Automatic Tuning of Sparse Matrix Kernels

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

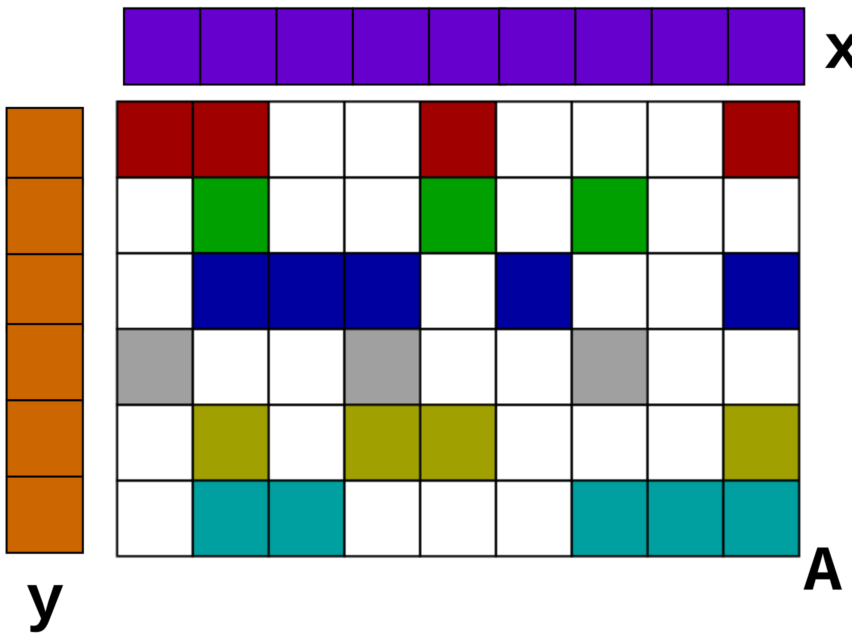
Motivation for Tuning Sparse Matrices

- 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** $A^TA \cdot x$, **2x** $A^2 \cdot x$

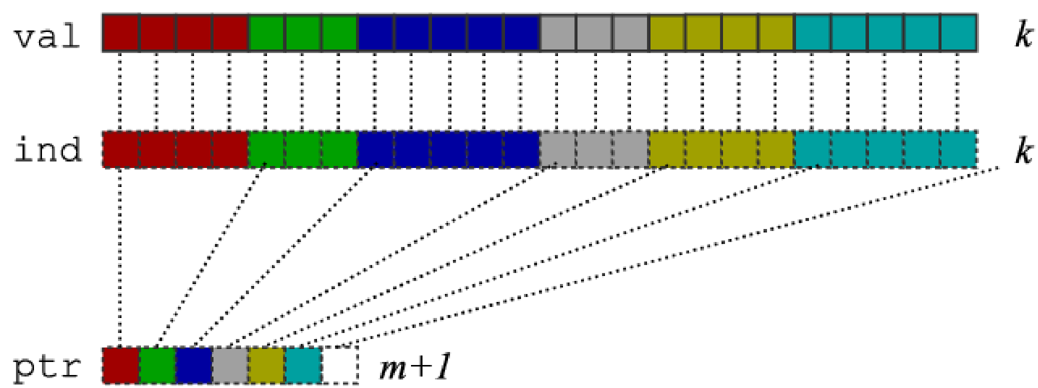
OSKI: Optimized Sparse Kernel Interface

- 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`**
  - Being integrated into PETSc

Compressed Sparse Row (CSR) Storage



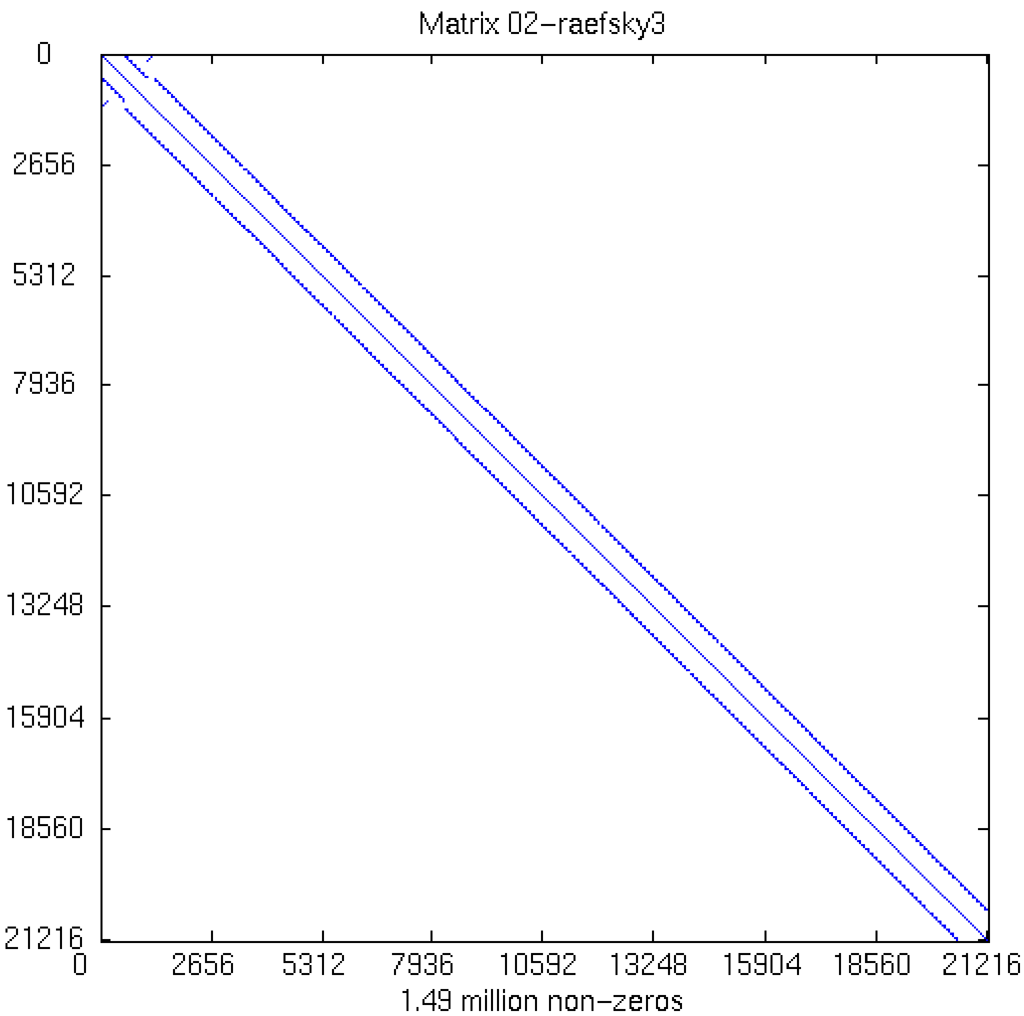
Representation of **A**



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]]
```

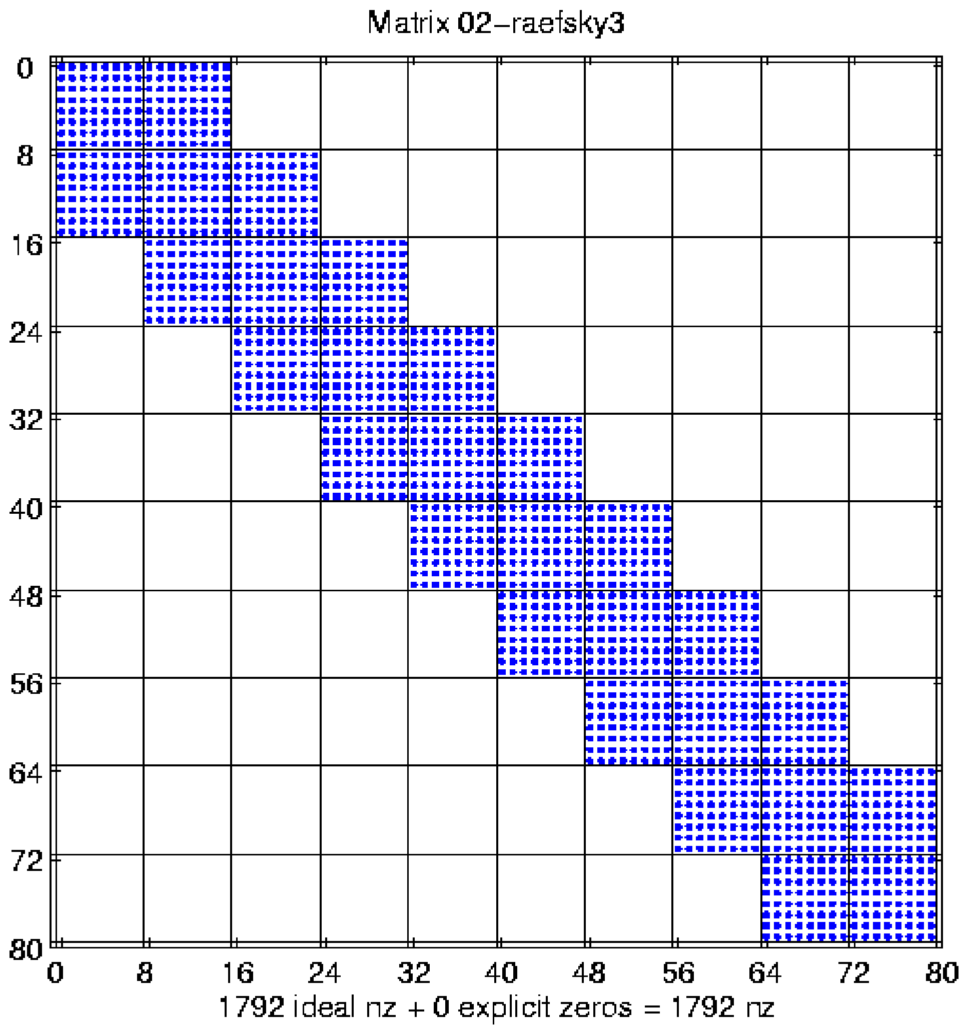
Example: The Difficulty of Tuning



- 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

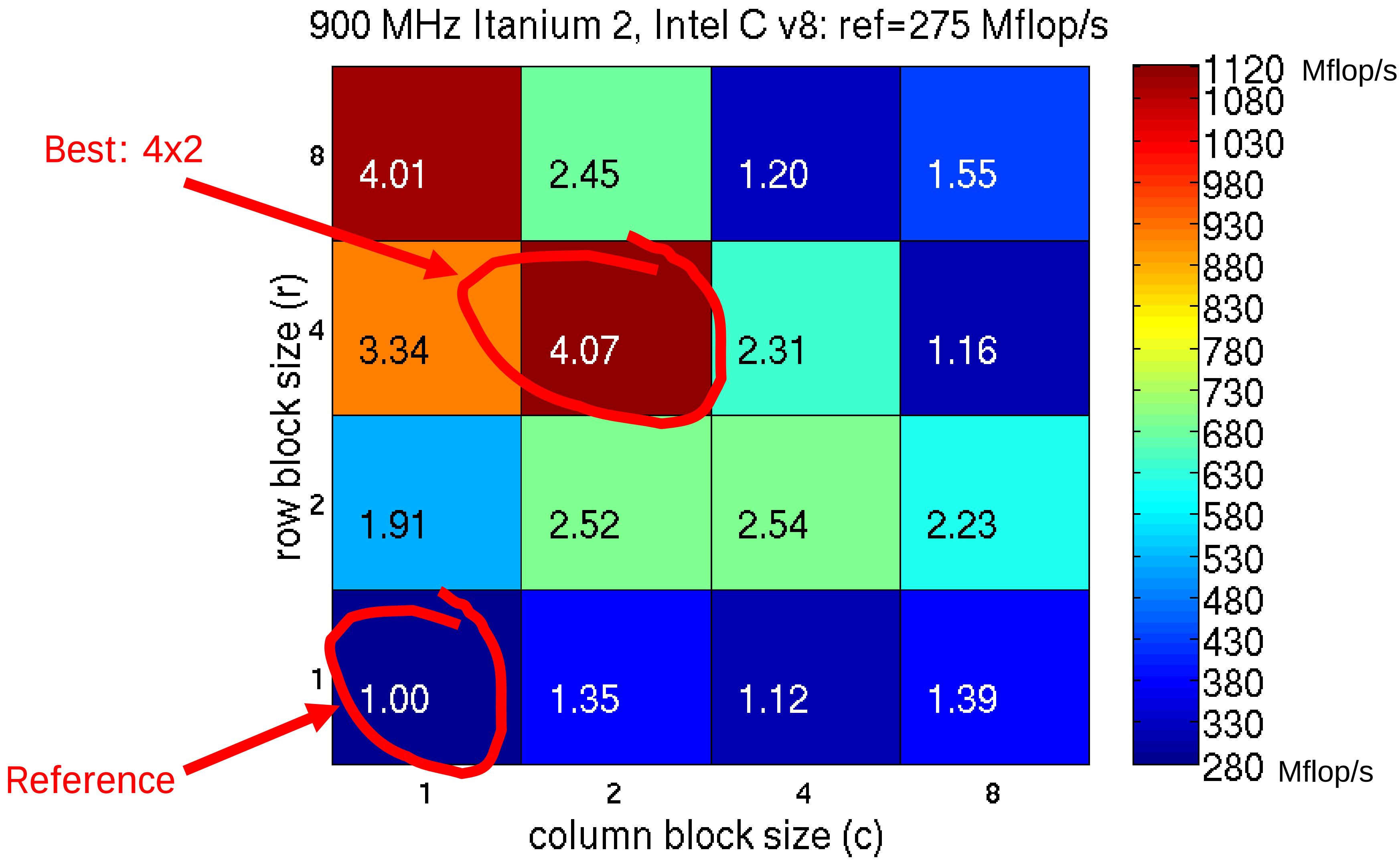
- **8x8** dense substructure

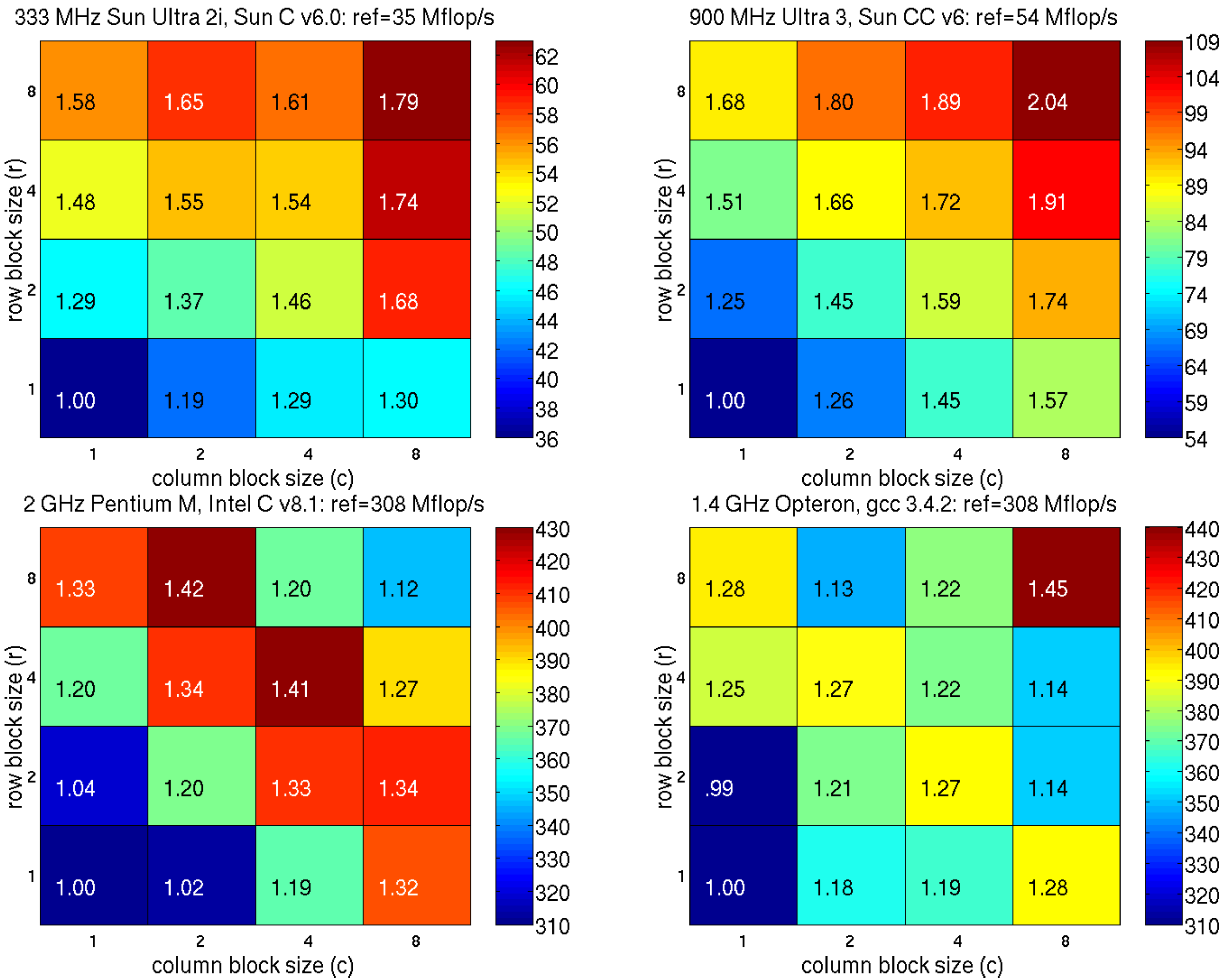
What We Expect

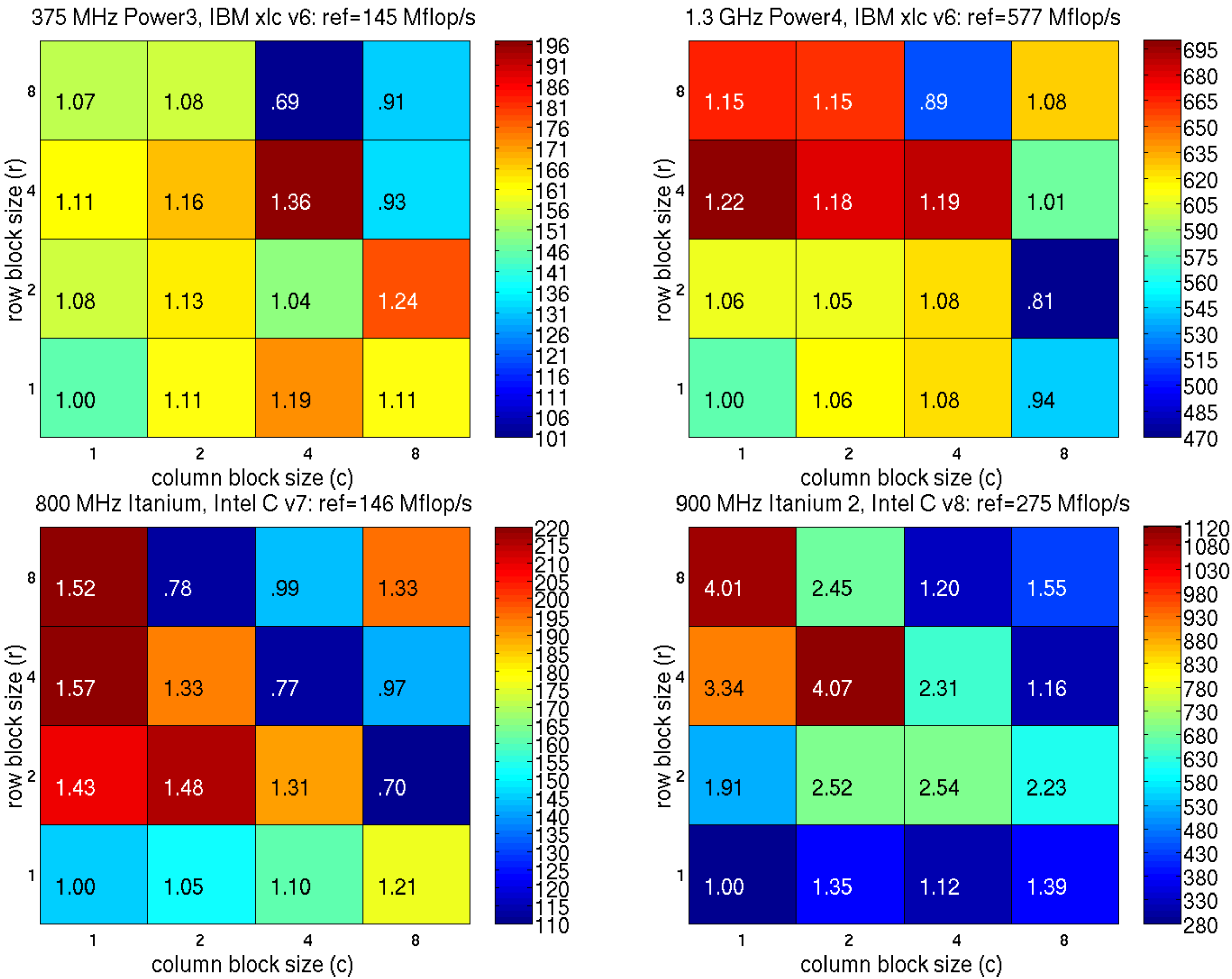
- Assume
  - Cost(SpMV) = time to read matrix
  - 1 double-word = 2 integers
  - r, c in {1, 2, 4, 8}
- CSR: 1 int / non-zero
- BCSR(r x c): 1 int / (r*c non-zeros)
- As r*c increases, speedup should
  - Increase smoothly
  - Approach 1.5

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

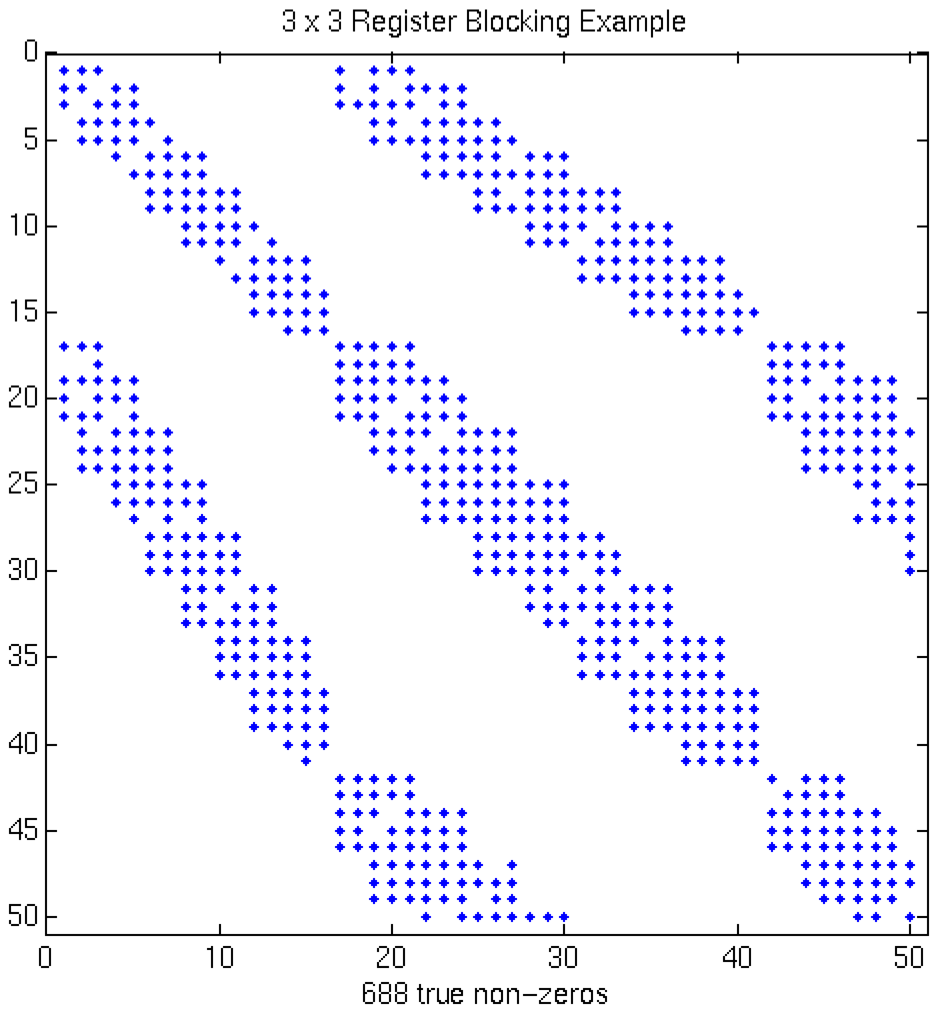
What We Get (The Need for Search)




333 MHz Sun Ultra 2i, Sun C v6.0: ref=35 Mflop/s
row block size (r)
column block size (c)
8: 1.58 1.65 1.61 1.79
4: 1.48 1.55 1.54 1.74
2: 1.29 1.37 1.46 1.68
1: 1.00 1.19 1.29 1.30
1 2 4 8
62 60 58 56 54 52 50 48 46 44 42 40 38 36
900 MHz Ultra 3, Sun CC v6: ref=54 Mflop/s
row block size (r)
column block size (c)
8: 1.68 1.80 1.89 2.04
4: 1.51 1.66 1.72 1.91
2: 1.25 1.45 1.59 1.74
1: 1.00 1.26 1.45 1.57
1 2 4 8
109 104 99 94 89 84 79 74 69 64 59 54
2 GHz Pentium M, Intel C v8.1: ref=308 Mflop/s
row block size (r)
column block size (c)
8: 1.33 1.42 1.20 1.12
4: 1.20 1.34 1.41 1.27
2: 1.04 1.20 1.33 1.34
1: 1.00 1.02 1.19 1.32
1 2 4 8
430 420 410 400 390 380 370 360 350 340 330 320 310
1.4 GHz Opteron, gcc 3.4.2: ref=308 Mflop/s
row block size (r)
column block size (c)
8: 1.28 1.13 1.22 1.45
4: 1.25 1.27 1.22 1.14
2: .99 1.21 1.27 1.14
1: 1.00 1.18 1.19 1.28
1 2 4 8
440 430 420 410 400 390 380 370 360 350 340 330 320 310

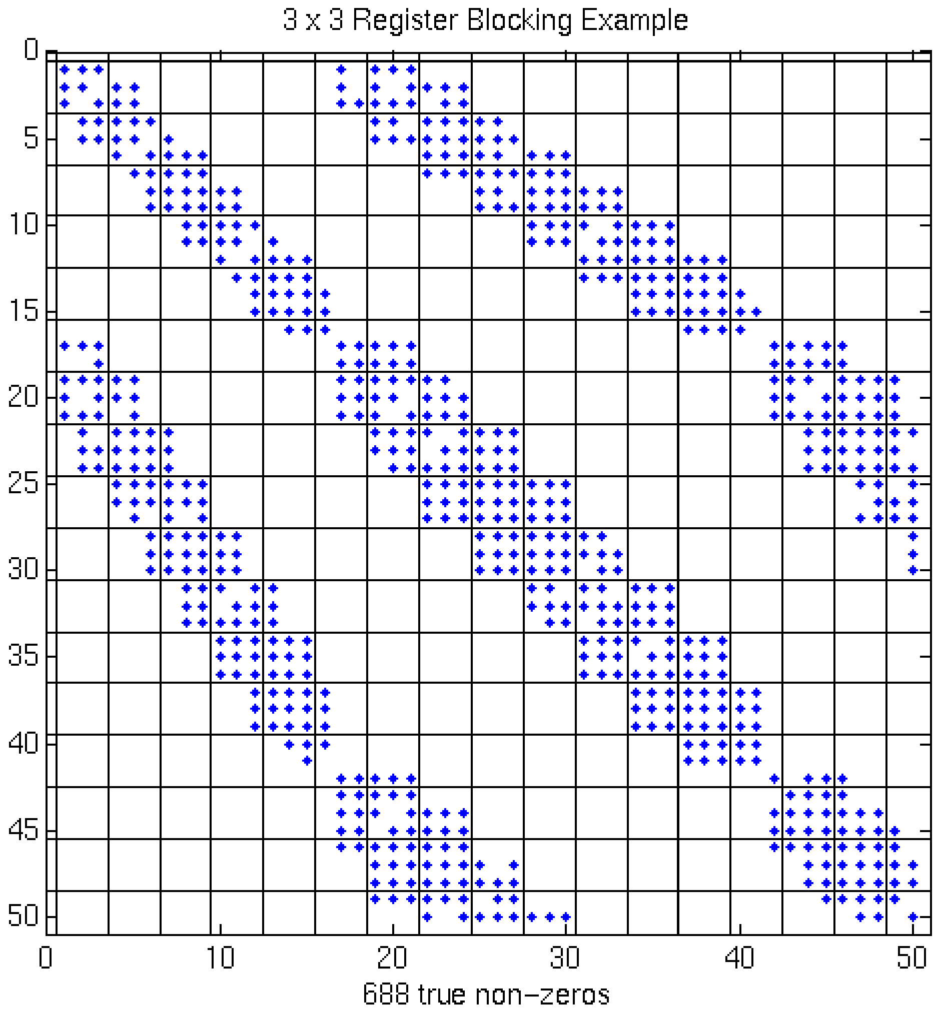

375 MHz Power3, IBM xlc v6: ref=145 Mflop/s
1.07 1.08 .69 .91
1.11 1.16 1.36 .93
1.08 1.13 1.04 1.24
1.00 1.11 1.19 1.11
196 191 186 181 176 171 166 161 156 151 146 141 136 131 126 121 116 111 106 101
1.3 GHz Power4, IBM xlc v6: ref=577 Mflop/s
1.15 1.15 .89 1.08
1.22 1.18 1.19 1.01
1.06 1.05 1.08 .81
1.00 1.06 1.08 .94
695 680 665 650 635 620 605 590 575 560 545 530 515 500 485 470
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
220 215 210 205 200 195 190 185 180 175 170 165 160 155 150 145 140 135 130 125 120 115 110
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
1120 1080 1030 980 930 880 830 780 730 680 630 580 530 480 430 380 330 280
row block size (r)
8 4 2 1
column block size (c)
1 2 4 8

Still More Surprises



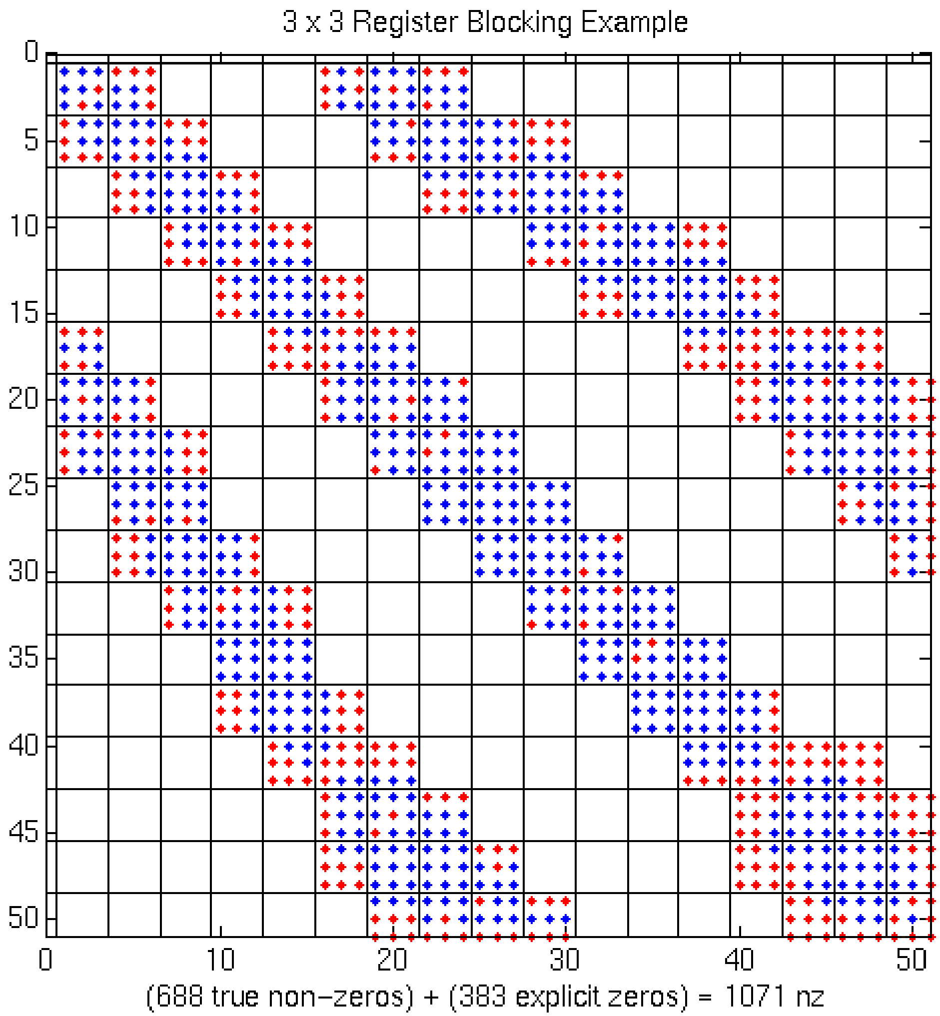
- More complicated non-zero structure in general

Still More Surprises



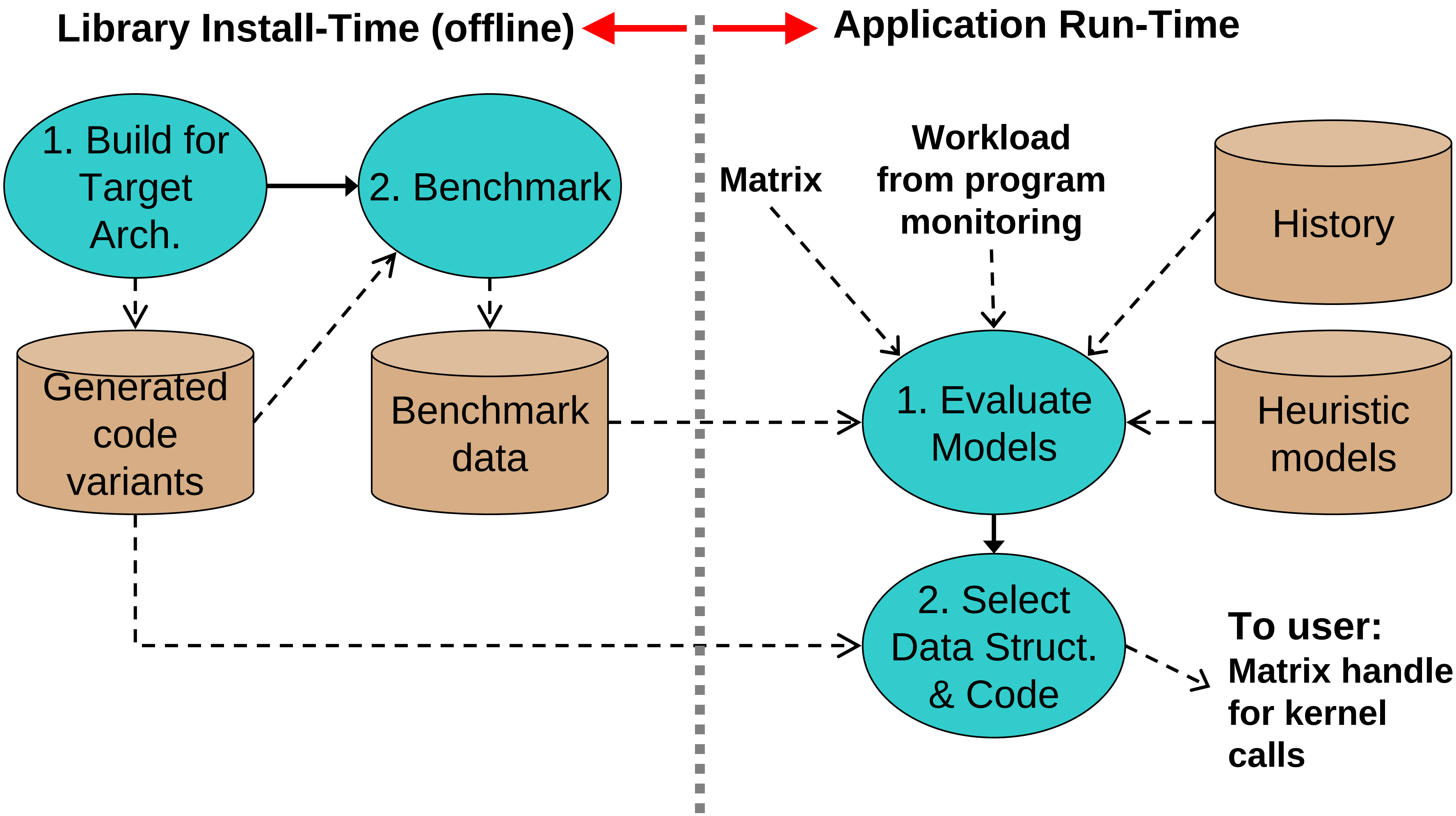
- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells

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!

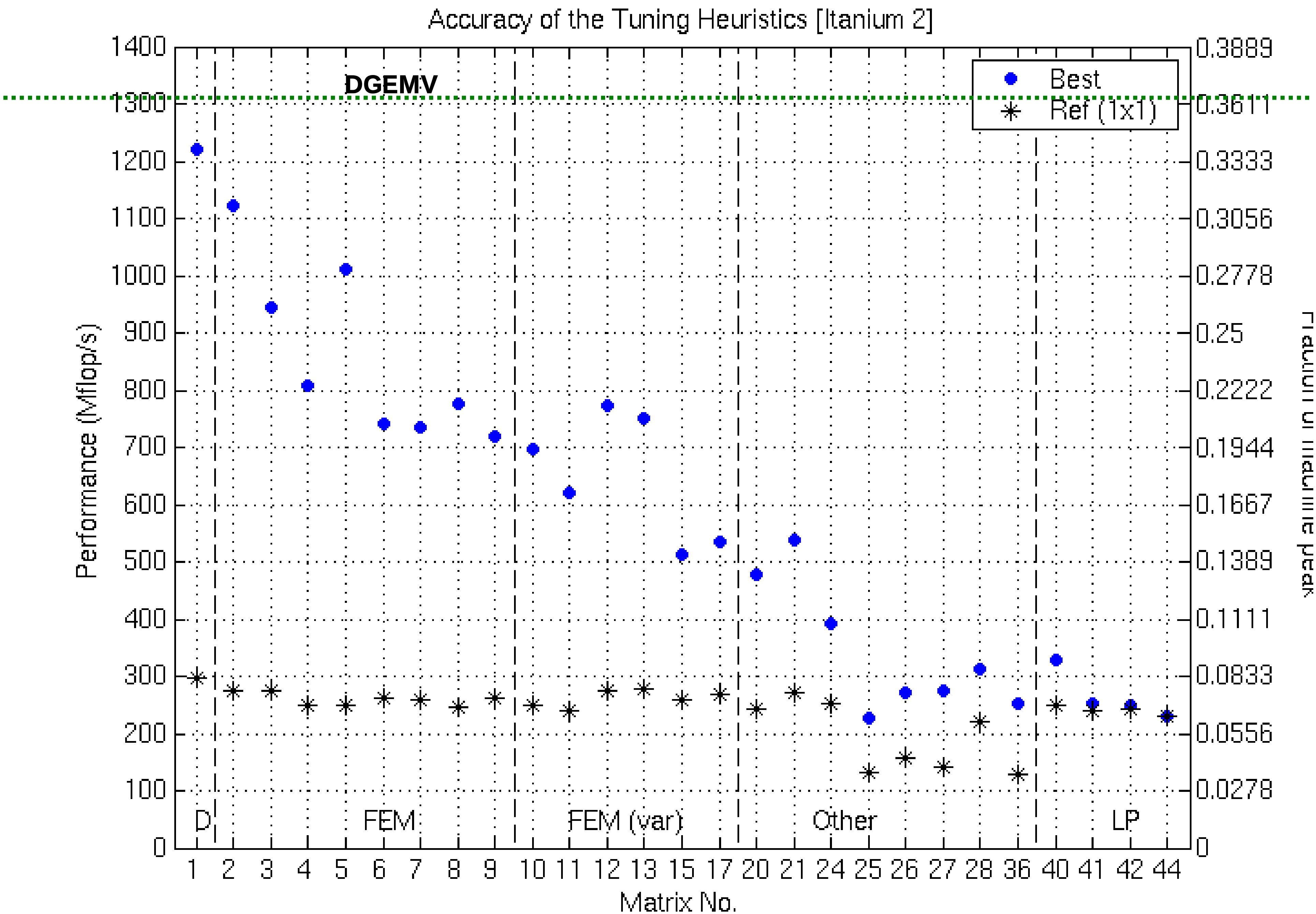
How OSKI Tunes (Overview)



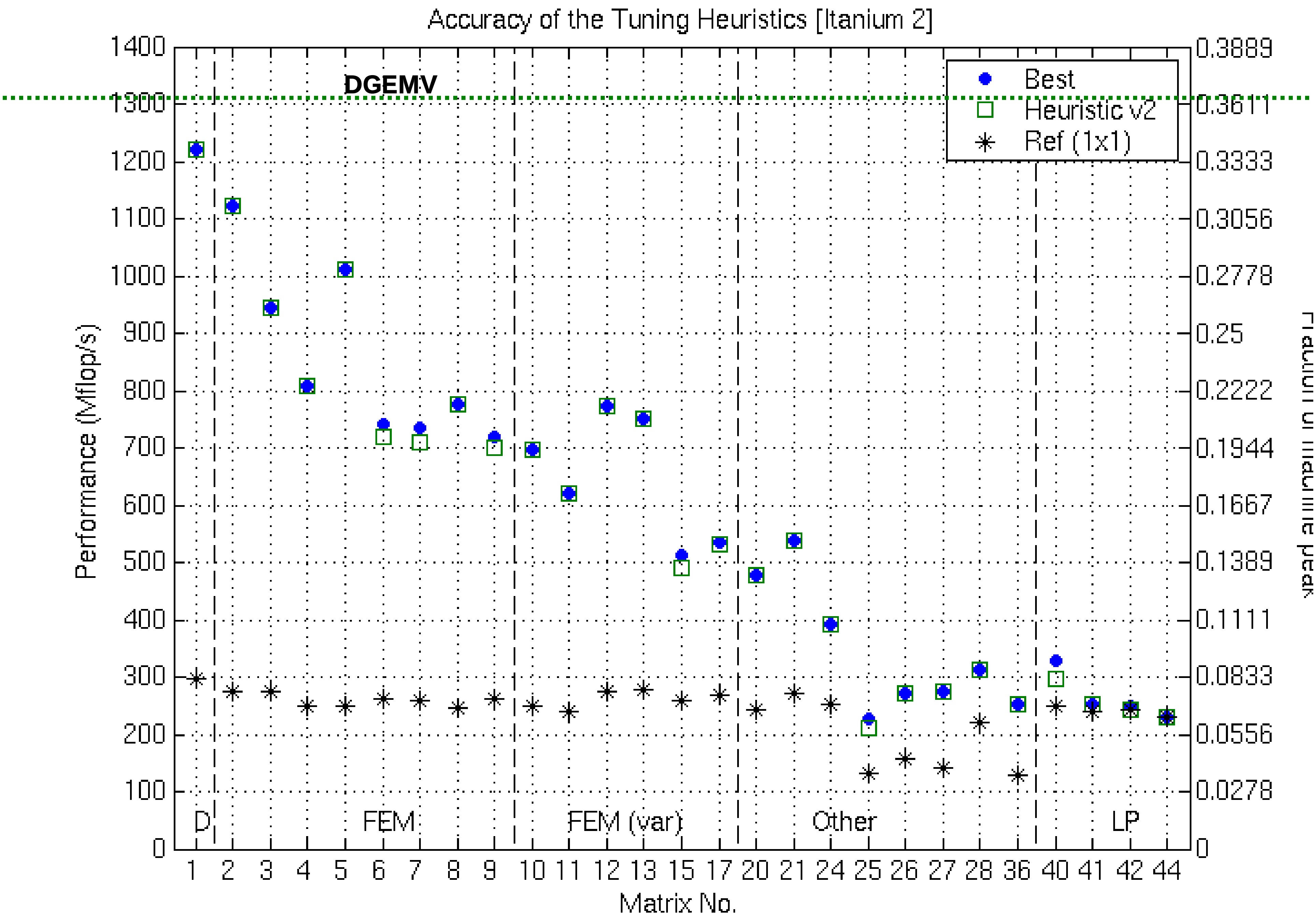
Extensibility: Advanced users may write & dynamically add “Code variants” and “Heuristic models” to system.

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”)

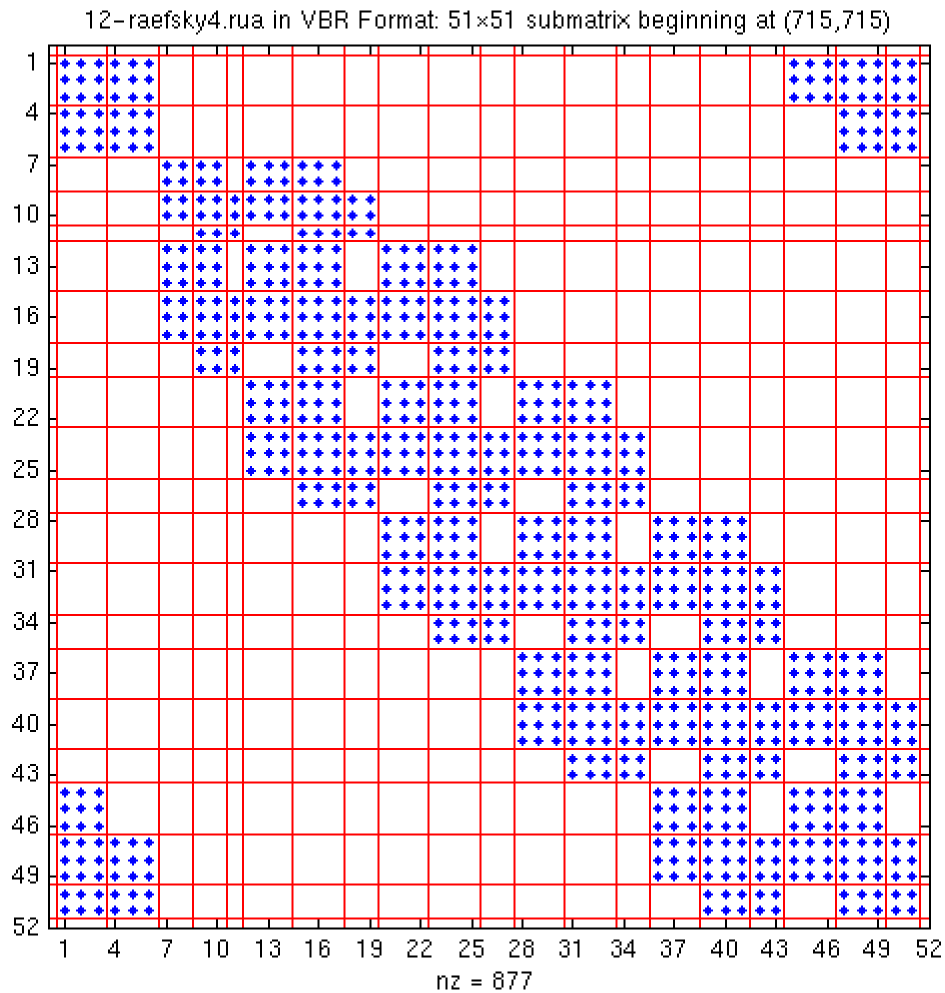
Calling OSKI: Interface Design

- 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

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)**

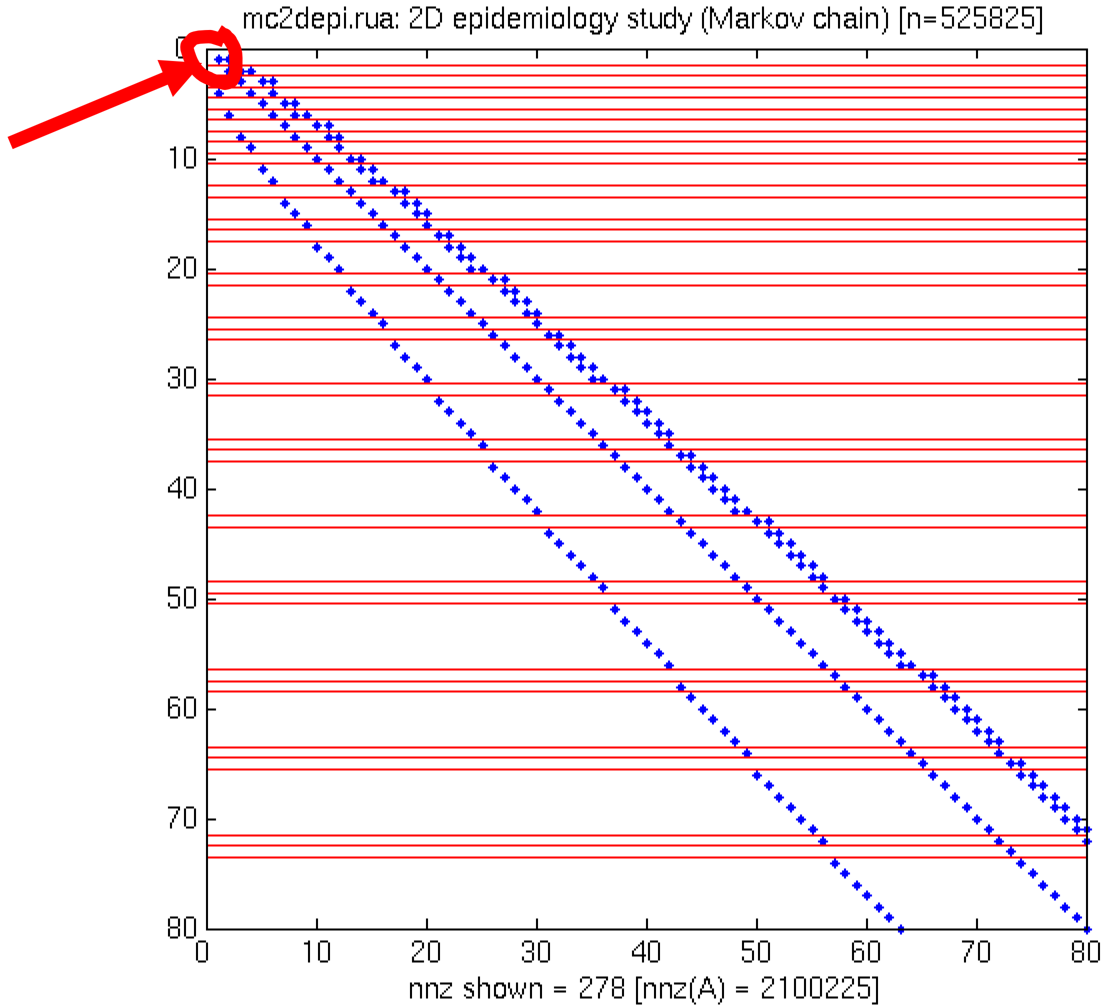
Example: Variable Block Structure



2.1x over CSR

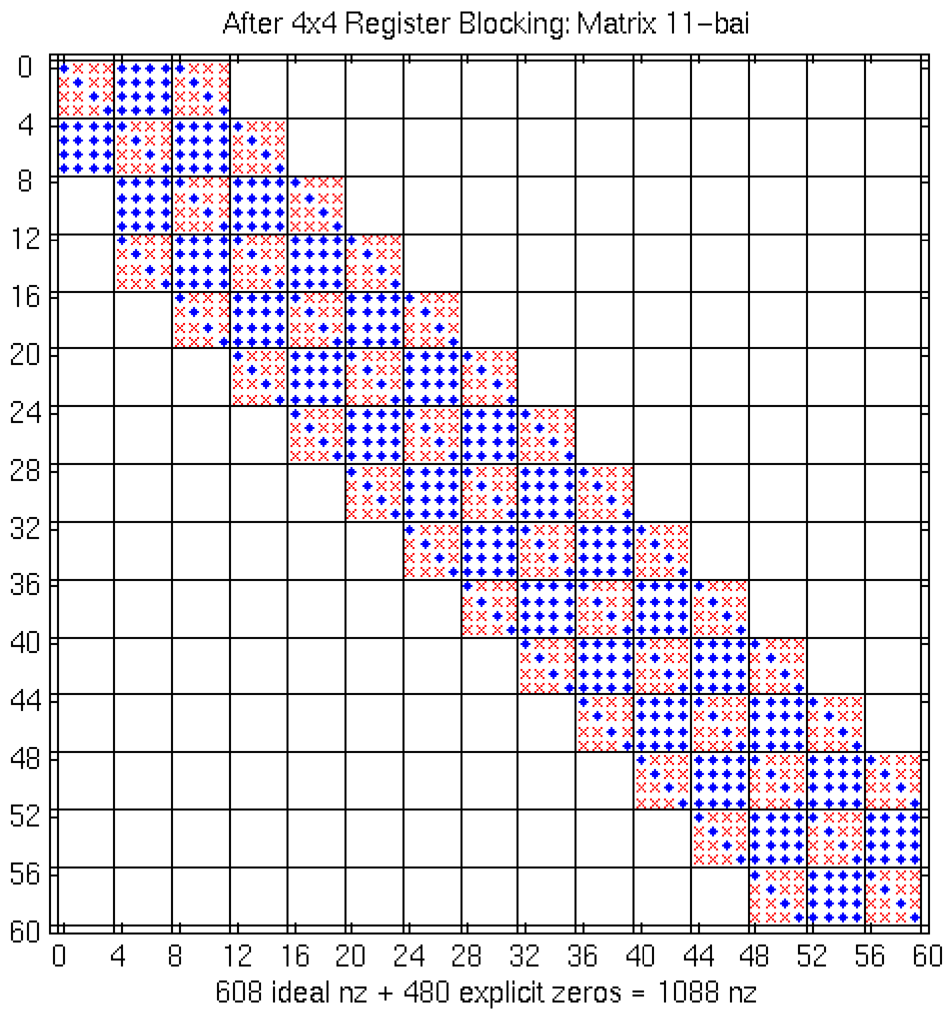
1.8x over RB

Example: Row-Segmented Diagonals

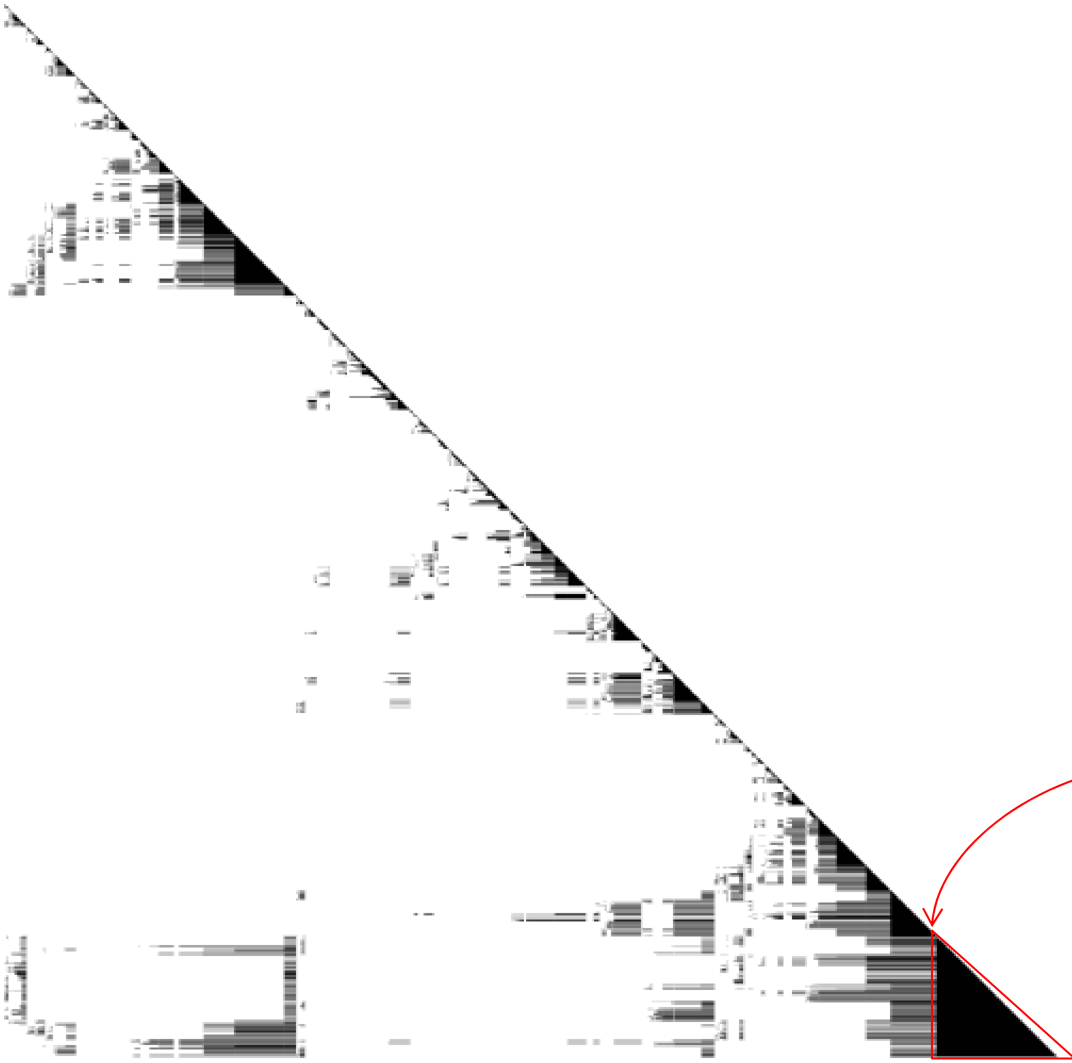


2x over CSR

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

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