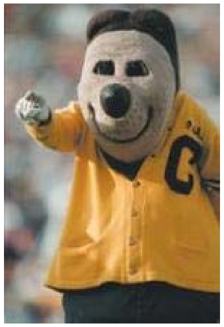
OSKI: Autotuned sparse matrix kernels



Richard Vuduc
LLNL / Georgia Tech [Work in this talk: Berkeley]

James Demmel, Kathy Yelick, …
UCB [BeBOP group]

CScADS Autotuning Workshop

What does the sparse case add to our conversation?

- Additional class of apps, *e.g.*, PageRank
- Data structure transformation – at run-time
  - Change is “semi-static”
  - How to manage run-time cost? Code gen?
  - Extra flops pay-off
  - Approach: Off-line benchmark + *cheap* run-time analysis & model

- Historical trends & snapshots “over time”
- Workloads and higher-level kernels
- Application adoption

(Personal) Historical Note

- Inspiration for OSKI has Bay Area roots
  - Profiling and feedback-directed compilation
    - Knuth (Stanford) ‘71: “An empirical study of FORTRAN programs”
    - Graham, Kessler, McKusick (UCB) ‘83: gprof
  - Memory hierarchy optimizations
    - Lam, Rothberg, Wolf (Stanford) ‘91
    - Pinar (LBL via UIUC), Heath 99 - for sparse mat-vec specifically
  - Automatic performance tuning
    - Bilmes, Asanovic, Chin, Demmel (UCB) ‘97: PHiPAC for dense matrix multiply
    - Im and Yelick (UCB) ‘99: SPARSITY for sparse mat-vec
- OSKI contributors
  - A. Gyulassy (UCD *via* UCB), S. Kamil (LBL/UCB), B. Lee (Harvard *via* UCB), HJ Moon (UCLA *via* UCB), R. Nishtala (UCB), …
  - A. Jain, S. Williams (UCB)

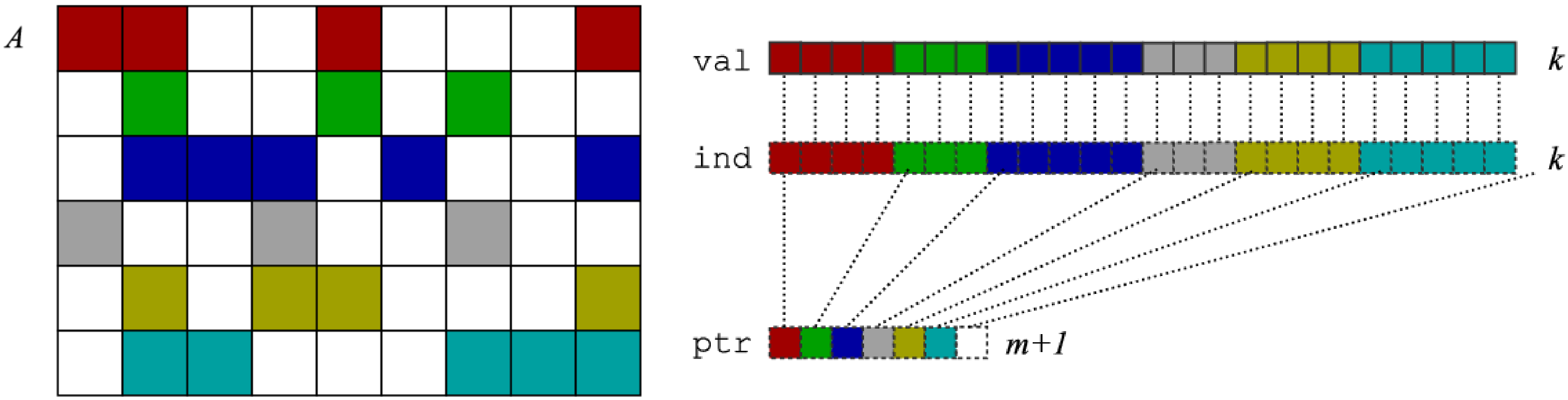
Why “autotune” sparse kernels?

- Sparse matrix-vector multiply < **10% peak, decreasing**
  - Indirect, irregular memory access
  - Low computational intensity *vs.* dense linear algebra
  - Depends on matrix (**run-time**) and machine
- Tuning is becoming more important
  - **2$\times$ speedup from tuning, will increase**
  - Manual tuning is difficult, getting harder
  - Tune target app, input, machine using **automated experiments**

OSKI: Optimized Sparse Kernel Interface

- Autotuned kernels for user's matrix & machine
  - BLAS-style interface: mat-vec (SpMV), tri. solve (TrSV), …
  - Hides complexity of single-core run-time tuning
  - Includes fast locality-aware kernels: $A^T A \cdot x$, $A^k \cdot x$, …
  - {32b, 64b}-int x {single, double} x {real, complex}
- Fast in practice
  - Standard SpMV < 10% peak, *vs.* up to **31%** with OSKI
  - Up to **4**× faster SpMV, **1.8**× triangular solve, **4x** $A^T A \cdot x$, …
- For "advanced" users & solver library writers
  - OSKI-PETSc; Trilinos (Heroux)
  - Adopted by ClearShape, Inc. for shipping product (2× speedup)

SpMV crash course:
Compressed Sparse Row (CSR) storage



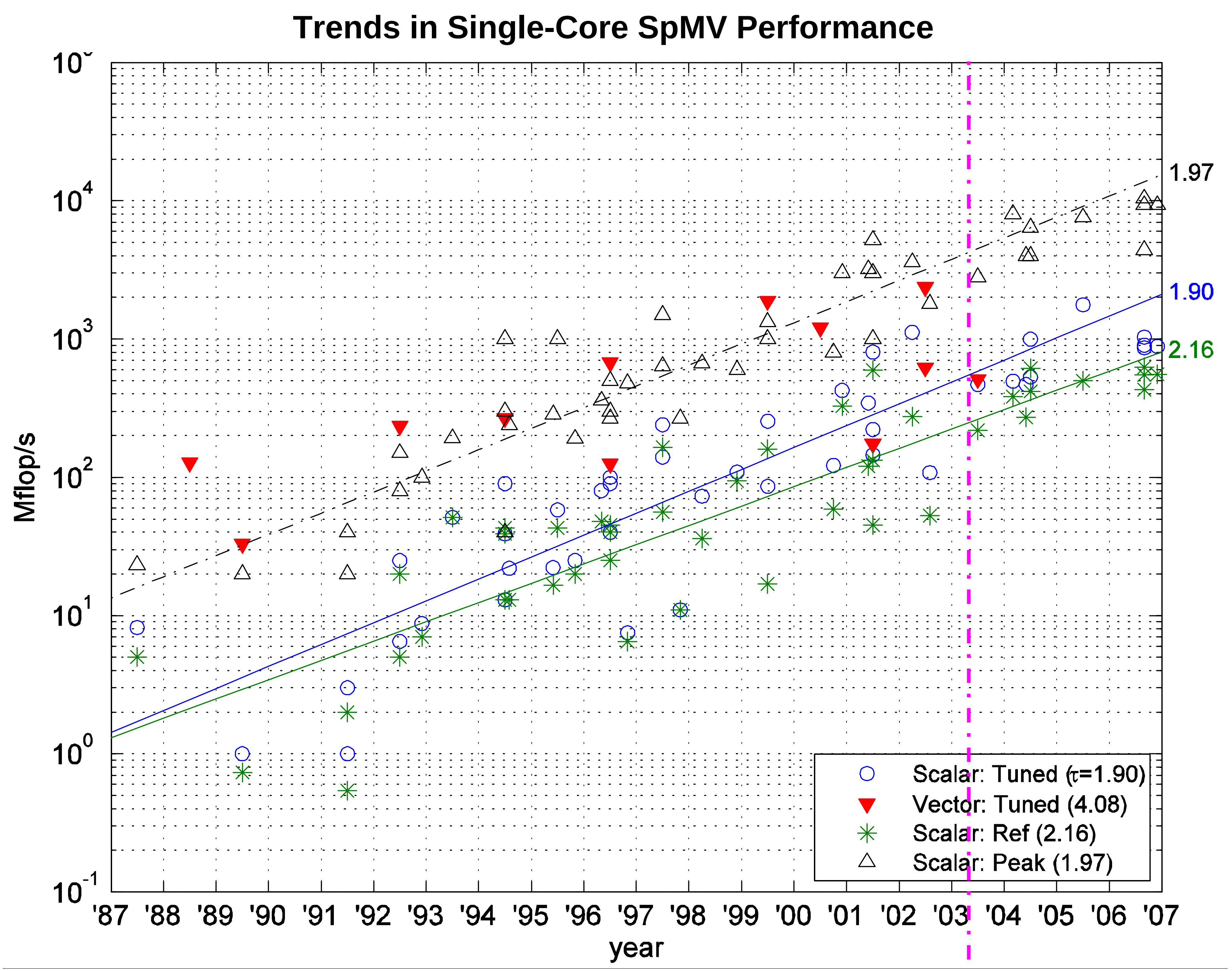
- Matrix-vector multiply: y = A*x
  - for all A(i, j): y(i) = y(i) + A(i, j) * x(j)

Irregular, indirect: x[ind[…]]
“Regularize?”

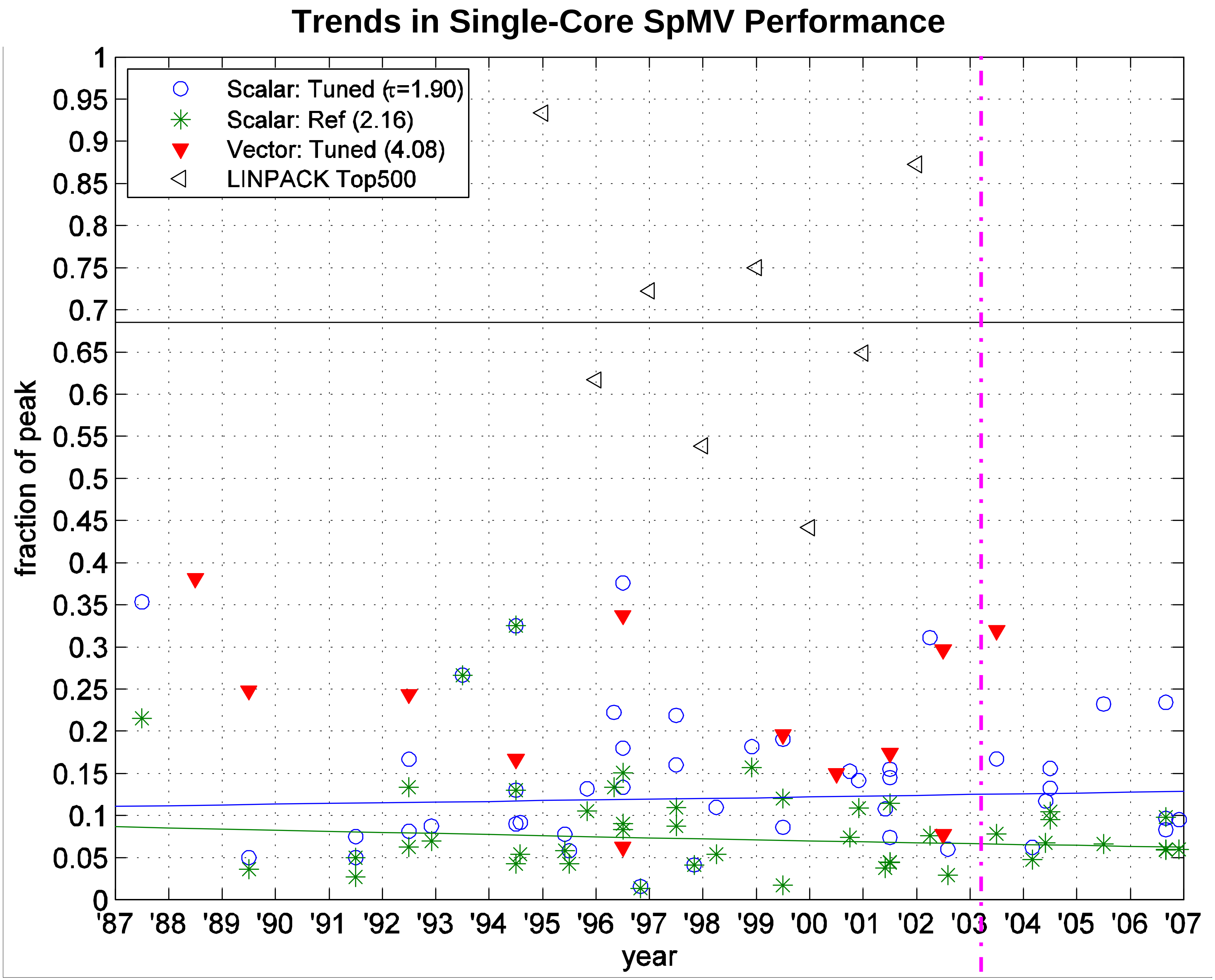
Dominant cost: Compress?

Trends: My predictions from 2003

- Need for "autotuning" will increase over time
  - So kindly approve my dissertation topic
- Example: SpMV, 1987 to present
  - Untuned: 10% of peak or less, **decreasing**
  - Tuned: 2× speedup, **increasing** over time
  - **Tuning is getting harder** (qualitative)
    - More complex machines & workloads
    - Parallelism

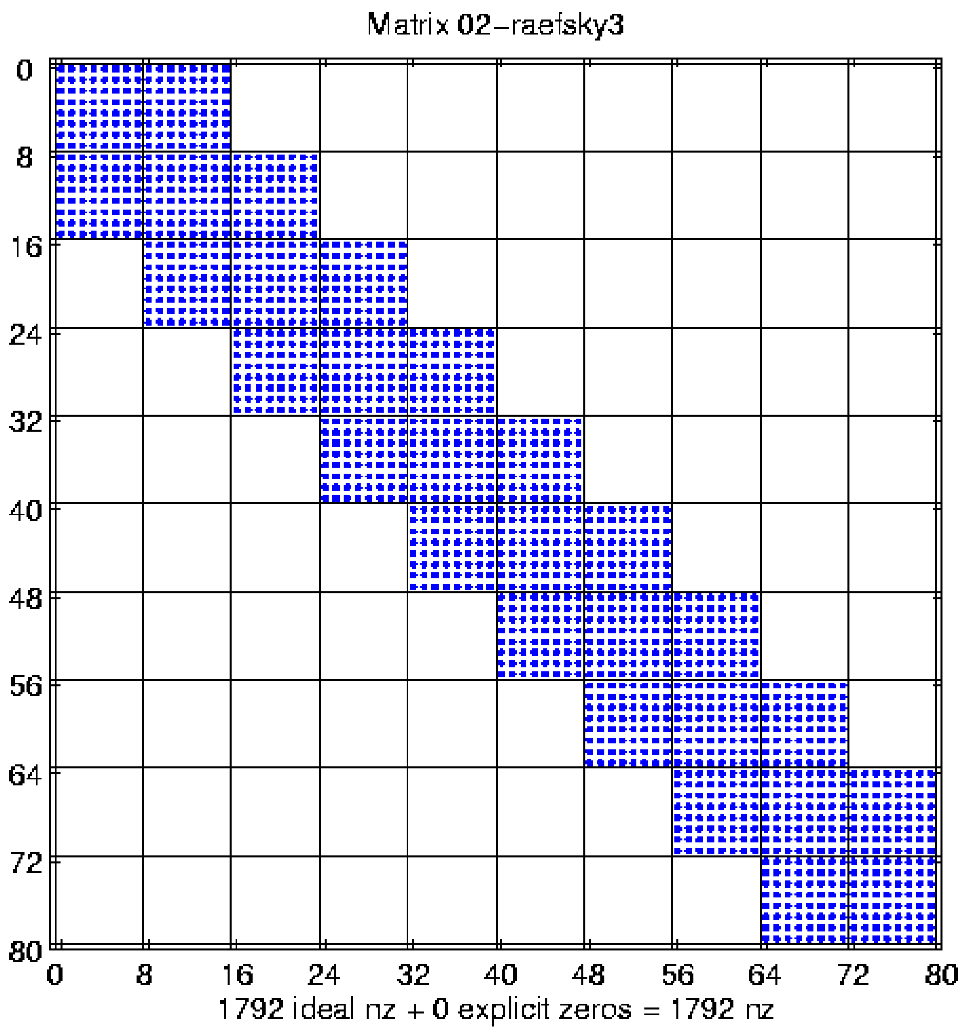


Trends in Single-Core SpMV Performance
Mflop/s
year
1.97
1.90
2.16
Scalar: Tuned (τ=1.90)
Vector: Tuned (4.08)
Scalar: Ref (2.16)
Scalar: Peak (1.97)
'87 '88 '89 '90 '91 '92 '93 '94 '95 '96 '97 '98 '99 '00 '01 '02 '03 '04 '05 '06 '07



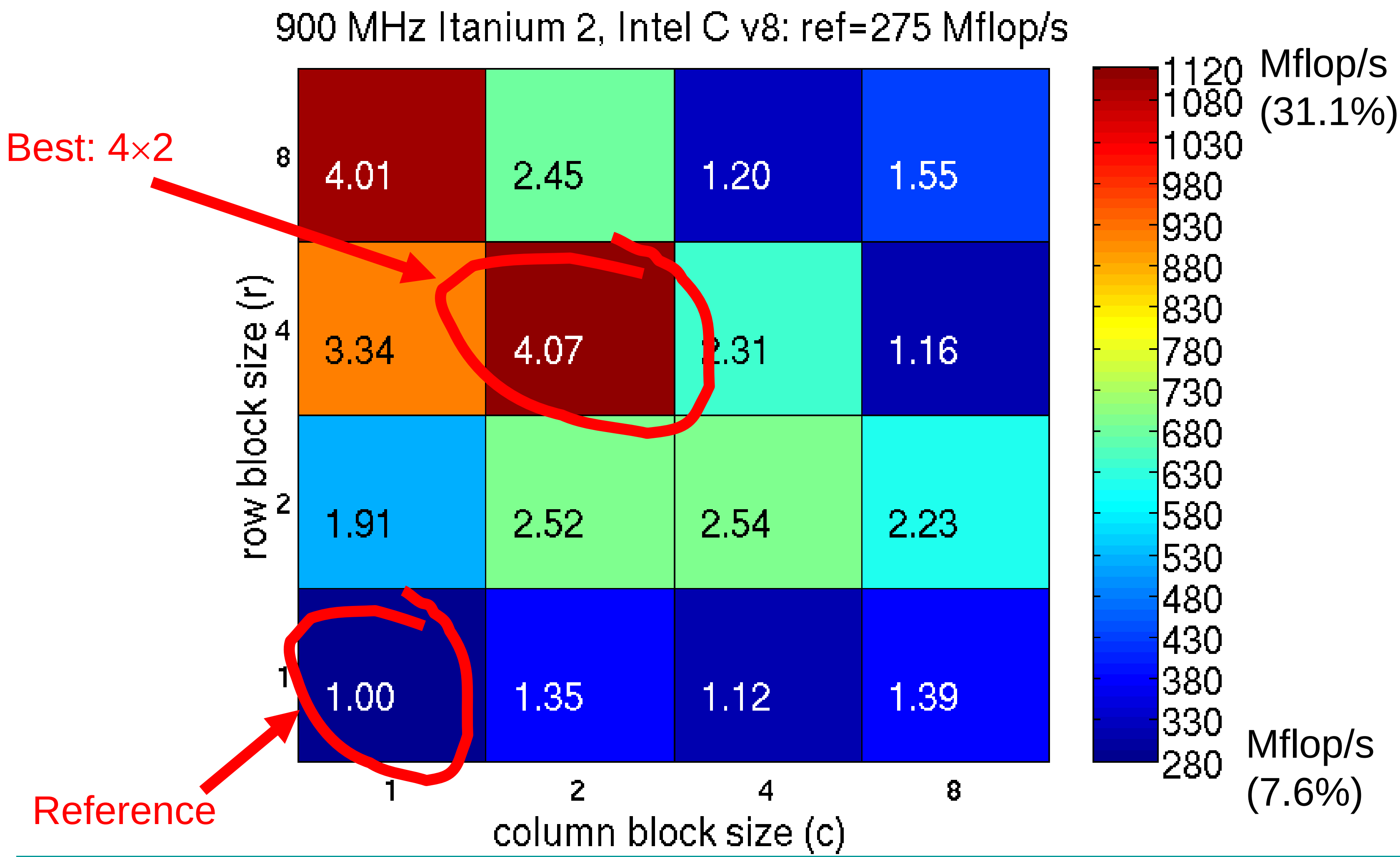
Trends in Single-Core SpMV Performance
Scalar: Tuned (τ=1.90)
Scalar: Ref (2.16)
Vector: Tuned (4.08)
LINPACK Top500
fraction of peak
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
'87 '88 '89 '90 '91 '92 '93 '94 '95 '96 '97 '98 '99 '00 '01 '02 '03 '04 '05 '06 '07
year

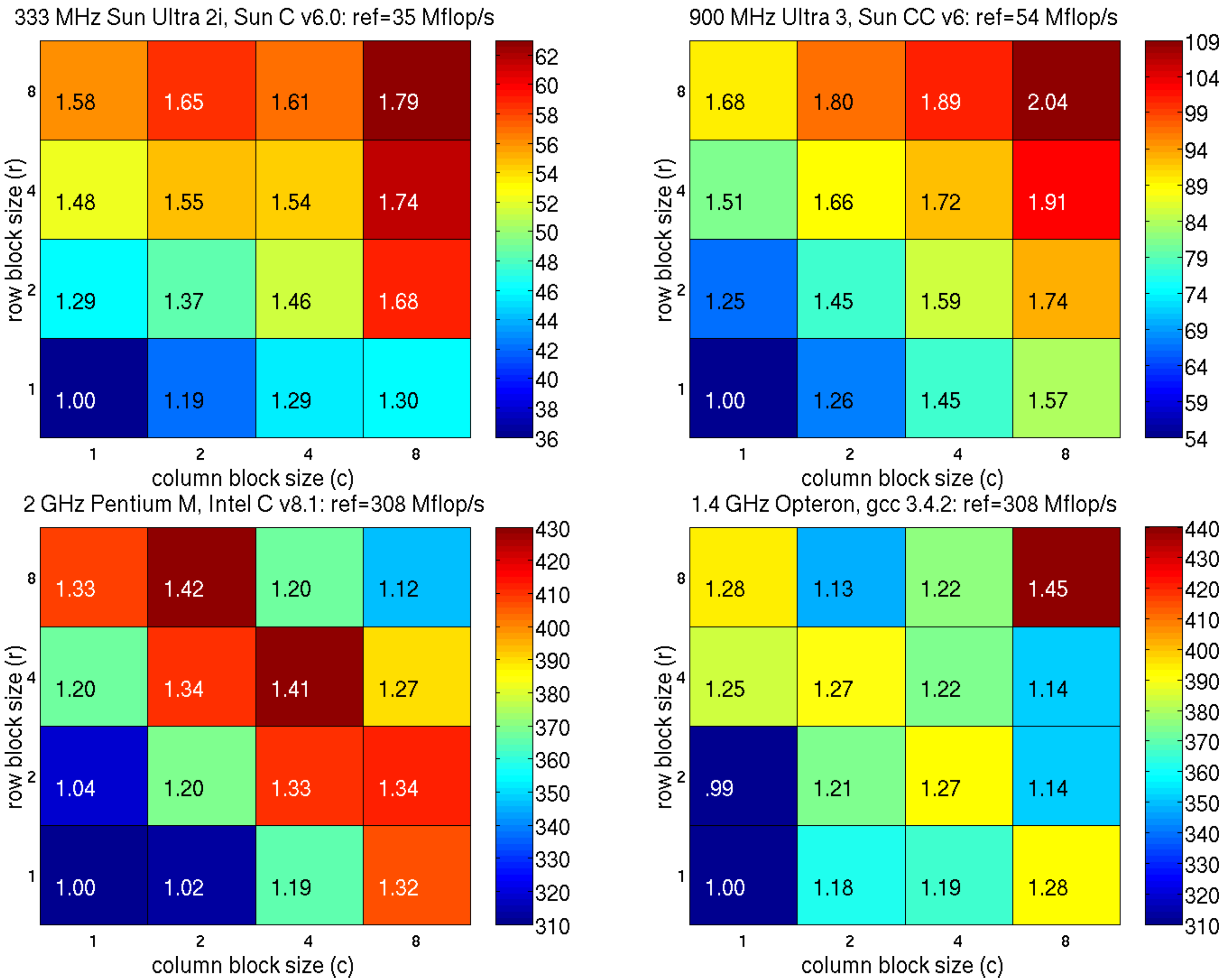
Experiment: How hard is SpMV tuning?



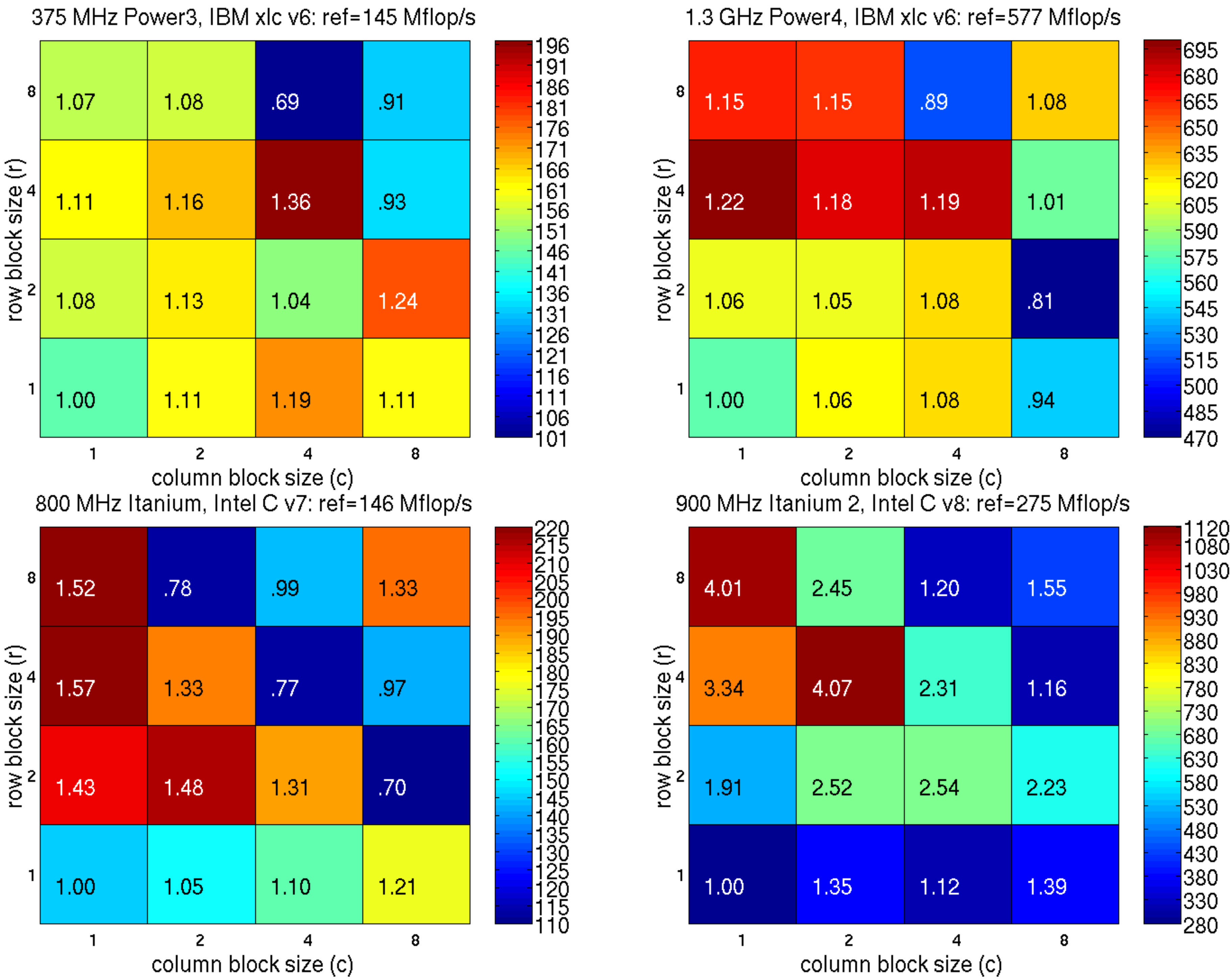
- Exploit 8×8 blocks
  - Store blocks & unroll
  - Compresses data
  - Regularizes accesses
- As r×c ↑, speed ↑

Speedups on Itanium 2: The need for search



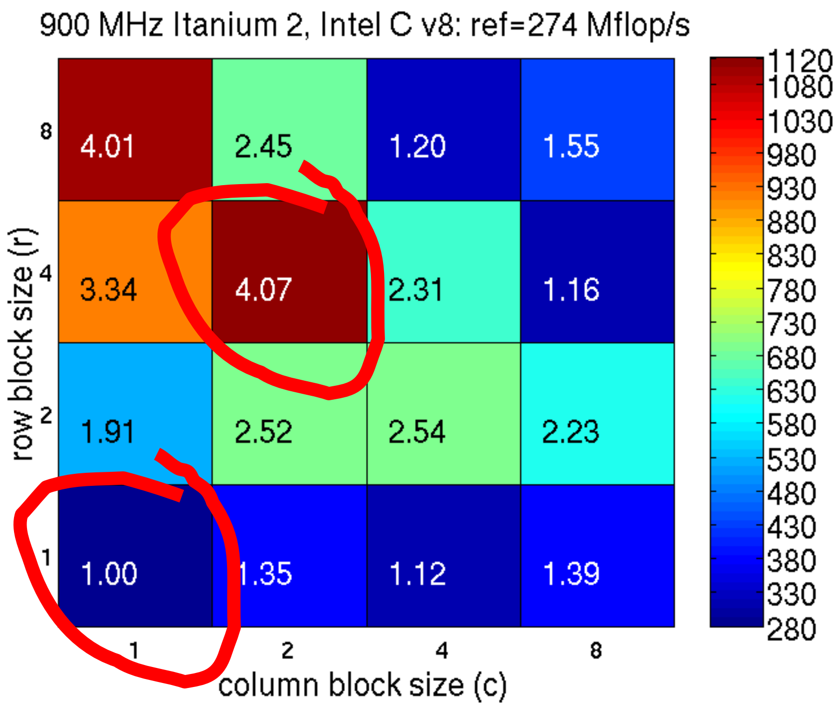


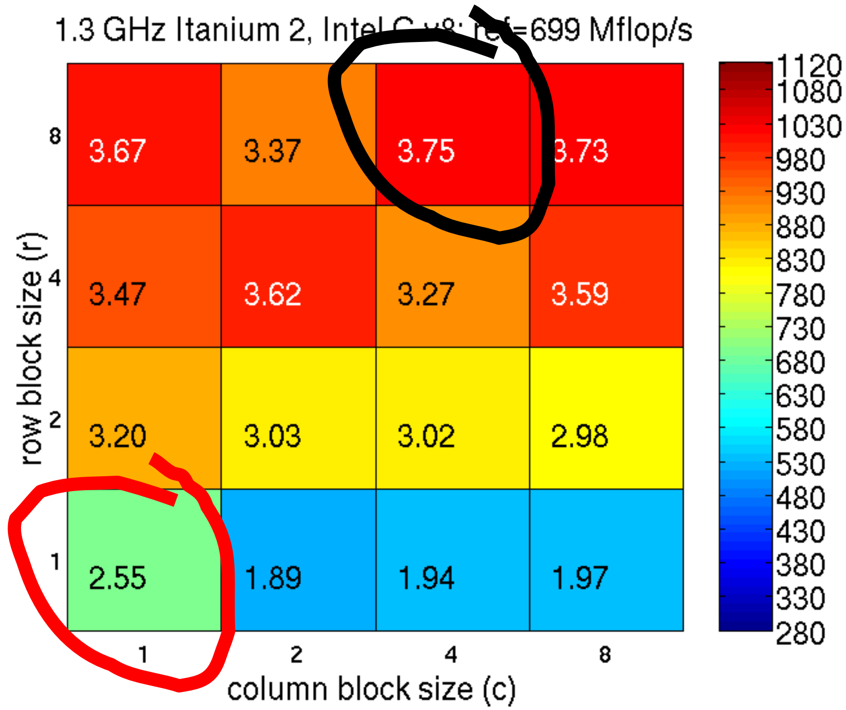
333 MHz Sun Ultra 2i, Sun C v6.0: ref=35 Mflop/s
900 MHz Ultra 3, Sun CC v6: ref=54 Mflop/s
2 GHz Pentium M, Intel C v8.1: ref=308 Mflop/s
1.4 GHz Opteron, gcc 3.4.2: ref=308 Mflop/s
row block size (r)
column block size (c)
1.58 1.65 1.61 1.79
1.48 1.55 1.54 1.74
1.29 1.37 1.46 1.68
1.00 1.19 1.29 1.30
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
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
1.28 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
1.3 GHz Power4, IBM xlc v6: ref=577 Mflop/s
800 MHz Itanium, Intel C v7: ref=146 Mflop/s
900 MHz Itanium 2, Intel C v8: ref=275 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
1.15 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
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
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

Better, worse, or about the same?

Better, worse, or about the same?
Itanium 2, 900 MHz → 1.3 GHz

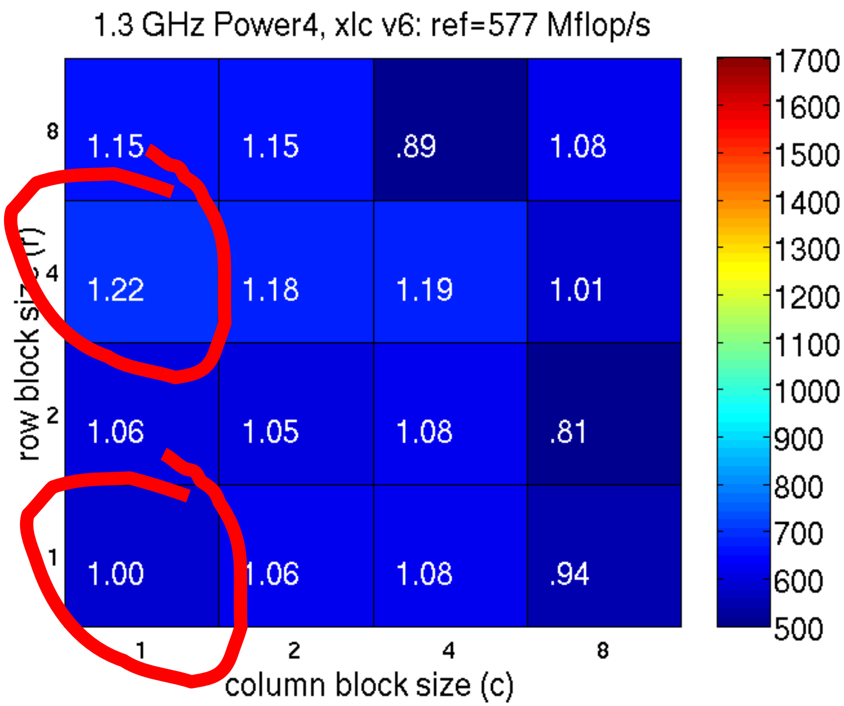


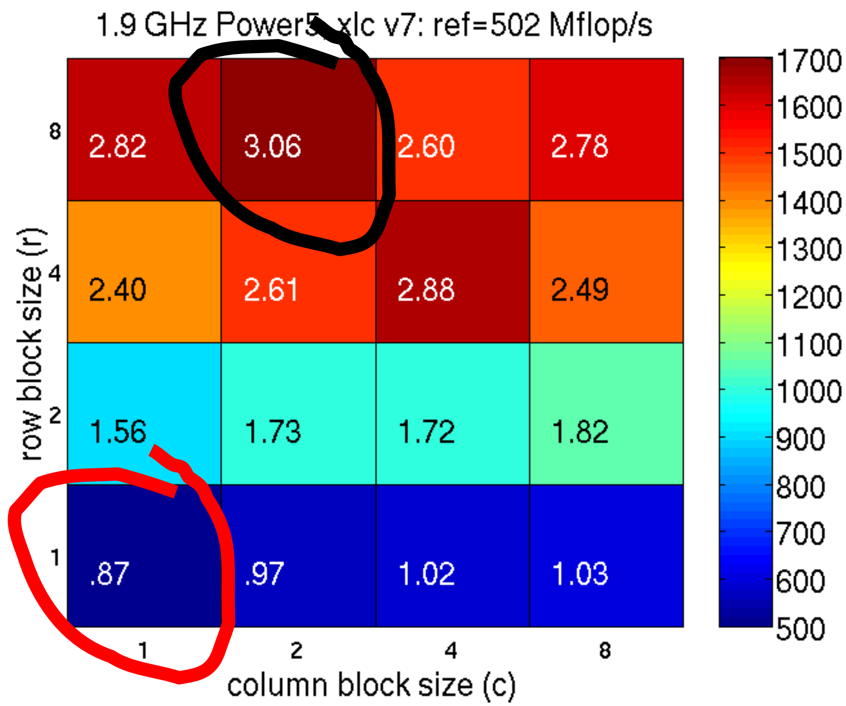


* Reference improves *

* Best possible worsens slightly *

Better, worse, or about the same?
Power4 → Power5

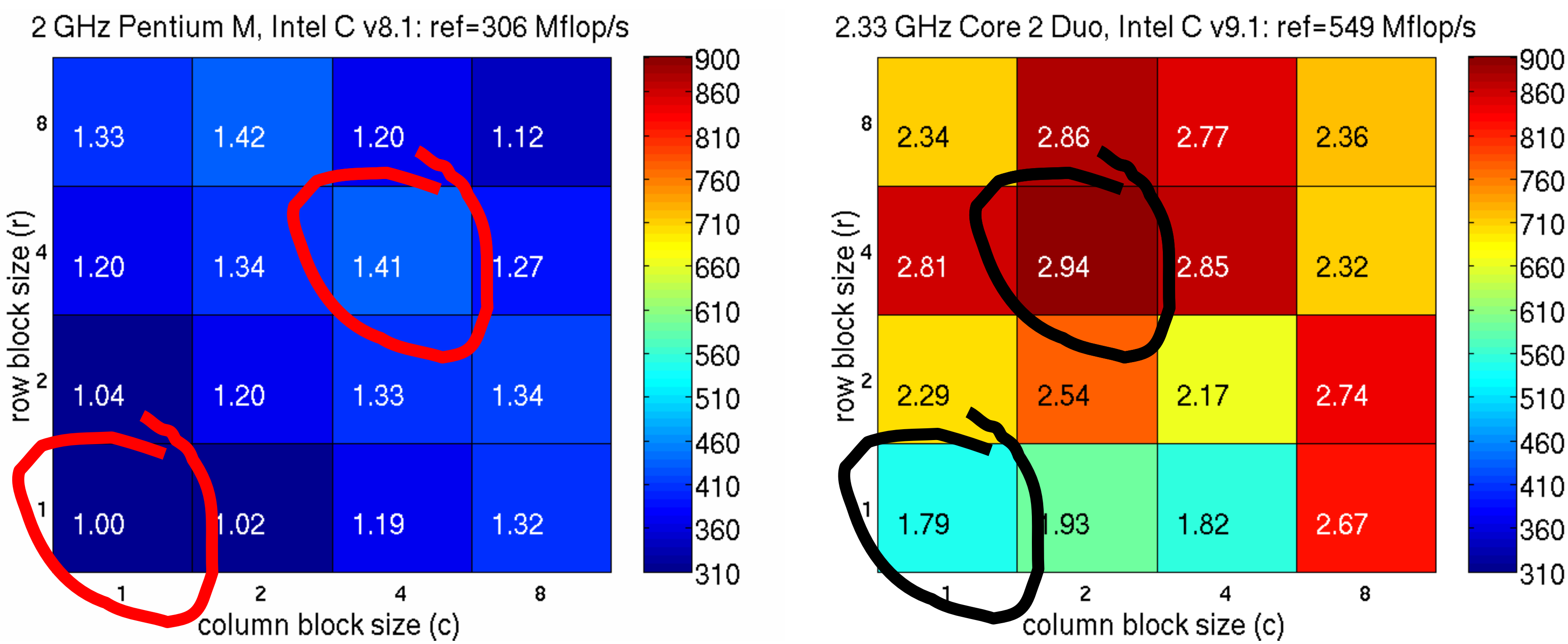




* Reference worsens! *

* Relative importance of tuning increases *

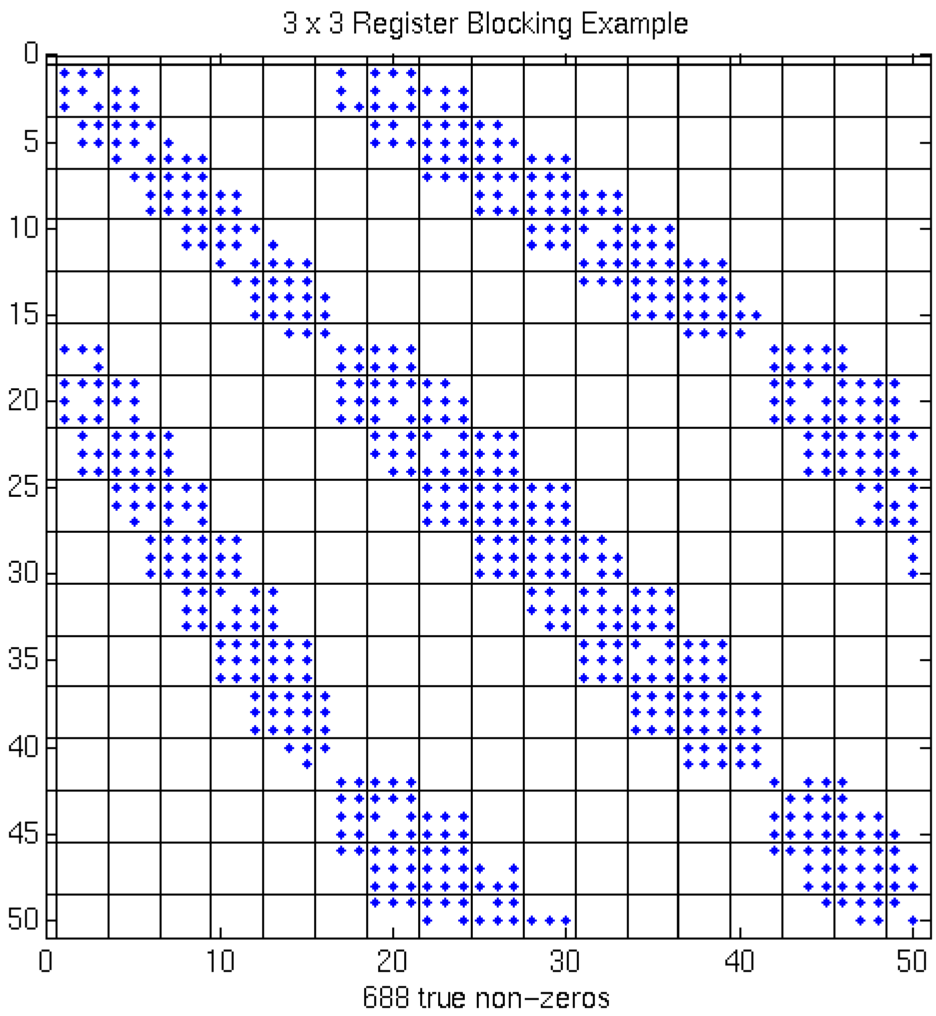
Better, worse, or about the same?
Pentium M → Core 2 Duo (1-core)



* Reference & best improve; relative speedup improves (~1.4 to 1.6×)

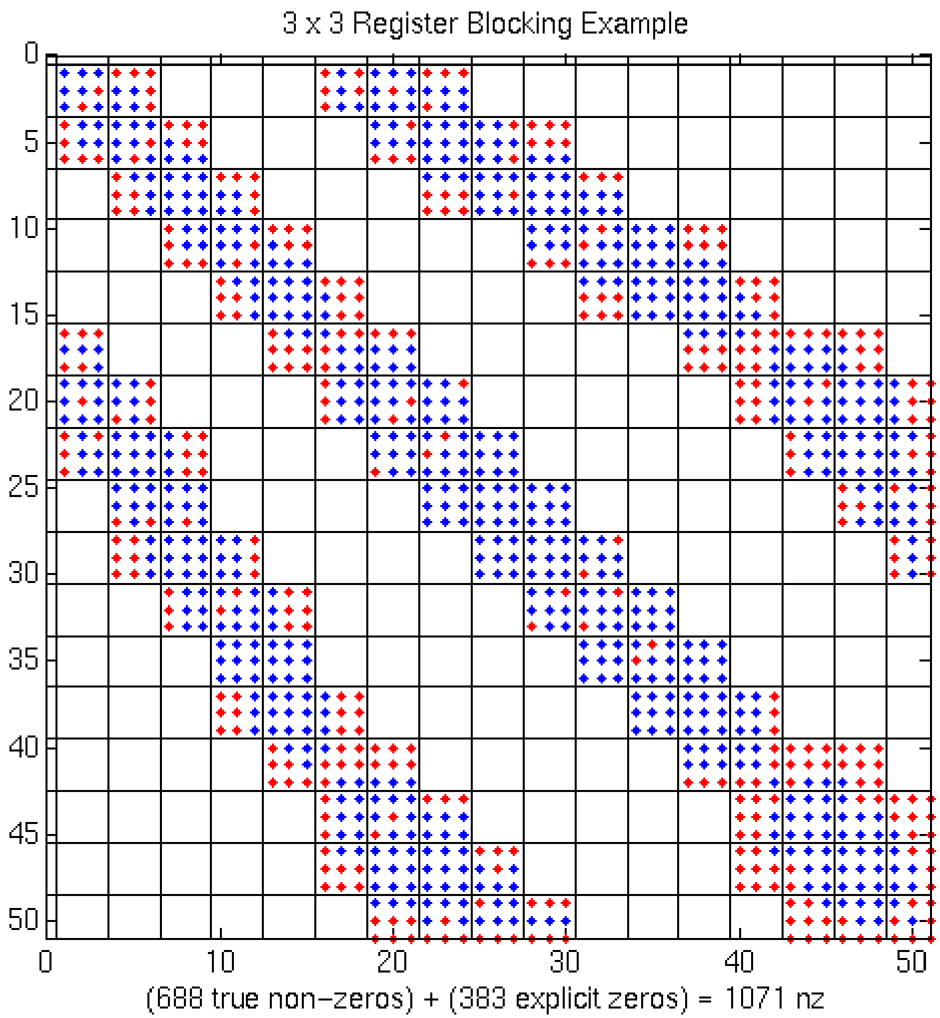
* Best decreases from 11% to 9.6% of peak *

More complex structures in practice



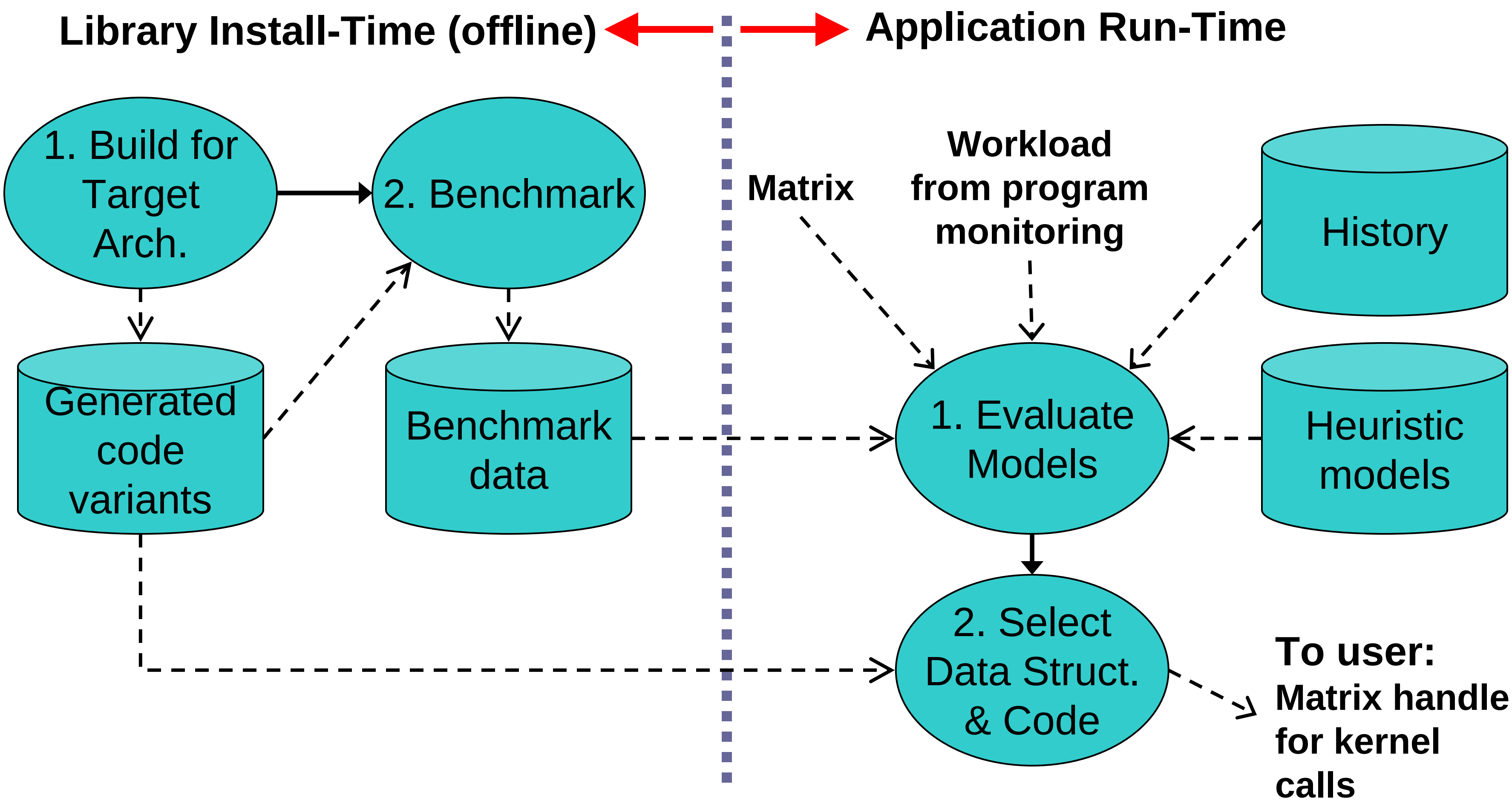
- Example: 3×3 blocking
  - Logical grid of 3×3 cells

Extra work can improve efficiency!



- Example: 3×3 blocking
  - Logical grid of 3×3 cells
  - Fill-in explicit zeros
  - Unroll 3x3 block multiplies
  - “Fill ratio” = 1.5
- On Pentium III: 1.5×
  - *i.e.*, 2/3 time

How OSKI tunes (Overview)



Heuristic model example: Select block size

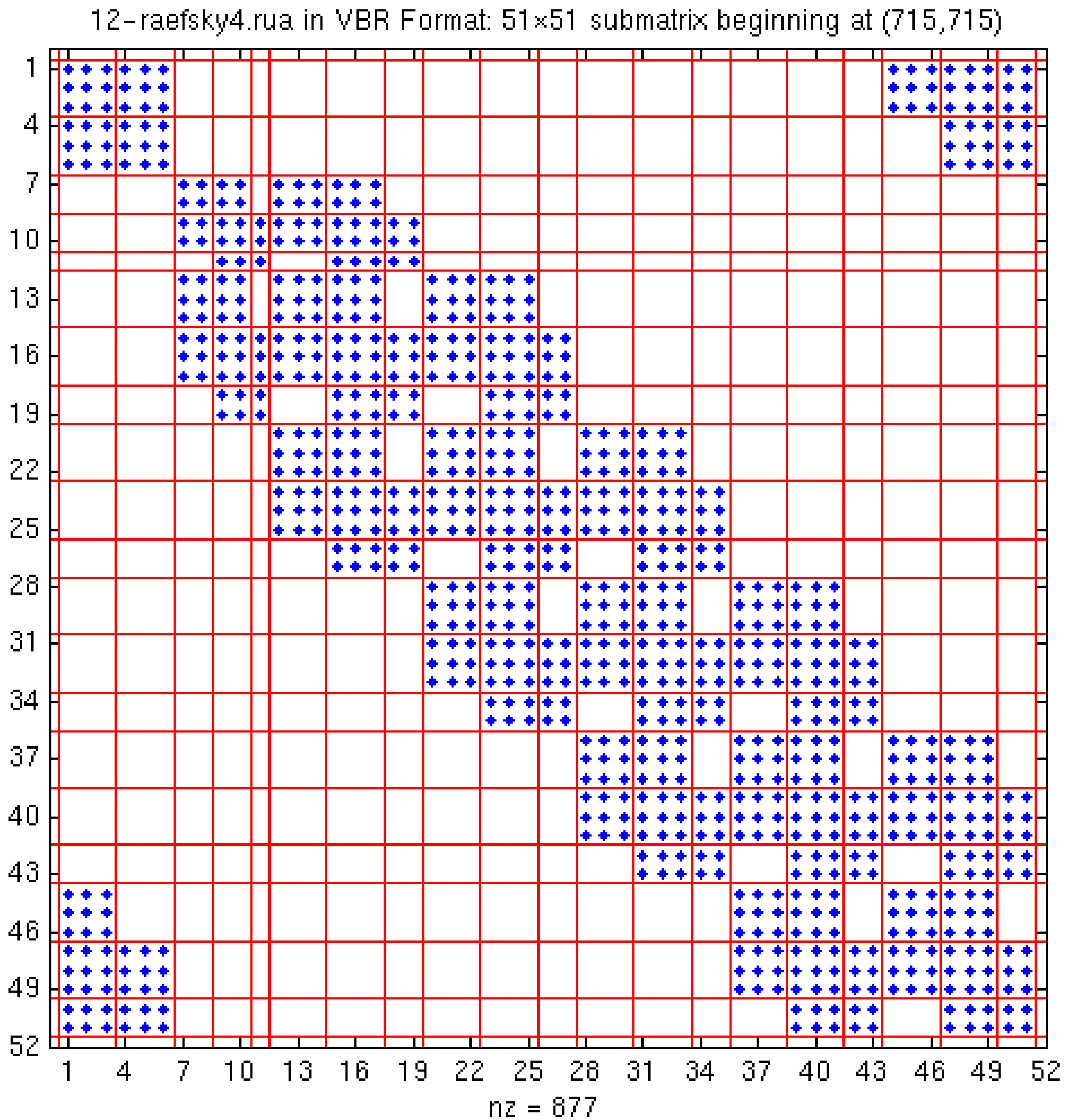
- Idea: Hybrid off-line / run-time model
  - Characterize machine with off-line benchmark
    - Precompute **Mflops(r, c)** using dense matrix for all r, c
    - Once per machine
  - Estimate matrix properties at run-time
    - Sample *A* to estimate **Fill(r, c)**
  - Run-time “search”
    - Select r, c to maximize **Mflops(r, c) / Fill(r, c)**
- In practice, selects (r, c) yielding perf. within 10% of best
- Run-time costs ~ 40 SpMVs
  - 80%+ = time to convert to new $r \times c$ format

Tunable optimization techniques

- Optimizations for SpMV
  - **Register blocking** (RB): up to 4× over CSR
  - Variable block splitting: 2.1× over CSR, 1.8× over RB
  - **Diagonals**: 2× over CSR
  - Reordering to create dense structure + splitting: 2× over CSR
  - **Symmetry**: 2.8× over CSR, 2.6× over RB
  - **Cache blocking**: 3× over CSR
  - **Multiple vectors (SpMM)**: 7× over CSR
  - And combinations…
- Sparse triangular solve
  - **Hybrid sparse/dense data structure**: 1.8× over CSR
- Higher-level kernels
  - $\boldsymbol{AA^T \cdot x}$ **or** $\boldsymbol{A^T A \cdot x}$: 4× over CSR, 1.8× over RB
  - $A^2 \cdot x$: 2× over CSR, 1.5× over RB

Structural splitting for complex patterns

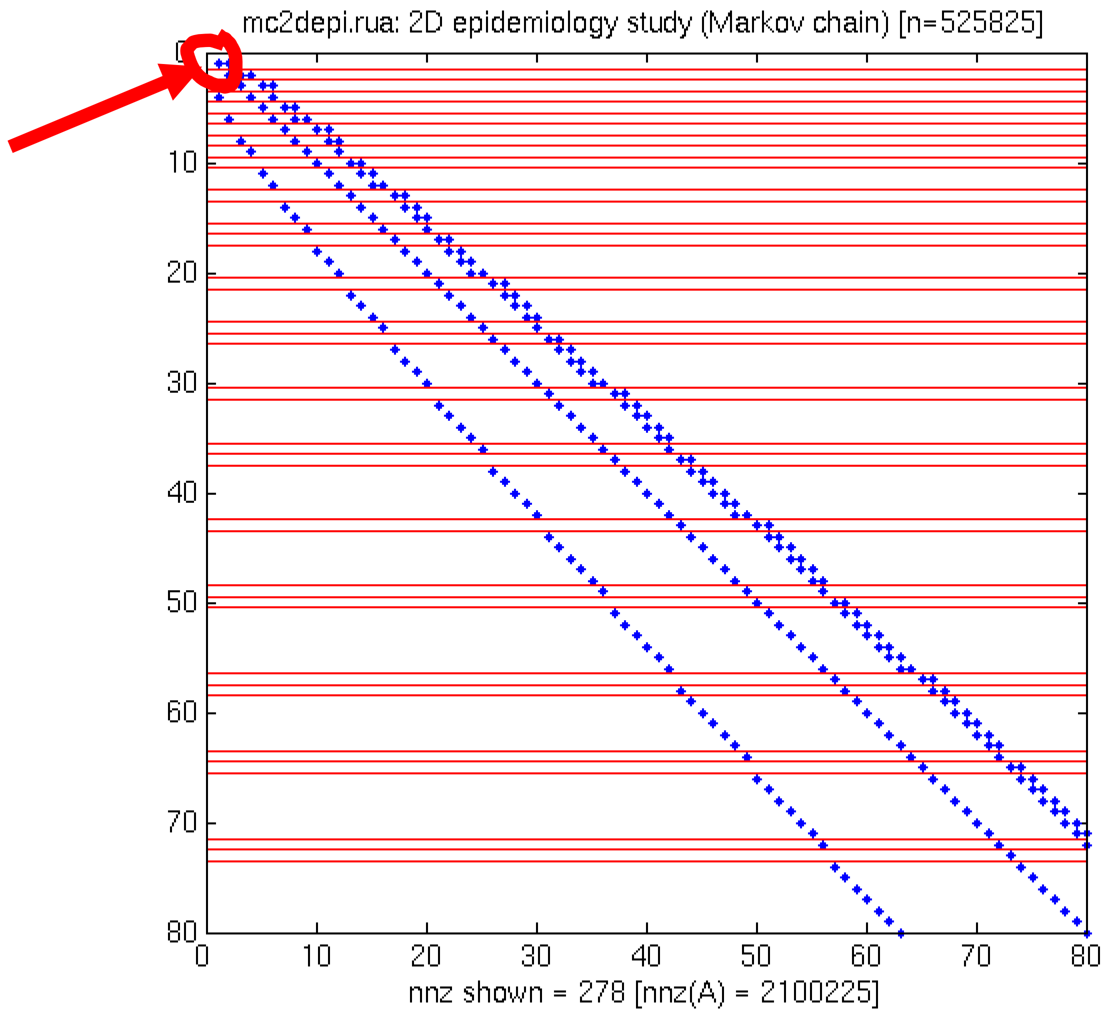
- Idea: Split $A = A_1 + A_2 + \ldots$, and tune A_i independently
  - Sample to detect "canonical" structures
  - Saves time and/or storage (avoid fill)
- Tuning knobs
  - Fill threshold, $.5 \leq \theta \leq 1$
  - Number of splittings, $2 \leq \boldsymbol{s} \leq 4$
  - Ordering of block sizes, $\boldsymbol{r_i} \times \boldsymbol{c_i}$; $\boldsymbol{r_s} \times \boldsymbol{c_s}$ = 1x1



2.1× **over CSR**

1.8× **over RB**

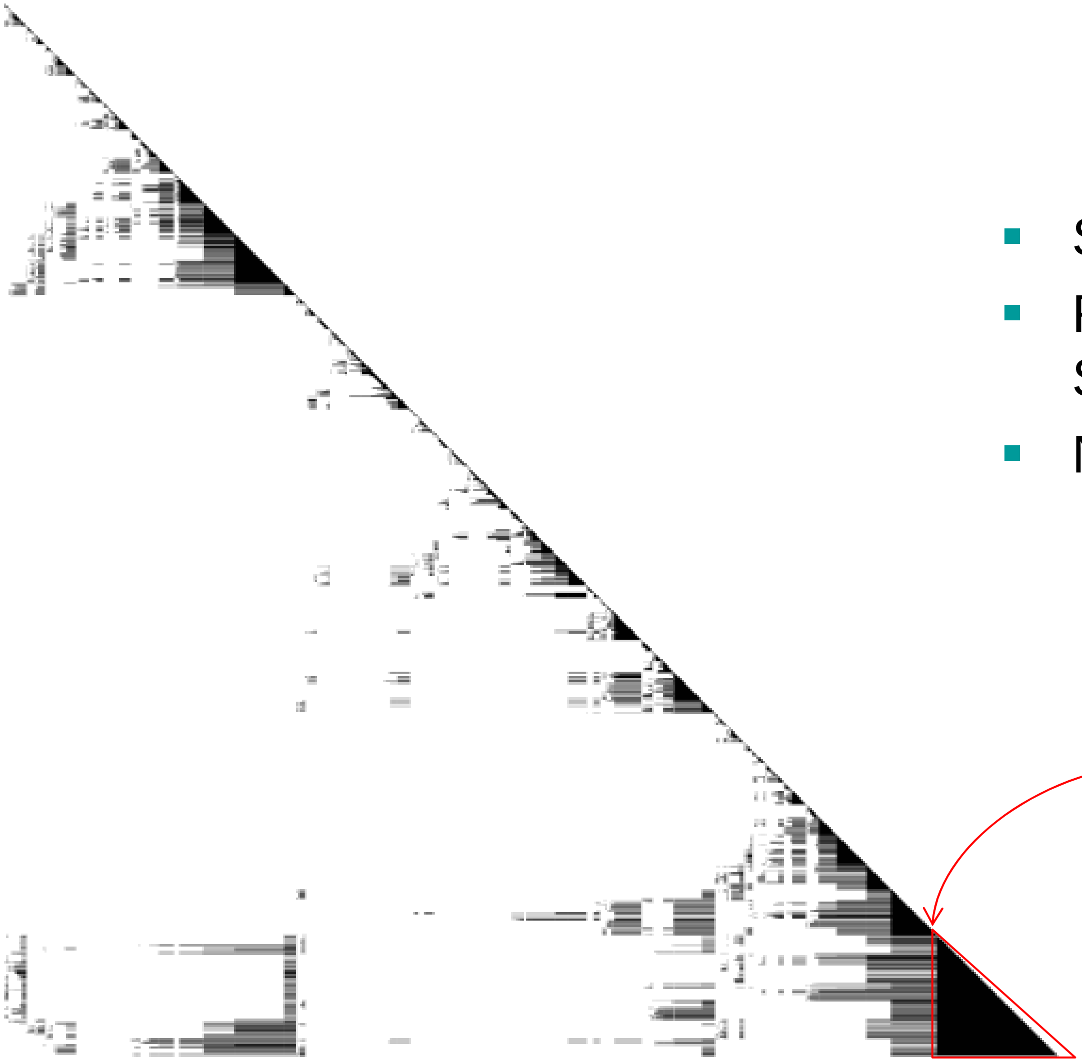
Example: Row-segmented diagonals



2×

over CSR

Dense sub-triangles for triangular solve



- Solve $Tx = b$ for x, T triangular
- 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+%!

Cache optimizations for $AA^T \cdot x$

- Idea: Interleave multiplication by A, A^T

$$AA^T x = \begin{pmatrix} a_1 & \cdots & a_n \end{pmatrix} \begin{pmatrix} a_1^{\,T} \\ \vdots \\ a_n^{\,T} \end{pmatrix} x = \sum_{i=1}^{n} a_i \left(a_i^{\,T} x \right)$$

"axpy"

dot product

- Combine with register optimizations: $a_i = r \times c$ block row

OSKI tunes for workloads

- Bi-conjugate gradients - equal mix of $A \cdot x$ and $A^T \cdot y$
  - 3×1: $A \cdot x$, $A^T \cdot y$ = 1053, 343 Mflop/s → 517 Mflop/s
  - 3×3: $A \cdot x$, $A^T \cdot y$ = 806, 826 Mflop/s → 816 Mflop/s
- Higher-level operation - ($A \cdot x$, $A^T \cdot y$) kernel
  - 3×1: 757 Mflop/s
  - 3×3: 1400 Mflop/s
- Workload tuning
  - Evaluate weighted sums of empirical models
  - Dynamic programming to evaluate alternatives

How to call OSKI in a “legacy” app

```
int* ptr = …, *ind = …;  double* val = …; /*  Matrix A, in CSR format */
double* x = …, *y = …; /*  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 );
r = ddot (x, y);  /* Some dense BLAS op on vectors */
```

How to call OSKI in a "legacy" app

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix A, in CSR format */
double* x = …, *y = …; /* Vectors */

/* Step 1: Create OSKI wrappers */
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 );
r = ddot (x, y);
```

How to call OSKI in a “legacy” app

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix A, in CSR format */
double* x = …, *y = …; /* Vectors */

/* Step 1: Create OSKI wrappers */
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);

/* Step 2: Call tune (with optional hints) */
oski_SetHintMatMult(A_tunable, …, 500);
oski_TuneMat (A_tunable);

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

How to call OSKI in a “legacy” app

```
int* ptr = …, *ind = …;  double* val = …; /* Matrix A, in CSR format */
double* x = …, *y = …; /* Vectors */

/* Step 1: Create OSKI wrappers */
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);

/* Step 2: Call tune (with optional hints) */
oski_SetHintMatMult(A_tunable, …, 500);
oski_TuneMat (A_tunable);

/* 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 3
r = ddot (x, y);
```

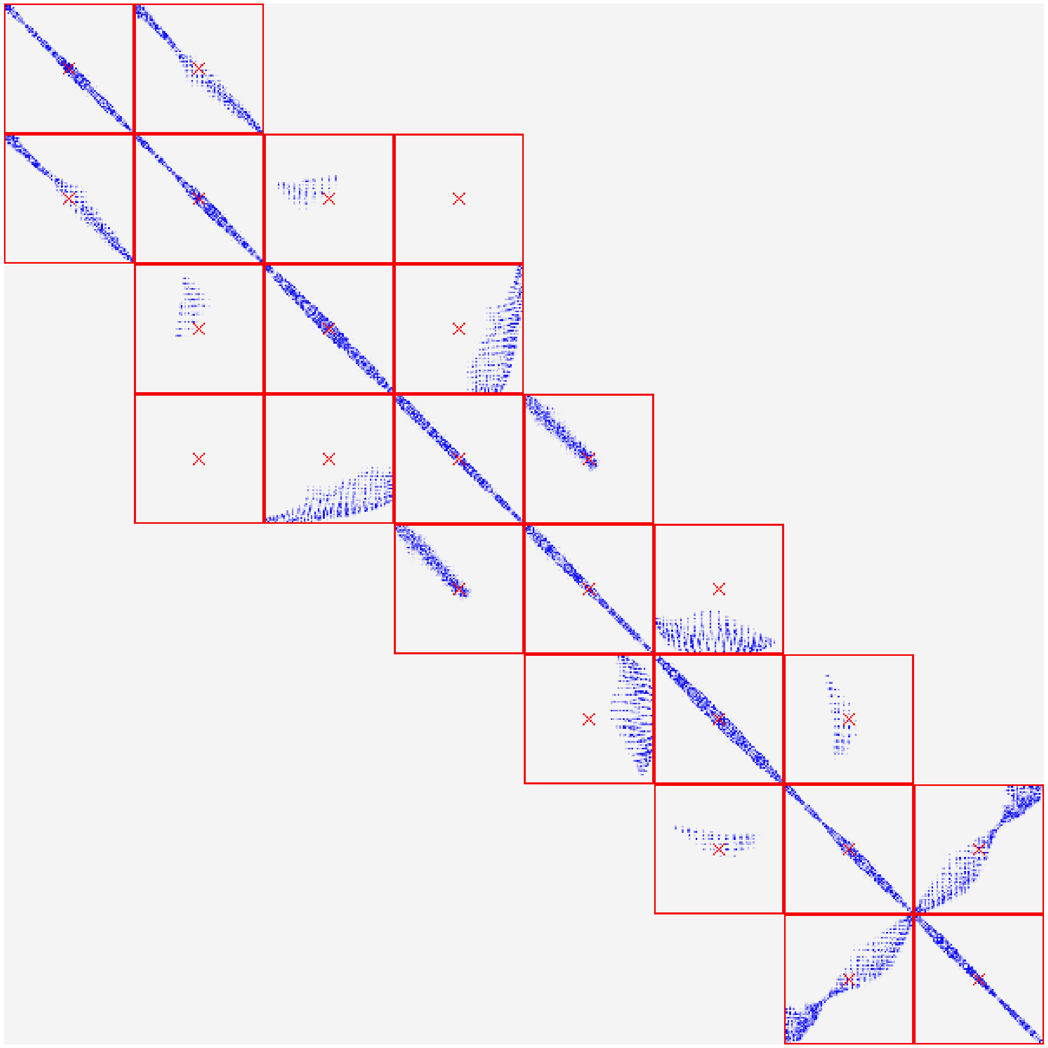
Other OSKI features

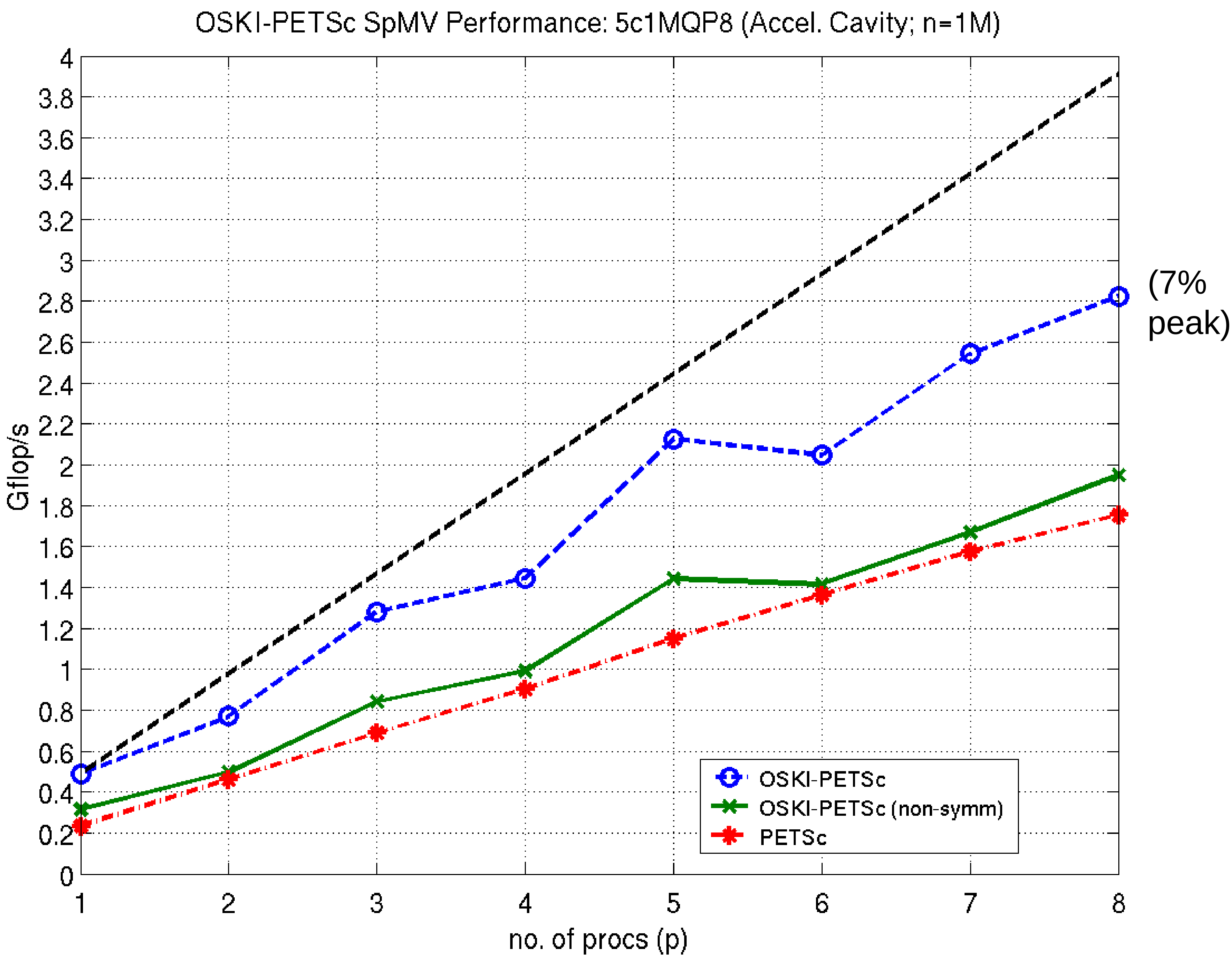
- Implicit tuning mode
- OSKI-Lua
  - Embedded scripting language w/ light footprint
  - Lists the sequence of data structure transformations used
- Get/set values
- “Plug-in” extensibility of new data structures

Examples of OSKI's early impact

- Integrating into major linear solver libraries
  - PETSc
  - Trilinos – R&D100 (Heroux)
- Early adopter: ClearShape, Inc.
  - Core product: lithography process simulator
  - 2× speedup on full simulation after using OSKI
- Proof-of-concept: SLAC T3P accelerator design app
  - SpMV dominates execution time
  - Symmetry, 2×2 block structure
  - 2× speedups over parallel PETSc on a Xeon cluster

SLAC T3P Matrix




OSKI-PETSc SpMV Performance: 5c1MQP8 (Accel. Cavity; n=1M)
Gflop/s
no. of procs (p)
(7% peak)
OSKI-PETSc
OSKI-PETSc (non-symm)
PETSc
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3
3.2
3.4
3.6
3.8
4
1
2
3
4
5
6
7
8

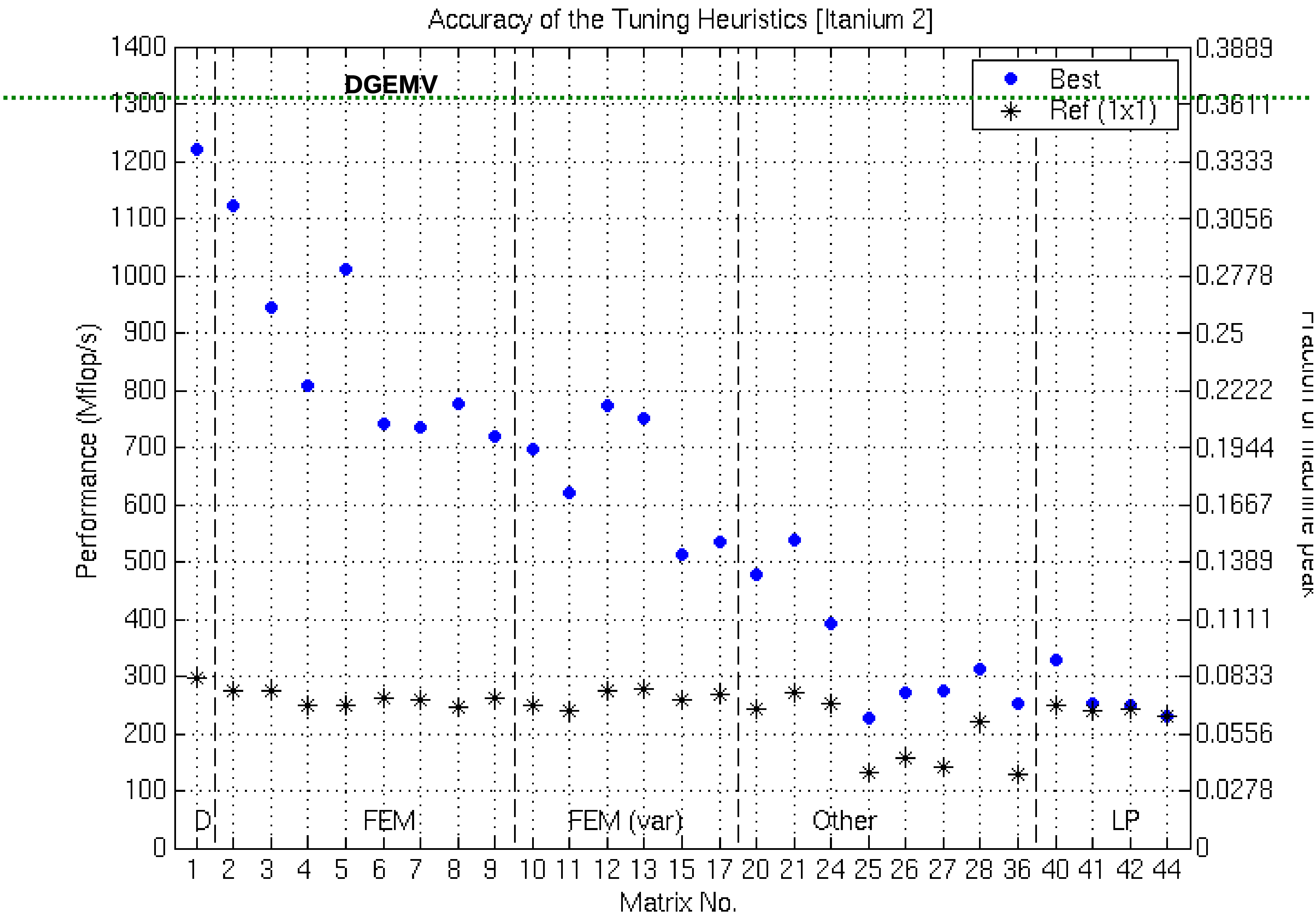
General theme: Aggressively exploit structure

- Application- and architecture-specific optimization
  - *E.g.*, Sparse matrix patterns
  - Robust performance in spite of architecture-specific peculiarities
  - Augment static models with benchmarking and search
- Short-term OSKI extensions
  - Integrate into large-scale apps, full-solver contexts
    - Accelerator design, plasma physics (DOE)
    - Geophysical simulation based on Block Lanczos ($A^T A * X$; LBL)
    - PRIMME eigensolver
  - Other kernels: Matrix triple products
  - Parallelism

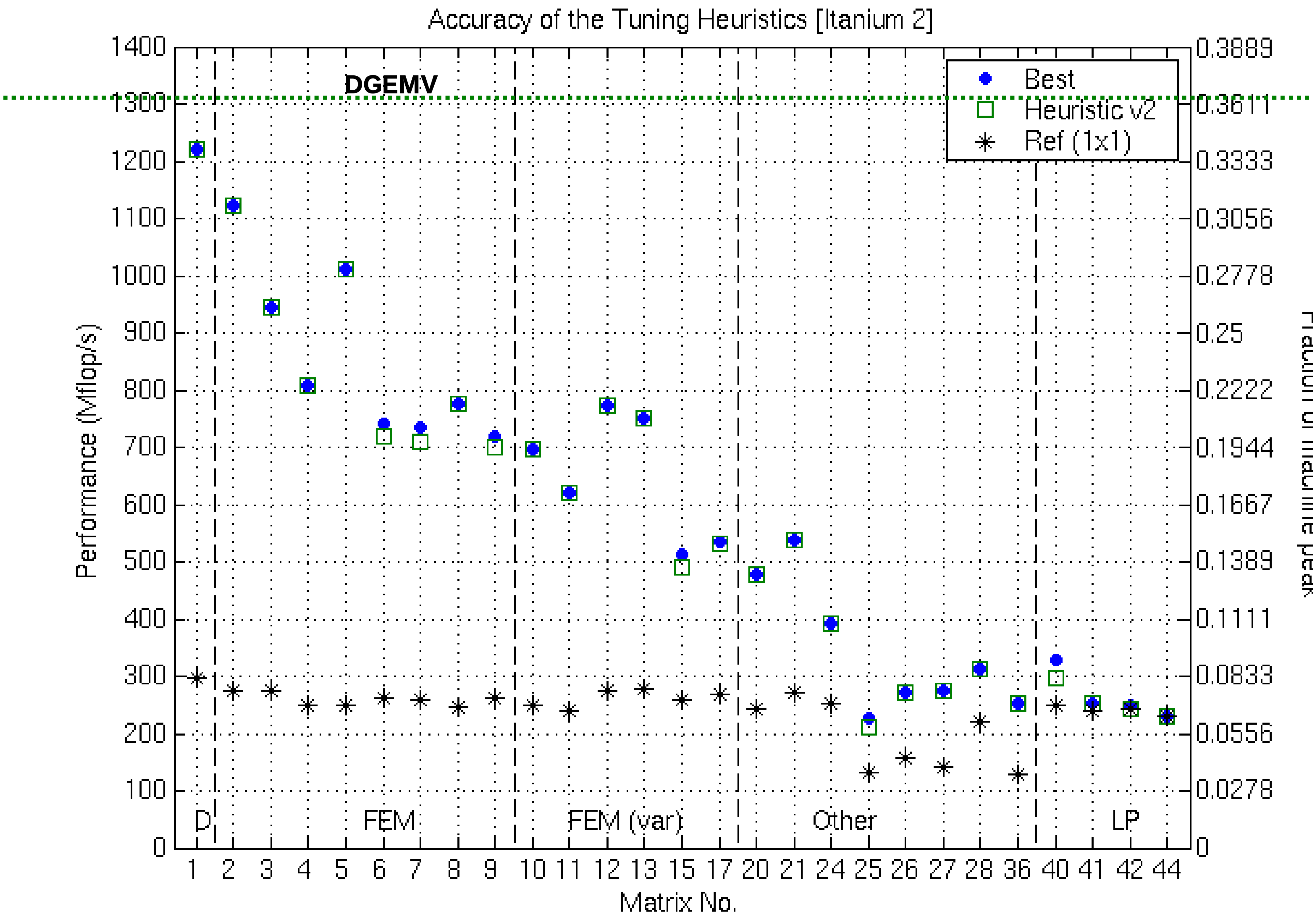
How to best generate all this code? Run-time?
{Data structure} x {kernel} x {low-level opt.}

- Optimizations for SpMV
  - **Register blocking** (RB): up to 4× over CSR
  - Variable block splitting: 2.1× over CSR, 1.8× over RB
  - **Diagonals**: 2× over CSR
  - Reordering to create dense structure + splitting: 2× over CSR
  - **Symmetry**: 2.8× over CSR, 2.6× over RB
  - **Cache blocking**: 3× over CSR
  - Multiple vectors (SpMM): 7× over CSR
  - And combinations…
- Sparse triangular solve
  - **Hybrid sparse/dense data structure**: 1.8× over CSR
- Higher-level kernels
  - $\boldsymbol{AA^T \cdot x}$ **or** $\boldsymbol{A^T A \cdot x}$: 4× over CSR, 1.8× over RB
  - $A^2 \cdot x$: 2× over CSR, 1.5× over RB

End



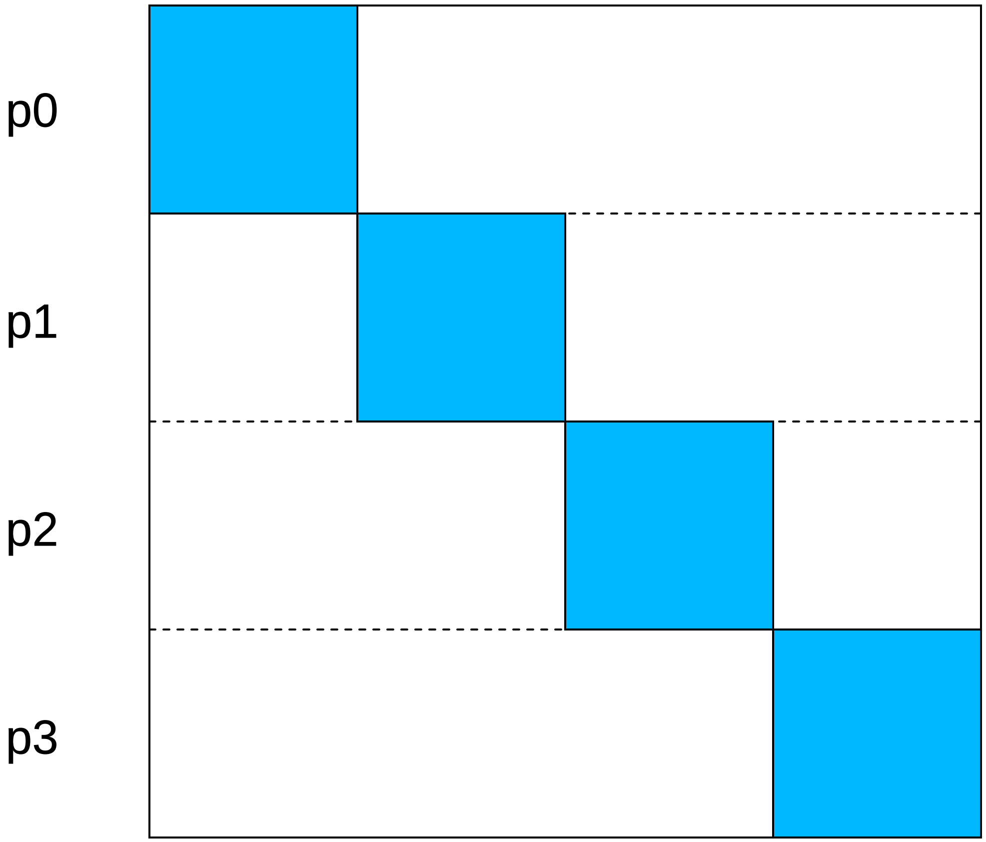
NOTE: “Fair” flops used (ops on explicit zeros not counted as “work”)



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

Quick-and-dirty Parallelism: OSKI-PETSc

- Extend PETSc's distributed memory SpMV (MATMPIAIJ)



- PETSc
  - Each process stores diag (all-local) and off-diag submatrices
- OSKI-PETSc:
  - Add OSKI wrappers
  - Each submatrix tuned independently