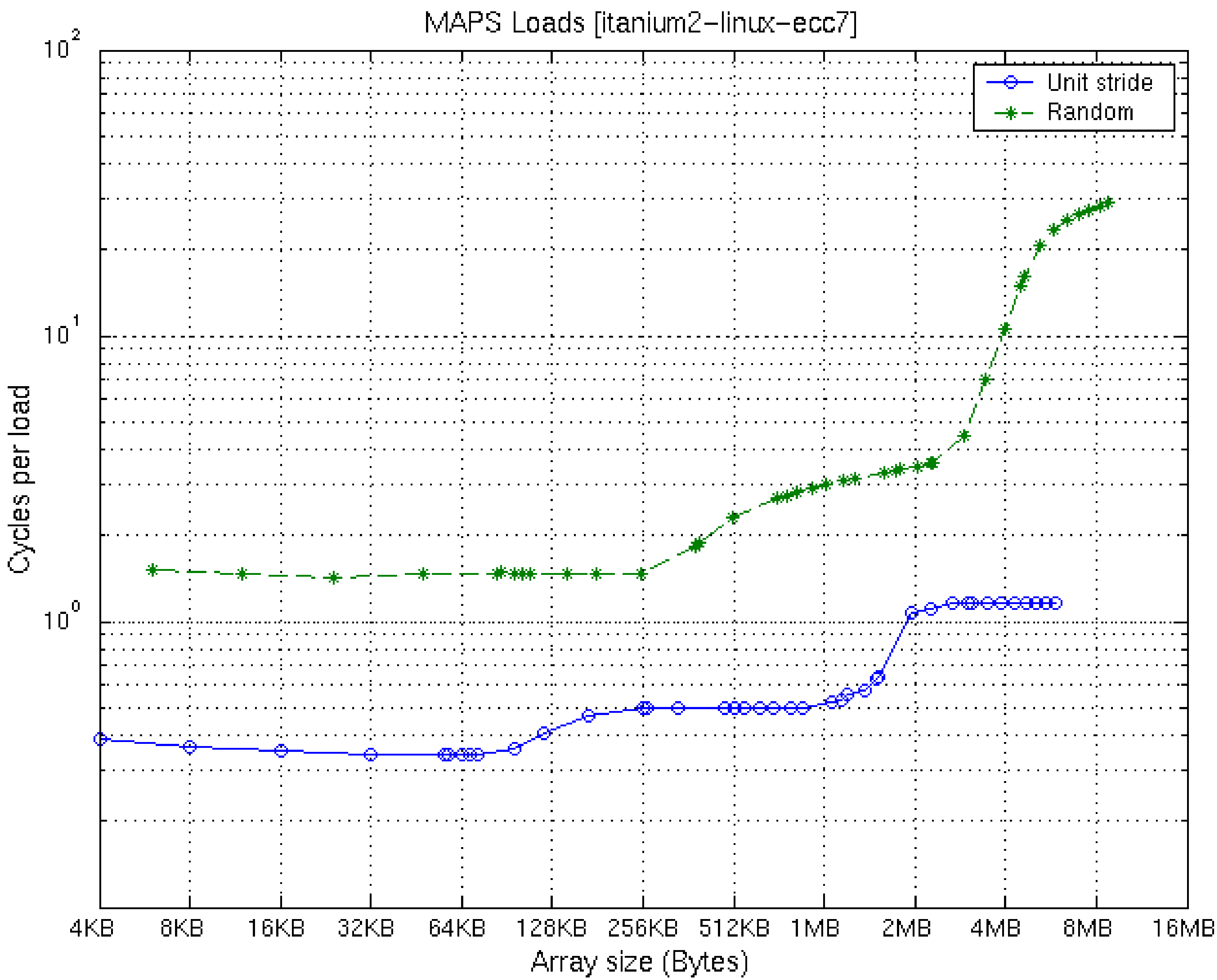
- Google approach
  - Approx. once a month: rank all pages using connectivity structure
    - Find dominant eigenvector of a matrix
  - At query-time: return list of pages ordered by rank
- Matrix: $A = \alpha G + (1-\alpha)(1/n)uu^T$
  - Markov model: Surfer follows link with probability α, jumps to a random page with probability $1-\alpha$
  - G is n x 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, $k \approx 25$

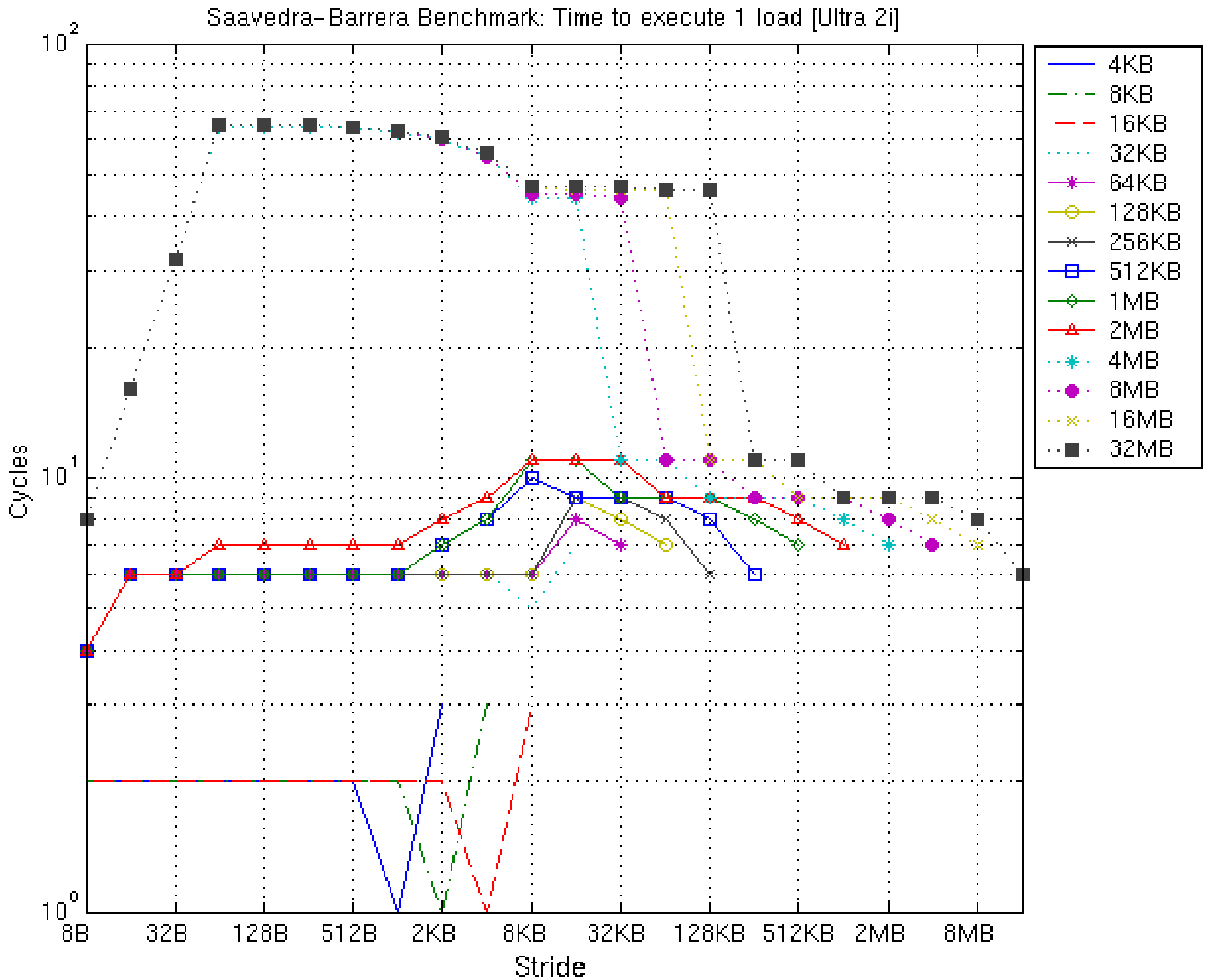


Performance Summary: Web Subgraph (1M x 1M, 3.1M nz) [Itanium 2−900, Intel C v7.0]
Reference
Register blocked
2x1 (1.50)
3x1 (1.99)
4x1 (2.49)
Performance (Mflop/s)
0
20
40
60
80
100
120
140
160
180
200
220
240
260
280
300
WebBase−1M (natural)
WebBase−1M (RCM)
WebBase−1M (MMD)
matrix (ordering)

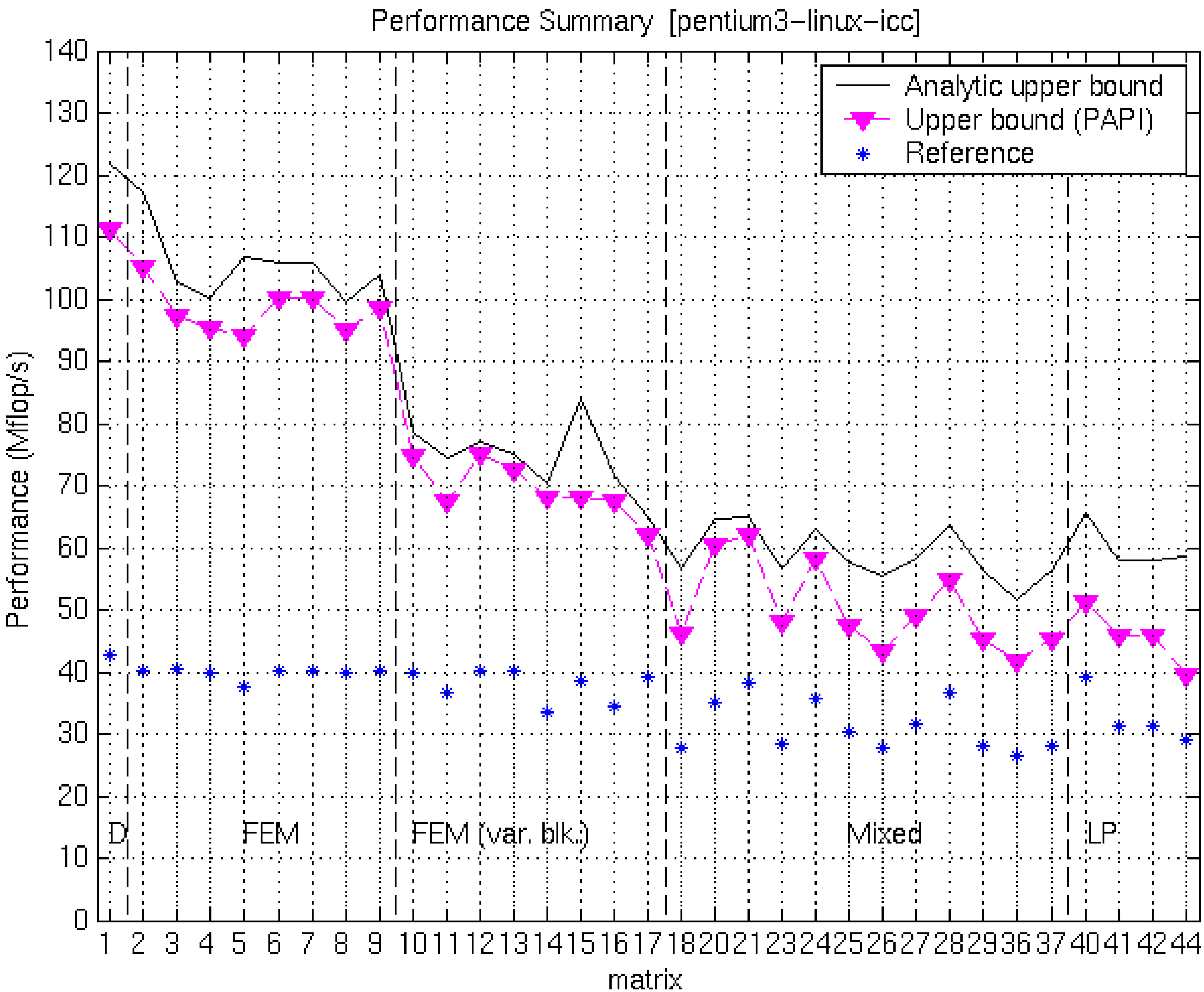


MAPS Loads [power4-aix]
Unit stride
Random
Cycles per load
10^3
10^2
10^1
10^0
4K 8K 16K 32K 64K 128K 256K 512K 1M 2M 4M 8M 16M 32M 64M 128M 256M
Array size (Bytes)

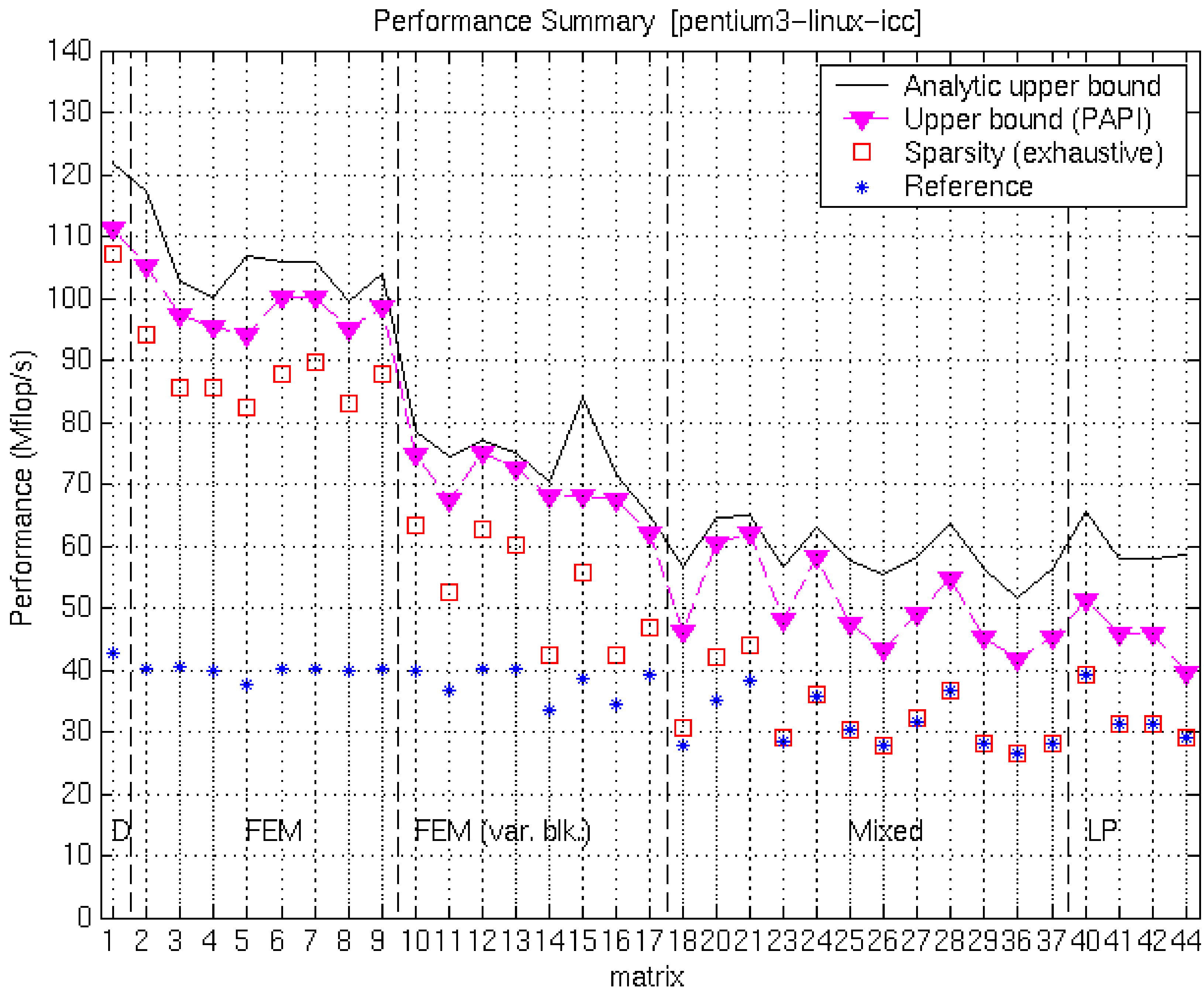

MAPS Loads [itanium2-linux-ecc7]
Unit stride
Random
Cycles per load
10^2
10^1
10^0
4KB
8KB
16KB
32KB
64KB
128KB
256KB
512KB
1MB
2MB
4MB
8MB
16MB
Array size (Bytes)



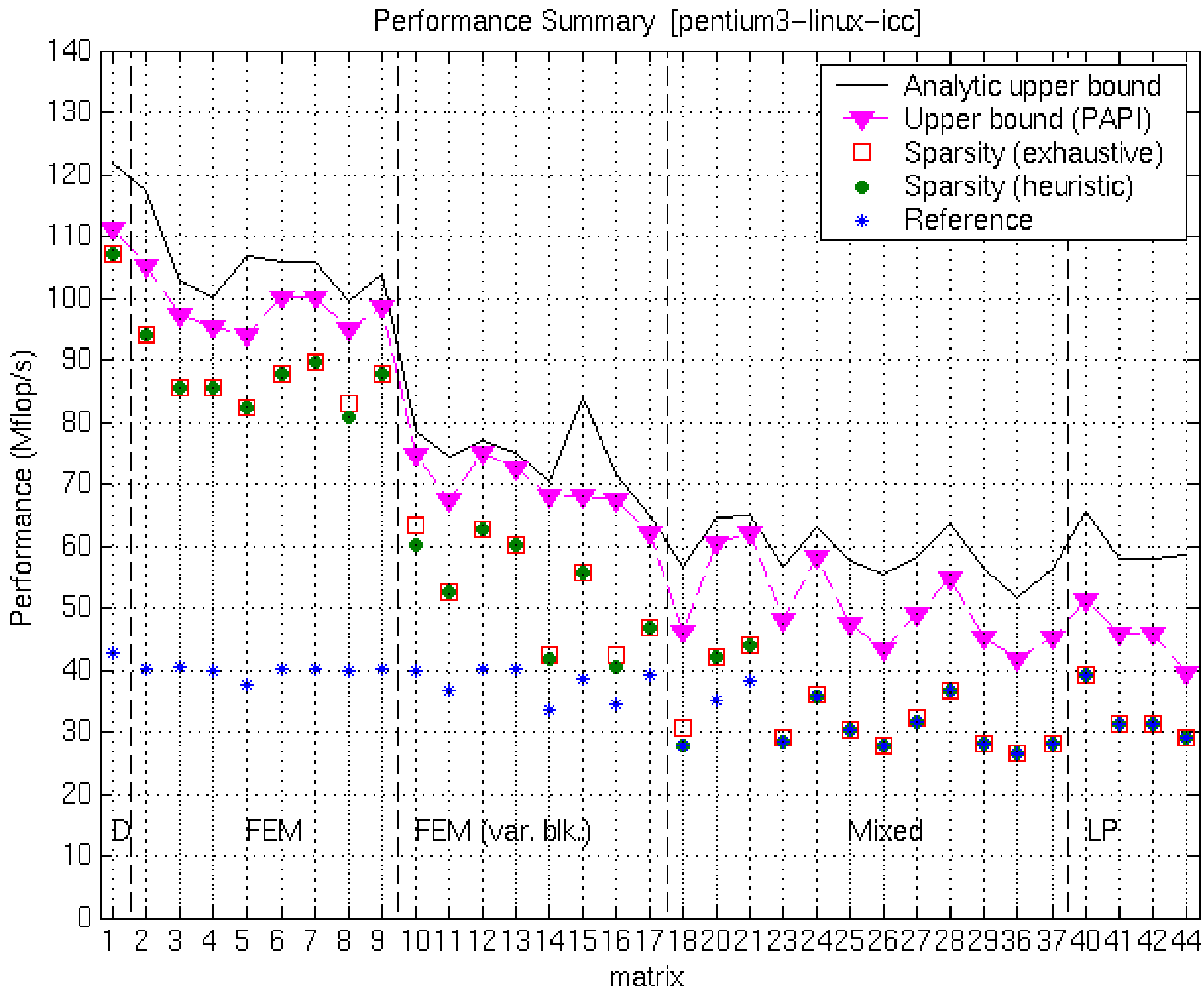
Saavedra-Barrera Benchmark: Time to execute 1 load [Ultra 2i]
Cycles
10^2
10^1
10^0
8B
32B
128B
512B
2KB
8KB
32KB
128KB
512KB
2MB
8MB
Stride
4KB
8KB
16KB
32KB
64KB
128KB
256KB
512KB
1MB
2MB
4MB
8MB
16MB
32MB



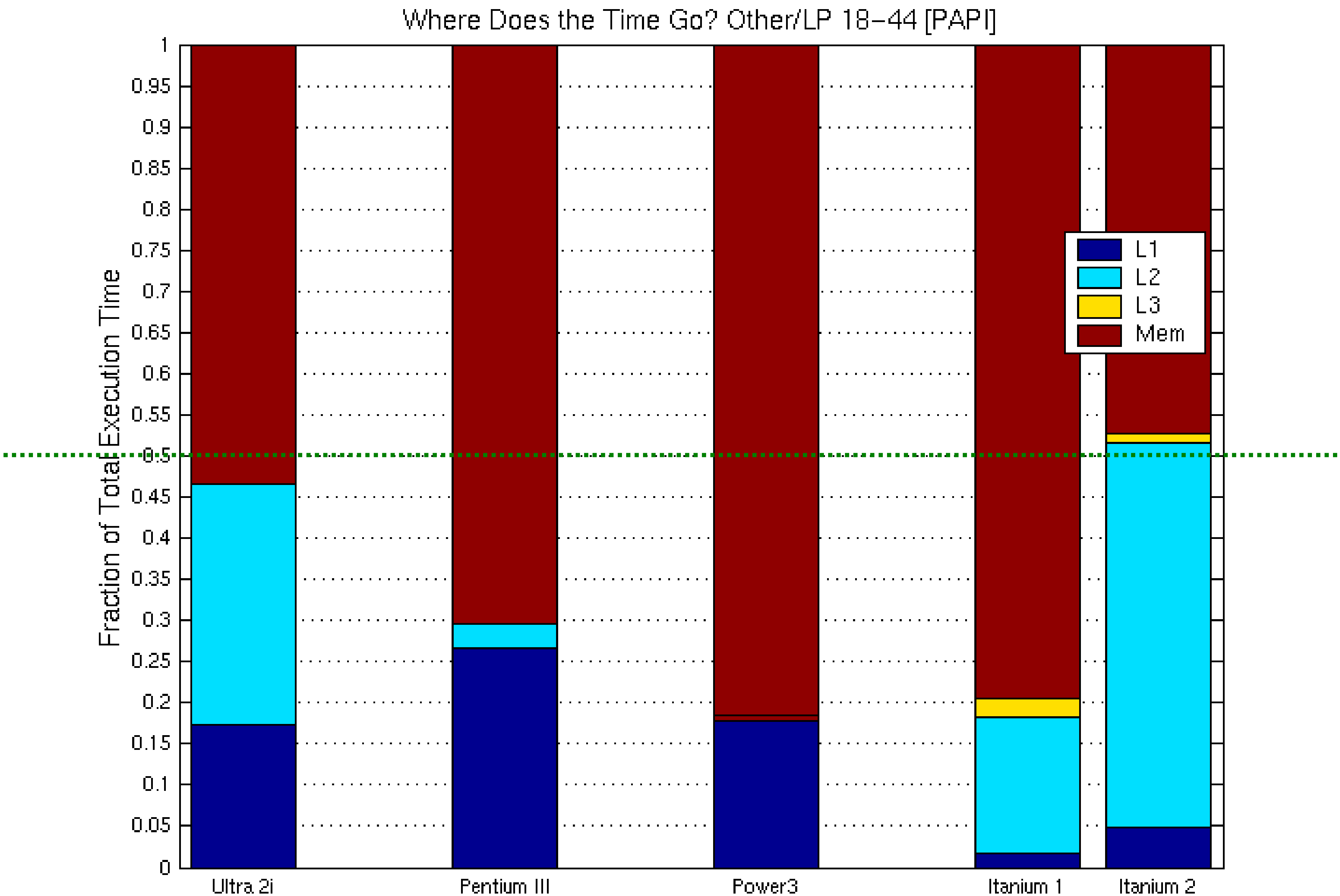
Performance Summary [pentium3-linux-icc]
Analytic upper bound
Upper bound (PAPI)
Reference
Performance (Mflop/s)
matrix
D
FEM
FEM (var. blk.)
Mixed
LP
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 21 23 24 25 26 27 28 29 36 37 40 41 42 44



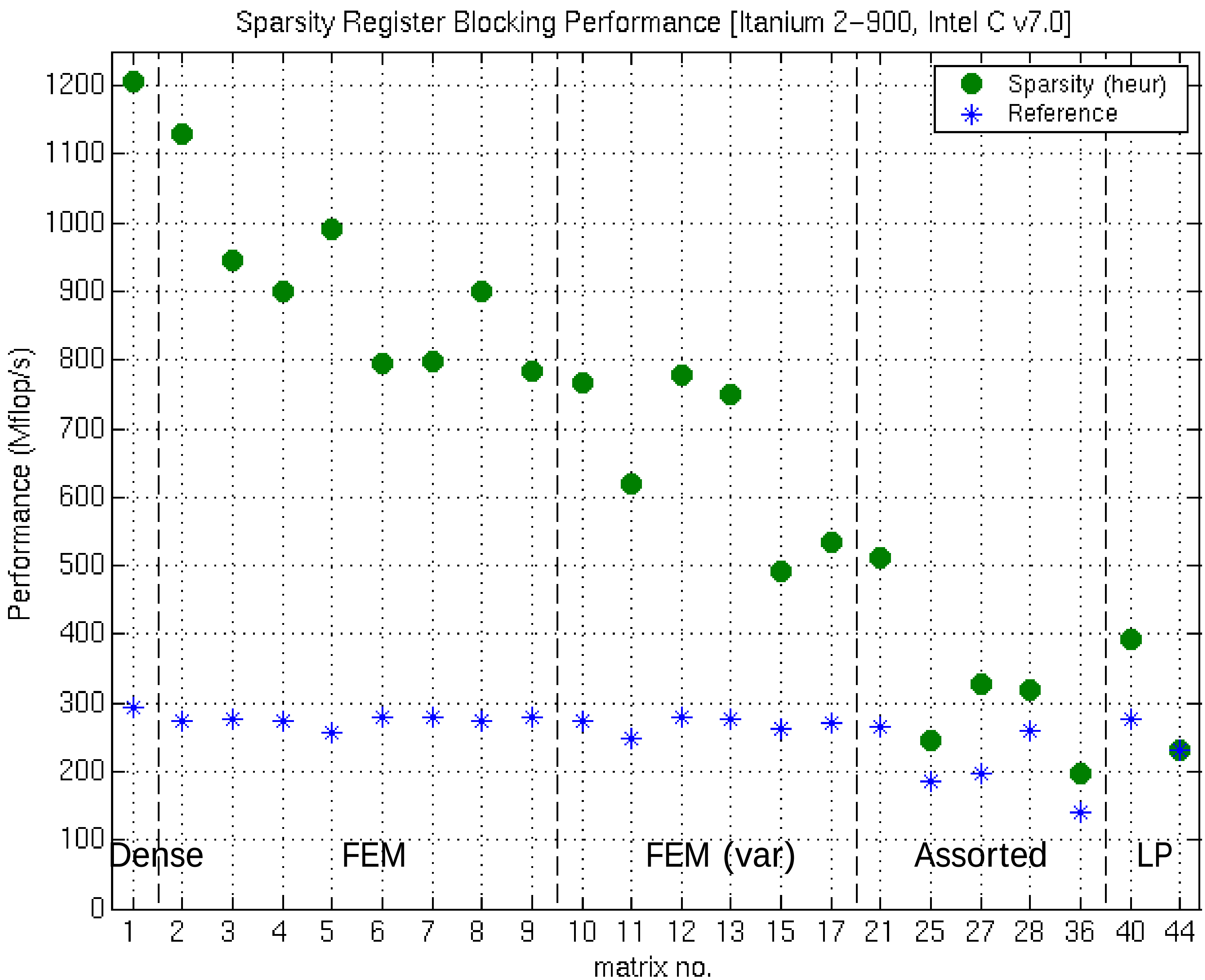
Performance Summary [pentium3-linux-icc]
Analytic upper bound
Upper bound (PAPI)
Sparsity (exhaustive)
Reference
Performance (Mflop/s)
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
D
FEM
FEM (var. blk.)
Mixed
LP
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 21 23 24 25 26 27 28 29 36 37 40 41 42 44
matrix



Performance Summary [pentium3-linux-icc]
Analytic upper bound
Upper bound (PAPI)
Sparsity (exhaustive)
Sparsity (heuristic)
Reference
Performance (Mflop/s)
matrix
D
FEM
FEM (var. blk.)
Mixed
LP
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 20 21 23 24 25 26 27 28 29 36 37 40 41 42 44



Where Does the Time Go? Other/LP 18−44 [PAPI]
Fraction of Total Execution Time
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
L1
L2
L3
Mem
Ultra 2i
Pentium III
Power3
Itanium 1
Itanium 2



Sparsity Register Blocking Performance [Itanium 2−900, Intel C v7.0]
Sparsity (heur)
Reference
Performance (Mflop/s)
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
Dense
FEM
FEM (var)
Assorted
LP
1
2
3
4
5
6
7
8
9
10
11
12
13
15
17
21
25
27
28
36
40
44
matrix no.

Tuning Sparse Triangular Solve (SpTS)

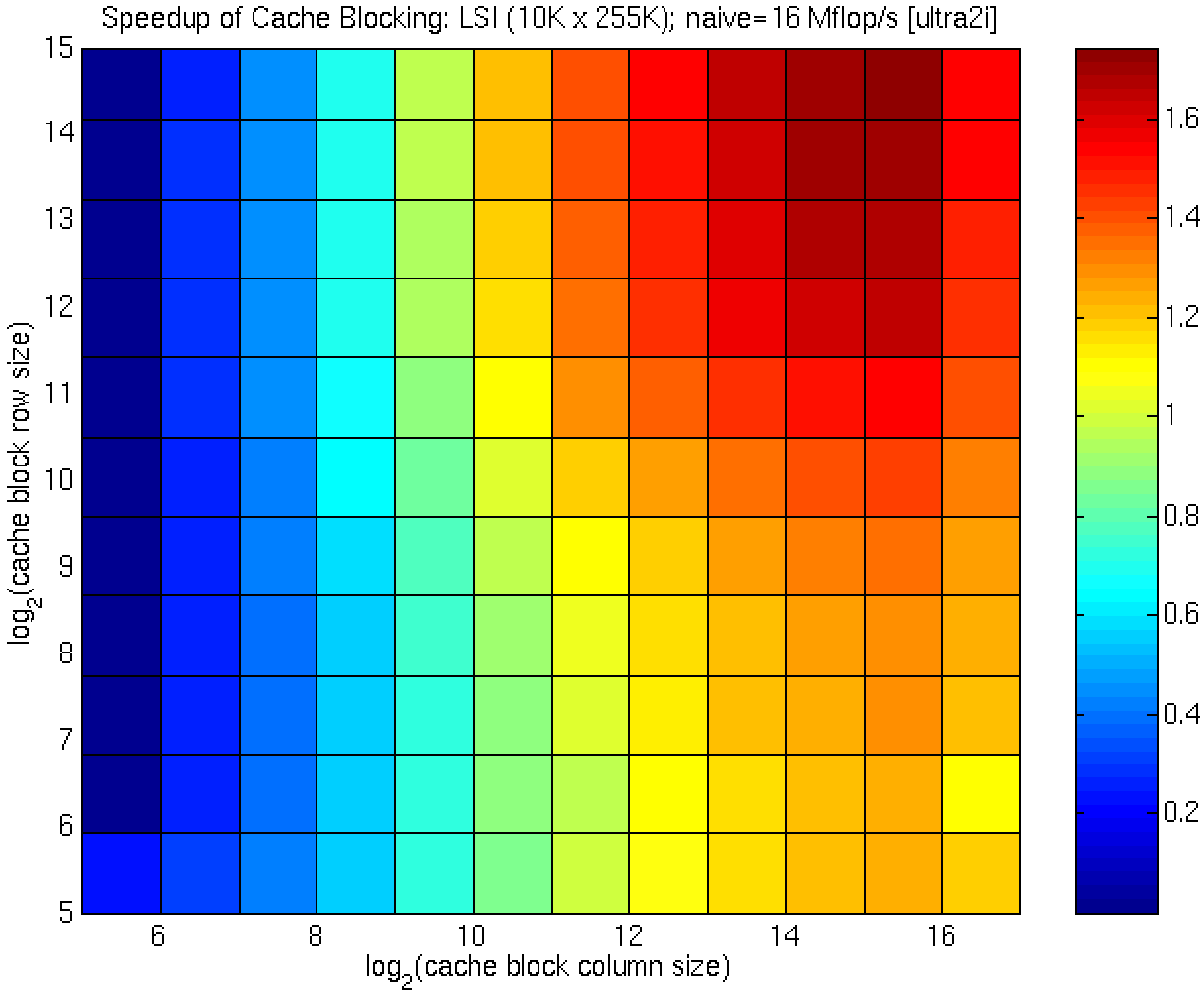
- Compute $x=L^{-1}*b$ where L sparse lower triangular, x & b dense
- L from sparse LU has rich dense substructure
  - Dense *trailing triangle* can account for 20—90% of matrix non-zeros
- SpTS optimizations
  - Split into sparse trapezoid and dense trailing triangle
  - Use tuned dense BLAS (DTRSV)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

Sparse Kernels and Optimizations

- Kernels
  - **Sparse matrix-vector multiply (SpMV): $y=A*x$**
  - Sparse triangular solve (SpTS): $x=T^{-1}*b$
  - $y=AA^T*x$, $y=A^TA*x$
  - 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

Berkeley Benchmarking and OPtimization Group
bebop.cs.berkeley.edu

Cache Blocked SpMV on LSI Matrix: Ultra 2i

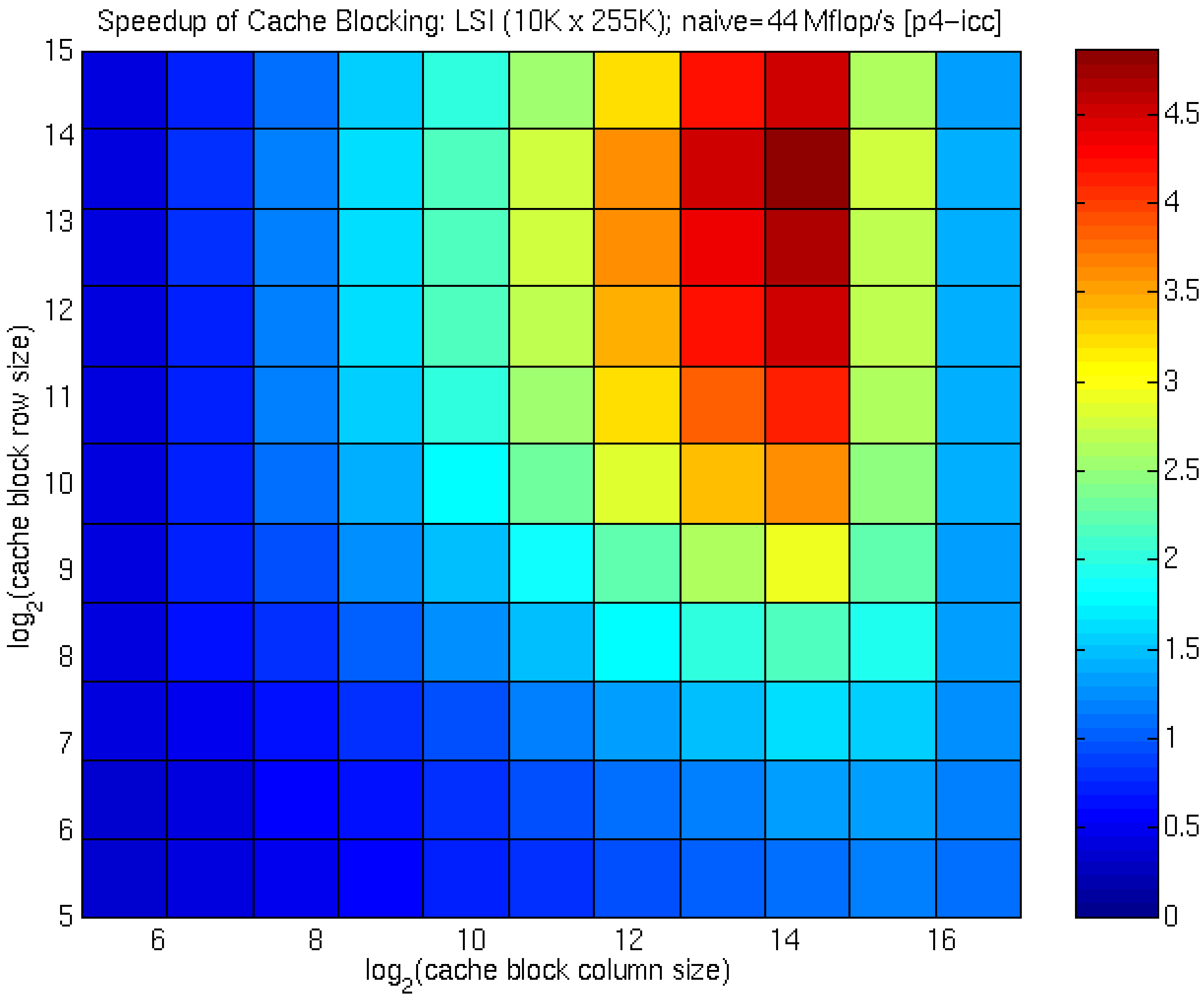


A
10k x 255k
3.7M non-zeros

Baseline:
16 Mflop/s

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

Cache Blocking on LSI Matrix: Pentium 4

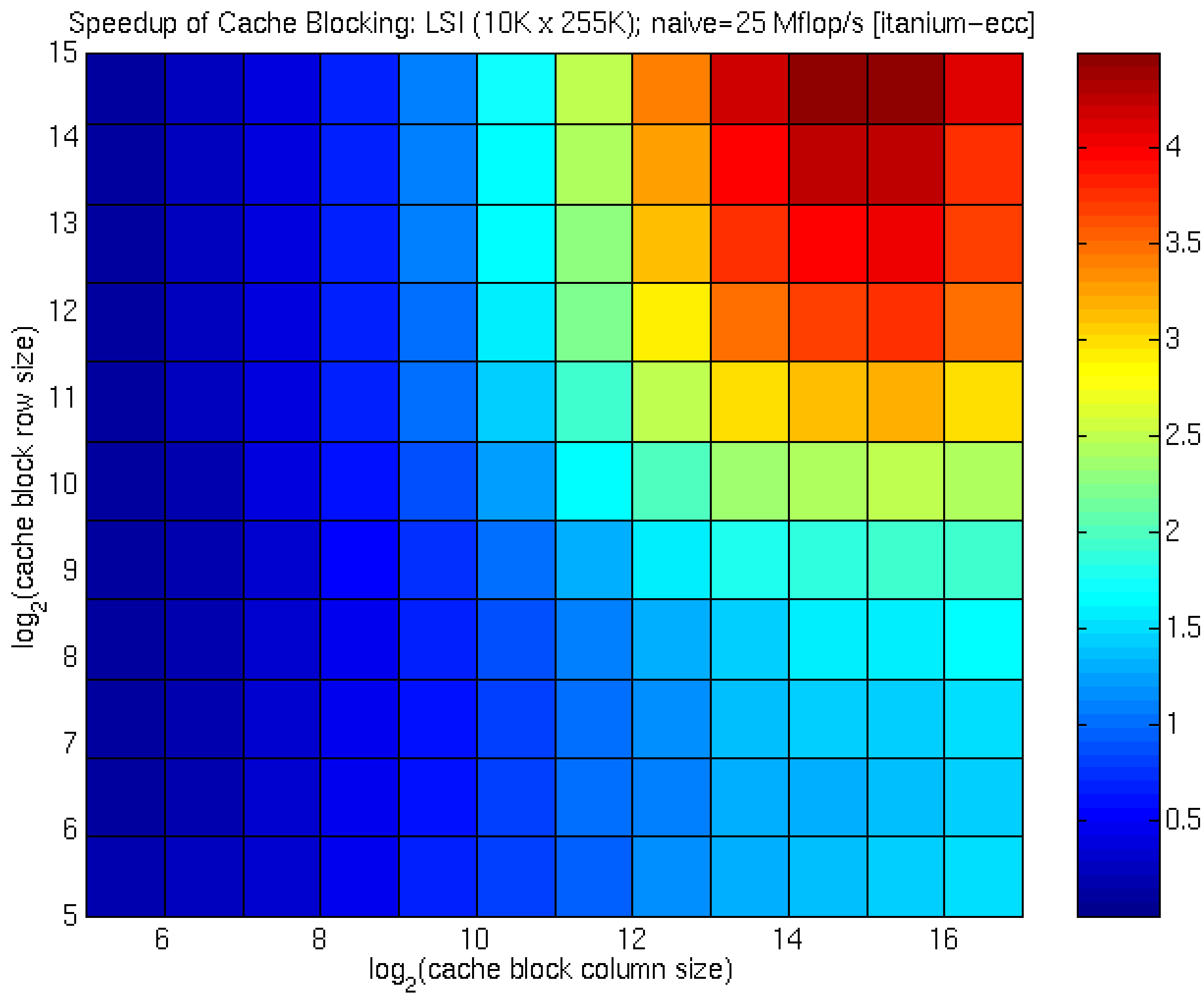


A
10k x 255k
3.7M non-zeros

Baseline:
44 Mflop/s

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

Cache Blocked SpMV on LSI Matrix: Itanium

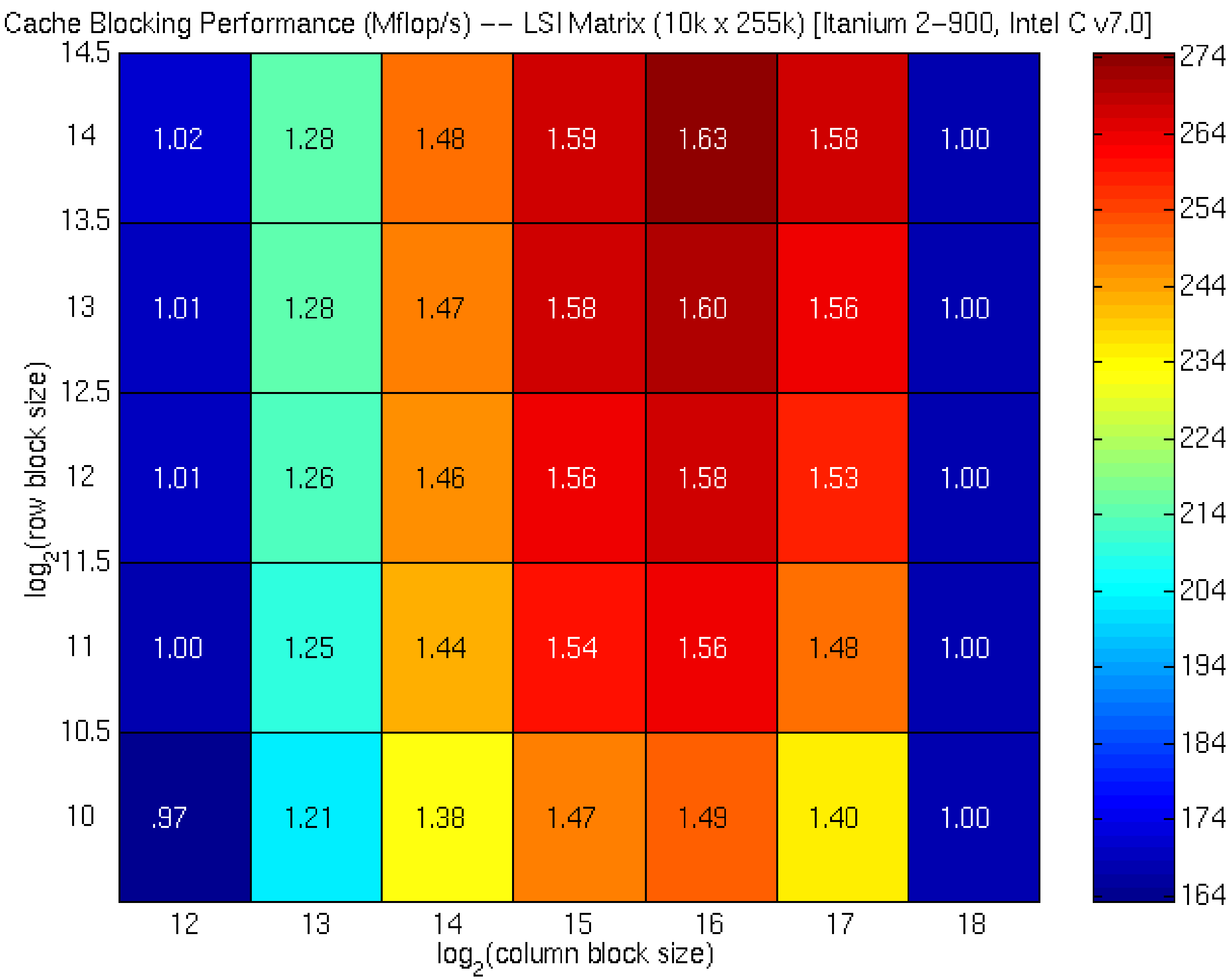


A
10k x 255k
3.7M non-zeros

Baseline:
25 Mflop/s

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

Cache Blocked SpMV on LSI Matrix: Itanium 2



A
10k x 255k
3.7M non-zeros

Baseline:
170 Mflop/s

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

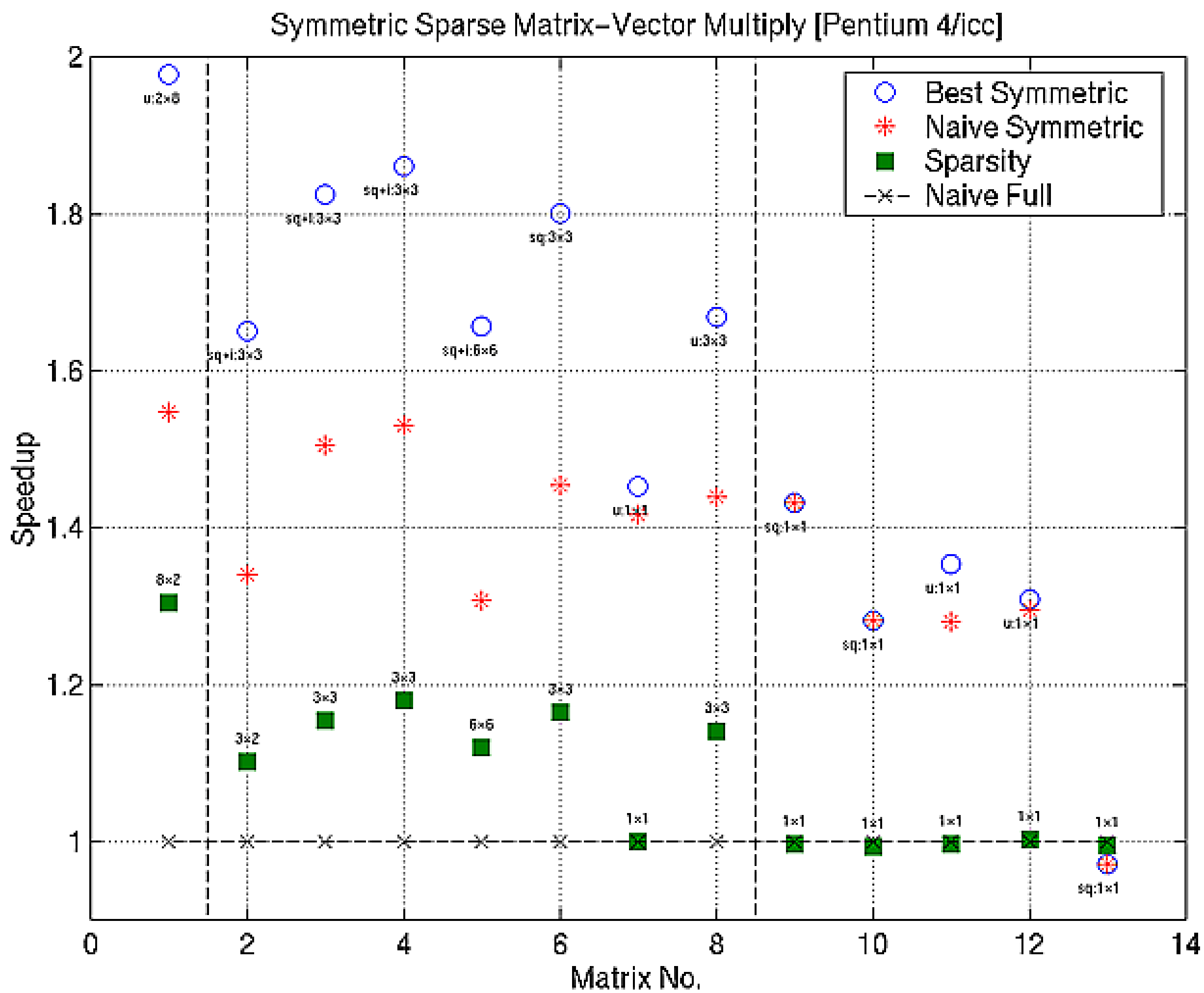
Summary and Questions

- 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 → multiple vector version [Haveliwala '02]
  - See www.cs.berkeley.edu/~richie/bebop/intel/google for more info, including more complete Itanium 2 results.

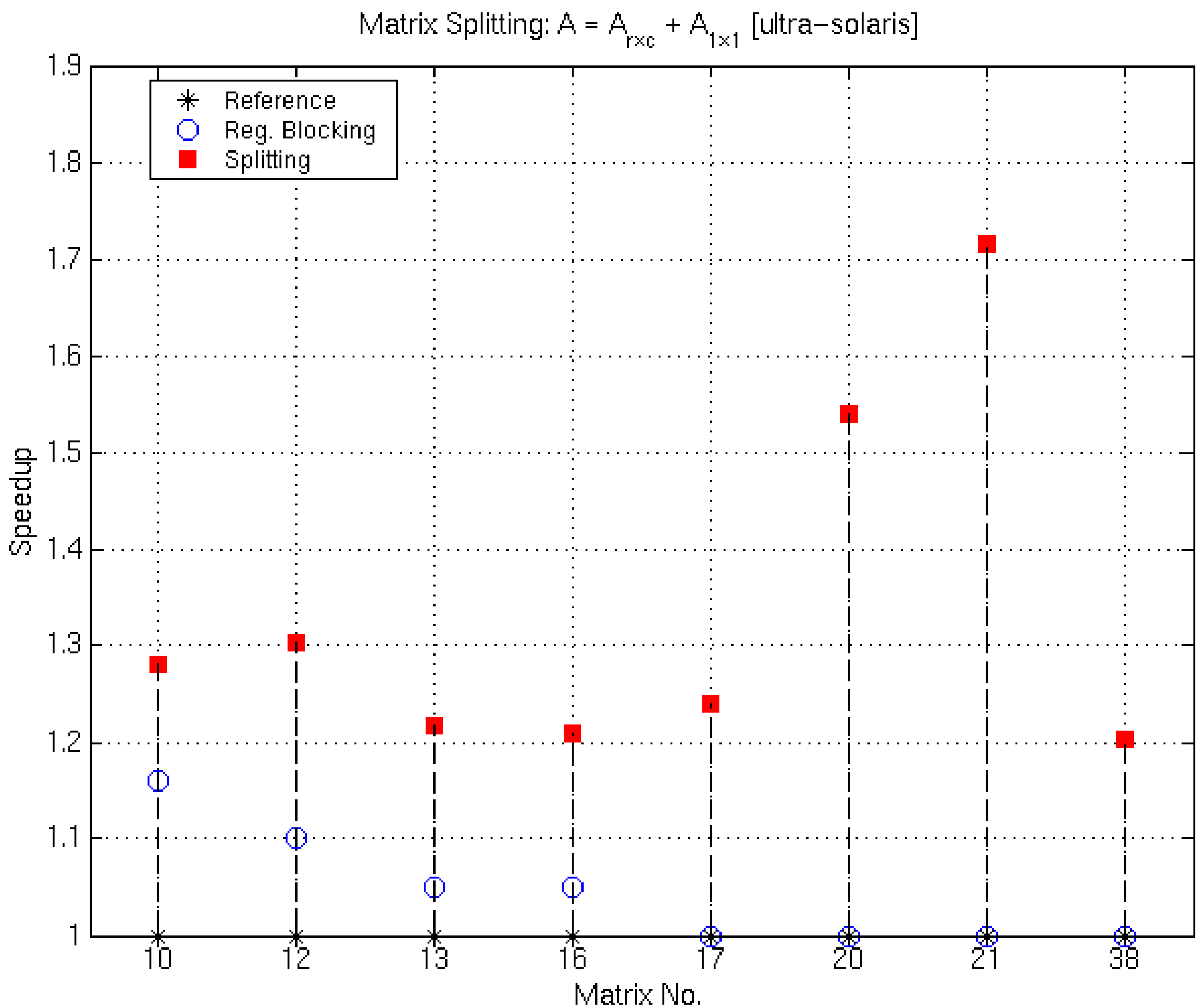
Exploiting Matrix Structure

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

Symmetric SpMV Performance: Pentium 4

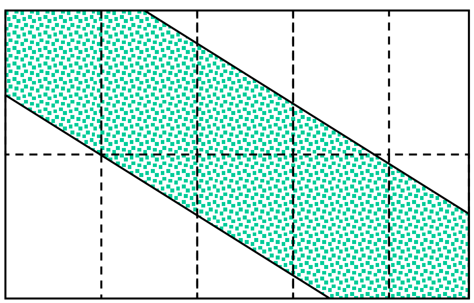


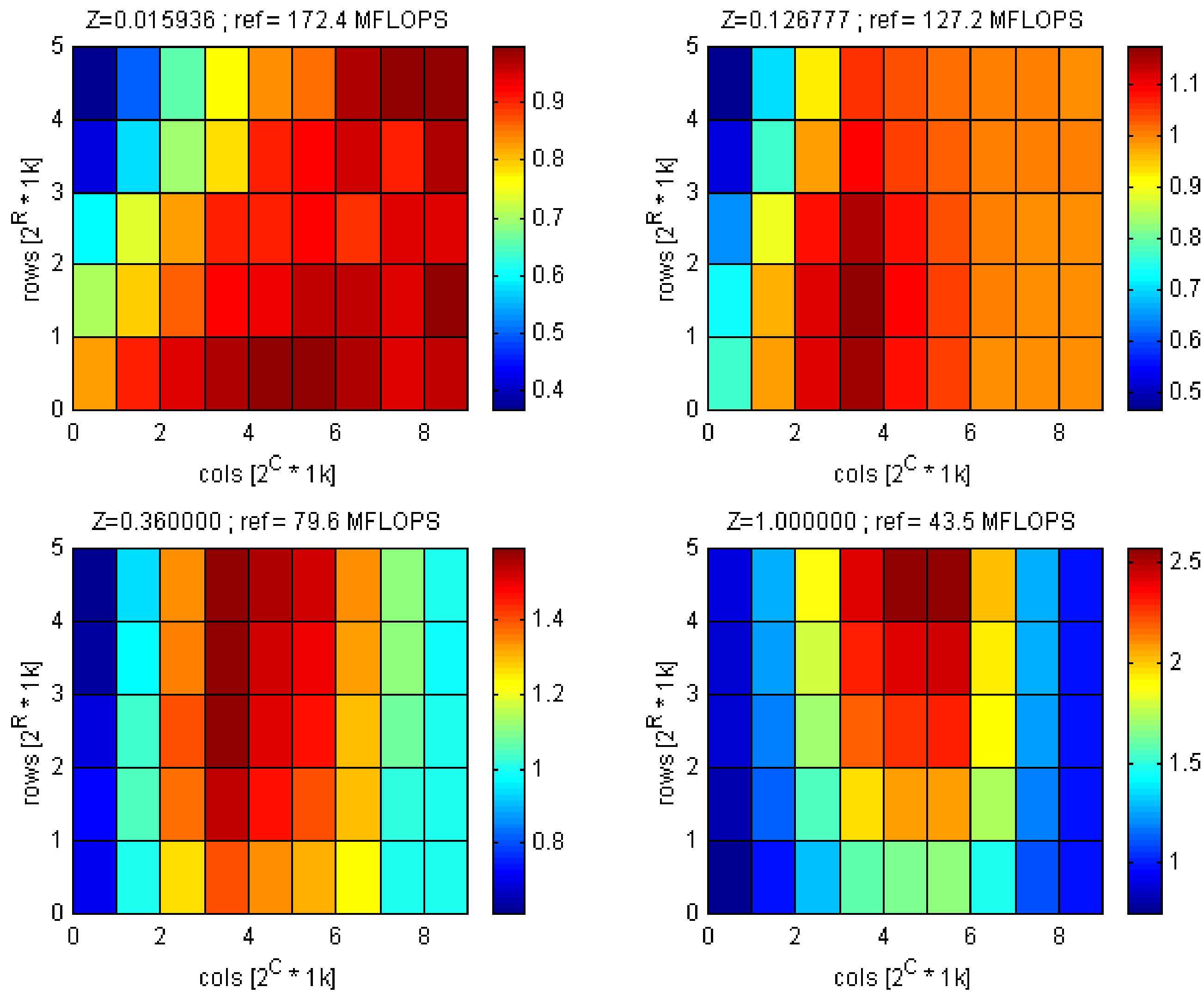
SpMV with Split Matrices: Ultra 2i



Cache Blocking on Random Matrices: Itanium

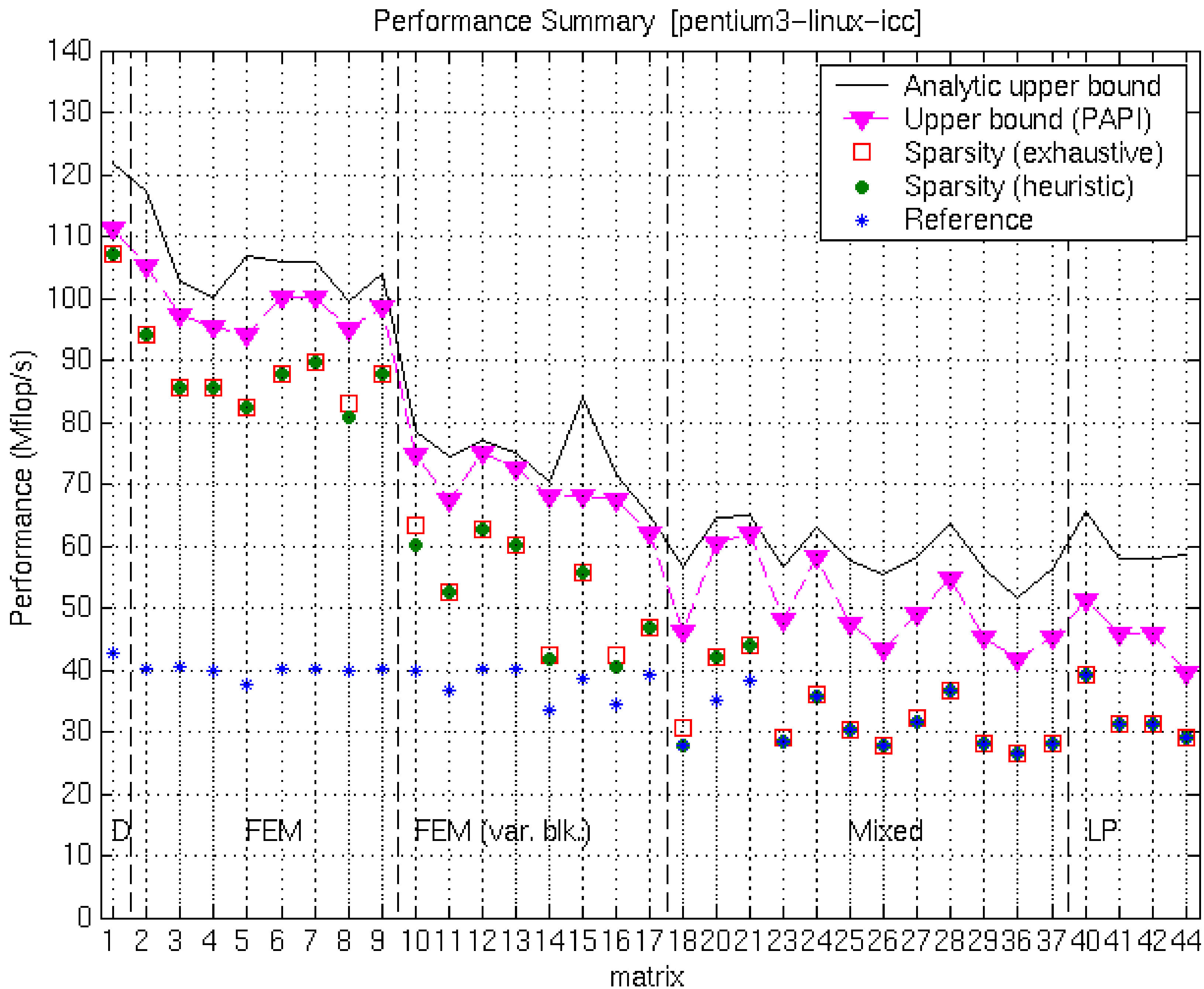
Speedup on four banded random matrices.



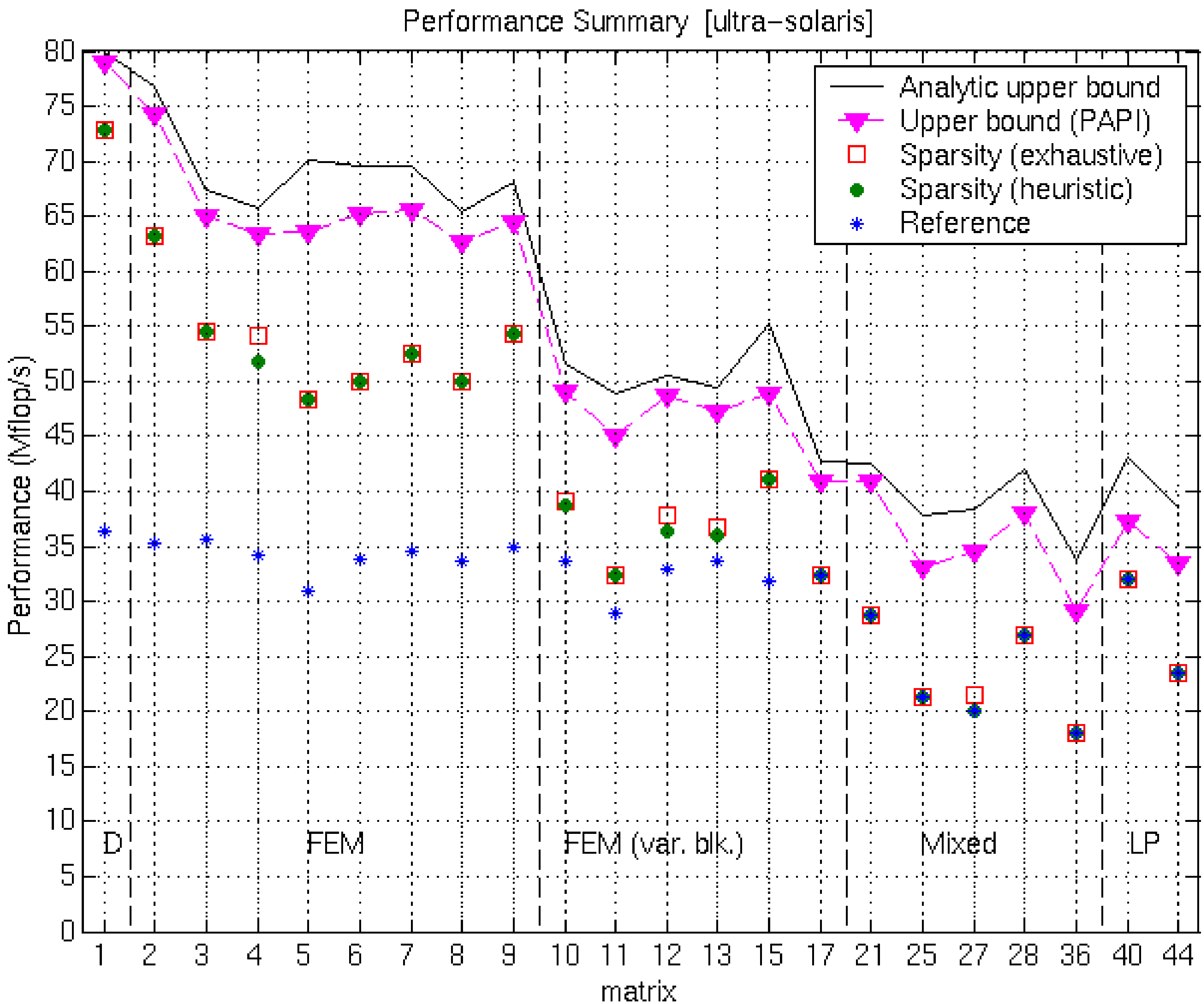


Sparse Kernels and Optimizations

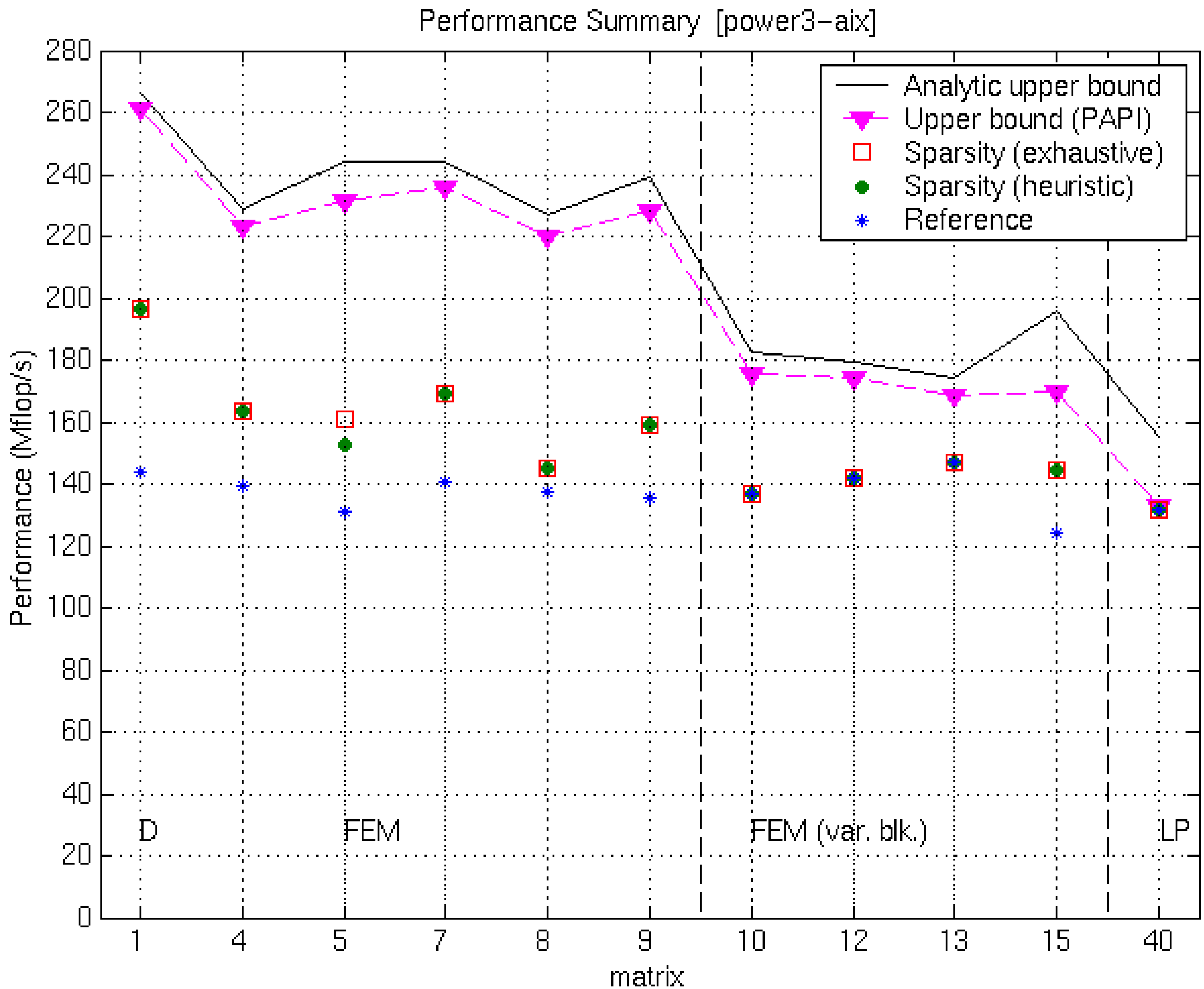
- Kernels
  - **Sparse matrix-vector multiply (SpMV): $y=A*x$**
  - Sparse triangular solve (SpTS): $x=T^{-1}*b$
  - $y=AA^T*x$, $y=A^TA*x$
  - Powers ($y=A^k*x$), sparse triple-product ($R*A*R^T$), …
- Optimization techniques (implementation space)
  - **Register blocking**
  - Cache blocking
  - **Multiple 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



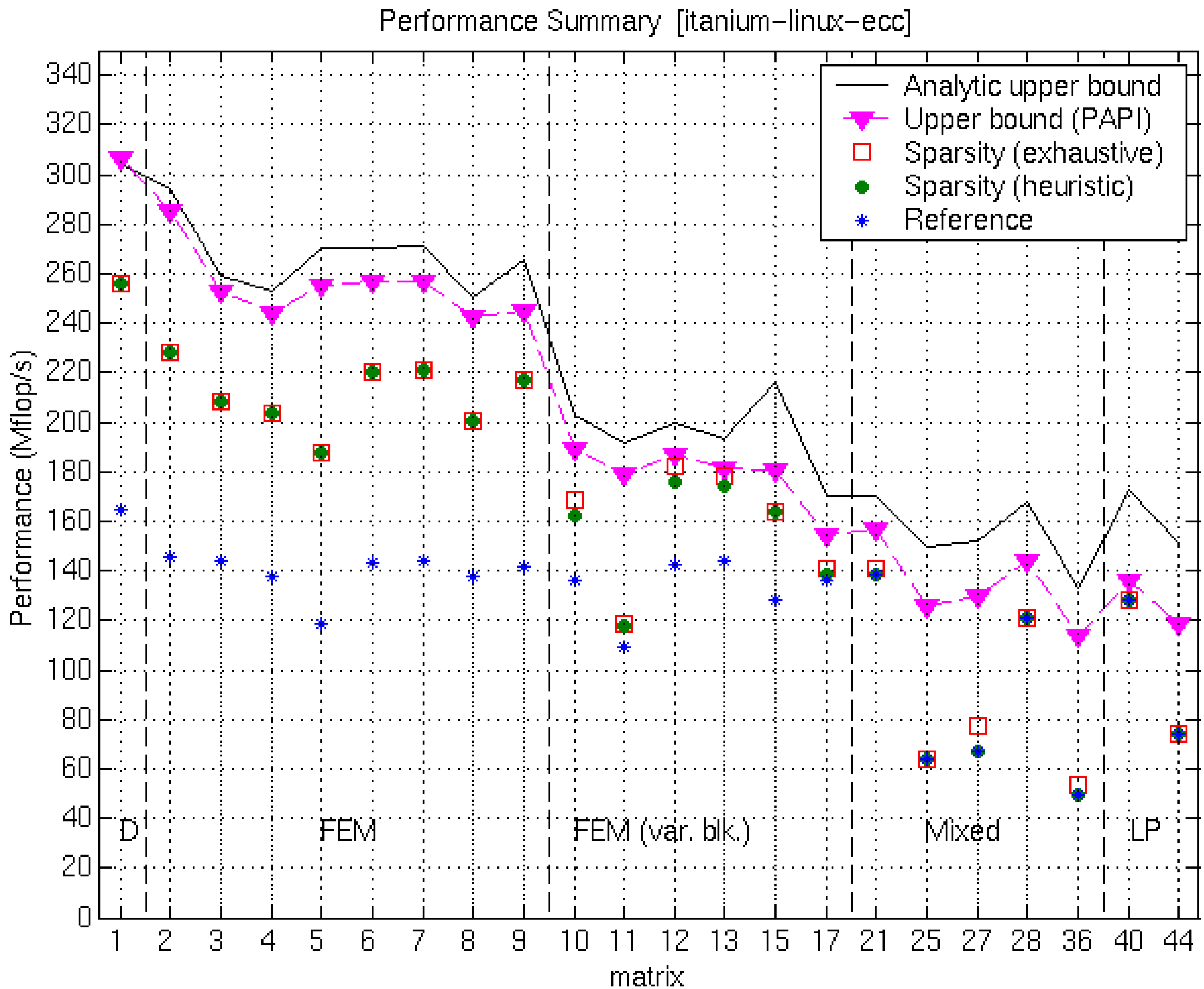
Performance Summary [pentium3−linux−icc]
Analytic upper bound
Upper bound (PAPI)
Sparsity (exhaustive)
Sparsity (heuristic)
Reference
Performance (Mflop/s)
matrix
D
FEM
FEM (var. blk.)
Mixed
LP
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
20
21
23
24
25
26
27
28
29
36
37
40
41
42
44



Performance Summary [ultra-solaris]
Performance (Mflop/s)
matrix
Analytic upper bound
Upper bound (PAPI)
Sparsity (exhaustive)
Sparsity (heuristic)
Reference
D
FEM
FEM (var. blk.)
Mixed
LP
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
1
2
3
4
5
6
7
8
9
10
11
12
13
15
17
21
25
27
28
36
40
44



Performance Summary [power3-aix]
Analytic upper bound
Upper bound (PAPI)
Sparsity (exhaustive)
Sparsity (heuristic)
Reference
Performance (Mflop/s)
matrix
D
FEM
FEM (var. blk.)
LP
0
20
40
60
80
100
120
140
160
180
200
220
240
260
280
1
4
5
7
8
9
10
12
13
15
40

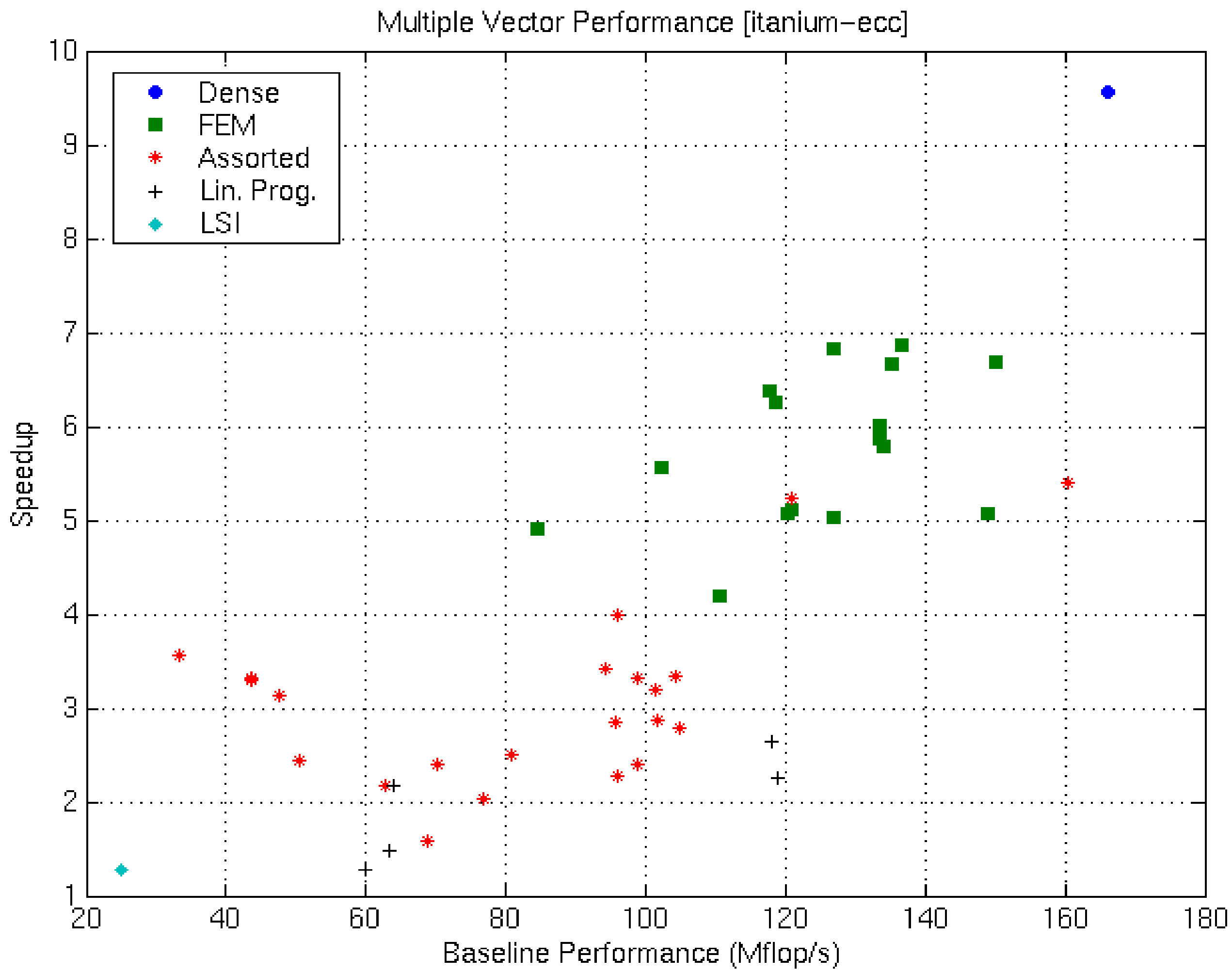


Performance Summary [itanium−linux−ecc]
Analytic upper bound
Upper bound (PAPI)
Sparsity (exhaustive)
Sparsity (heuristic)
Reference
Performance (Mflop/s)
0
20
40
60
80
100
120
140
160
180
200
220
240
260
280
300
320
340
D
FEM
FEM (var. blk.)
Mixed
LP
1
2
3
4
5
6
7
8
9
10
11
12
13
15
17
21
25
27
28
36
40
44
matrix

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

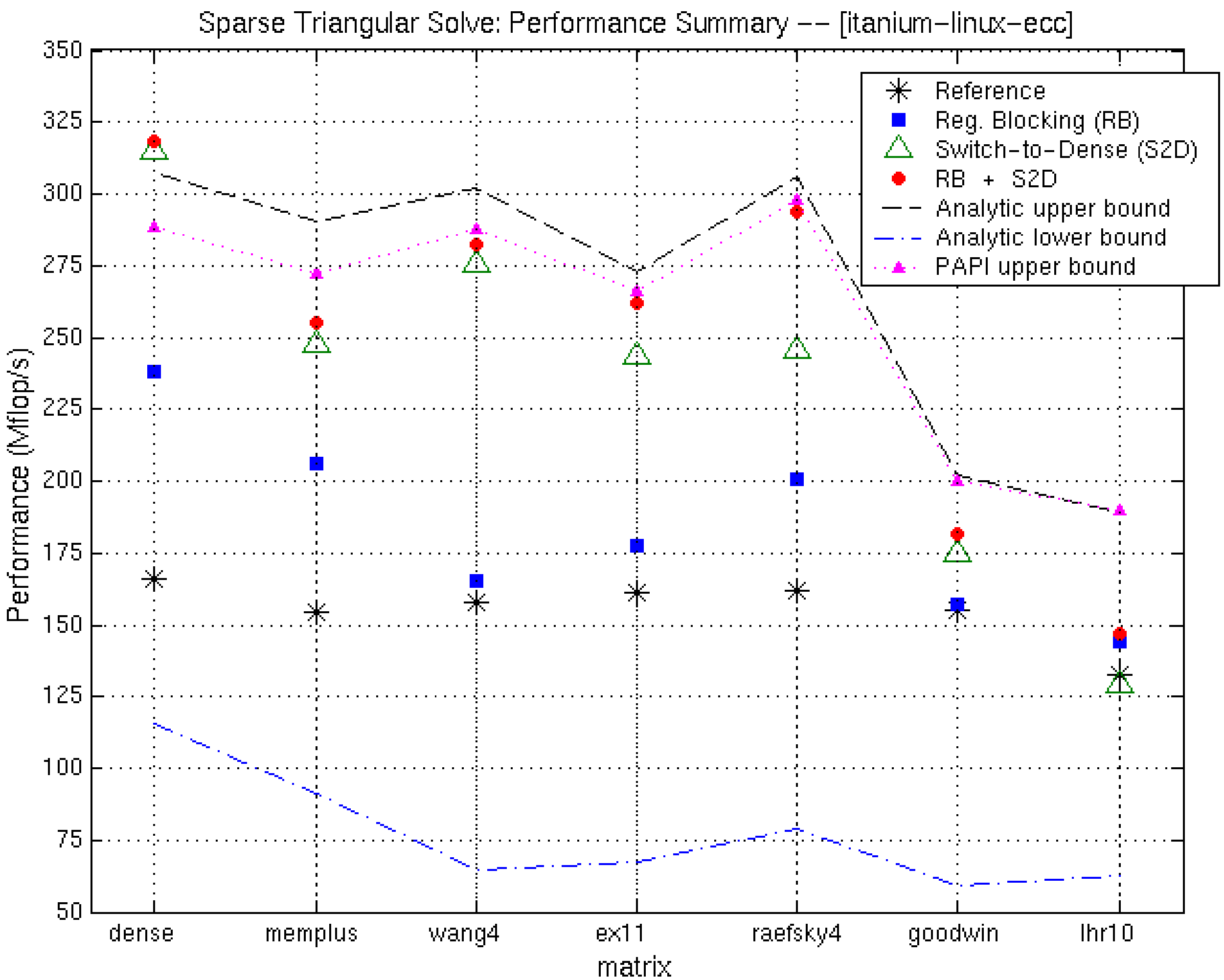
Multiple Vector Performance: Itanium



Sparse Kernels and Optimizations

- Kernels
  - Sparse matrix-vector multiply (SpMV): $y=A*x$
  - **Sparse triangular solve (SpTS): $x=T^{-1}*b$**
  - $y=AA^T*x$, $y=A^TA*x$
  - 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

SpTS Performance: Itanium



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

Sparse Kernels and Optimizations

- Kernels
  - Sparse matrix-vector multiply (SpMV): $y=A*x$
  - Sparse triangular solve (SpTS): $x=T^{-1}*b$
  - **$y=AA^T*x$, $y=A^TA*x$**
  - 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

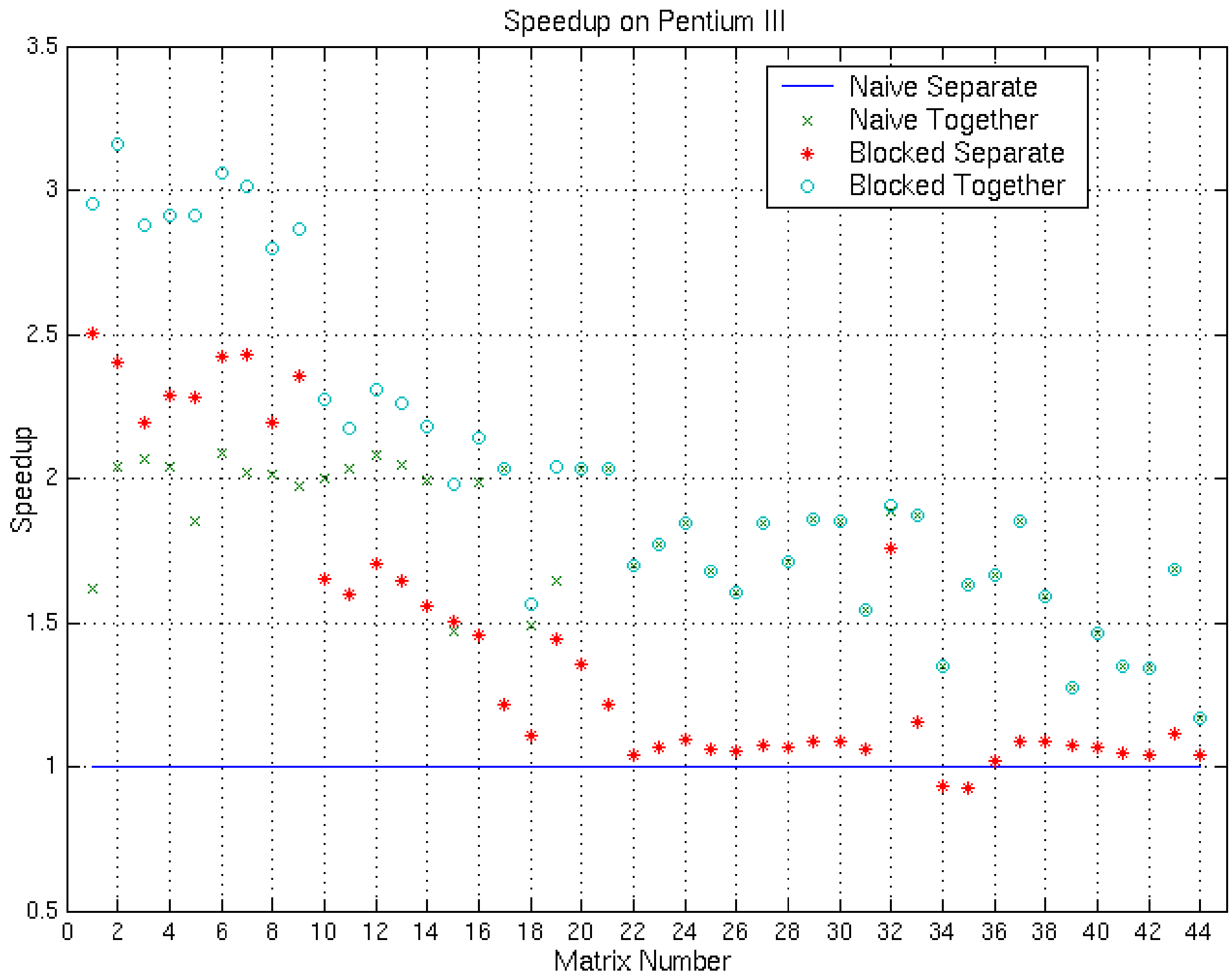
Optimizing AA^T*x

- 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:

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

- 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

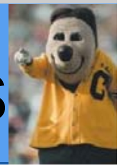
Related Work

- 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

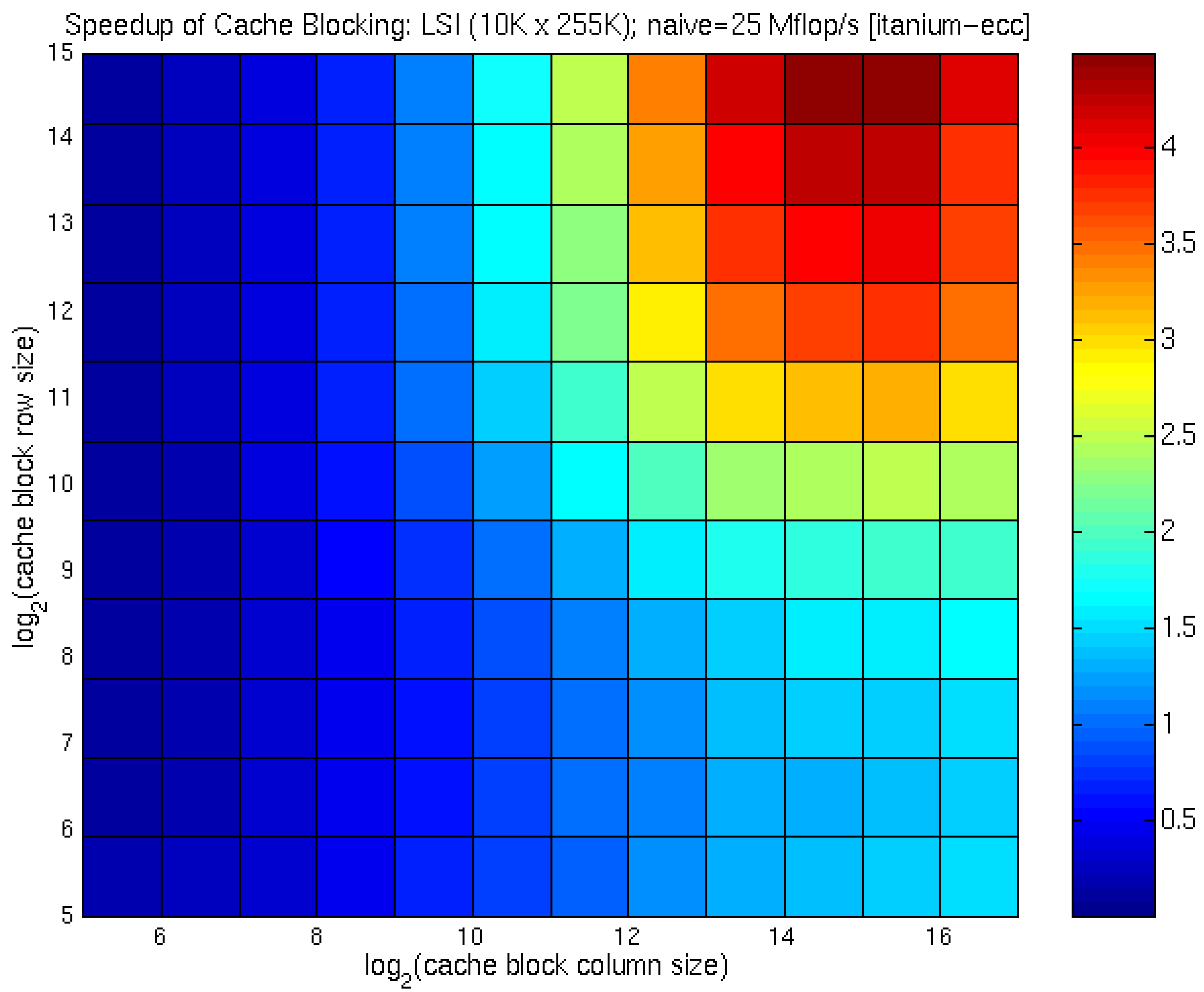
- Compiler analysis, models
  - CROPS [Carter, Ferrante, *et al*.]; Serial sparse tiling [Strout '01]
  - TUNE [Chatterjee, *et al*.]
  - Iterative compilation [O'Boyle, *et al*., '98]
  - Broadway compiler [Guyer & Lin, '99]
  - [Brewer '95], ADAPT [Voss '00]
- Sparse BLAS interfaces
  - BLAST Forum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

Context: Creating High-Performance Libraries

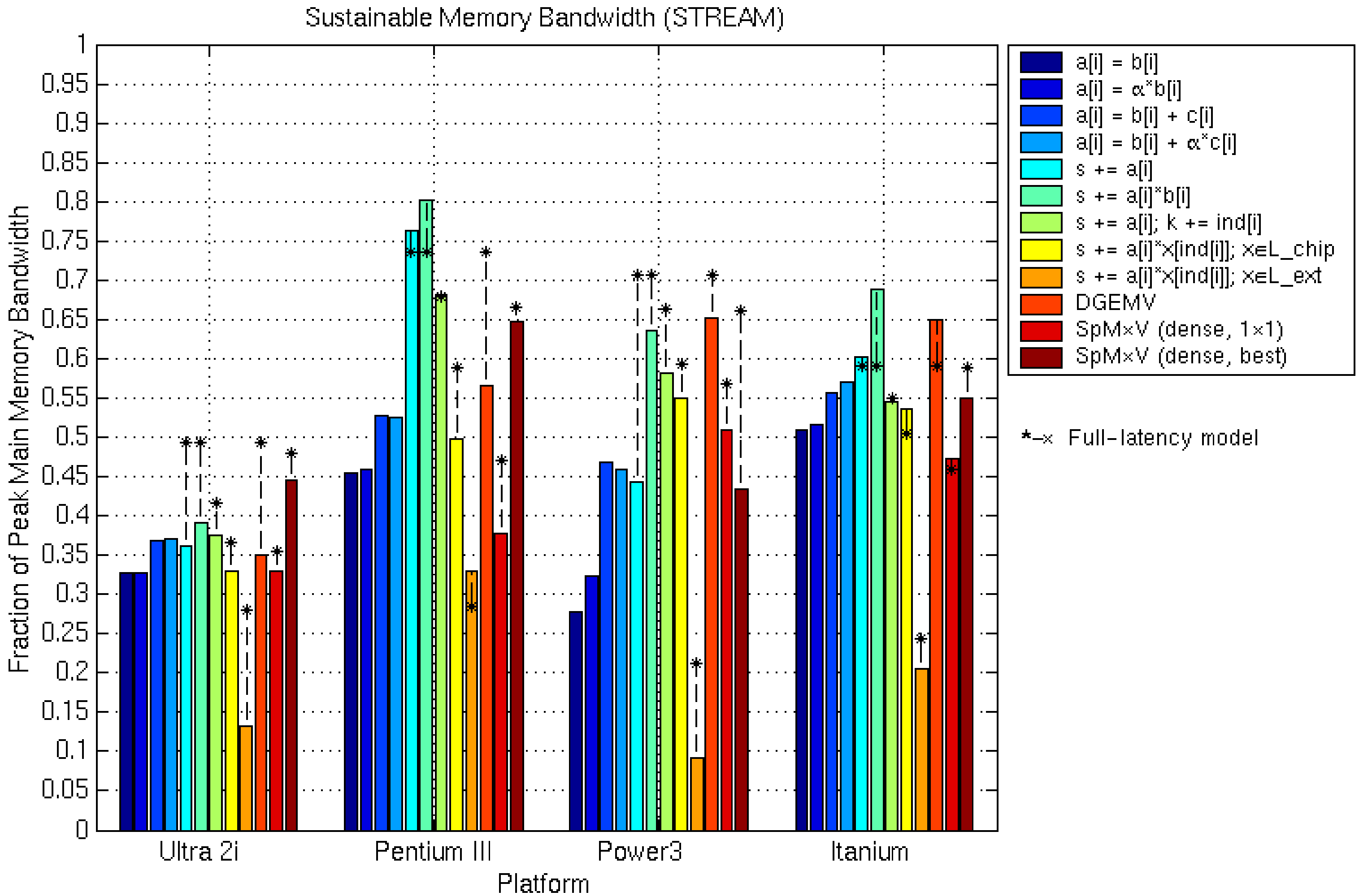


- Application performance dominated by a few *computational kernels*
- Today: Kernels hand-tuned by vendor or user
- Performance tuning challenges
  - Performance is a complicated function of kernel, architecture, compiler, and workload
  - Tedious and time-consuming
- Successful automated approaches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

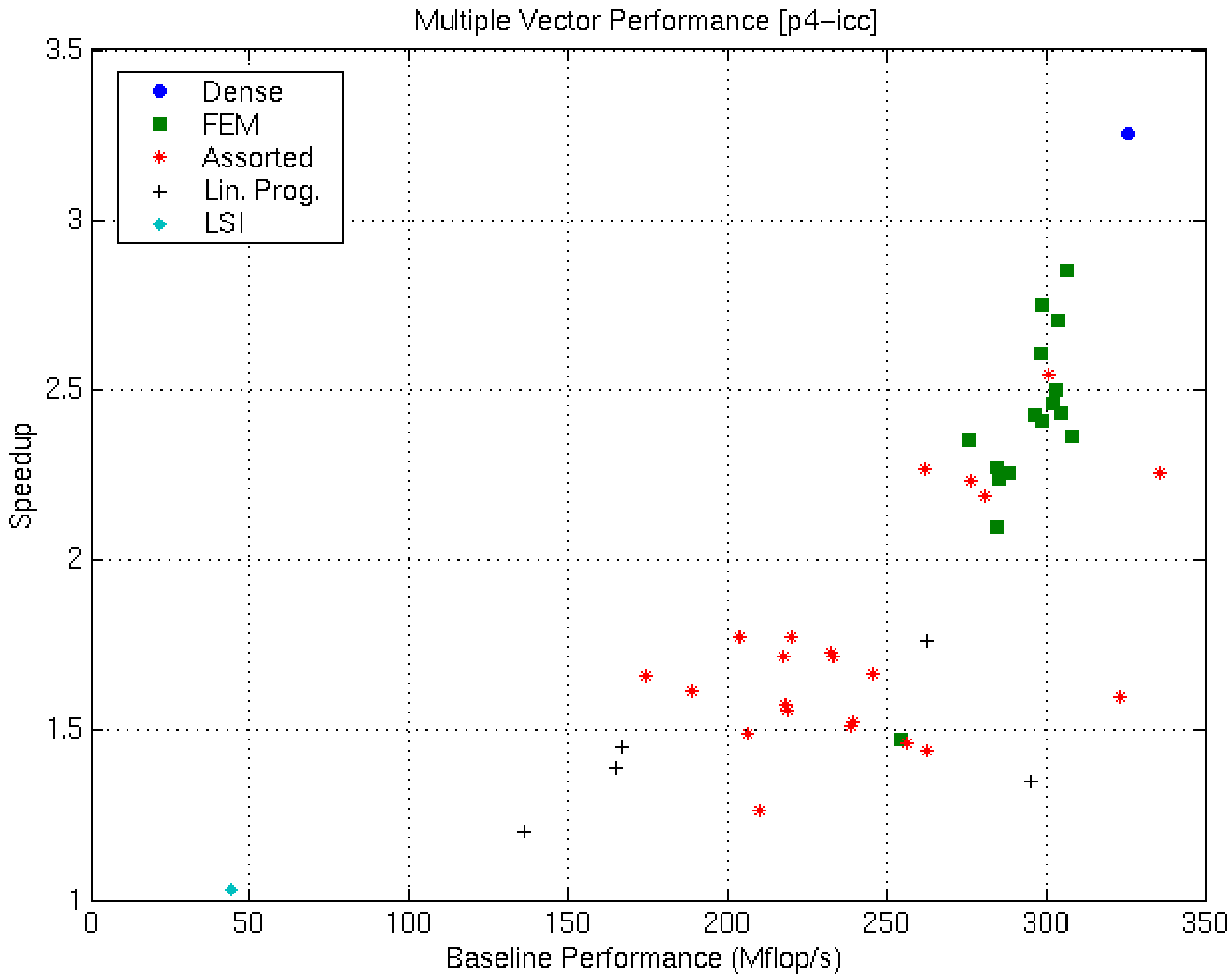
Cache Blocked SpMV on LSI Matrix: Itanium



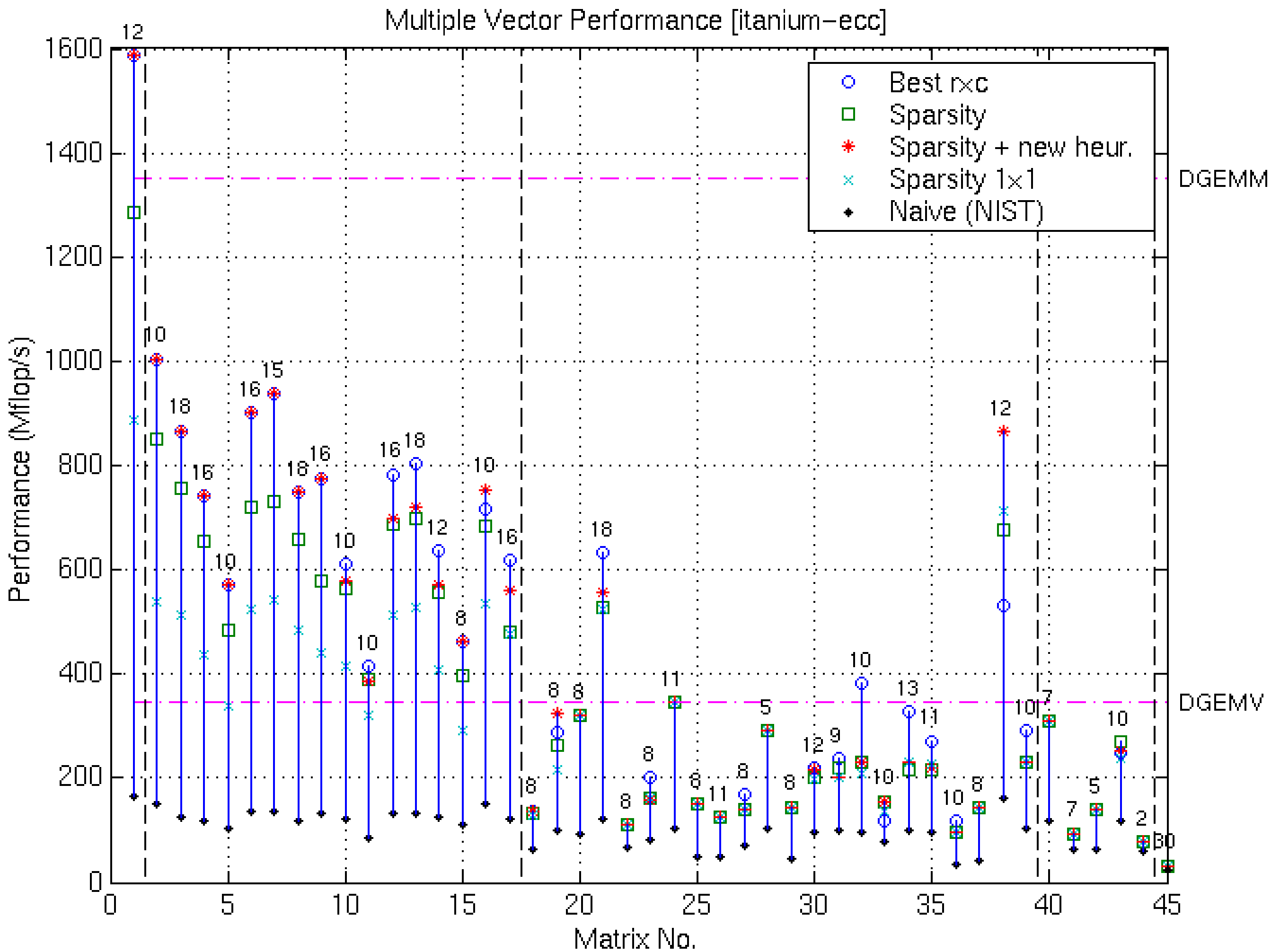
Sustainable Memory Bandwidth



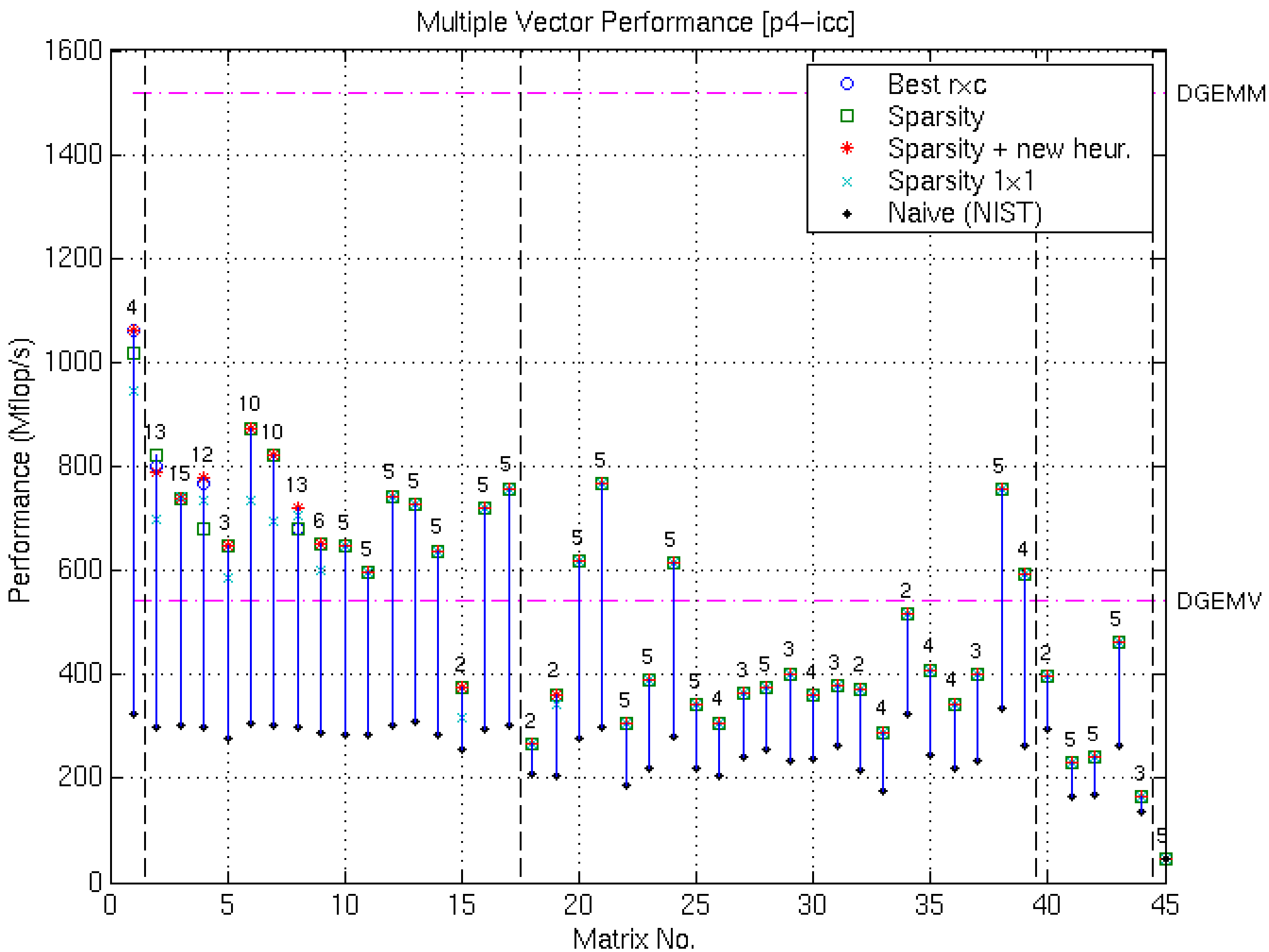
Multiple Vector Performance: Pentium 4



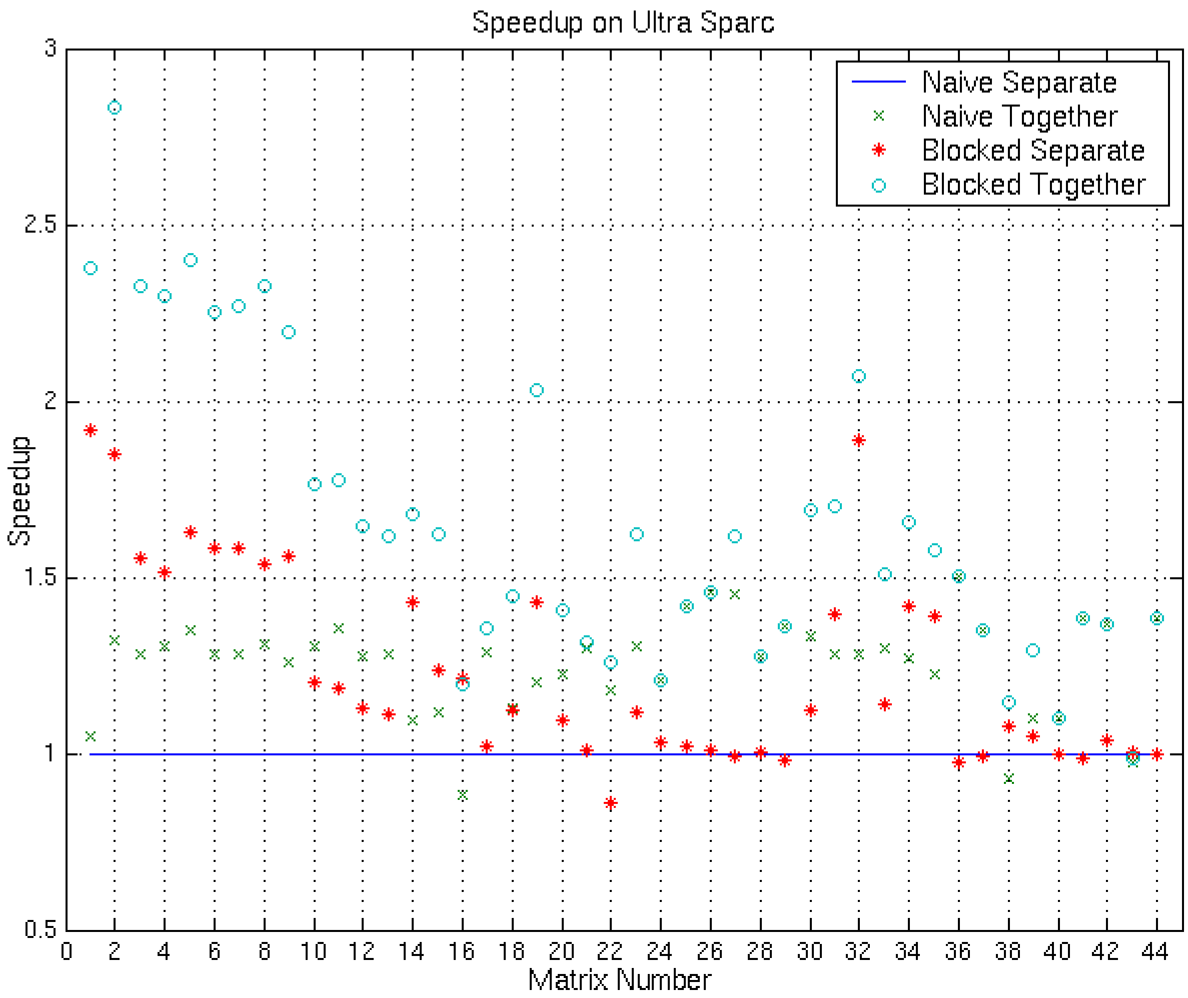
Multiple Vector Performance: Itanium



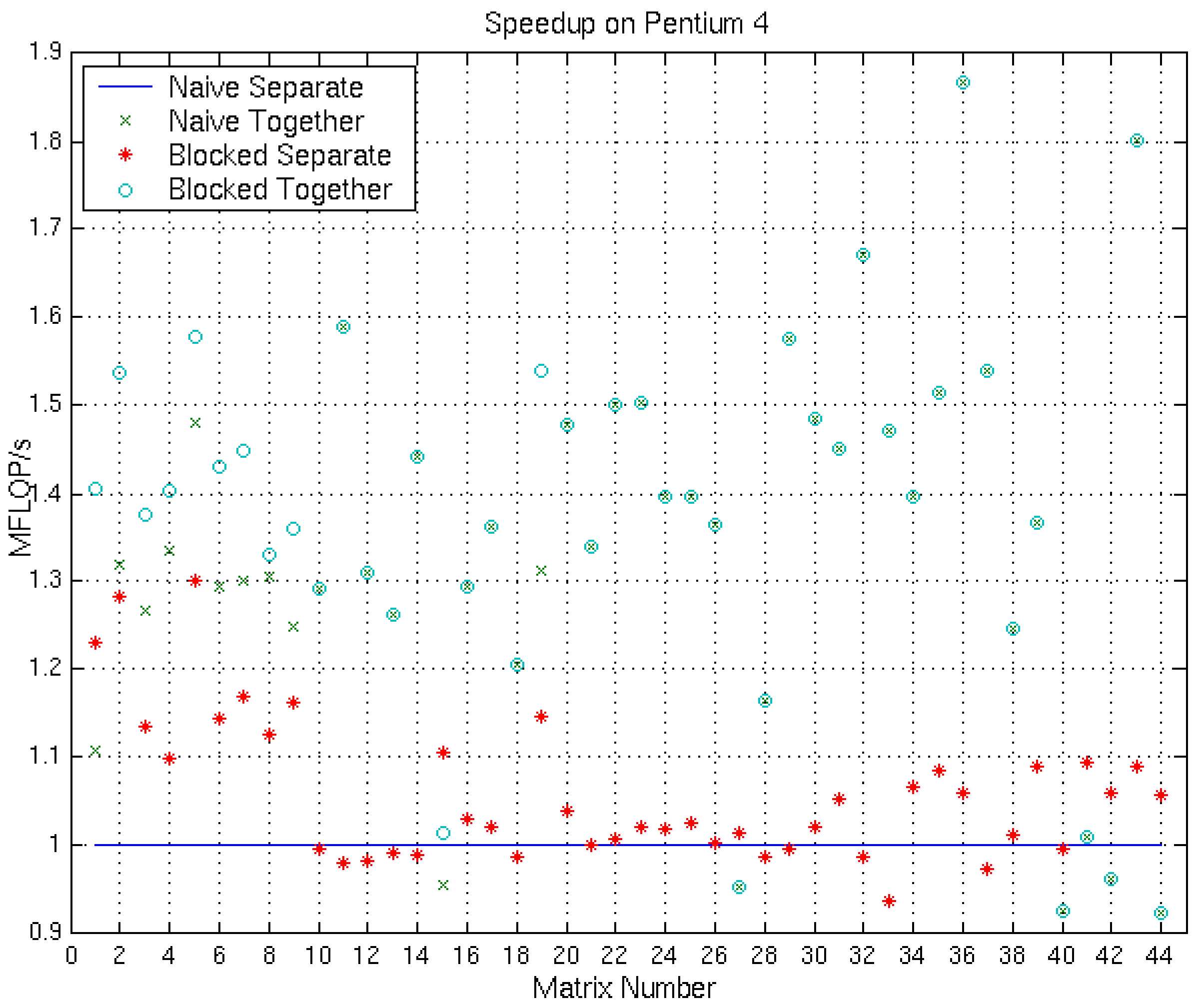
Multiple Vector Performance: Pentium 4



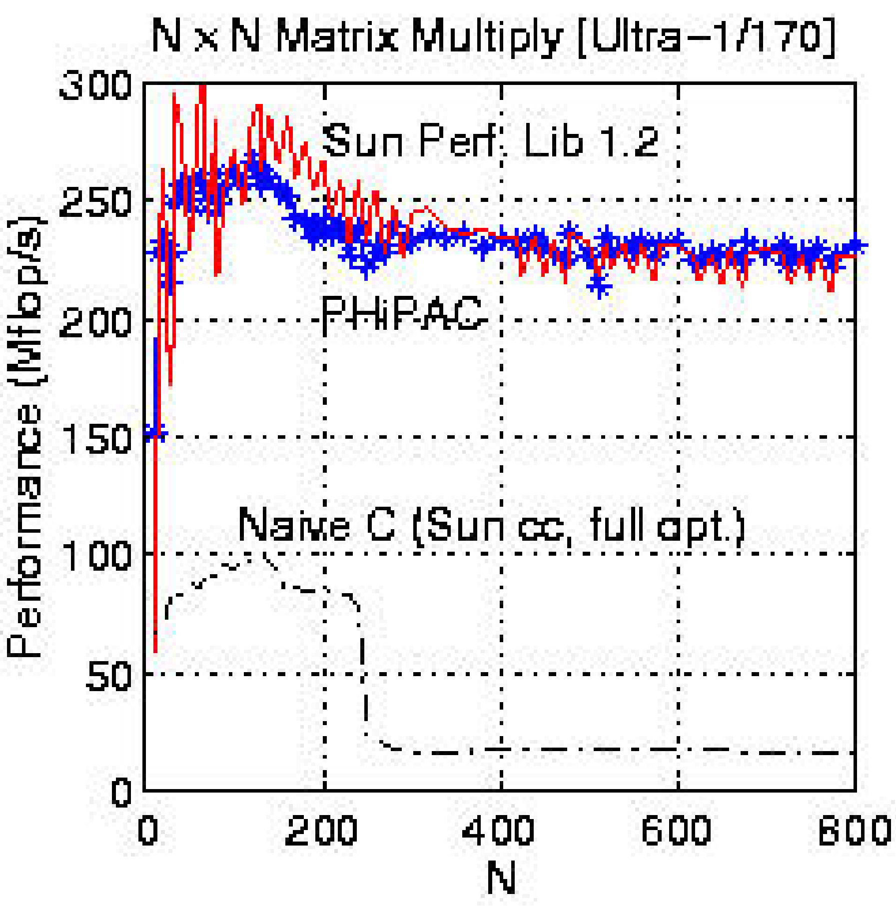
Optimized AA^T*x Performance: Ultra 2i

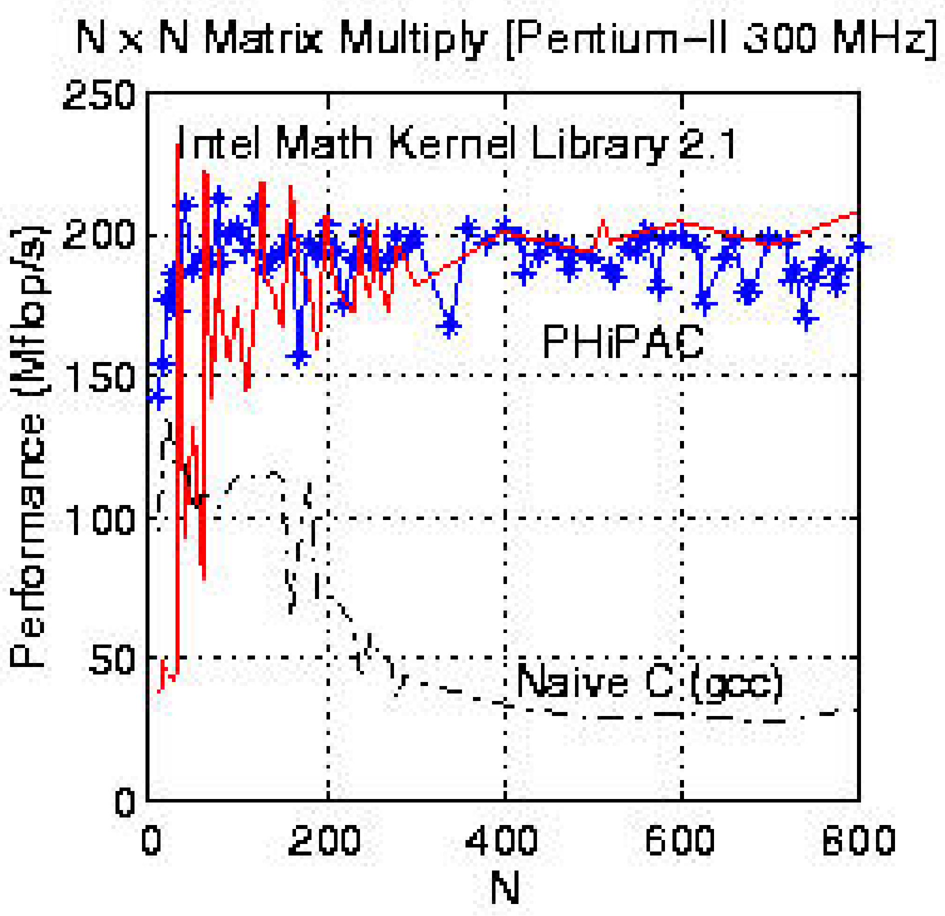


Optimized AA^T*x Performance: Pentium 4

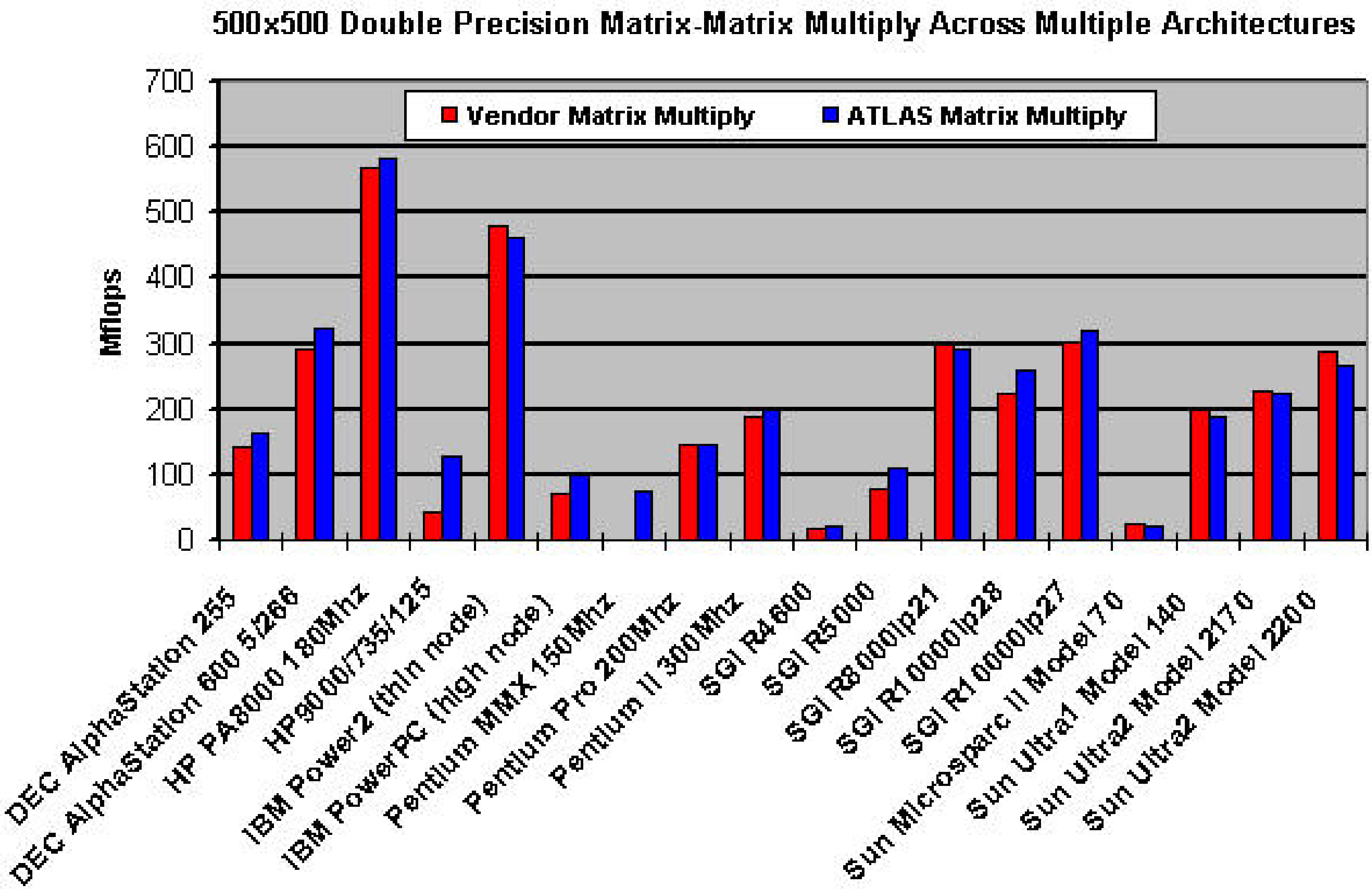


Tuning Pays Off—PHiPAC



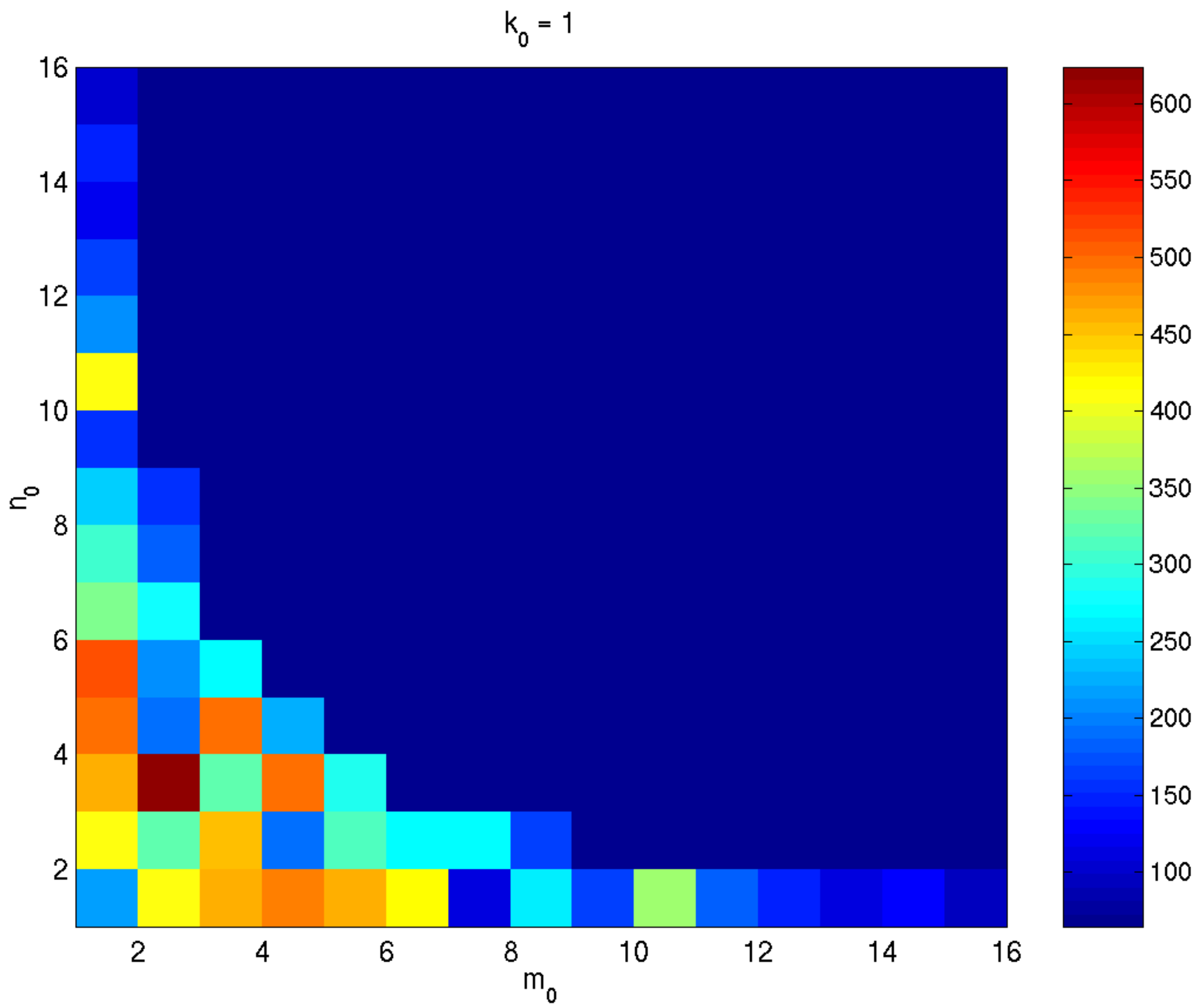


Tuning pays off – ATLAS



Extends applicability of PHIPAC; Incorporated in Matlab (with rest

Register Tile Sizes (Dense Matrix Multiply)

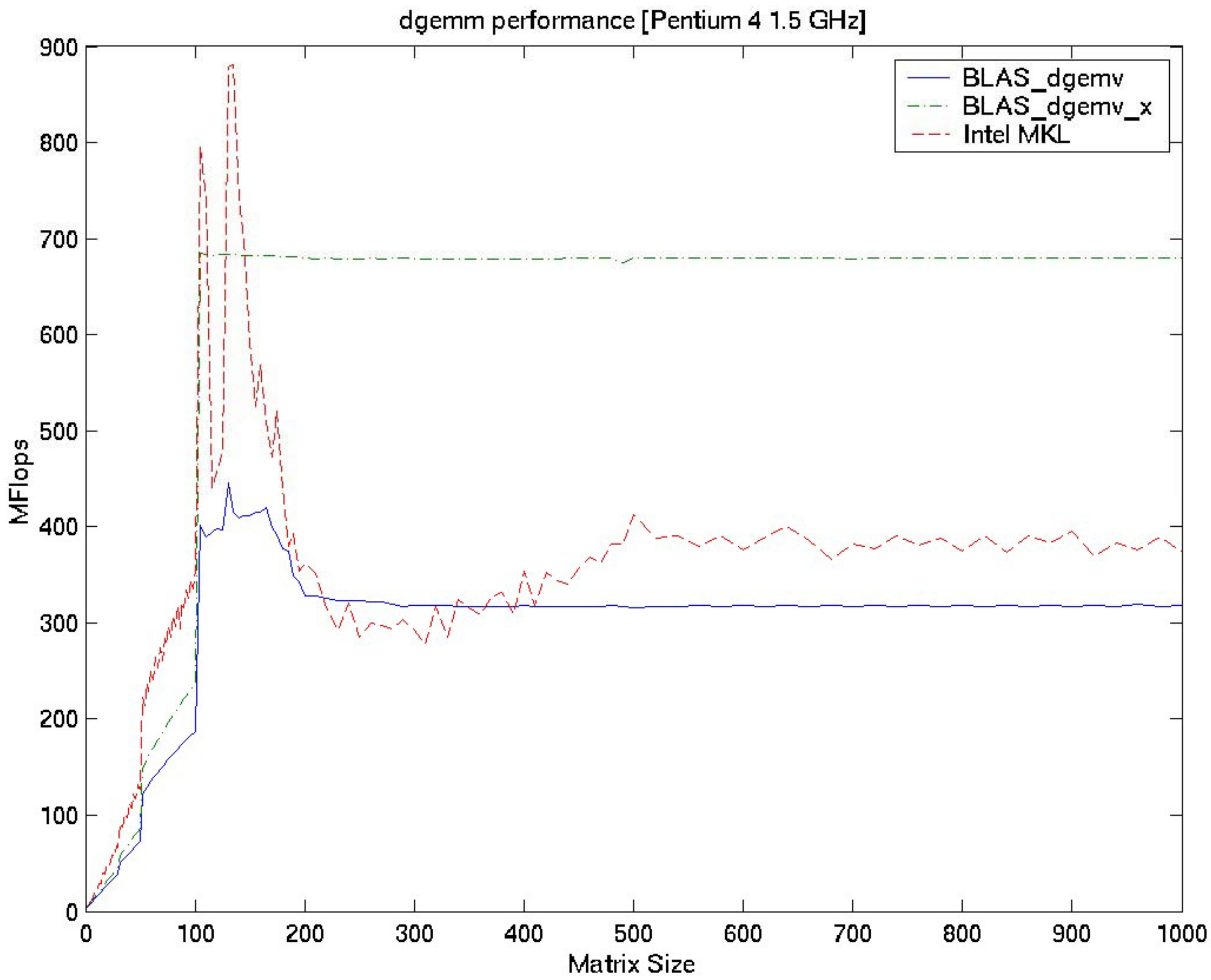


333 MHz Sun Ultra 2i

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) , … , s(n) have f-bit fractions, SUM has F>f bit fraction
  - Consider following algorithm for $S = \Sigma_{i=1,n}\ s(i)$
    - Sort so that $|s(1)| \geq |s(2)| \geq \cdots \geq |s(n)|$
    - 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 $\leq$ 1.5 ulps
    - If $n = 2^{F-f} + 2$, then error $\leq 2^{2f-F}$ ulps (can be >> 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 $n \leq 2^{16} + 1 = 65537$