CS267 – Lecture 15

Automatic Performance Tuning and Sparse-Matrix-Vector-Multiplication (SpMV)

James Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr16

Outline

- Motivation for Automatic Performance Tuning
- Results for sparse matrix kernels
- OSKI = Optimized Sparse Kernel Interface
  - pOSKI for multicore
- Tuning Higher Level Algorithms
- Future Work, Class Projects

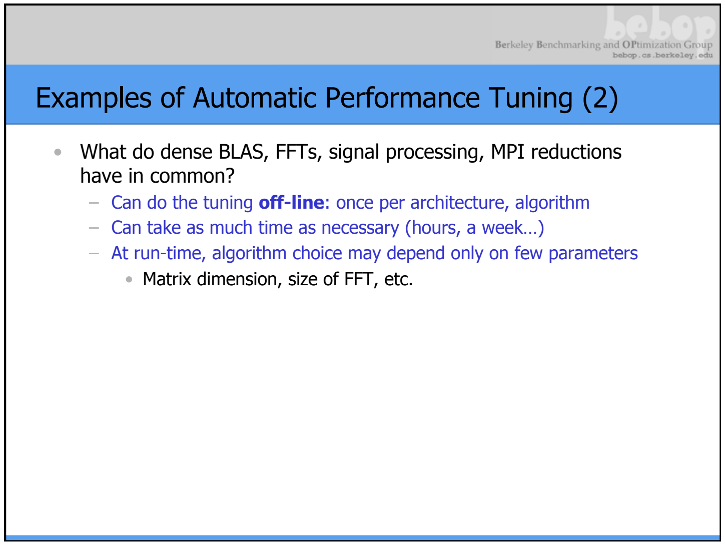
- BeBOP: Berkeley Benchmarking and Optimization Group
  - Many results shown from current and former members
  - Meet weekly Th 12:30-2, in 380 Soda

Motivation for Automatic Performance Tuning

- Writing high performance software is hard
  - Make programming easier while getting high speed
- Ideal: program in your favorite high level language (Matlab, Python, …) and get a high fraction of peak performance
- Reality: Best algorithm (and its implementation) can depend strongly on the problem, computer architecture, compiler,…
  - Best choice can depend on knowing a lot of applied mathematics and computer science
- How much of this can we teach?
- How much of this can we automate?

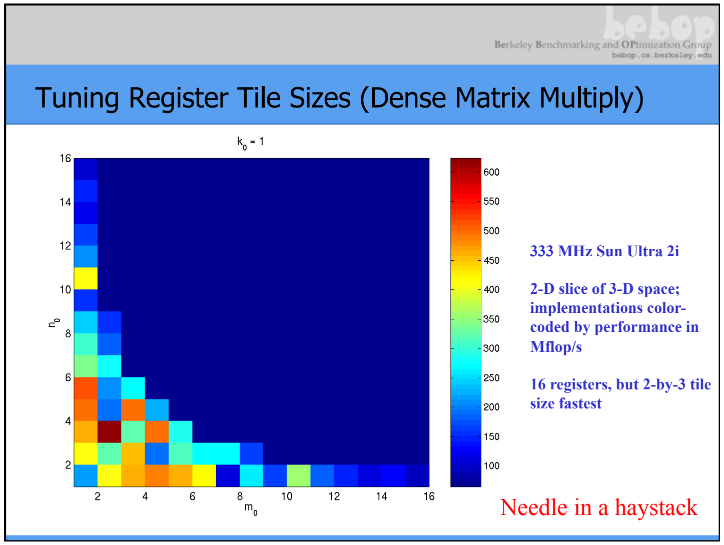
Examples of Automatic Performance Tuning (1)

- Dense BLAS
  - Sequential
  - PHiPAC (UCB), then ATLAS (UTK) (used in Matlab)
  - math-atlas.sourceforge.net/
  - Internal vendor tools
- Fast Fourier Transform (FFT) & variations
  - Sequential and Parallel
  - FFTW (MIT)
  - www.fftw.org
- Digital Signal Processing
  - SPIRAL: www.spiral.net (CMU)
- Communication Collectives (UCB, UTK)
- Rose (LLNL), Bernoulli (Cornell), Telescoping Languages (Rice), …
- More projects, conferences, government reports, …



Examples of Automatic Performance Tuning (2)

- What do dense BLAS, FFTs, signal processing, MPI reductions have in common?
  - Can do the tuning **off-line**: once per architecture, algorithm
  - Can take as much time as necessary (hours, a week…)
  - At run-time, algorithm choice may depend only on few parameters
    - Matrix dimension, size of FFT, etc.



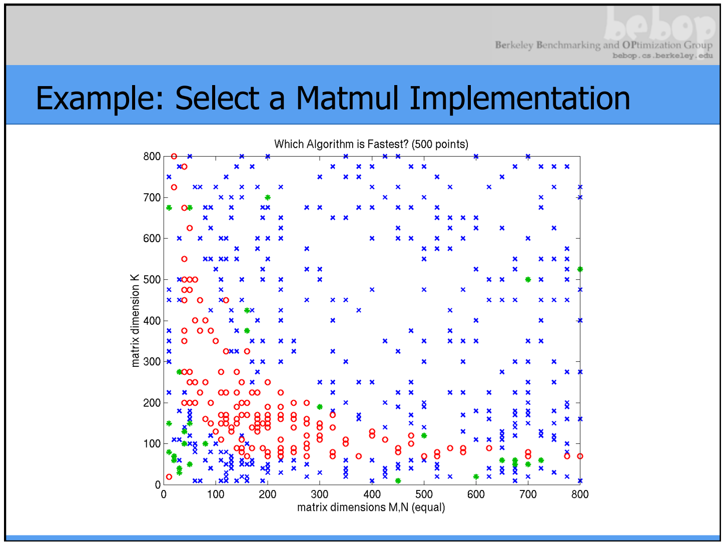
Tuning Register Tile Sizes (Dense Matrix Multiply)

333 MHz Sun Ultra 2i

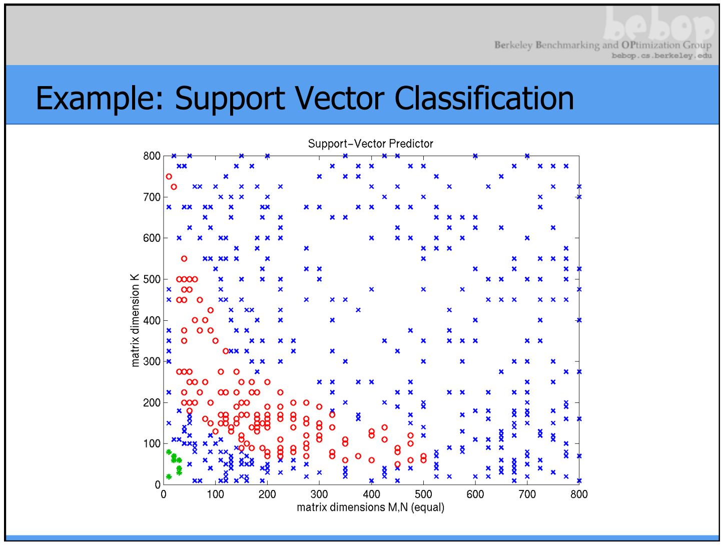
2-D slice of 3-D space; implementations color-coded by performance in Mflop/s

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

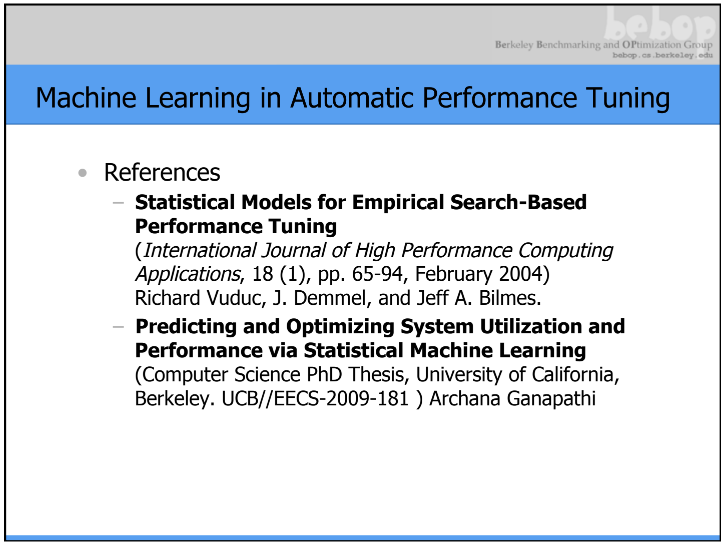
Needle in a haystack



Example: Select a Matmul Implementation

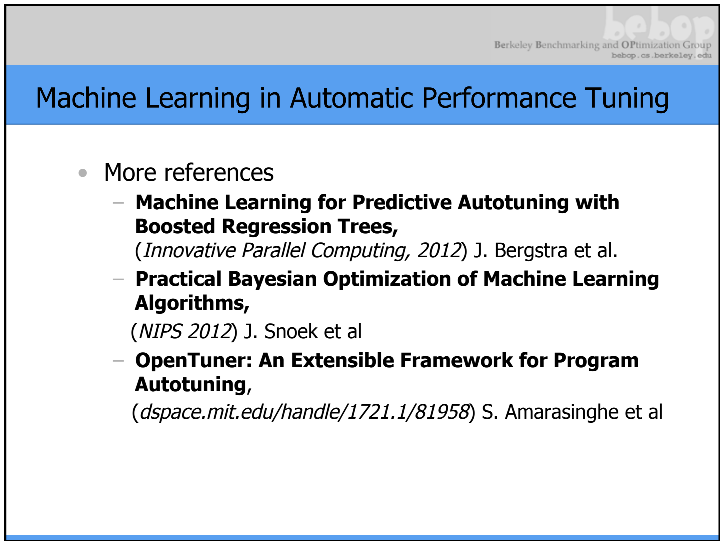


Example: Support Vector Classification



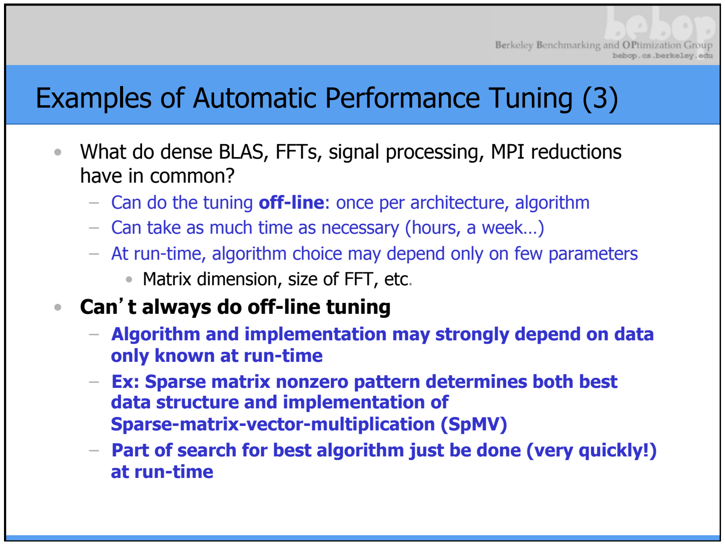
Machine Learning in Automatic Performance Tuning

- References
  - **Statistical Models for Empirical Search-Based Performance Tuning**
    (*International Journal of High Performance Computing Applications*, 18 (1), pp. 65-94, February 2004)
    Richard Vuduc, J. Demmel, and Jeff A. Bilmes.
  - **Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning**
    (Computer Science PhD Thesis, University of California, Berkeley. UCB//EECS-2009-181) Archana Ganapathi



Machine Learning in Automatic Performance Tuning

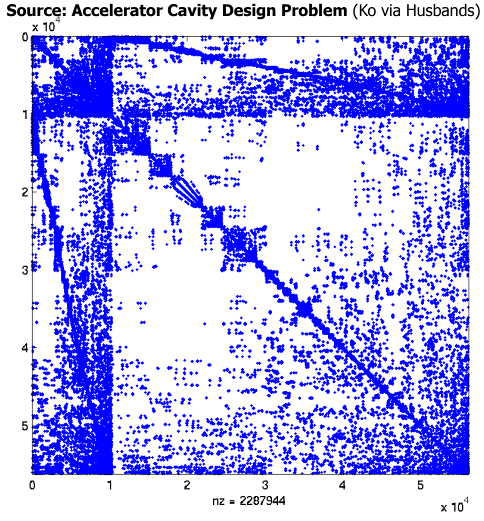
- More references
  - **Machine Learning for Predictive Autotuning with Boosted Regression Trees,**
    (*Innovative Parallel Computing, 2012*) J. Bergstra et al.
  - **Practical Bayesian Optimization of Machine Learning Algorithms,**
    (*NIPS 2012*) J. Snoek et al
  - **OpenTuner: An Extensible Framework for Program Autotuning,**
    (*dspace.mit.edu/handle/1721.1/81958*) S. Amarasinghe et al

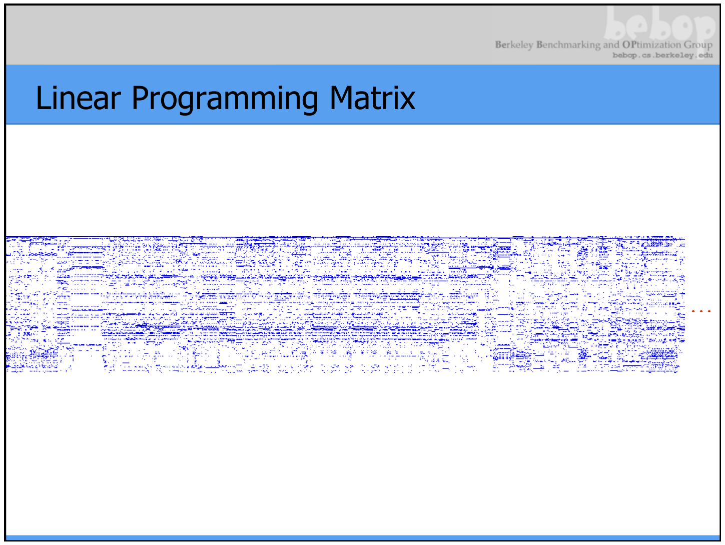


Examples of Automatic Performance Tuning (3)

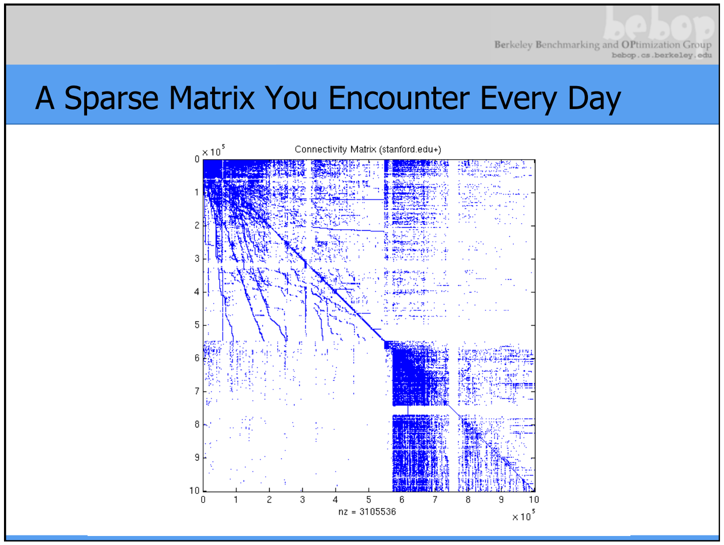
- What do dense BLAS, FFTs, signal processing, MPI reductions have in common?
  - Can do the tuning **off-line**: once per architecture, algorithm
  - Can take as much time as necessary (hours, a week…)
  - At run-time, algorithm choice may depend only on few parameters
    - Matrix dimension, size of FFT, etc.
- **Can't always do off-line tuning**
  - **Algorithm and implementation may strongly depend on data only known at run-time**
  - **Ex: Sparse matrix nonzero pattern determines both best data structure and implementation of Sparse-matrix-vector-multiplication (SpMV)**
  - **Part of search for best algorithm just be done (very quickly!) at run-time**

Source: Accelerator Cavity Design Problem (Ko via Husbands)

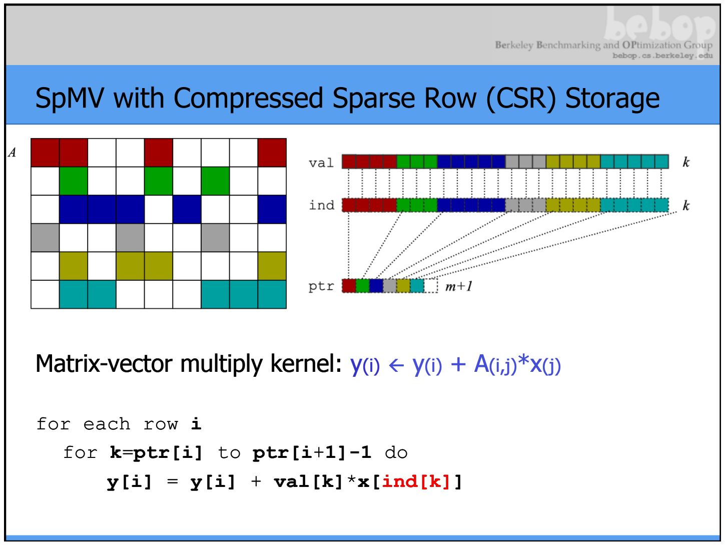




Linear Programming Matrix



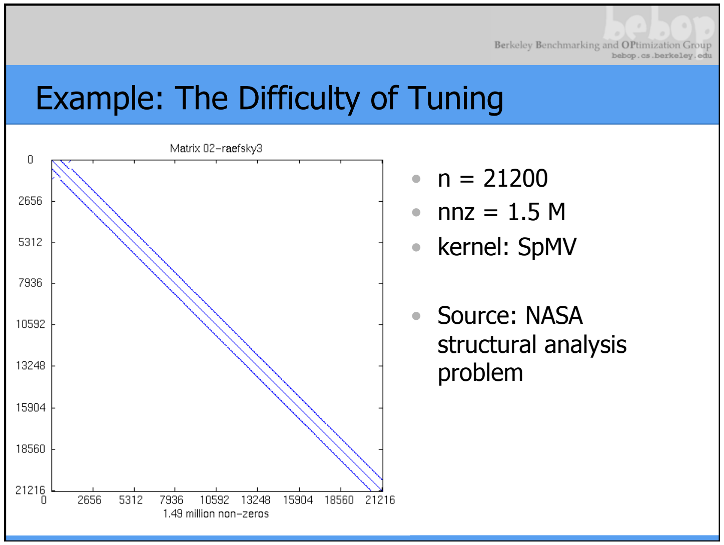
A Sparse Matrix You Encounter Every Day



SpMV with Compressed Sparse Row (CSR) Storage

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

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

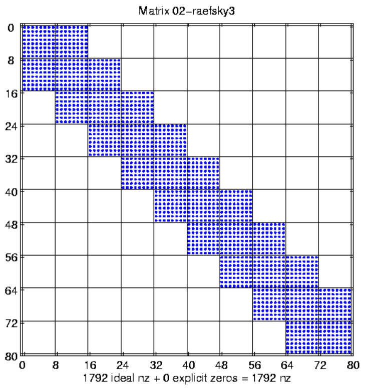


Example: The Difficulty of Tuning

- n = 21200
- nnz = 1.5 M
- kernel: SpMV

- Source: NASA structural analysis problem

Example: The Difficulty of Tuning

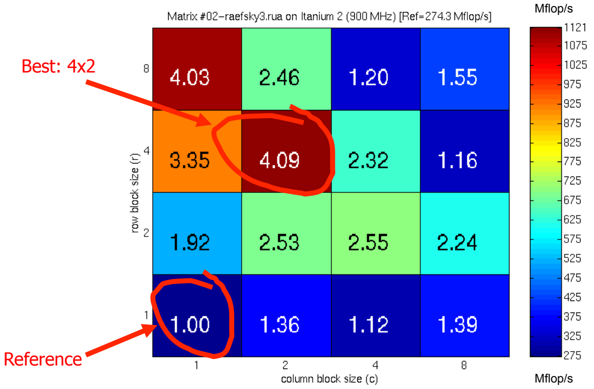


- n = 21200
- nnz = 1.5 M
- kernel: SpMV
- Source: NASA structural analysis problem
- **8x8** dense substructure

Taking advantage of block structure in SpMV

- Bottleneck is time to get matrix from memory
  - Only 2 flops for each nonzero in matrix
- Don' t store each nonzero with index, instead store each nonzero r-by-c block with index
  - Storage drops by up to 2x, if rc >> 1, all 32-bit quantities
  - Time to fetch matrix from memory decreases
- Change both data structure and algorithm
  - Need to pick r and c
  - Need to change algorithm accordingly
- In example, is r=c=8 best choice?
  - Minimizes storage, so looks like a good idea…

Speedups on Itanium 2: The Need for Search

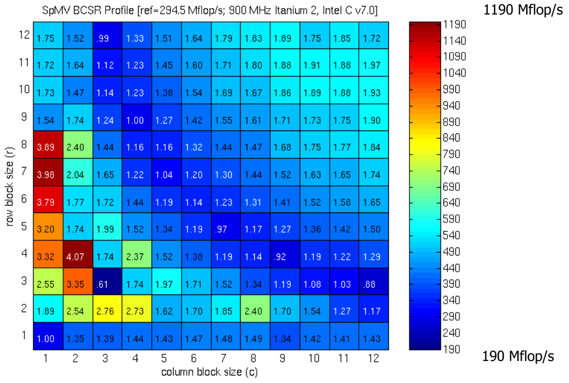


Best: 4x2

Reference

row block size (r) \ column block size (c)	1	2	4	8
8	4.03	2.46	1.20	1.55
4	3.35	4.09	2.32	1.16
2	1.92	2.53	2.55	2.24
1	1.00	1.36	1.12	1.39

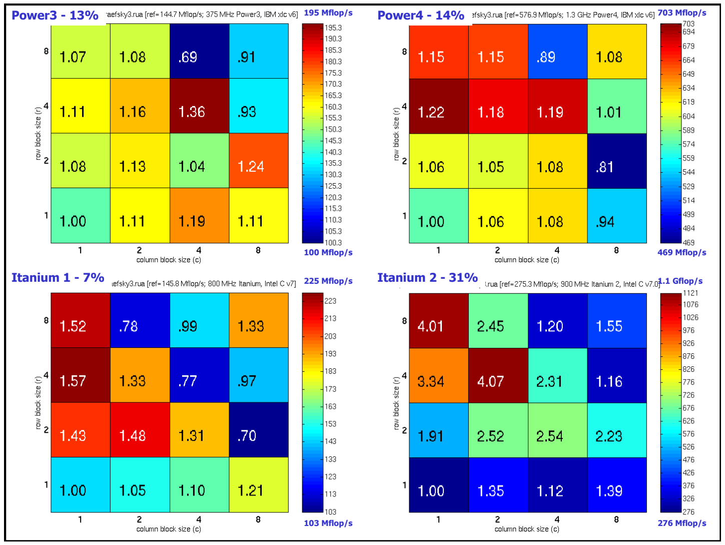
Register Profile: Itanium 2

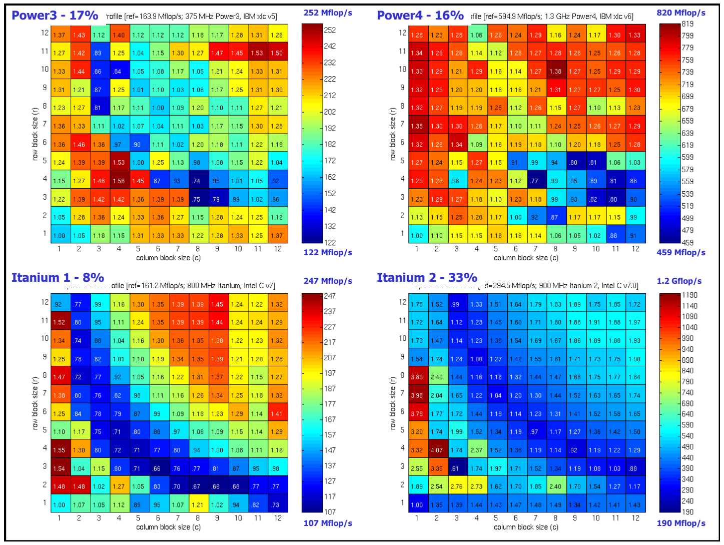


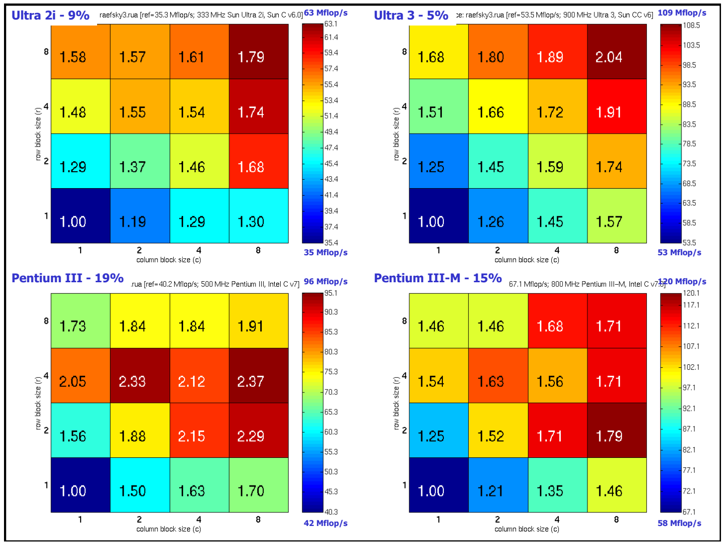
1190 Mflop/s

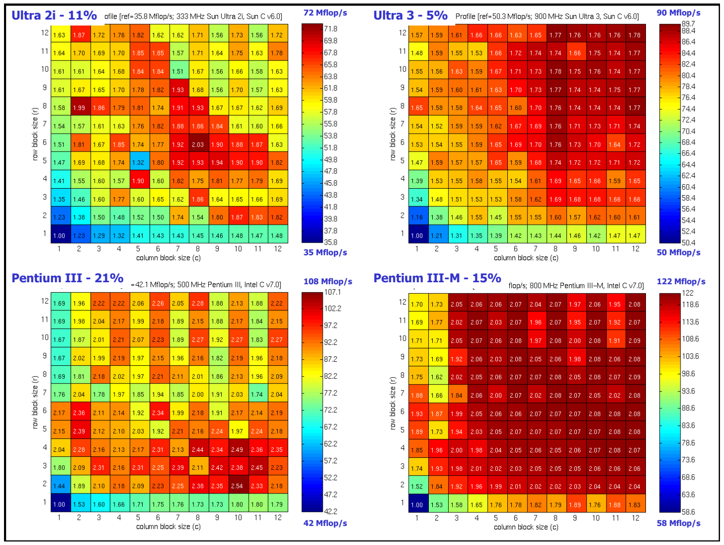
190 Mflop/s

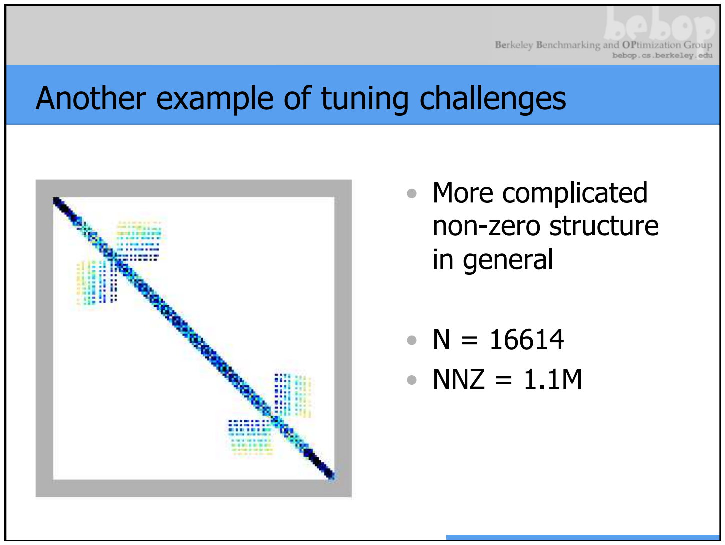
row block size (r) \ column block size (c)	1	2	3	4	5	6	7	8	9	10	11	12
12	1.75	1.52	.99	1.33	1.51	1.64	1.79	1.83	1.89	1.75	1.85	1.72
11	1.72	1.64	1.12	1.23	1.45	1.60	1.71	1.80	1.88	1.91	1.88	1.97
10	1.73	1.47	1.14	1.23	1.38	1.54	1.69	1.67	1.86	1.89	1.88	1.93
9	1.54	1.74	1.24	1.00	1.27	1.42	1.55	1.61	1.71	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


Power3 - 13%
raefsky3.rua [ref=144.7 Mflop/s; 375 MHz Power3, IBM xlc v6]
195 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
100 Mflop/s
Power4 - 14%
efsky3.rua [ref=576.9 Mflop/s; 1.3 GHz Power4, IBM xlc v6]
703 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
469 Mflop/s
Itanium 1 - 7%
aefsky3.rua [ref=145.8 Mflop/s; 800 MHz Itanium, Intel C v7]
225 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
103 Mflop/s
Itanium 2 - 31%
.rua [ref=275.3 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]
1.1 Gflop/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
276 Mflop/s
row block size (r)
column block size (c)


Power3 - 17%
rofile [ref=163.9 Mflop/s; 375 MHz Power3, IBM xlc v5]
252 Mflop/s
122 Mflop/s
Power4 - 16%
file [ref=594.9 Mflop/s; 1.3 GHz Power4, IBM xlc v6]
820 Mflop/s
459 Mflop/s
Itanium 1 - 8%
ofile [ref=161.2 Mflop/s; 800 MHz Itanium, Intel C v7]
247 Mflop/s
107 Mflop/s
Itanium 2 - 33%
ef=294.5 Mflop/s; 900 MHz Itanium 2, Intel C v7.0]
1.2 Gflop/s
190 Mflop/s
row block size (r)
column block size (c)

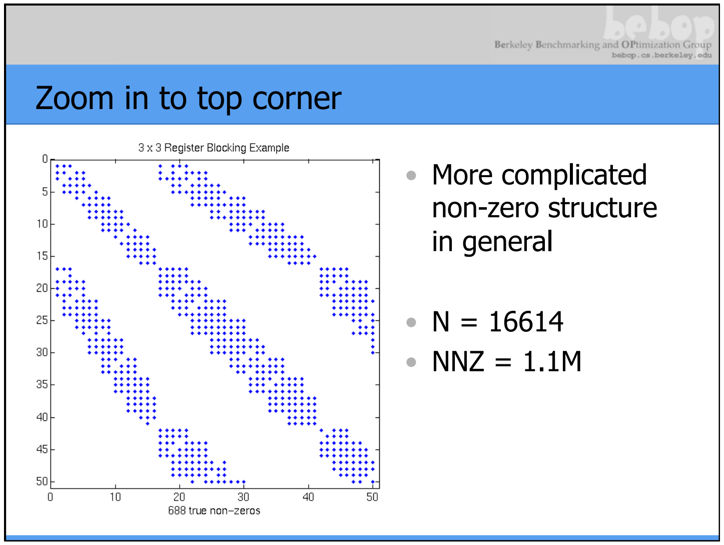

Ultra 2i - 9%
raefsky3.rua [ref=35.3 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]
63 Mflop/s
1.58 1.57 1.61 1.79
1.48 1.55 1.54 1.74
1.29 1.37 1.46 1.68
1.00 1.19 1.29 1.30
35 Mflop/s
Ultra 3 - 5%
ce: raefsky3.rua [ref=53.5 Mflop/s; 900 MHz Ultra 3, Sun CC v6]
109 Mflop/s
1.68 1.80 1.89 2.04
1.51 1.66 1.72 1.91
1.25 1.45 1.59 1.74
1.00 1.26 1.45 1.57
53 Mflop/s
Pentium III - 19%
.rua [ref=40.2 Mflop/s; 500 MHz Pentium III, Intel C v7]
96 Mflop/s
1.73 1.84 1.84 1.91
2.05 2.33 2.12 2.37
1.56 1.88 2.15 2.29
1.00 1.50 1.63 1.70
42 Mflop/s
Pentium III-M - 15%
67.1 Mflop/s; 800 MHz Pentium III–M, Intel C v7.0]
120 Mflop/s
1.46 1.46 1.68 1.71
1.54 1.63 1.56 1.71
1.25 1.52 1.71 1.79
1.00 1.21 1.35 1.46
58 Mflop/s
row block size (r)
column block size (c)


Ultra 2i - 11%
ofile [ref=35.8 Mflop/s; 333 MHz Sun Ultra 2i, Sun C v6.0]
72 Mflop/s
35 Mflop/s
Ultra 3 - 5%
Profile [ref=50.3 Mflop/s; 900 MHz Sun Ultra 3, Sun C v6.0]
90 Mflop/s
50 Mflop/s
Pentium III - 21%
=42.1 Mflop/s; 500 MHz Pentium III, Intel C v7.0]
108 Mflop/s
42 Mflop/s
Pentium III-M - 15%
flop/s; 800 MHz Pentium III–M, Intel C v7.0]
122 Mflop/s
58 Mflop/s
row block size (r)
column block size (c)



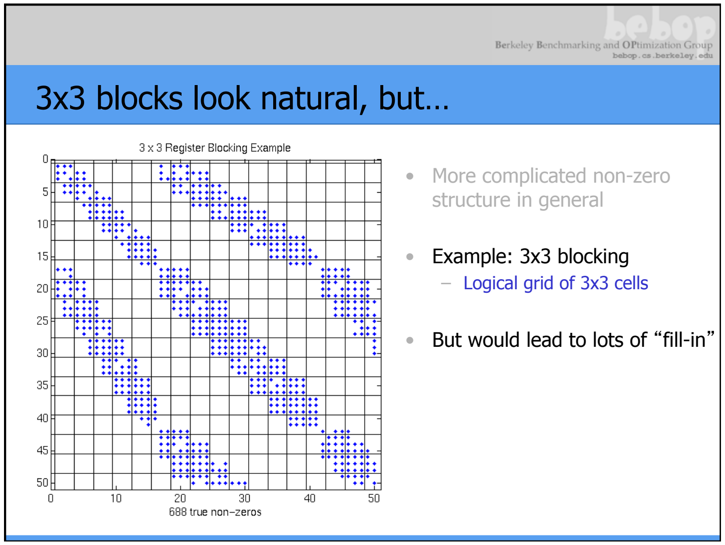
Another example of tuning challenges

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



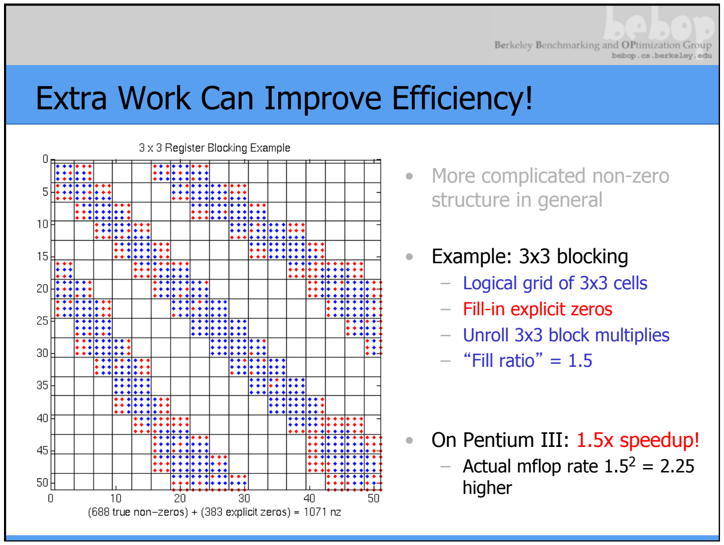
Zoom in to top corner

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



3x3 blocks look natural, but…

- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
- But would lead to lots of "fill-in"



Extra Work Can Improve Efficiency!

3 x 3 Register Blocking Example
0
5
10
15
20
25
30
35
40
45
50
0
10
20
30
40
50
(688 true non−zeros) + (383 explicit zeros) = 1071 nz

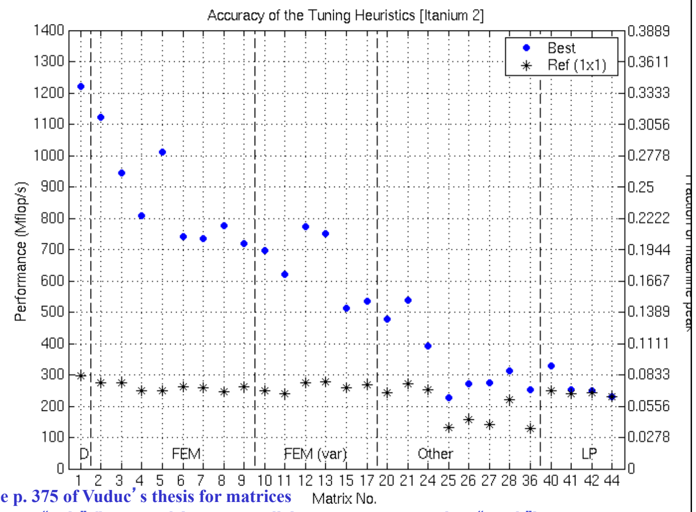
- More complicated non-zero structure in general
- Example: 3x3 blocking
  - Logical grid of 3x3 cells
  - Fill-in explicit zeros
  - Unroll 3x3 block multiplies
  - "Fill ratio" = 1.5
- On Pentium III: 1.5x speedup!
  - Actual mflop rate 1.5^2 = 2.25 higher

Automatic Register Block Size Selection

- Selecting the r x c block size
  - **Off-line benchmark**
    - Precompute **Mflops(r,c)** using dense A for each r x c
    - Once per machine/architecture
  - **Run-time "search"**
    - Sample *A* to estimate **Fill(r,c)** for each r x c
  - **Run-time heuristic model**
    - Choose r, c to minimize **time ~ Fill(r,c) / Mflops(r,c)**

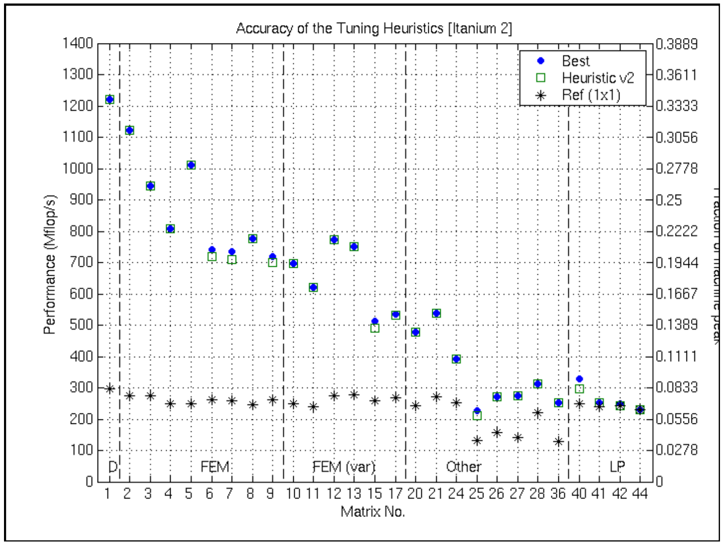
Accurate and Efficient Adaptive Fill Estimation

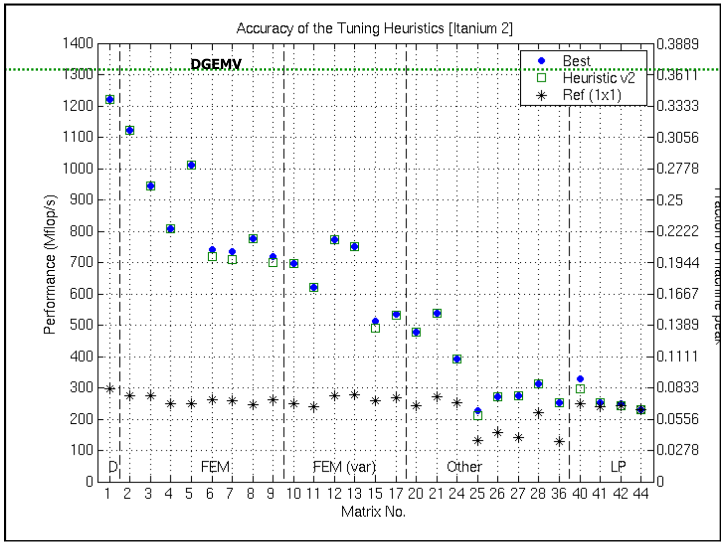
- Idea: Sample matrix
  - Fraction of matrix to sample: *s* ∈ [0,1]
  - Cost ~ O(*s* * nnz)
  - Control cost by controlling *s*
    - Search at run-time: the constant matters!
- Control *s* automatically by computing statistical confidence intervals
  - Idea: Monitor variance
- Cost of tuning
  - Lower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

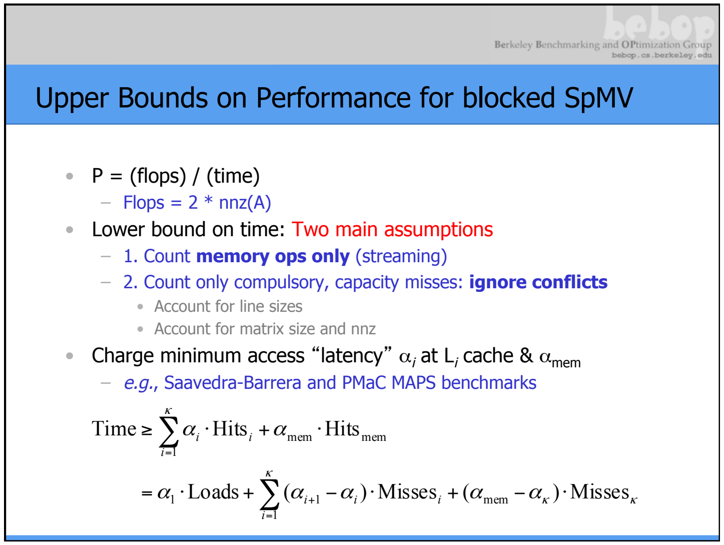


See p. 375 of Vuduc's thesis for matrices

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





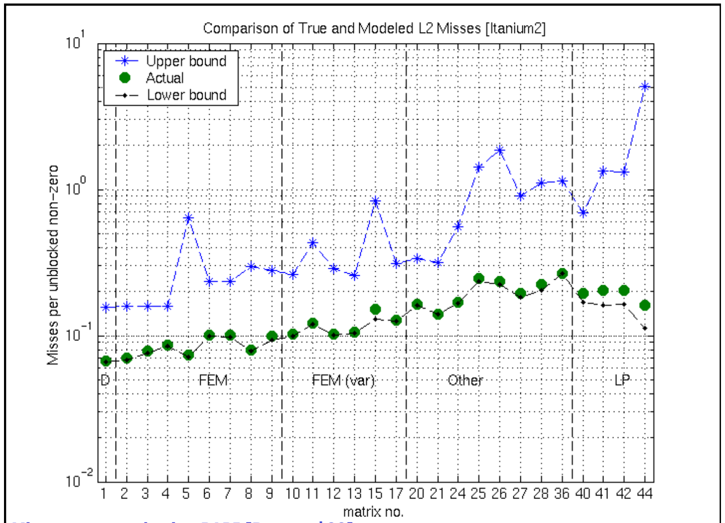

bebop
Berkeley Benchmarking and OPtimization Group
bebop.cs.berkeley.edu

Upper Bounds on Performance for blocked SpMV

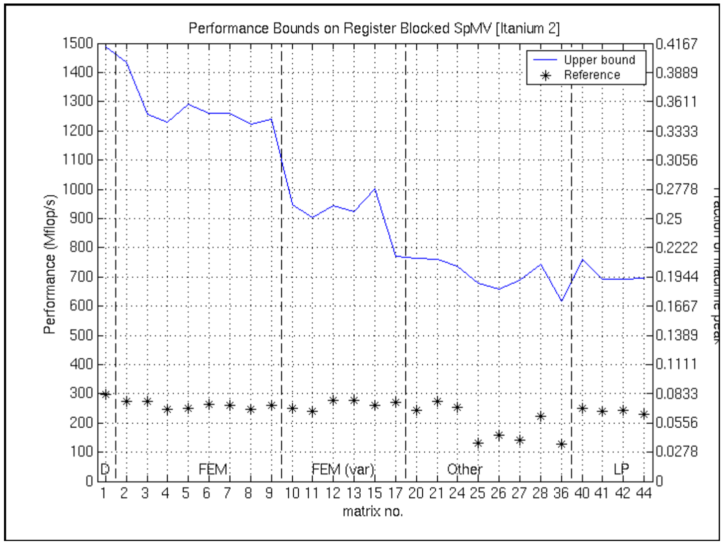
- P = (flops) / (time)
  - Flops = 2 * nnz(A)
- Lower bound on time: Two main assumptions
  - 1. Count **memory ops only** (streaming)
  - 2. Count only compulsory, capacity misses: **ignore conflicts**
    - Account for line sizes
    - Account for matrix size and nnz
- Charge minimum access "latency"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

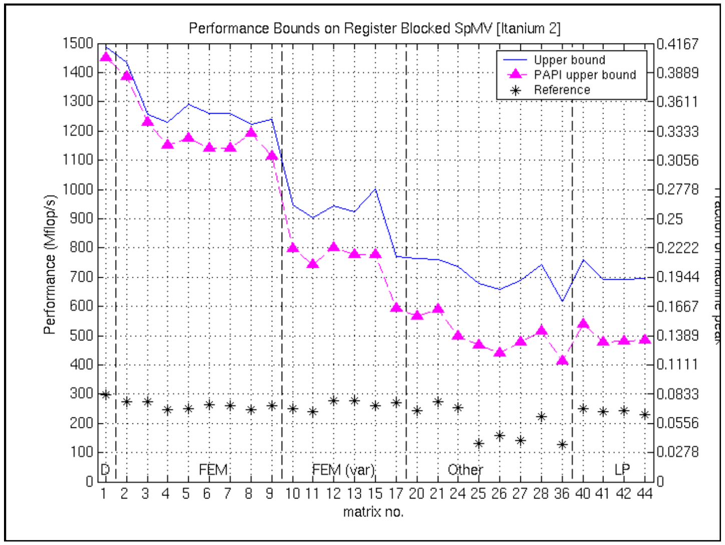
$$\text{Time} \geq \sum_{i=1}^{\kappa} \alpha_i \cdot \text{Hits}_i + \alpha_{\text{mem}} \cdot \text{Hits}_{\text{mem}}$$

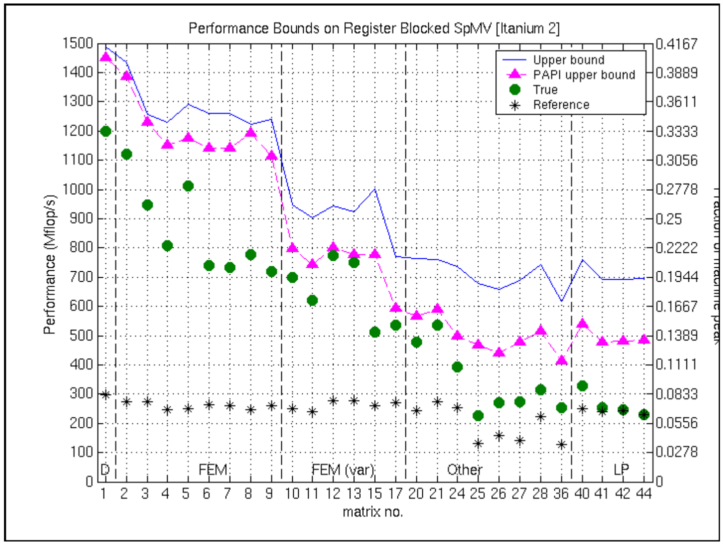
$$= \alpha_1 \cdot \text{Loads} + \sum_{i=1}^{\kappa} (\alpha_{i+1} - \alpha_i) \cdot \text{Misses}_i + (\alpha_{\text{mem}} - \alpha_\kappa) \cdot \text{Misses}_\kappa$$

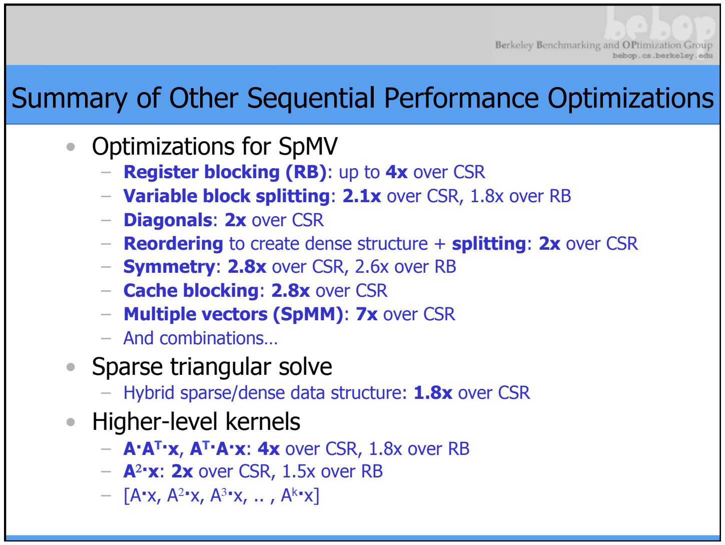


Misses measured using PAPI [Browne '00]



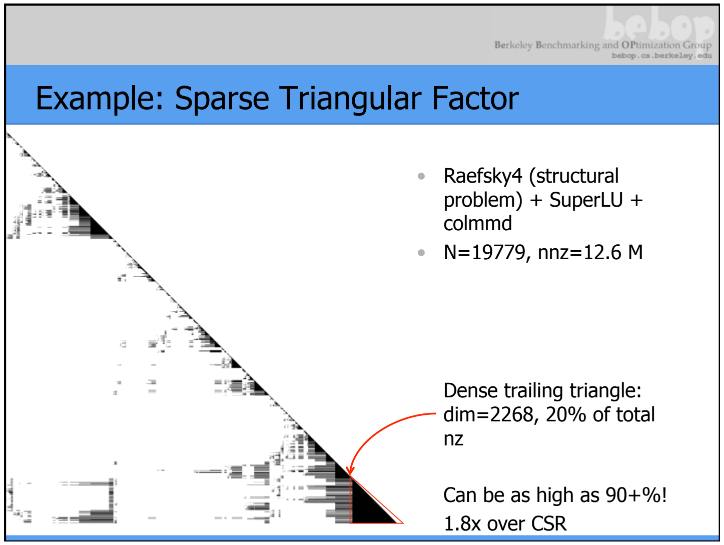






Summary of Other Sequential Performance Optimizations

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

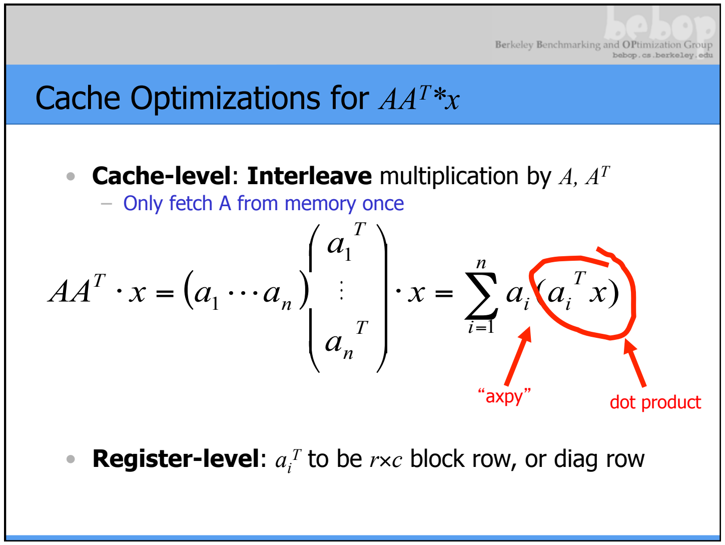


Example: Sparse Triangular Factor

- 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

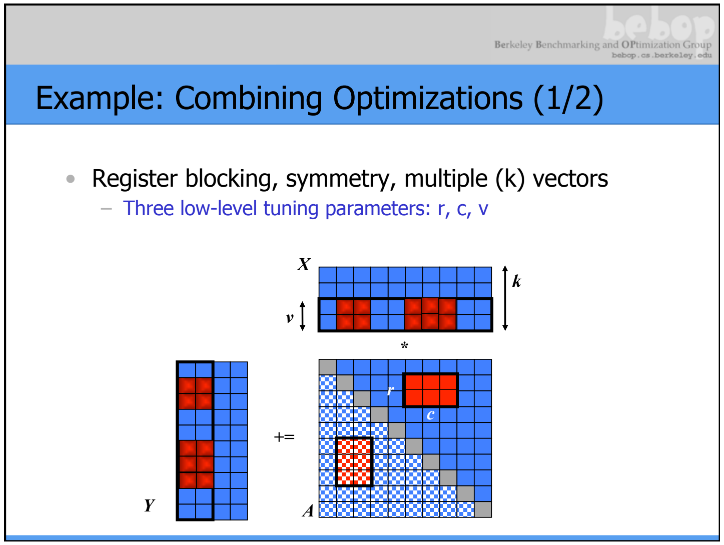
- **Cache-level**: **Interleave** multiplication by A, A^T
  - Only fetch A from memory once

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

"axpy"

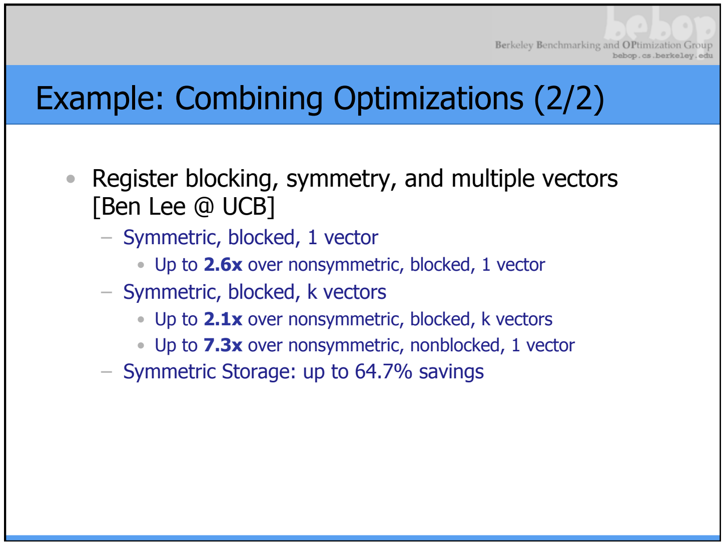
dot product

- **Register-level**: a_i^T to be $r\times c$ block row, or diag row



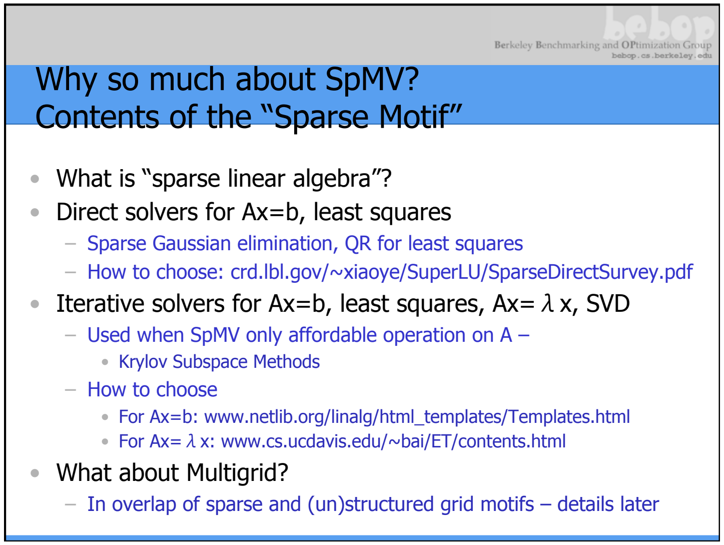
Example: Combining Optimizations (1/2)

- Register blocking, symmetry, multiple (k) vectors
  - Three low-level tuning parameters: r, c, v



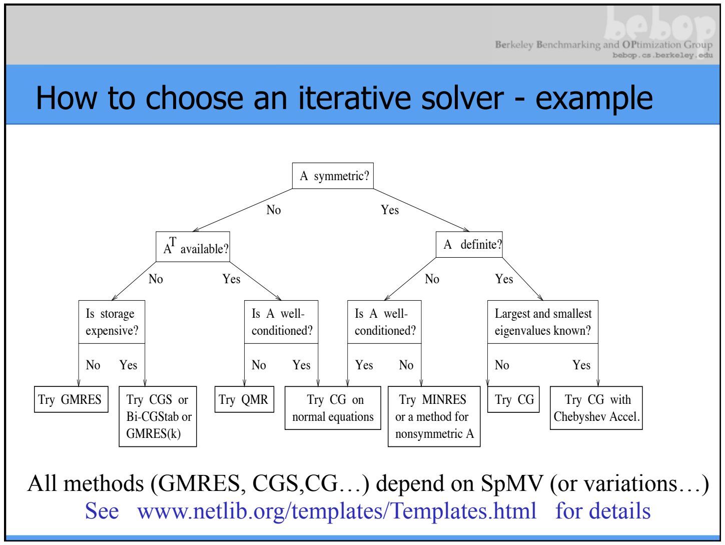
Example: Combining Optimizations (2/2)

- Register blocking, symmetry, and multiple vectors [Ben Lee @ UCB]
  - Symmetric, blocked, 1 vector
    - Up to **2.6x** over nonsymmetric, blocked, 1 vector
  - Symmetric, blocked, k vectors
    - Up to **2.1x** over nonsymmetric, blocked, k vectors
    - Up to **7.3x** over nonsymmetric, nonblocked, 1 vector
  - Symmetric Storage: up to 64.7% savings



Why so much about SpMV? Contents of the "Sparse Motif"

- What is "sparse linear algebra"?
- Direct solvers for Ax=b, least squares
  - Sparse Gaussian elimination, QR for least squares
  - How to choose: crd.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf
- Iterative solvers for Ax=b, least squares, Ax= λ x, SVD
  - Used when SpMV only affordable operation on A –
    - Krylov Subspace Methods
  - How to choose
    - For Ax=b: www.netlib.org/linalg/html_templates/Templates.html
    - For Ax= λ x: www.cs.ucdavis.edu/~bai/ET/contents.html
- What about Multigrid?
  - In overlap of sparse and (un)structured grid motifs – details later

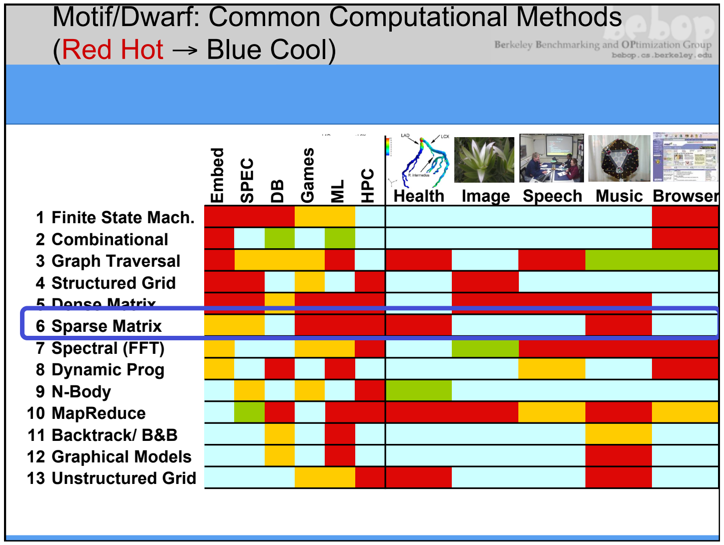


How to choose an iterative solver - example

A symmetric?
No
Yes
A^T available?
No
Yes
A definite?
No
Yes
Is storage expensive?
No
Yes
Is A well-conditioned?
No
Yes
Is A well-conditioned?
Yes
No
Largest and smallest eigenvalues known?
No
Yes
Try GMRES
Try CGS or Bi-CGStab or GMRES(k)
Try QMR
Try CG on normal equations
Try MINRES or a method for nonsymmetric A
Try CG
Try CG with Chebyshev Accel.

All methods (GMRES, CGS,CG…) depend on SpMV (or variations…)

See www.netlib.org/templates/Templates.html for details

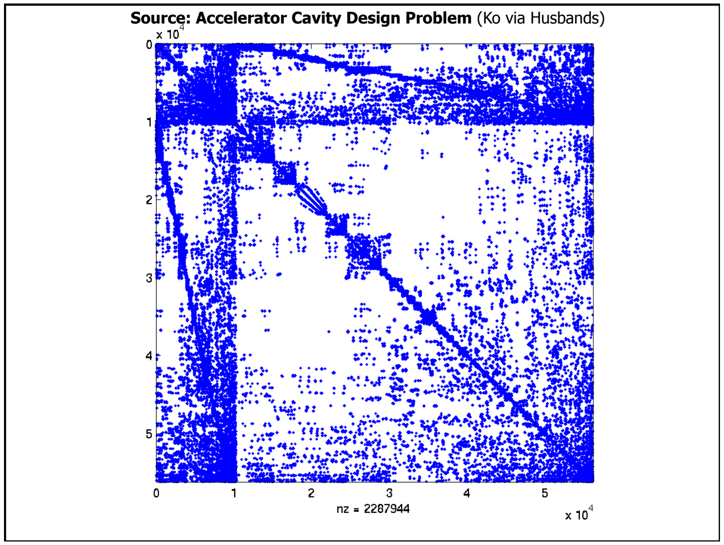


Motif/Dwarf: Common Computational Methods (Red Hot → Blue Cool)

	Embed	SPEC	DB	Games	ML	HPC	Health	Image	Speech	Music	Browser
1 Finite State Mach.	🟥	🟥	🟥	🟨	🟨	🟦	🟦	🟦	🟦	🟦	🟥
2 Combinational	🟥	🟦	🟩	🟦	🟩	🟦	🟦	🟦	🟦	🟦	🟥
3 Graph Traversal	🟥	🟨	🟨	🟨	🟥	🟦	🟥	🟦	🟥	🟩	🟩
4 Structured Grid	🟥	🟥	🟦	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦
5 Dense Matrix	🟥	🟥	🟨	🟥	🟥	🟥	🟦	🟥	🟥	🟥	🟦
6 Sparse Matrix	🟨	🟨	🟦	🟥	🟥	🟥	🟥	🟦	🟦	🟥	🟦
7 Spectral (FFT)	🟨	🟦	🟦	🟨	🟨	🟥	🟦	🟩	🟥	🟥	🟥
8 Dynamic Prog	🟨	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟨	🟦	🟥
9 N-Body	🟦	🟨	🟦	🟨	🟦	🟥	🟩	🟦	🟦	🟦	🟦
10 MapReduce	🟦	🟩	🟥	🟦	🟥	🟥	🟥	🟥	🟨	🟥	🟨
11 Backtrack/ B&B	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟨	🟦
12 Graphical Models	🟦	🟦	🟨	🟦	🟥	🟦	🟦	🟦	🟦	🟥	🟦
13 Unstructured Grid	🟦	🟦	🟦	🟨	🟨	🟥	🟥	🟦	🟦	🟥	🟦

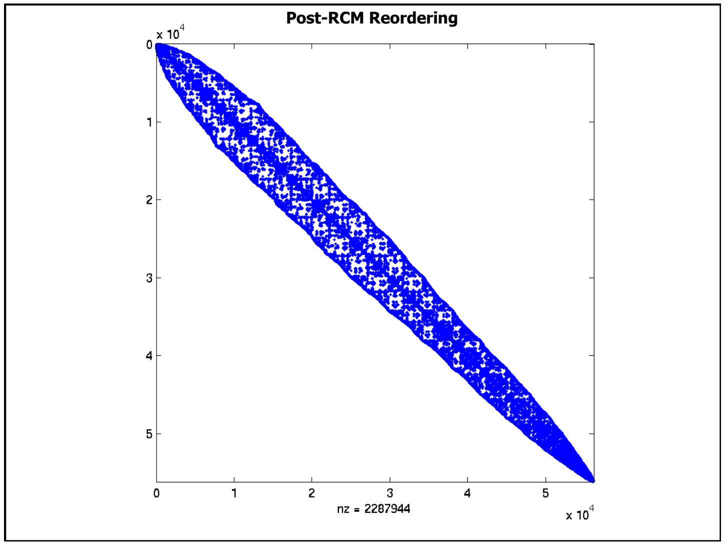
Potential Impact on Applications: Omega3P

- Application: accelerator cavity design [Ko]
- Relevant optimization techniques
  - **Symmetric storage**
  - **Register blocking**
  - **Reordering, to create more dense blocks**
    - **Reverse Cuthill-McKee ordering to reduce bandwidth**
      - **Do Breadth-First-Search, number nodes in reverse order visited**
    - **Traveling Salesman Problem-based ordering to create blocks**
      - **Nodes = columns of A**
      - **Weights(u, v) = no. of nonzeros u, v have in common**
      - **Tour = ordering of columns**
      - **Choose maximum weight tour**
      - **See [Pinar & Heath '97]**
- 2.1x speedup on Power 4

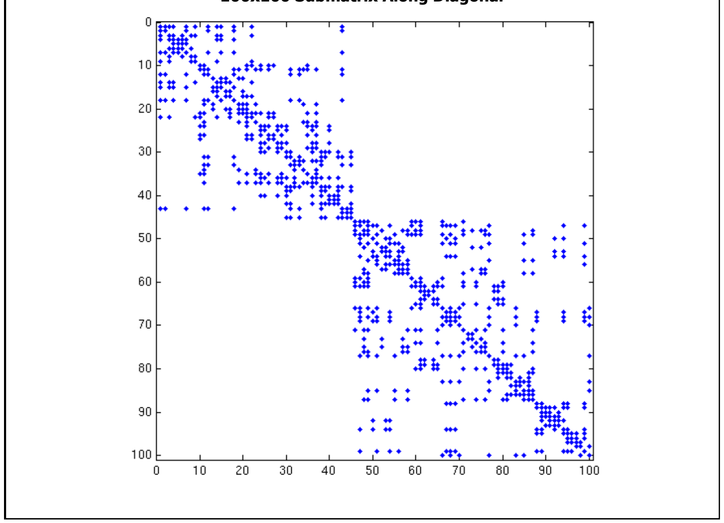


Source: Accelerator Cavity Design Problem (Ko via Husbands)

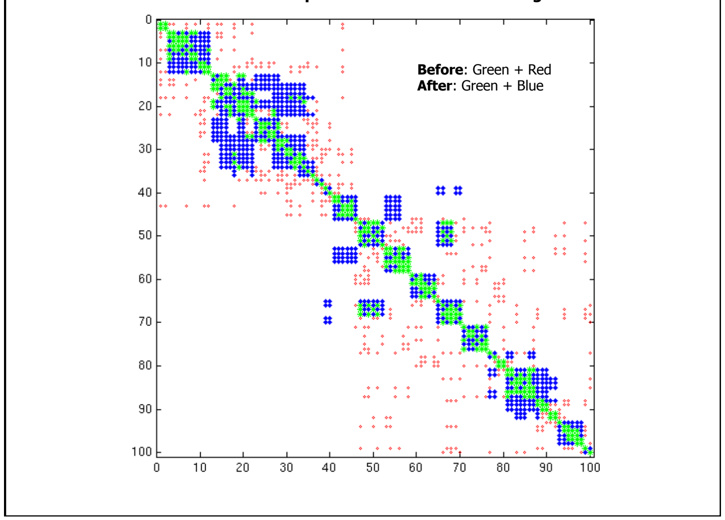
Post-RCM Reordering



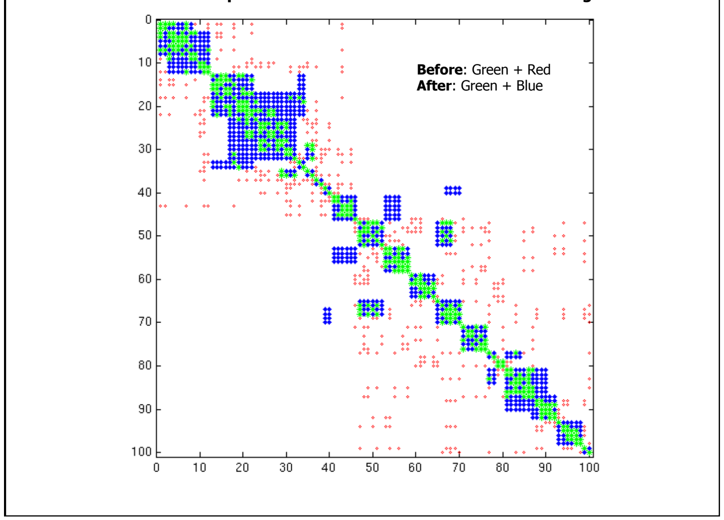
100x100 Submatrix Along Diagonal

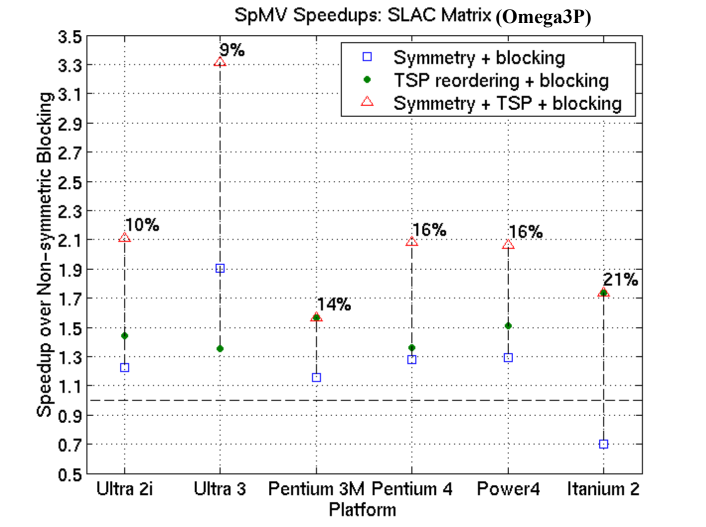


“Microscopic” Effect of RCM Reordering



“Microscopic” Effect of Combined RCM+TSP Reordering





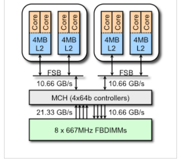
How do permutations affect algorithms?

- A = original matrix, A^P = A with permuted rows, columns
- Naïve approach: permute x, multiply $y=A^Px$, permute y
- Faster way to solve Ax=b
  - Write $A^P = P^TAP$ where P is a permutation matrix
  - Solve $A^Px^P = P^Tb$ for x^P, using SpMV with A^P, then let $x = Px^P$
  - Only need to permute vectors twice, not twice per iteration
- Faster way to solve $Ax= \lambda x$
  - A and A^P have same eigenvalues, no vectors to permute!
  - $A^Px^P = \lambda x^P$ implies $Ax = \lambda x$ where $x = Px^P$
- Where else do optimizations change higher level algorithms? More later…

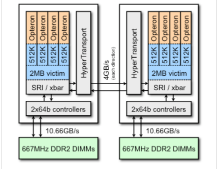
Tuning SpMV on Multicore

Multicore SMPs Used

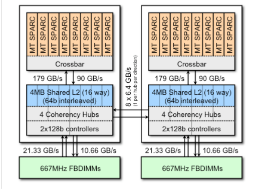
Intel Xeon E5345 (Clovertown)



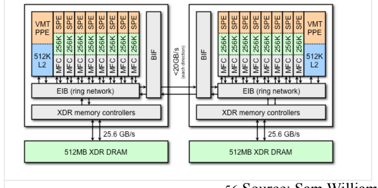
AMD Opteron 2356 (Barcelona)



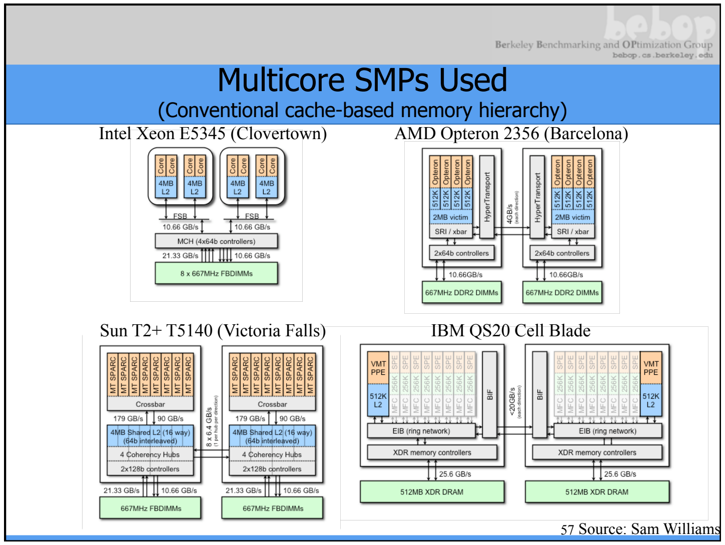
Sun T2+ T5140 (Victoria Falls)



IBM QS20 Cell Blade



Source: Sam Williams



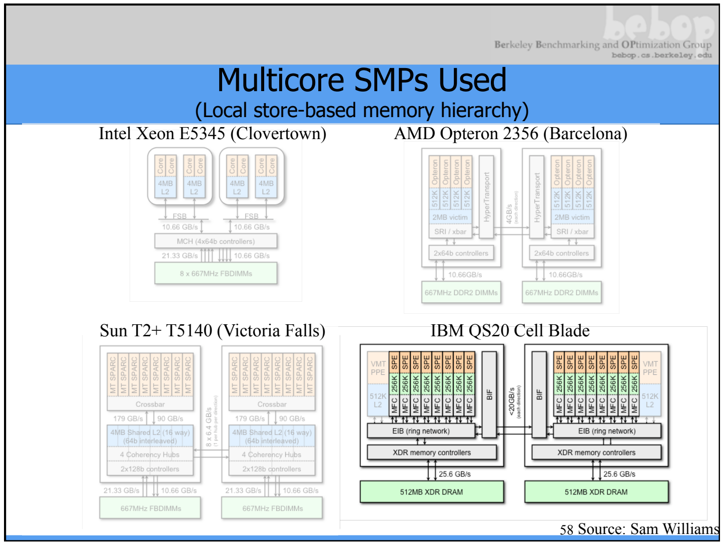
Multicore SMPs Used

(Conventional cache-based memory hierarchy)

Intel Xeon E5345 (Clovertown) AMD Opteron 2356 (Barcelona)

Sun T2+ T5140 (Victoria Falls) IBM QS20 Cell Blade

Source: Sam Williams



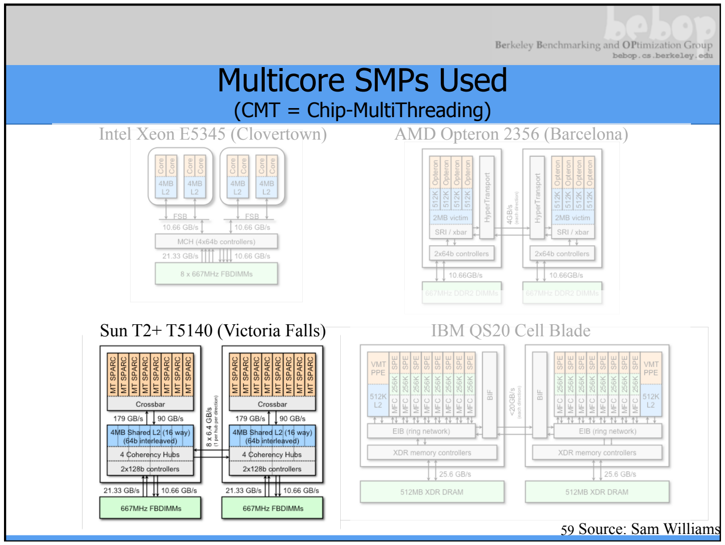
Multicore SMPs Used

(Local store-based memory hierarchy)

Intel Xeon E5345 (Clovertown) AMD Opteron 2356 (Barcelona)

Sun T2+ T5140 (Victoria Falls) IBM QS20 Cell Blade

Source: Sam Williams



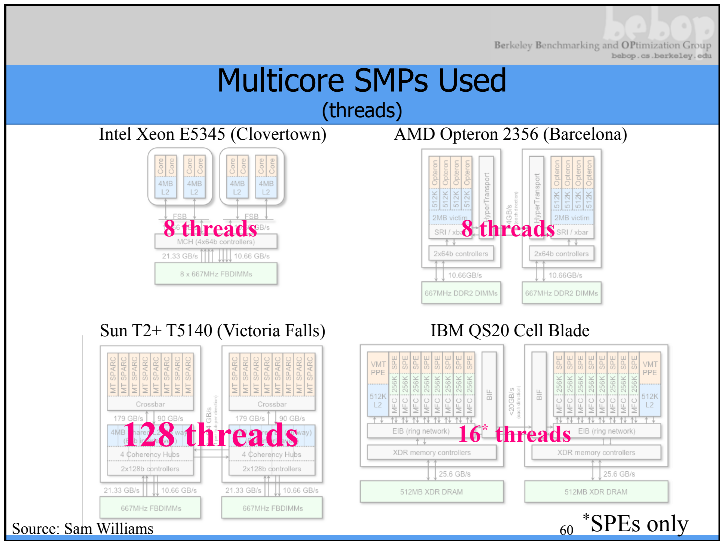
Multicore SMPs Used

(CMT = Chip-MultiThreading)

Intel Xeon E5345 (Clovertown) AMD Opteron 2356 (Barcelona)

Sun T2+ T5140 (Victoria Falls) IBM QS20 Cell Blade

Source: Sam Williams



Multicore SMPs Used

(threads)

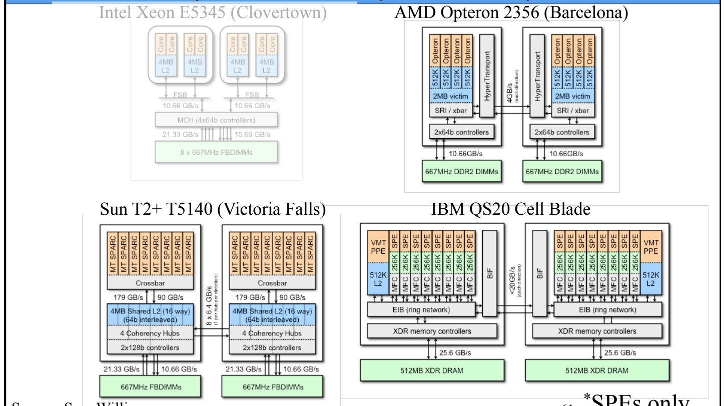
Intel Xeon E5345 (Clovertown) AMD Opteron 2356 (Barcelona)

Sun T2+ T5140 (Victoria Falls) IBM QS20 Cell Blade

Source: Sam Williams *SPEs only

Multicore SMPs Used

(Non-Uniform Memory Access - NUMA)

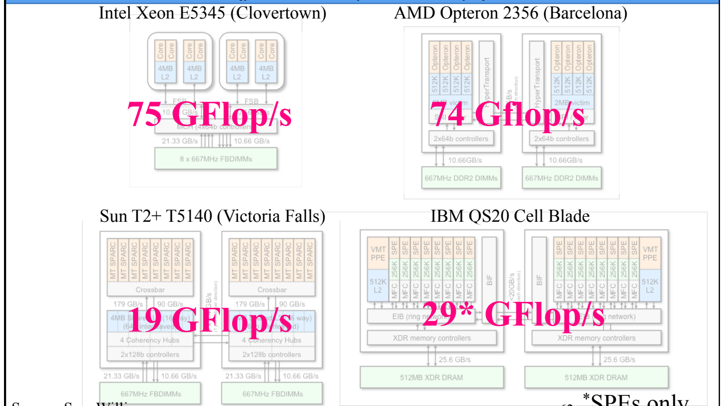


Source: Sam Williams

*SPEs only

Multicore SMPs Used

(peak double precision flops)

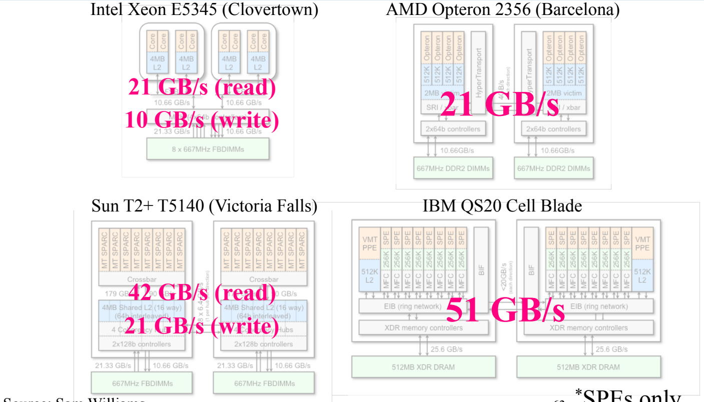

Intel Xeon E5345 (Clovertown)
AMD Opteron 2356 (Barcelona)
75 GFlop/s
74 Gflop/s
Sun T2+ T5140 (Victoria Falls)
IBM QS20 Cell Blade
19 GFlop/s
29* GFlop/s

Source: Sam Williams

*SPEs only

Multicore SMPs Used

(Total DRAM bandwidth)

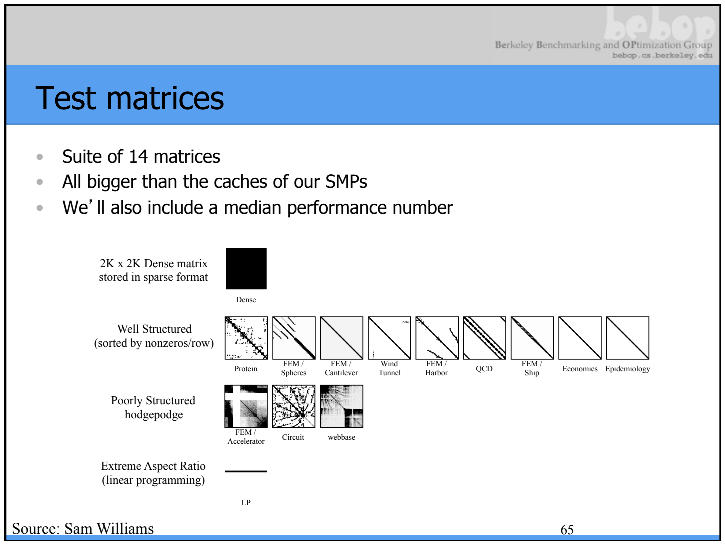


Source: Sam Williams

*SPEs only

Results from
"Auto-tuning Sparse Matrix-Vector Multiplication (SpMV)"

Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, James Demmel, "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms", Supercomputing (SC), 2007.

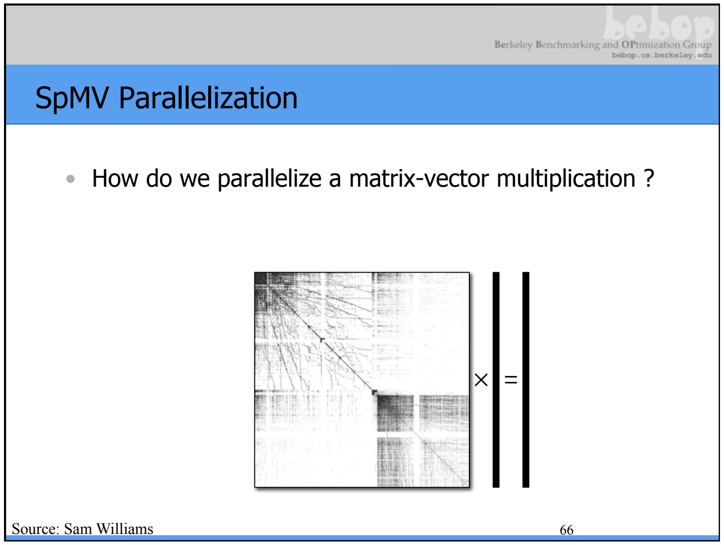


Test matrices

- Suite of 14 matrices
- All bigger than the caches of our SMPs
- We’ ll also include a median performance number

2K x 2K Dense matrix stored in sparse format
Dense
Well Structured (sorted by nonzeros/row)
Protein
FEM / Spheres
FEM / Cantilever
Wind Tunnel
FEM / Harbor
QCD
FEM / Ship
Economics
Epidemiology
Poorly Structured hodgepodge
FEM / Accelerator
Circuit
webbase
Extreme Aspect Ratio (linear programming)
LP

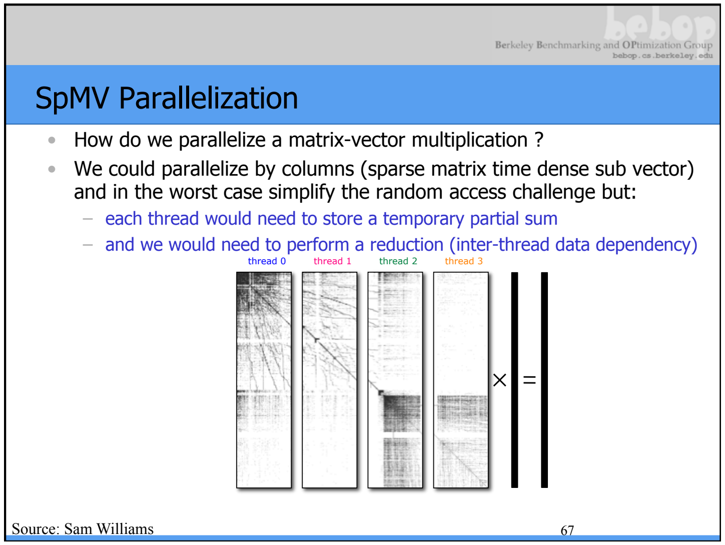
Source: Sam Williams



SpMV Parallelization

- How do we parallelize a matrix-vector multiplication ?

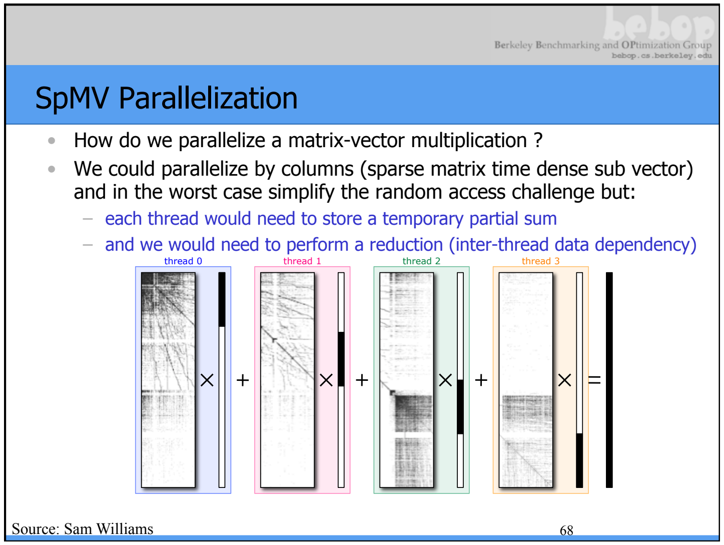
Source: Sam Williams



SpMV Parallelization

- How do we parallelize a matrix-vector multiplication ?
- We could parallelize by columns (sparse matrix time dense sub vector) and in the worst case simplify the random access challenge but:
  - each thread would need to store a temporary partial sum
  - and we would need to perform a reduction (inter-thread data dependency)

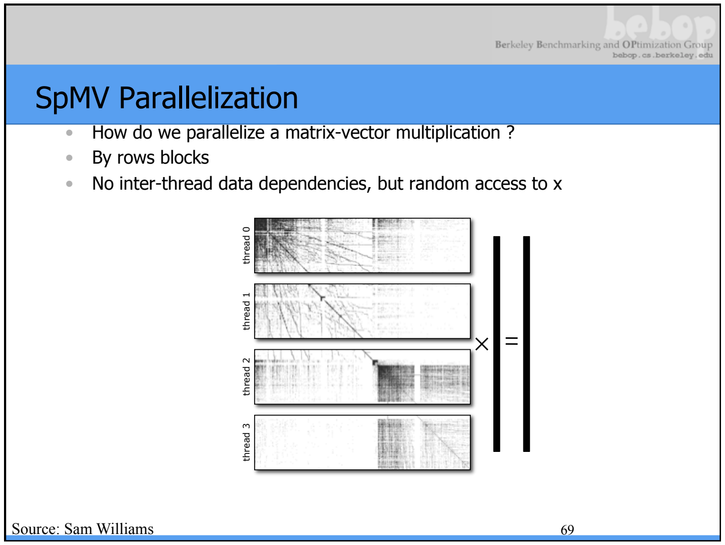
Source: Sam Williams



SpMV Parallelization

- How do we parallelize a matrix-vector multiplication ?
- We could parallelize by columns (sparse matrix time dense sub vector) and in the worst case simplify the random access challenge but:
  - each thread would need to store a temporary partial sum
  - and we would need to perform a reduction (inter-thread data dependency)

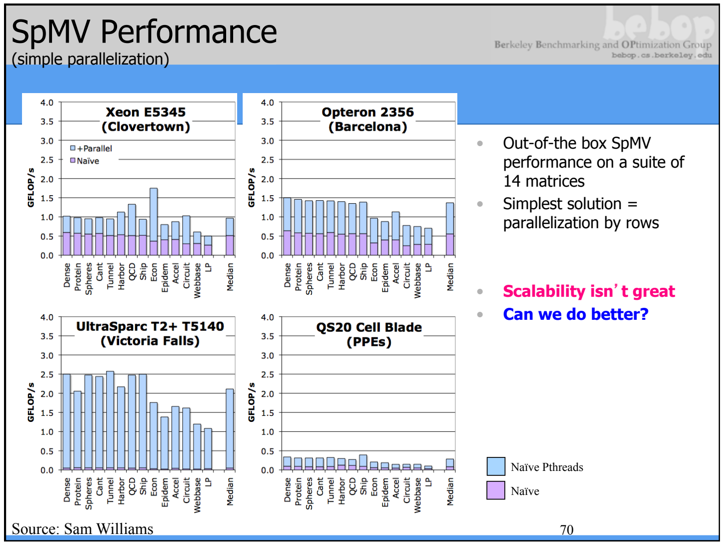
Source: Sam Williams



SpMV Parallelization

- How do we parallelize a matrix-vector multiplication ?
- By rows blocks
- No inter-thread data dependencies, but random access to x

Source: Sam Williams



SpMV Performance

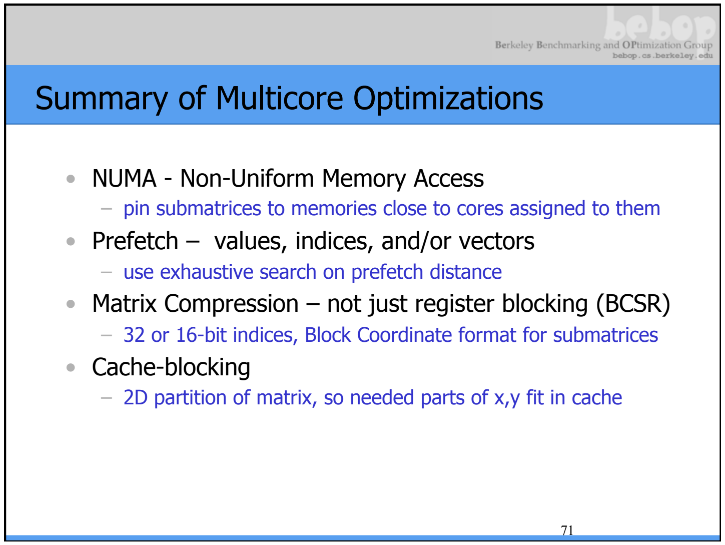
(simple parallelization)

- Out-of-the box SpMV performance on a suite of 14 matrices
- Simplest solution = parallelization by rows

- **Scalability isn't great**
- **Can we do better?**

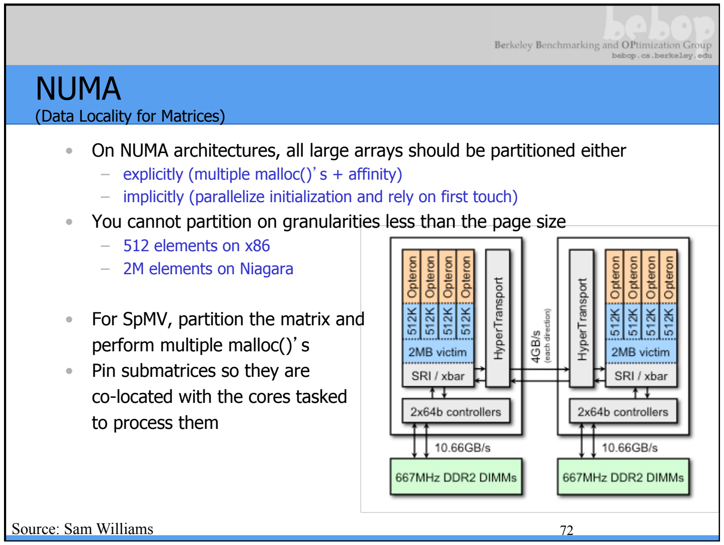
□ Naïve Pthreads
□ Naïve

Source: Sam Williams



Summary of Multicore Optimizations

- NUMA - Non-Uniform Memory Access
  - pin submatrices to memories close to cores assigned to them
- Prefetch – values, indices, and/or vectors
  - use exhaustive search on prefetch distance
- Matrix Compression – not just register blocking (BCSR)
  - 32 or 16-bit indices, Block Coordinate format for submatrices
- Cache-blocking
  - 2D partition of matrix, so needed parts of x,y fit in cache



NUMA

(Data Locality for Matrices)

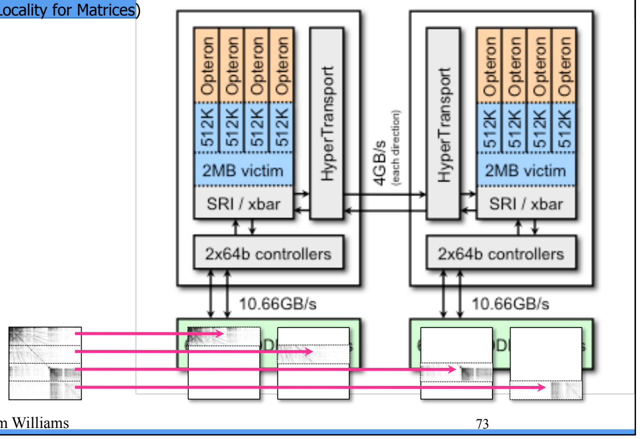
- On NUMA architectures, all large arrays should be partitioned either
  - explicitly (multiple malloc()'s + affinity)
  - implicitly (parallelize initialization and rely on first touch)
- You cannot partition on granularities less than the page size
  - 512 elements on x86
  - 2M elements on Niagara

- For SpMV, partition the matrix and perform multiple malloc()'s
- Pin submatrices so they are co-located with the cores tasked to process them

Source: Sam Williams

NUMA

(Data Locality for Matrices)



Source: Sam Williams

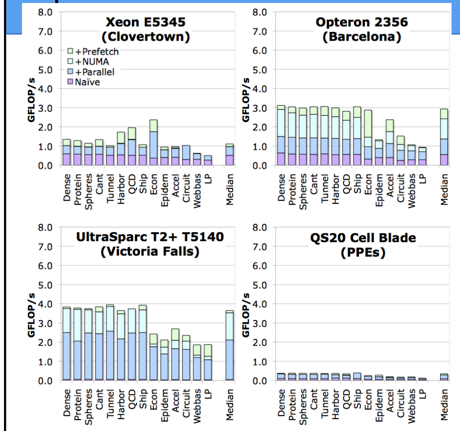
Prefetch for SpMV

- SW prefetch injects more MLP into the memory subsystem.
- Supplement HW prefetchers
- Can try to prefetch the
  - values
  - indices
  - source vector
  - *or any combination thereof*
- In general, should only insert one prefetch per cache line (works best on unrolled code)

```
for(all rows){
  y0 = 0.0;
  y1 = 0.0;
  y2 = 0.0;
  y3 = 0.0;
  for(all tiles in this row){
    PREFETCH(V+i+PFDistance);
    y0+=V[i  ]*X[C[i]]
    y1+=V[i+1]*X[C[i]]
    y2+=V[i+2]*X[C[i]]
    y3+=V[i+3]*X[C[i]]
  }
  y[r+0] = y0;
  y[r+1] = y1;
  y[r+2] = y2;
  y[r+3] = y3;
}
```

Source: Sam Williams

SpMV Performance



- NUMA-aware allocation is essential on memory-bound NUMA SMPs.
- Explicit software prefetching can boost bandwidth and change cache replacement policies
- Cell PPEs are likely latency-limited.
- used **exhaustive** search for best prefetch distance

Source: Sam Williams

Matrix Compression

- Goal: minimize memory traffic
- Register blocking
  - Choose block size to minimize memory traffic
  - Only power-of-2 block sizes
  - Simplifies search, achieves most of the possible speedup
- Shorter indices
  - 32-bit, or 16-bit if possible
- Different sparse matrix formats
  - BCSR – Block compressed sparse row
    - Like CSR but with register blocks
  - BCOO – Block coordinate
    - Stores row and column index of each register block
    - Better on very sparse sub-blocks (see cache blocking later)

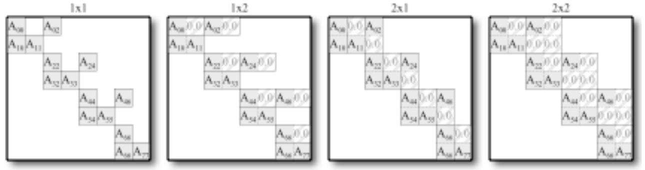
ILP/DLP vs Bandwidth

- In the multicore era, which is the bigger issue?
  - a lack of ILP/DLP (a major advantage of BCSR)
  - insufficient memory bandwidth per core
- There are many architectures that when running low arithmetic intensity kernels, there is so little available memory bandwidth per core that you won't notice a complete lack of ILP
- Perhaps we should concentrate on **minimizing memory traffic** rather than maximizing ILP/DLP
- Rather than benchmarking every combination, just **Select the register blocking that minimizes the matrix foot print.**

Source: Sam Williams

Matrix Compression Strategies

- Register blocking creates small dense tiles
  - better ILP/DLP
  - reduced overhead per nonzero

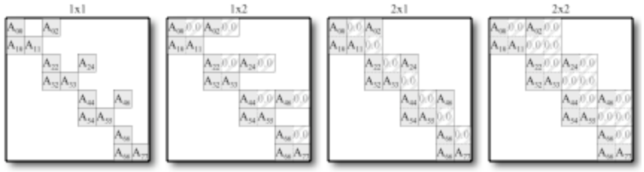


- Let each thread select a unique register blocking
- In this work,
  - we only considered power-of-two register blocks
  - select the register blocking that minimizes memory traffic

Source: Sam Williams

Matrix Compression Strategies

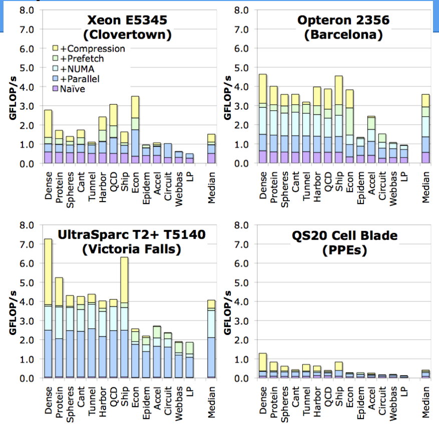
- Where possible we may encode indices with less than 32 bits
- We may also select different matrix formats



- In this work,
  - we considered 16-bit and 32-bit indices (relative to thread's start)
  - we explored BCSR/BCOO (GCSR in book chapter)

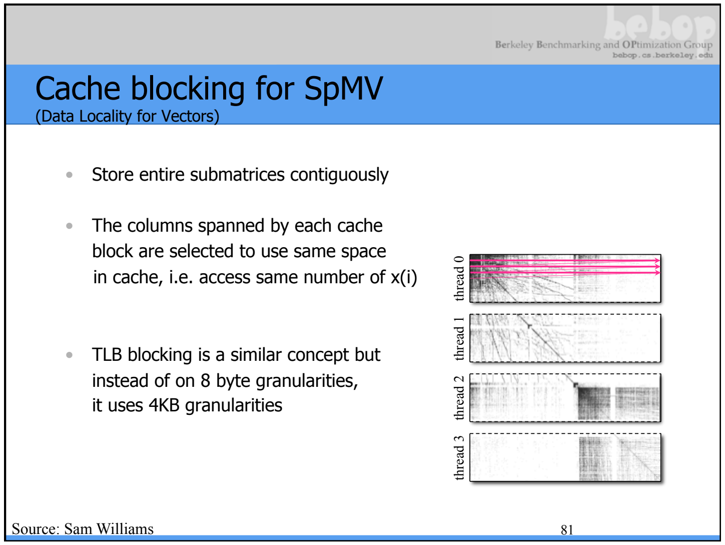
Source: Sam Williams

SpMV Performance



- After maximizing memory bandwidth, the only hope is to minimize memory traffic.
- Compression: exploit
  - register blocking
  - other formats
  - smaller indices
- Use a traffic minimization **heuristic** rather than search
- Benefit is clearly matrix-dependent.
- Register blocking enables efficient software prefetching (one per cache line)

Source: Sam Williams

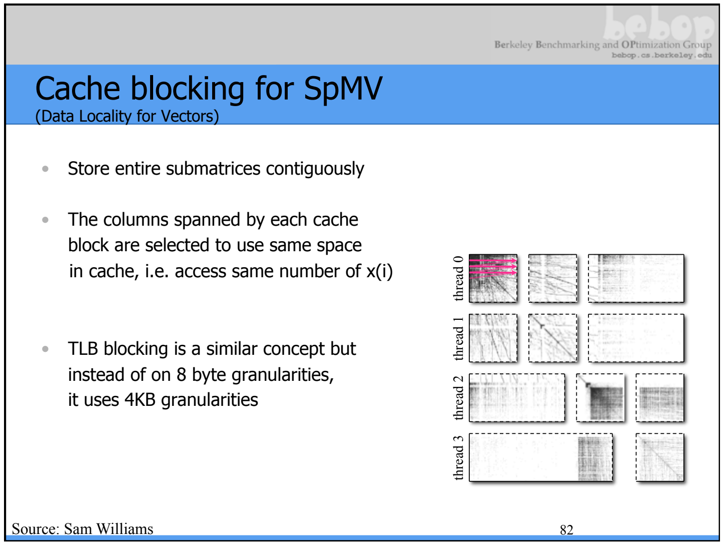


Cache blocking for SpMV

(Data Locality for Vectors)

- Store entire submatrices contiguously
- The columns spanned by each cache block are selected to use same space in cache, i.e. access same number of x(i)
- TLB blocking is a similar concept but instead of on 8 byte granularities, it uses 4KB granularities

Source: Sam Williams

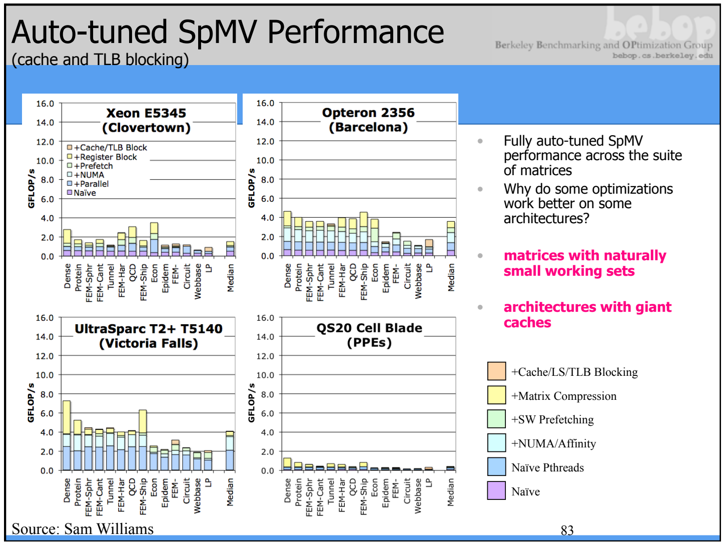


Cache blocking for SpMV

(Data Locality for Vectors)

- Store entire submatrices contiguously
- The columns spanned by each cache block are selected to use same space in cache, i.e. access same number of x(i)
- TLB blocking is a similar concept but instead of on 8 byte granularities, it uses 4KB granularities

Source: Sam Williams



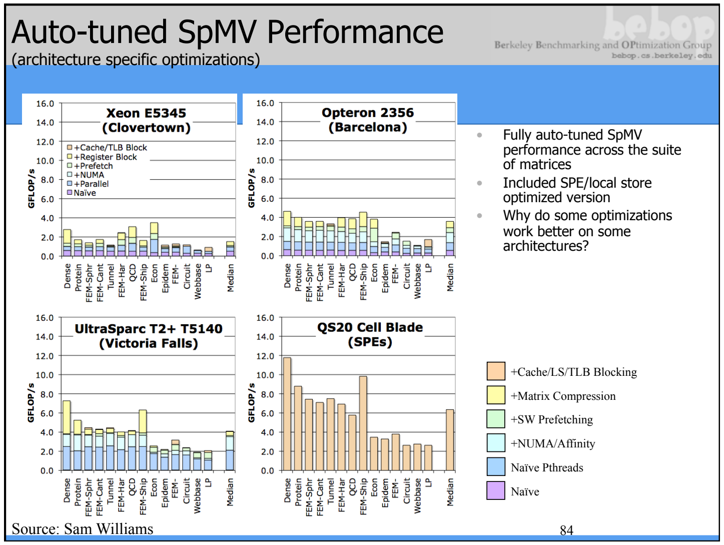
Auto-tuned SpMV Performance

(cache and TLB blocking)

- Fully auto-tuned SpMV performance across the suite of matrices
- Why do some optimizations work better on some architectures?
- **matrices with naturally small working sets**
- **architectures with giant caches**

- 🟧 +Cache/LS/TLB Blocking
- 🟨 +Matrix Compression
- 🟩 +SW Prefetching
- ▢ +NUMA/Affinity
- 🟦 Naïve Pthreads
- 🟪 Naïve

Source: Sam Williams



Auto-tuned SpMV Performance

(architecture specific optimizations)

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

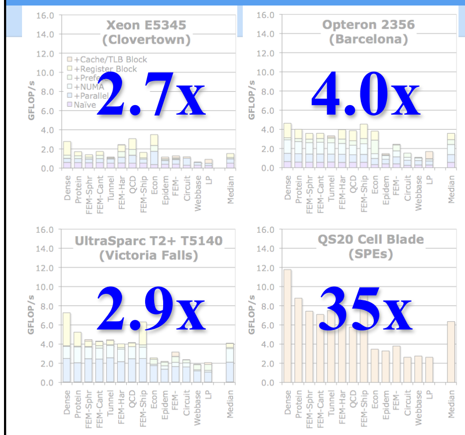
- 🟧 +Cache/LS/TLB Blocking
- 🟨 +Matrix Compression
- 🟩 +SW Prefetching
- ▢ +NUMA/Affinity
- 🟦 Naïve Pthreads
- 🟪 Naïve

Source: Sam Williams

Auto-tuned SpMV Performance

(max speedup)

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



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

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

Source: Sam Williams

85

Optimized Sparse Kernel Interface - pOSKI

Berkeley Benchmarking

bebop.cs.berkeley.edu/poski

- Provides sparse kernels automatically tuned for user's matrix & machine
  - BLAS-style functionality: SpMV, Ax & A^Ty
  - Hides complexity of run-time tuning
- Based on OSKI – bebop.cs.berkeley.edu/oski
  - Autotuner for sequential sparse matrix operations:
    - SpMV (Ax and A^Tx), A^TAx, solve sparse triangular systems, …
  - So far pOSKI only does multicore optimizations of SpMV
  - Up to 4.5x faster SpMV (Ax) on Intel Sandy Bridge E
- Work by the Berkeley Benchmarking and Optimization (BeBop) group

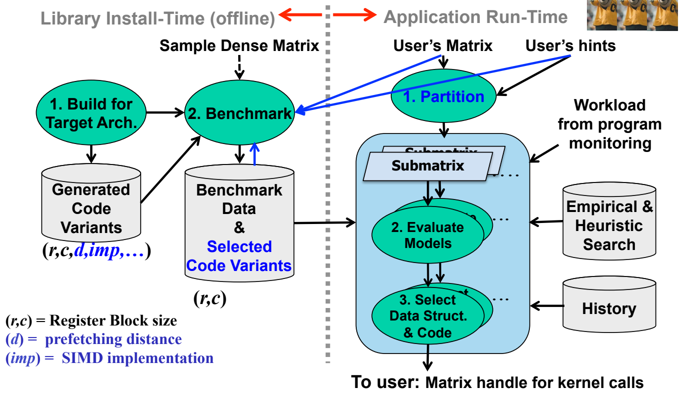
Optimizations in pOSKI, so far

Berkeley Benchmarking

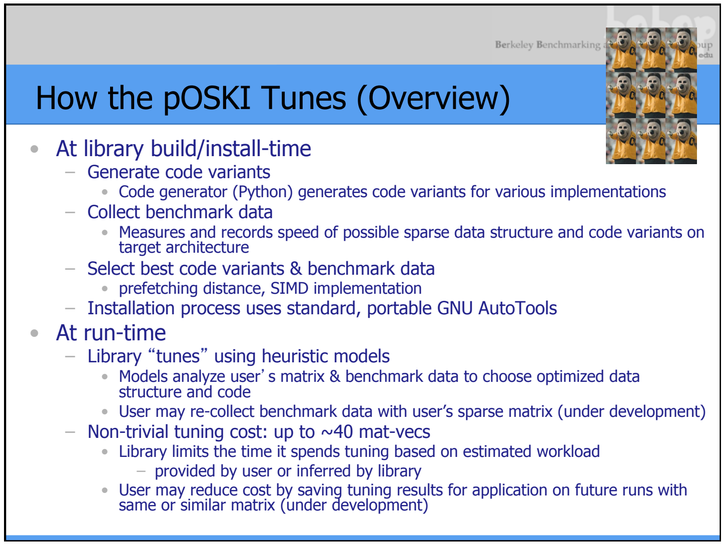
- Fully automatic heuristics for
  - Sparse matrix-vector multiply (Ax, A^Tx)
    - Register-level blocking, Thread-level blocking
    - SIMD, software prefetching, software pipelining, loop unrolling
    - NUMA-aware allocations
- "Plug-in" extensibility
  - Very advanced users may write their own heuristics, create new data structures/code variants and dynamically add them to the system, using embedded scripting language Lua
- Other optimizations that could be added
  - Cache-level blocking, Reordering (RCM, TSP), variable block structure, index compressing, Symmetric storage, etc.

How the pOSKI Tunes (Overview)

Berkeley Benchmarking

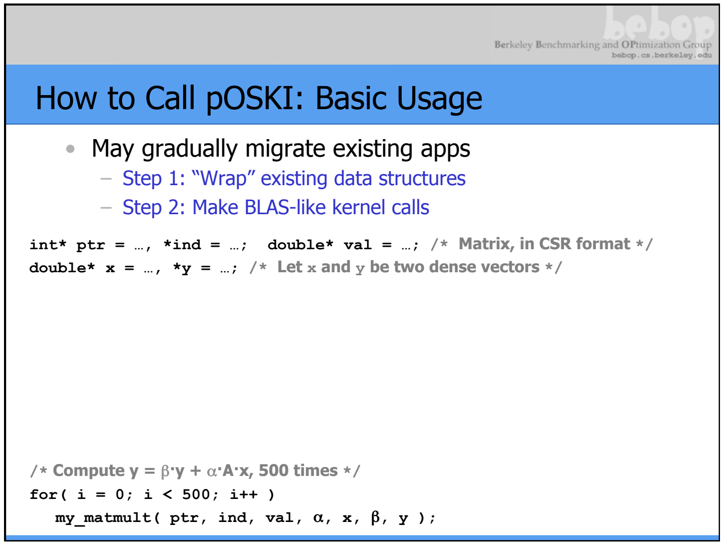

Library Install-Time (offline)
Application Run-Time
Sample Dense Matrix
User's Matrix
User's hints
1. Build for Target Arch.
2. Benchmark
1. Partition
Workload from program monitoring
Generated Code Variants
(r,c,d,imp,…)
Benchmark Data & Selected Code Variants
(r,c)
Submatrix
2. Evaluate Models
3. Select Data Struct. & Code
Empirical & Heuristic Search
History
To user: Matrix handle for kernel calls

(r,c) = Register Block size
(d) = prefetching distance
(imp) = SIMD implementation



How the pOSKI Tunes (Overview)

- At library build/install-time
  - Generate code variants
    - Code generator (Python) generates code variants for various implementations
  - Collect benchmark data
    - Measures and records speed of possible sparse data structure and code variants on target architecture
  - Select best code variants & benchmark data
    - prefetching distance, SIMD implementation
  - Installation process uses standard, portable GNU AutoTools
- At run-time
  - Library "tunes" using heuristic models
    - Models analyze user' s matrix & benchmark data to choose optimized data structure and code
    - User may re-collect benchmark data with user's sparse matrix (under development)
  - Non-trivial tuning cost: up to ~40 mat-vecs
    - Library limits the time it spends tuning based on estimated workload
      - provided by user or inferred by library
    - User may reduce cost by saving tuning results for application on future runs with same or similar matrix (under development)



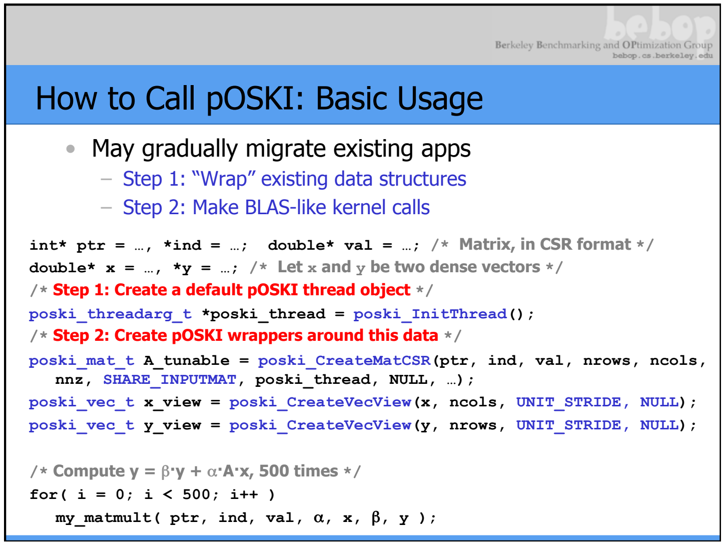
How to Call pOSKI: Basic Usage

- 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 );
```

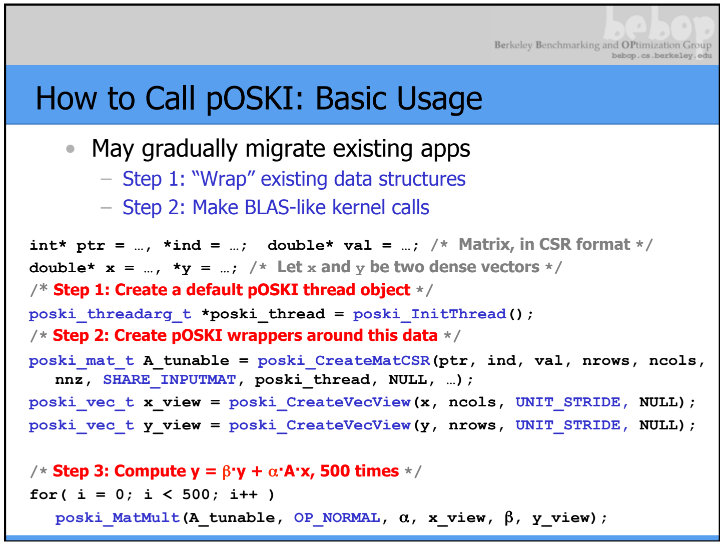


How to Call pOSKI: Basic Usage

- 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 a default pOSKI thread object */
poski_threadarg_t *poski_thread = poski_InitThread();
/* Step 2: Create pOSKI wrappers around this data */
poski_mat_t A_tunable = poski_CreateMatCSR(ptr, ind, val, nrows, ncols,
  nnz, SHARE_INPUTMAT, poski_thread, NULL, …);
poski_vec_t x_view = poski_CreateVecView(x, ncols, UNIT_STRIDE, NULL);
poski_vec_t y_view = poski_CreateVecView(y, nrows, UNIT_STRIDE, NULL);

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

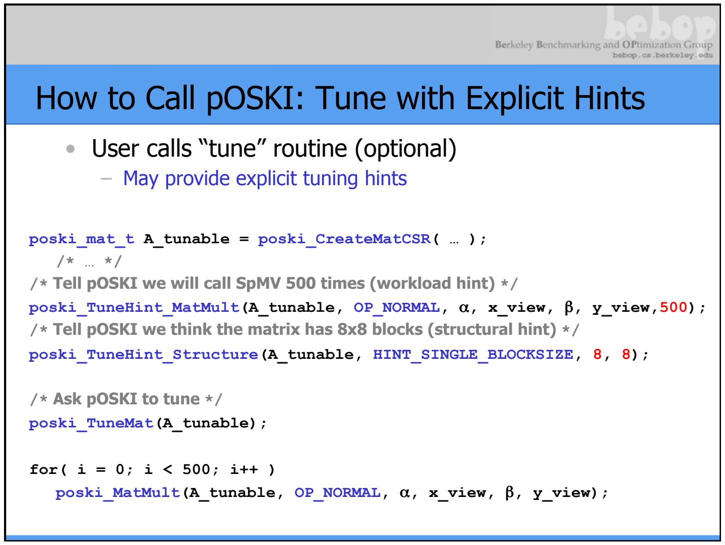


How to Call pOSKI: Basic Usage

- 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 a default pOSKI thread object */
poski_threadarg_t *poski_thread = poski_InitThread();
/* Step 2: Create pOSKI wrappers around this data */
poski_mat_t A_tunable = poski_CreateMatCSR(ptr, ind, val, nrows, ncols,
  nnz, SHARE_INPUTMAT, poski_thread, NULL, …);
poski_vec_t x_view = poski_CreateVecView(x, ncols, UNIT_STRIDE, NULL);
poski_vec_t y_view = poski_CreateVecView(y, nrows, UNIT_STRIDE, NULL);

/* Step 3: Compute y = β·y + α·A·x, 500 times */
for( i = 0; i < 500; i++ )
  poski_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view);
```



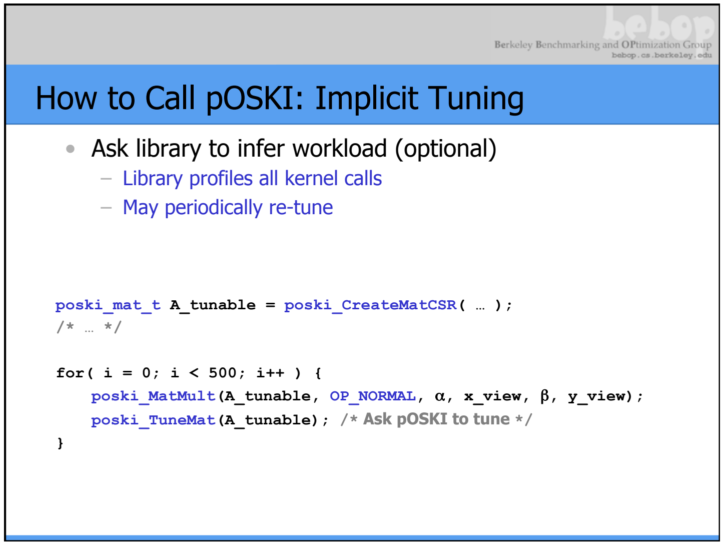
How to Call pOSKI: Tune with Explicit Hints

- User calls “tune” routine (optional)
  - May provide explicit tuning hints

```
poski_mat_t A_tunable = poski_CreateMatCSR( … );
   /* … */
/* Tell pOSKI we will call SpMV 500 times (workload hint) */
poski_TuneHint_MatMult(A_tunable, OP_NORMAL, α, x_view, β, y_view,500);
/* Tell pOSKI we think the matrix has 8x8 blocks (structural hint) */
poski_TuneHint_Structure(A_tunable, HINT_SINGLE_BLOCKSIZE, 8, 8);

/* Ask pOSKI to tune */
poski_TuneMat(A_tunable);

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

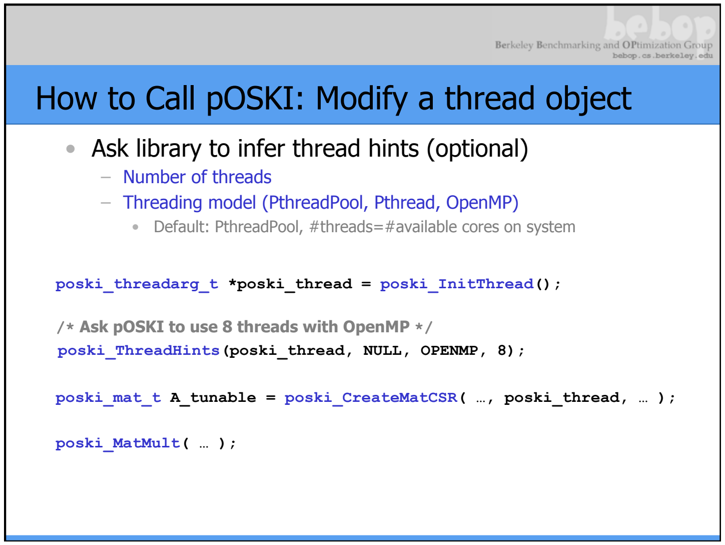


How to Call pOSKI: Implicit Tuning

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

```
poski_mat_t A_tunable = poski_CreateMatCSR( … );
/* … */

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



How to Call pOSKI: Modify a thread object

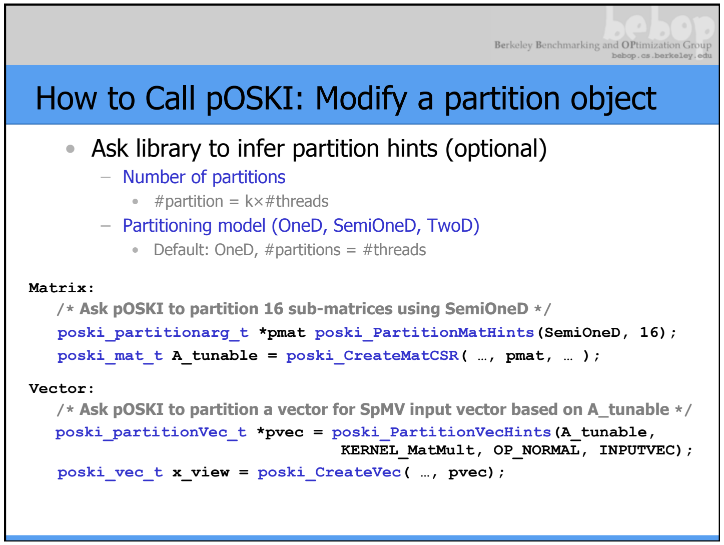
- Ask library to infer thread hints (optional)
  - Number of threads
  - Threading model (PthreadPool, Pthread, OpenMP)
    - Default: PthreadPool, #threads=#available cores on system

```
poski_threadarg_t *poski_thread = poski_InitThread();

/* Ask pOSKI to use 8 threads with OpenMP */
poski_ThreadHints(poski_thread, NULL, OPENMP, 8);

poski_mat_t A_tunable = poski_CreateMatCSR( …, poski_thread, … );

poski_MatMult( … );
```



How to Call pOSKI: Modify a partition object

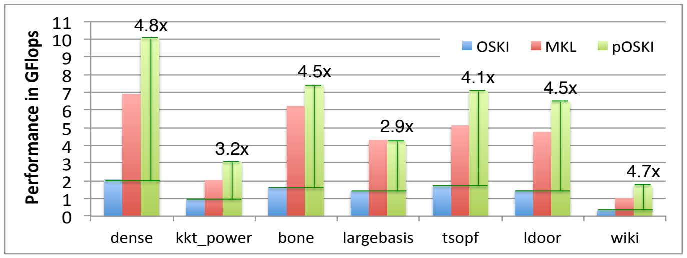
- Ask library to infer partition hints (optional)
  - Number of partitions
    - #partition = k×#threads
  - Partitioning model (OneD, SemiOneD, TwoD)
    - Default: OneD, #partitions = #threads

```
Matrix:
   /* Ask pOSKI to partition 16 sub-matrices using SemiOneD */
   poski_partitionarg_t *pmat poski_PartitionMatHints(SemiOneD, 16);
   poski_mat_t A_tunable = poski_CreateMatCSR( …, pmat, … );

Vector:
   /* Ask pOSKI to partition a vector for SpMV input vector based on A_tunable */
   poski_partitionVec_t *pvec = poski_PartitionVecHints(A_tunable,
                                    KERNEL_MatMult, OP_NORMAL, INPUTVEC);
   poski_vec_t x_view = poski_CreateVec( …, pvec);
```

Performance on Intel Sandy Bridge E

- Jaketown: i7-3960X @ 3.3 GHz
- #Cores: 6 (2 threads per core), L3:15MB
- pOSKI SpMV (Ax) with double precision floating point
- MKL Sparse BLAS Level 2: *mkl_dcsrmv()*



Is tuning SpMV all we can do?

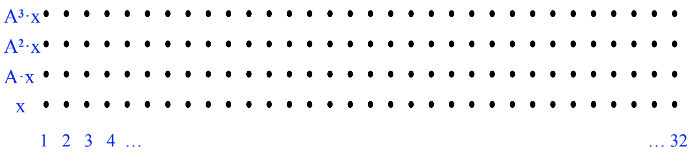
- Iterative methods all depend on it
- But speedups are limited
  - Just 2 flops per nonzero
  - Communication costs dominate
- Can we beat this bottleneck?
- Need to look at next level in stack:
  - What do algorithms that use SpMV do?
  - Can we reorganize them to avoid communication?
- Only way significant speedups will be possible

Tuning Higher Level Algorithms than SpMV

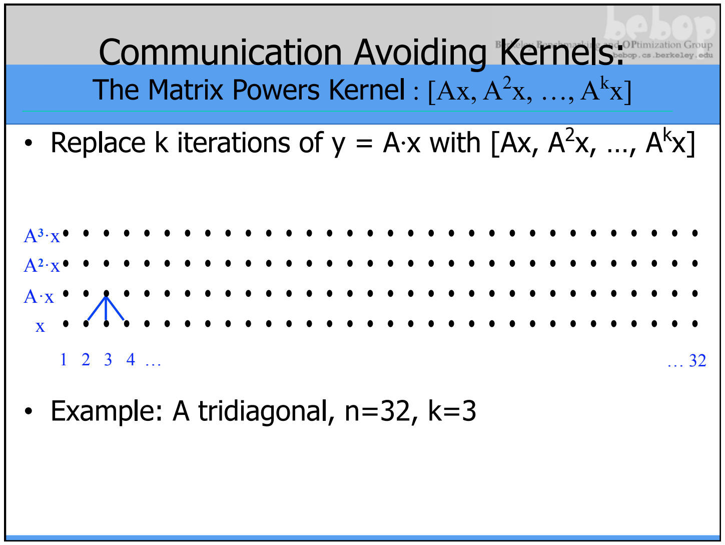
- We almost always do many SpMVs, not just one
  - "Krylov Subspace Methods" (KSMs) for Ax=b, Ax = λ x
    - Conjugate Gradients, GMRES, Lanczos, …
  - Do a sequence of k SpMVs to get vectors $[x_1 , \dots , x_k]$
  - Find best solution x as linear combination of $[x_1 , \dots , x_k]$
- Main cost is k SpMVs
- Since communication usually dominates, can we do better?
- Goal: make communication cost independent of k
  - Parallel case: O(log P) messages, not O(k log P) - optimal
    - same bandwidth as before
  - Sequential case: O(1) messages and bandwidth, not O(k) - optimal
- Achievable when matrix partitionable with low surface-to-volume ratio

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

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



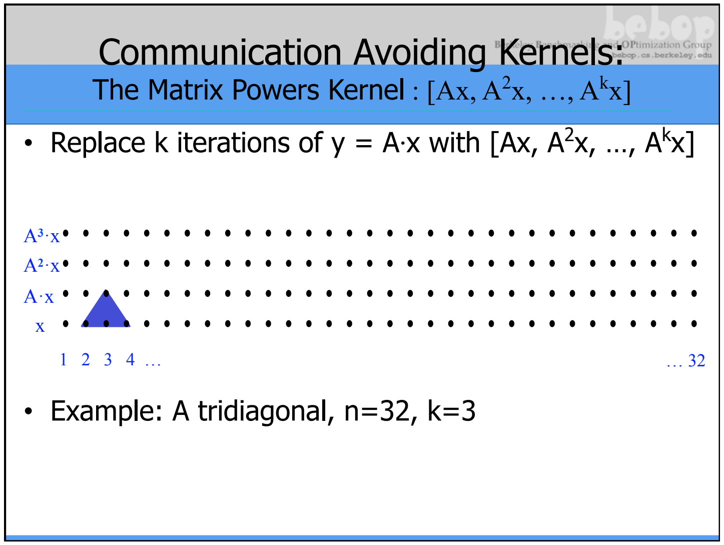
- Example: A tridiagonal, n=32, k=3
- Works for any "well-partitioned" A



Communication Avoiding Kernels: The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

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

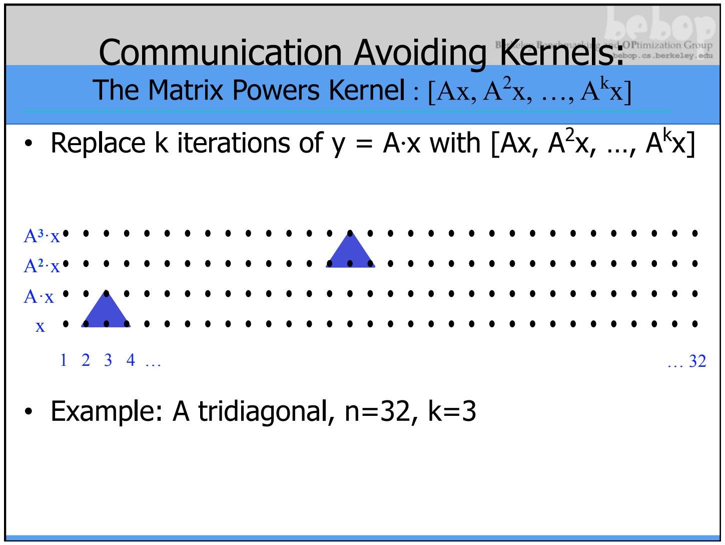
- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels: The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

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

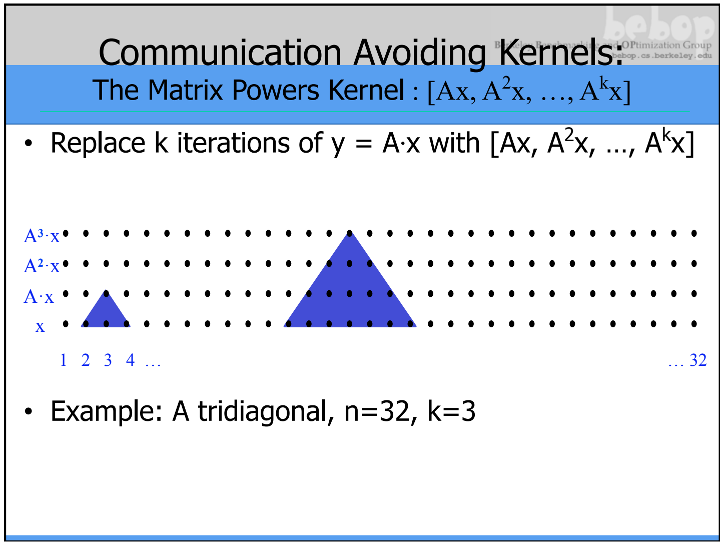
- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels: The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

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

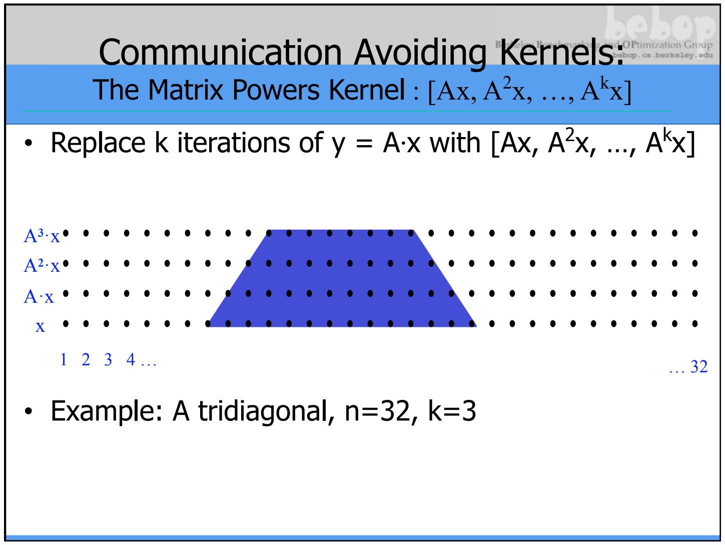
- Example: A tridiagonal, n=32, k=3



Communication Avoiding Kernels: The Matrix Powers Kernel : $[Ax, A^2x, \ldots, A^kx]$

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

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

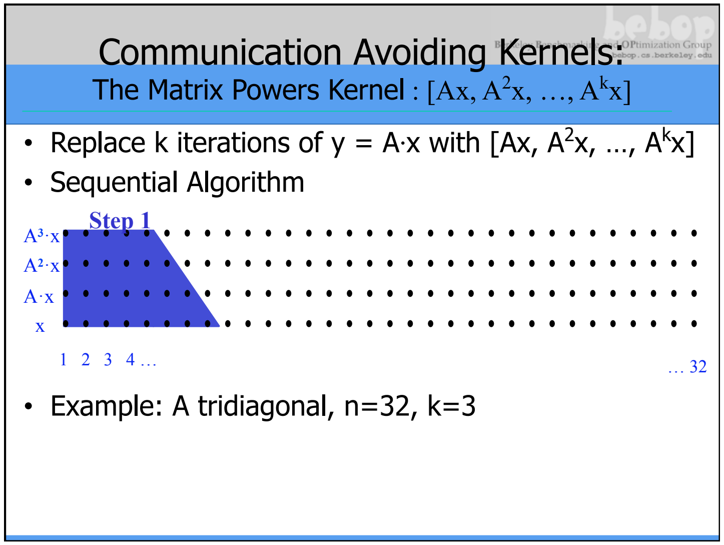


Communication Avoiding Kernels:

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

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

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

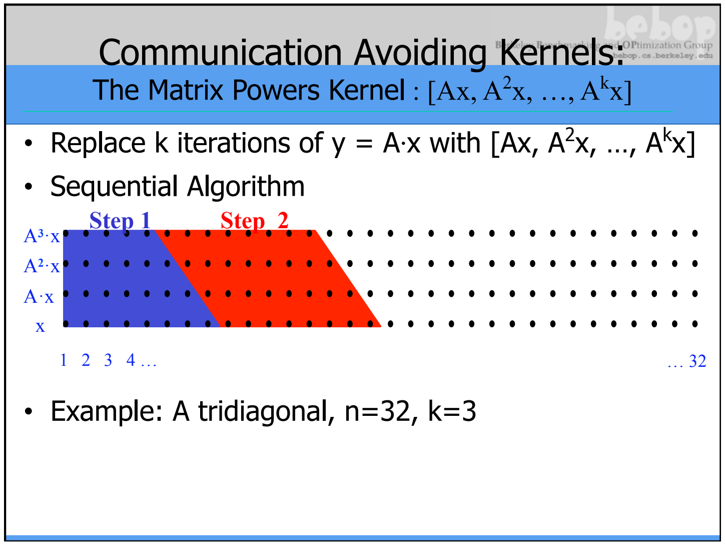


Communication Avoiding Kernels:

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

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

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

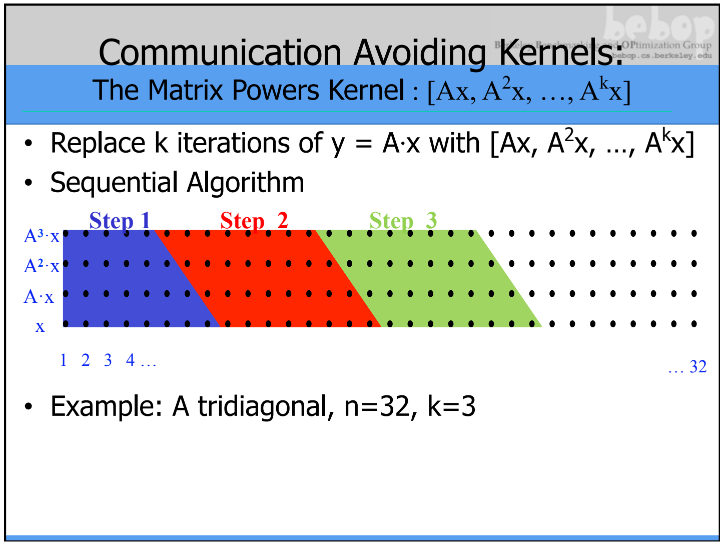


Communication Avoiding Kernels:

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

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

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

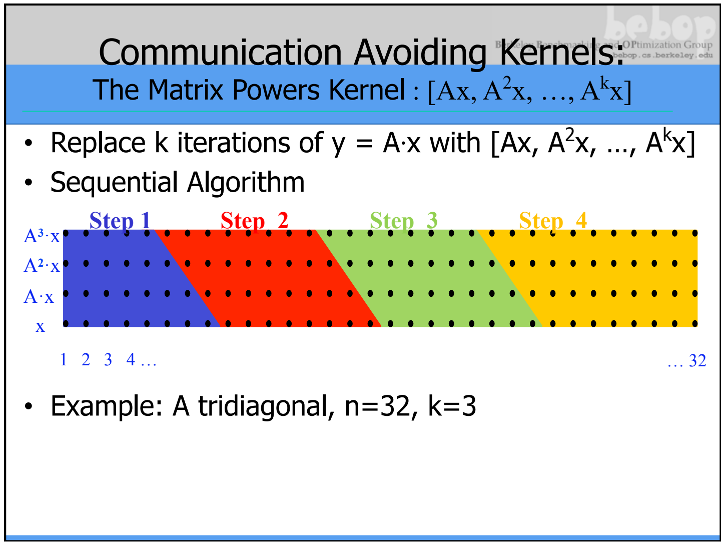


Communication Avoiding Kernels:

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

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

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

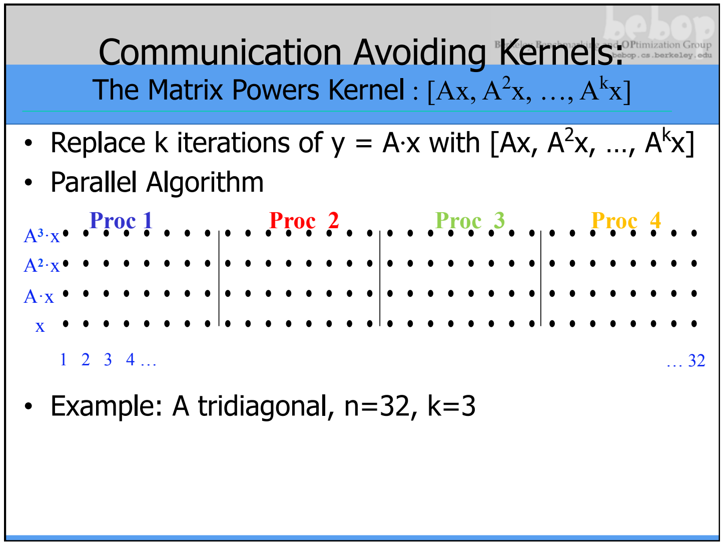


Communication Avoiding Kernels:

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

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

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

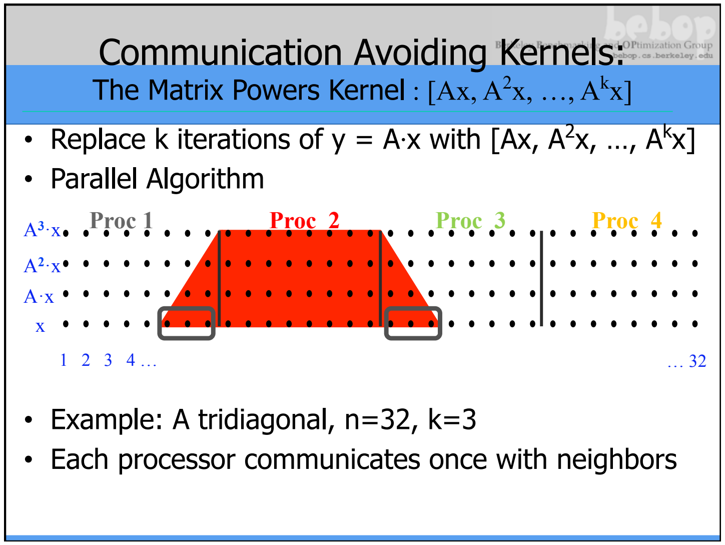


Communication Avoiding Kernels:

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

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

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

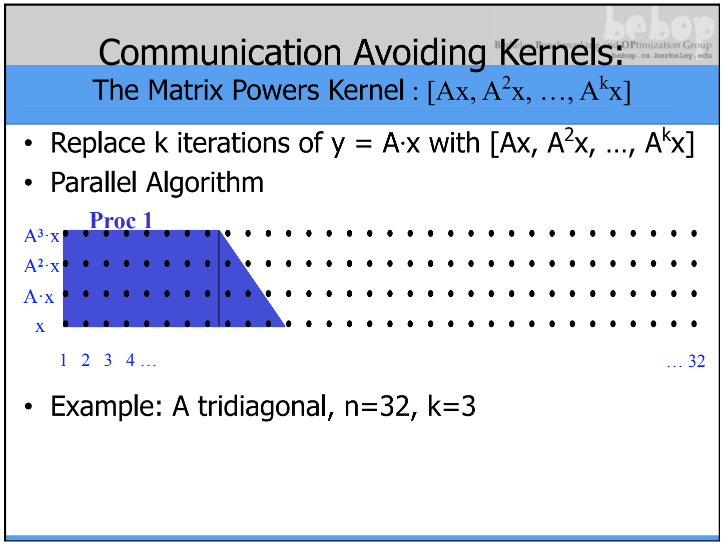


Communication Avoiding Kernels:

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

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

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



Communication Avoiding Kernels:

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

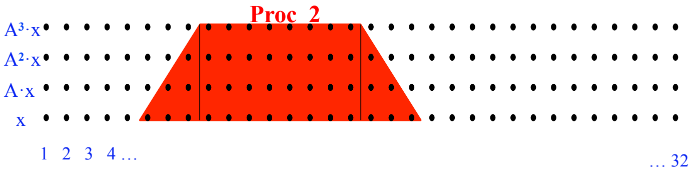
- Replace k iterations of $y = A \cdot x$ with $[Ax, A^2x, \ldots, A^kx]$
- Parallel Algorithm

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

Communication Avoiding Kernels:

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

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

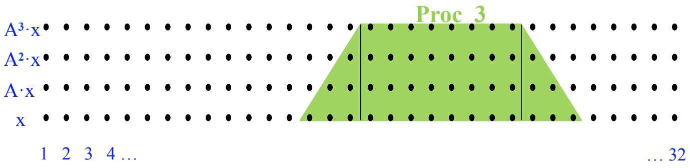


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

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

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

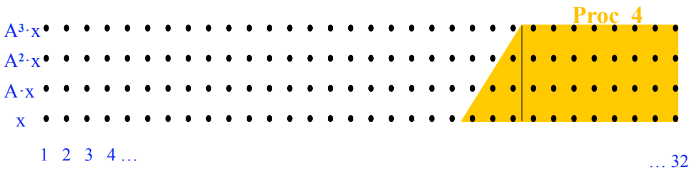


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

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

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

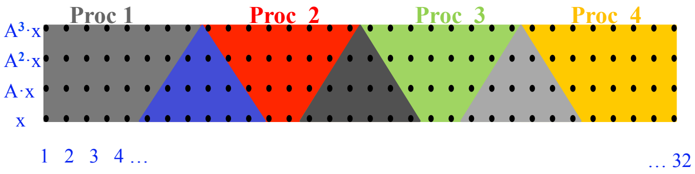


- Example: A tridiagonal, n=32, k=3

Communication Avoiding Kernels:

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

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



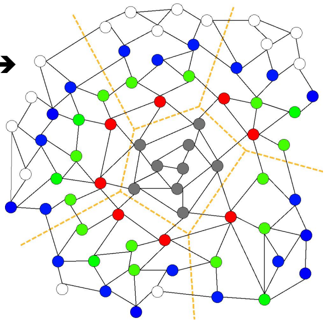
- Example: A tridiagonal, n=32, k=3
- Each processor works on (overlapping) trapezoid

Communication Avoiding Kernels:
The Matrix Powers Kernel : [$Ax, A^2x, \ldots, A^kx$]

Same idea works for general sparse matrices

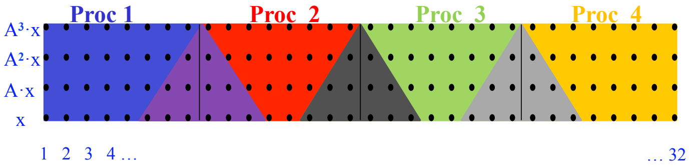
Simple block-row partitioning ➔
(hyper)graph partitioning

Top-to-bottom processing ➔
Traveling Salesman Problem



Communication Avoiding Kernels:
The Matrix Powers Kernel : [$Ax, A^2x, \ldots, A^kx$]

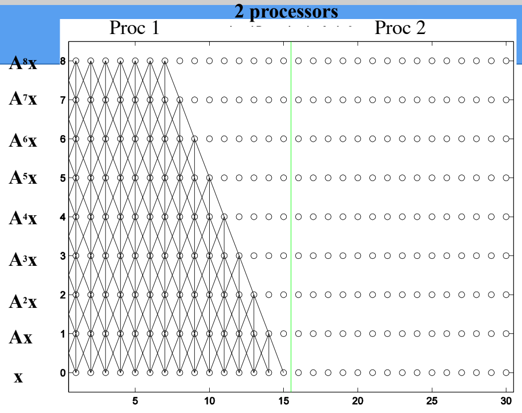
- Replace k iterations of y = A·x with [$Ax, A^2x, \ldots, A^kx$]
- Parallel Algorithm



- Example: A tridiagonal, n=32, k=3
- Entries in overlapping regions (triangles) computed redundantly

Locally Dependent Entries for [x,Ax,…,A^8x], A tridiagonal

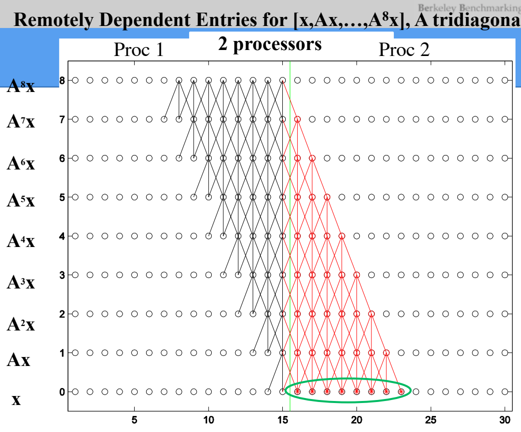
2 processors



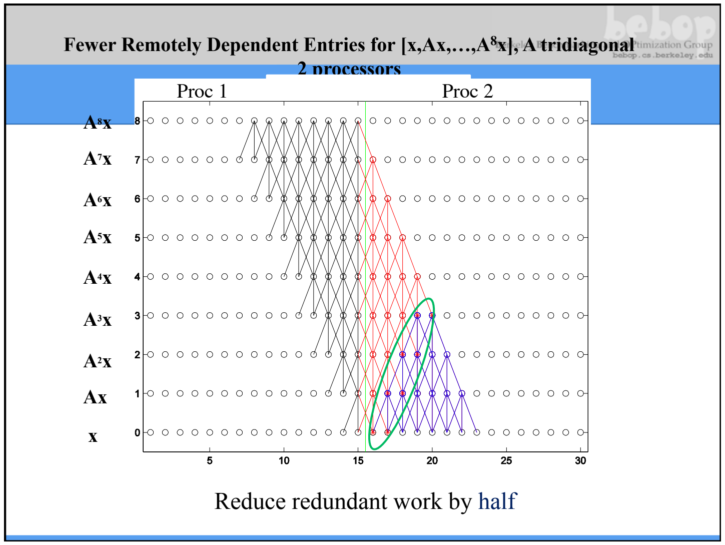
Can be computed without communication
k=8 fold reuse of A

Remotely Dependent Entries for [x,Ax,…,A^8x], A tridiagonal

2 processors

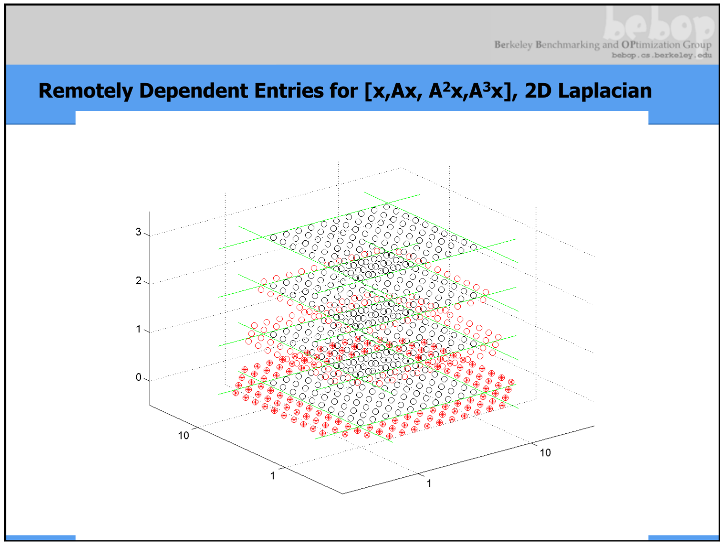


One message to get data needed to compute remotely dependent entries, ***not k=8***

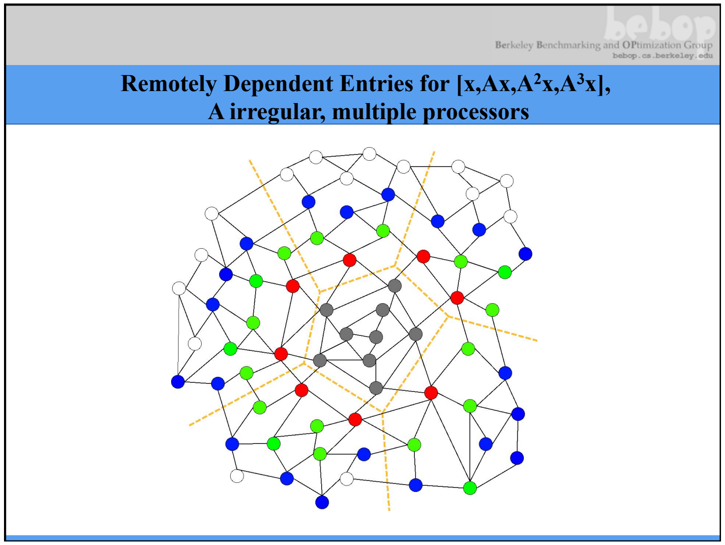


Fewer Remotely Dependent Entries for $[x,Ax,\ldots,A^8x]$, A tridiagonal 2 processors

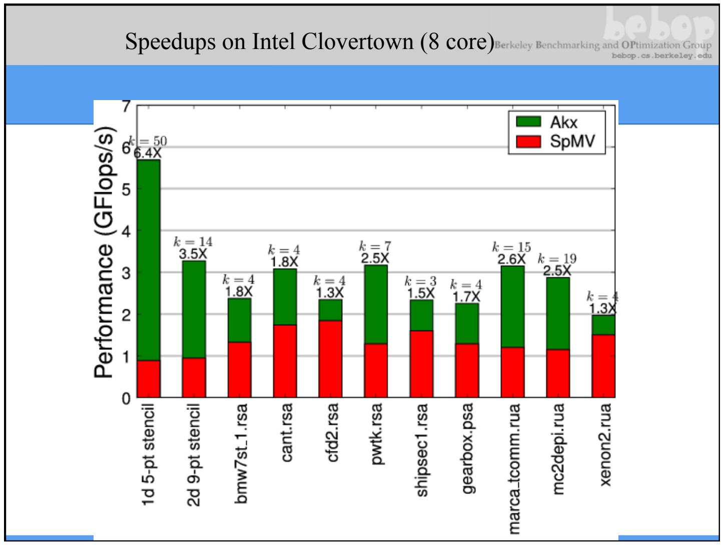
Reduce redundant work by half



Remotely Dependent Entries for $[x,Ax,A^2x,A^3x]$, 2D Laplacian



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



Speedups on Intel Clovertown (8 core)

Performance Results

- Measured Multicore (Clovertown) speedups up to 6.4x
- Measured/Modeled sequential OOC speedup up to 3x
- Modeled parallel Petascale speedup up to 6.9x
- Modeled parallel Grid speedup up to 22x

- Sequential speedup due to bandwidth, works for many problem sizes
- Parallel speedup due to latency, works for smaller problems on many processors
- Multicore results used both techniques

Avoiding Communication in Iterative Linear Algebra

- k-steps of typical iterative solver for sparse Ax=b or Ax= λ x
  - Does k SpMVs with starting vector
  - Finds "best" solution among all linear combinations of these k+1 vectors
  - Many such "Krylov Subspace Methods"
    - Conjugate Gradients, GMRES, Lanczos, Arnoldi, …
- Goal: minimize communication in Krylov Subspace Methods
  - Assume matrix "well-partitioned," with modest surface-to-volume ratio
  - Parallel implementation
    - Conventional: O(k log p) messages, because k calls to SpMV
    - **New: O(log p) messages - optimal**
  - Serial implementation
    - Conventional: O(k) moves of data from slow to fast memory
    - **New**: **O(1) moves of data – optimal**
- Lots of speed up possible (modeled and measured)
  - Price: some redundant computation
- Much prior work
  - See theses of Mark Hoemmen, Erin Carson, other papers at bebop.cs.berkeley.edu

Minimizing Communication of GMRES to solve Ax=b

- GMRES: find x in span{b,Ab,…,A^kb} minimizing || Ax-b $||_2$
- Cost of k steps of standard GMRES vs new GMRES

Standard GMRES

```
for i=1 to k
   w = A · v(i-1)
   MGS(w, v(0),…,v(i-1))
   update v(i), H
endfor
solve LSQ problem with H
```

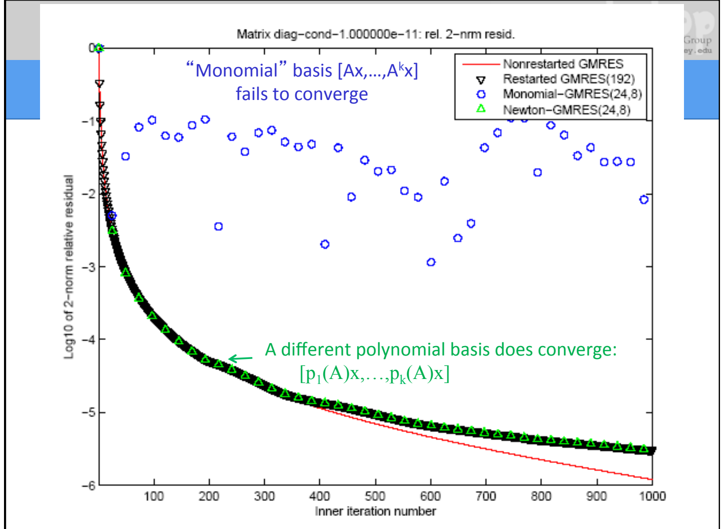
Sequential: #words_moved =
O(k·nnz) from SpMV
+ O(k^2·n) from MGS
Parallel: #messages =
O(k) from SpMV
+ O(k^2 · log p) from MGS

Communication-avoiding GMRES

```
W = [ v, Av, A²v, … , Aᵏv ]
[Q,R] = TSQR(W)  …  "Tall Skinny QR"
Build H from R, solve LSQ problem
```

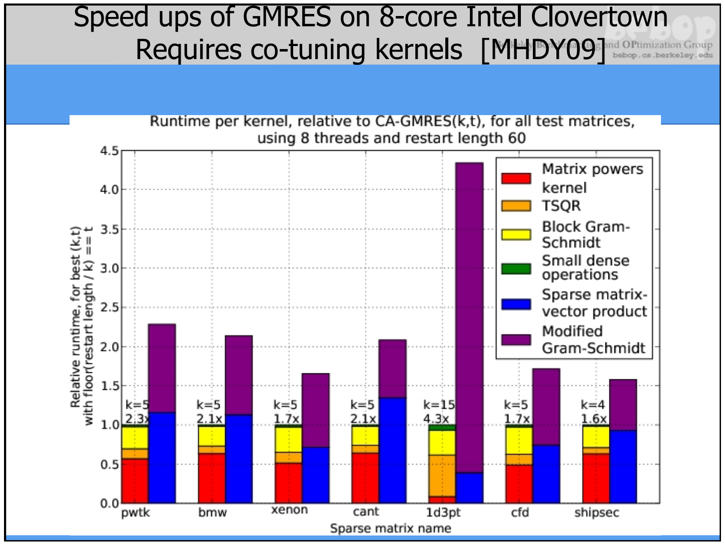
Sequential: #words_moved =
O(nnz) from SpMV
+ O(k·n) from TSQR
Parallel: #messages =
O(1) from computing W
+ O(log p) from TSQR

•Oops – W from power method, precision lost!



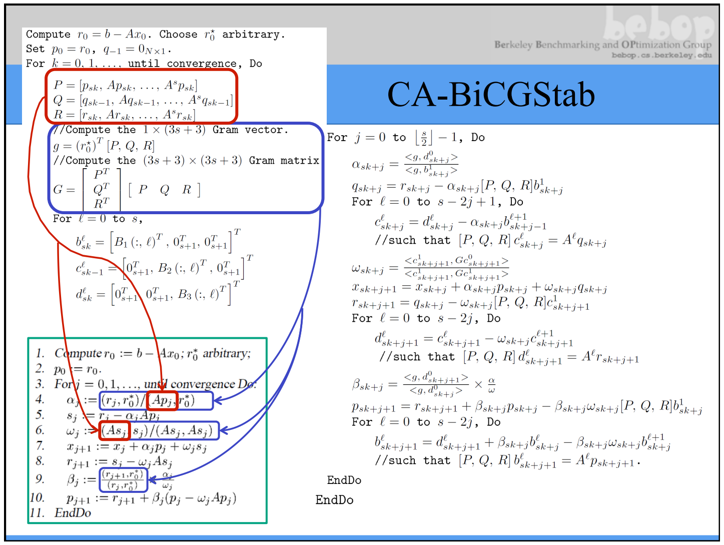
"Monomial" basis [Ax,…,A^kx] fails to converge

A different polynomial basis does converge: [p_1(A)x,…,p_k(A)x]



Speed ups of GMRES on 8-core Intel Clovertown Requires co-tuning kernels [MHDY09]

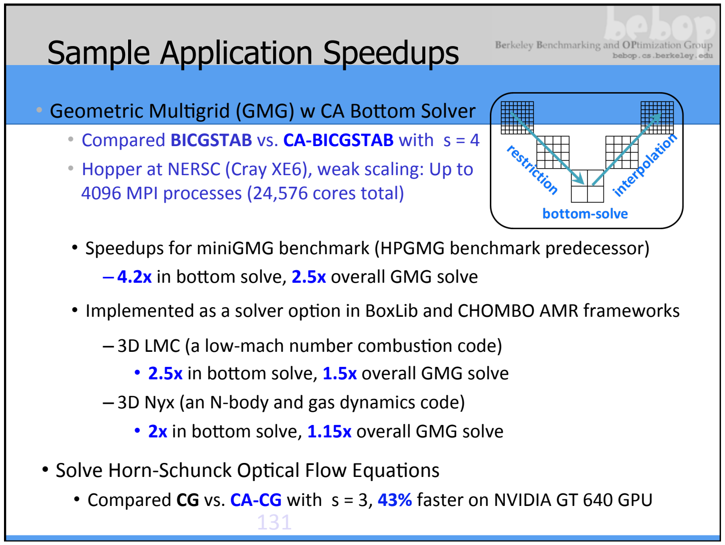
Runtime per kernel, relative to CA-GMRES(k,t), for all test matrices, using 8 threads and restart length 60



CA-BiCGStab

```
Compute $r_0 = b - Ax_0$. Choose $r_0^\star$ arbitrary.
Set $p_0 = r_0$, $q_{-1} = 0_{N\times 1}$.
For $k = 0, 1, \ldots,$ until convergence, Do
    $P = [p_{sk}, Ap_{sk}, \ldots, A^s p_{sk}]$
    $Q = [q_{sk-1}, Aq_{sk-1}, \ldots, A^s q_{sk-1}]$
    $R = [r_{sk}, Ar_{sk}, \ldots, A^s r_{sk}]$
    //Compute the $1 \times (3s+3)$ Gram vector.
    $g = (r_0^\star)^T [P, Q, R]$
    //Compute the $(3s+3) \times (3s+3)$ Gram matrix
    $G = \begin{bmatrix} P^T \\ Q^T \\ R^T \end{bmatrix} \begin{bmatrix} P & Q & R \end{bmatrix}$
    For $\ell = 0$ to $s$,
        $b_{sk}^\ell = \left[B_1(:,\ell)^T, 0_{s+1}^T, 0_{s+1}^T\right]^T$
        $c_{sk-1}^\ell = \left[0_{s+1}^T, B_2(:,\ell)^T, 0_{s+1}^T\right]^T$
        $d_{sk}^\ell = \left[0_{s+1}^T, 0_{s+1}^T, B_3(:,\ell)^T\right]^T$
    For $j = 0$ to $\lfloor \frac{s}{2} \rfloor - 1$, Do
        $\alpha_{sk+j} = \frac{<g, d_{sk+j}^0>}{<g, b_{sk+j}^1>}$
        $q_{sk+j} = r_{sk+j} - \alpha_{sk+j}[P, Q, R]b_{sk+j}^1$
        For $\ell = 0$ to $s - 2j + 1$, Do
            $c_{sk+j}^\ell = d_{sk+j}^\ell - \alpha_{sk+j} b_{sk+j-1}^{\ell+1}$
            //such that $[P, Q, R]\, c_{sk+j}^\ell = A^\ell q_{sk+j}$
        $\omega_{sk+j} = \frac{<c_{sk+j+1}^1, Gc_{sk+j+1}^0>}{<c_{sk+j+1}^1, Gc_{sk+j+1}^1>}$
        $x_{sk+j+1} = x_{sk+j} + \alpha_{sk+j} p_{sk+j} + \omega_{sk+j} q_{sk+j}$
        $r_{sk+j+1} = q_{sk+j} - \omega_{sk+j}[P, Q, R]c_{sk+j+1}^1$
        For $\ell = 0$ to $s - 2j$, Do
            $d_{sk+j+1}^\ell = c_{sk+j+1}^\ell - \omega_{sk+j} c_{sk+j+1}^{\ell+1}$
            //such that $[P, Q, R]\, d_{sk+j+1}^\ell = A^\ell r_{sk+j+1}$
        $\beta_{sk+j} = \frac{<g, d_{sk+j+1}^0>}{<g, d_{sk+j}^0>} \times \frac{\alpha}{\omega}$
        $p_{sk+j+1} = r_{sk+j+1} + \beta_{sk+j} p_{sk+j} - \beta_{sk+j}\omega_{sk+j}[P, Q, R]b_{sk+j}^1$
        For $\ell = 0$ to $s - 2j$, Do
            $b_{sk+j+1}^\ell = d_{sk+j+1}^\ell + \beta_{sk+j} b_{sk+j}^\ell - \beta_{sk+j}\omega_{sk+j} b_{sk+j}^{\ell+1}$
            //such that $[P, Q, R]\, b_{sk+j+1}^\ell = A^\ell p_{sk+j+1}$.
    EndDo
EndDo
```

```
Compute $r_0 := b - Ax_0$; $r_0^*$ arbitrary;
$p_0 := r_0$.
For $j = 0, 1, \ldots,$ until convergence Do:
    $\alpha_j := (r_j, r_0^*)/(Ap_j, r_0^*)$
    $s_j := r_j - \alpha_j Ap_j$
    $\omega_j := (As_j, s_j)/(As_j, As_j)$
    $x_{j+1} := x_j + \alpha_j p_j + \omega_j s_j$
    $r_{j+1} := s_j - \omega_j As_j$
    $\beta_j := \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)} \times \frac{\alpha_j}{\omega_j}$
    $p_{j+1} := r_{j+1} + \beta_j(p_j - \omega_j Ap_j)$
EndDo
```

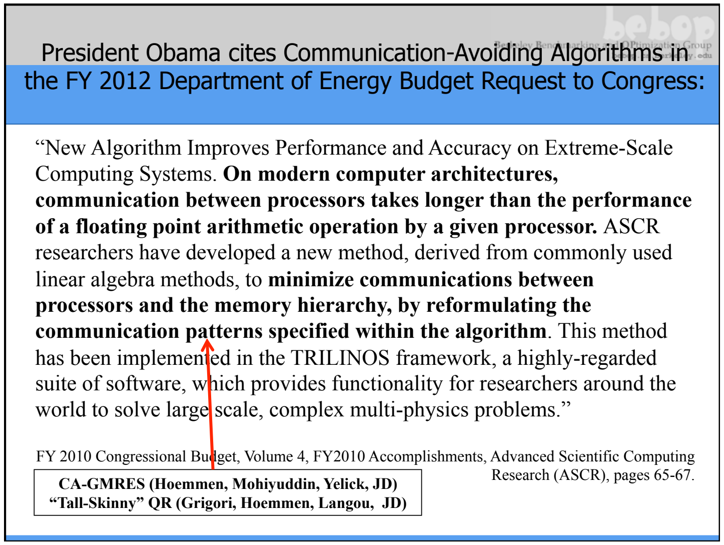


Sample Application Speedups

- Geometric Multigrid (GMG) w CA Bottom Solver
  - Compared **BICGSTAB** vs. **CA-BICGSTAB** with s = 4
  - Hopper at NERSC (Cray XE6), weak scaling: Up to 4096 MPI processes (24,576 cores total)

restriction
interpolation
bottom-solve

- Speedups for miniGMG benchmark (HPGMG benchmark predecessor)
  - **4.2x** in bottom solve, **2.5x** overall GMG solve
- Implemented as a solver option in BoxLib and CHOMBO AMR frameworks
  - 3D LMC (a low-mach number combustion code)
    - **2.5x** in bottom solve, **1.5x** overall GMG solve
  - 3D Nyx (an N-body and gas dynamics code)
    - **2x** in bottom solve, **1.15x** overall GMG solve
- Solve Horn-Schunck Optical Flow Equations
  - Compared **CG** vs. **CA-CG** with s = 3, **43%** faster on NVIDIA GT 640 GPU

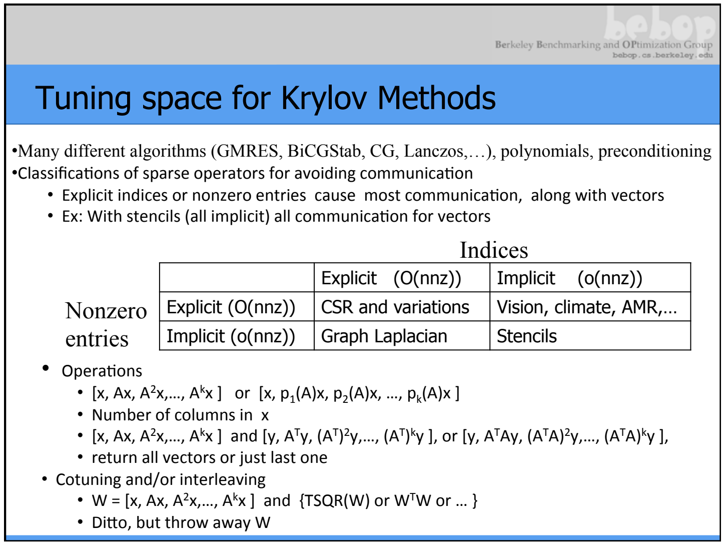


President Obama cites Communication-Avoiding Algorithms in the FY 2012 Department of Energy Budget Request to Congress:

"New Algorithm Improves Performance and Accuracy on Extreme-Scale Computing Systems. **On modern computer architectures, communication between processors takes longer than the performance of a floating point arithmetic operation by a given processor.** ASCR researchers have developed a new method, derived from commonly used linear algebra methods, to **minimize communications between processors and the memory hierarchy, by reformulating the communication patterns specified within the algorithm**. This method has been implemented in the TRILINOS framework, a highly-regarded suite of software, which provides functionality for researchers around the world to solve large scale, complex multi-physics problems."

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

CA-GMRES (Hoemmen, Mohiyuddin, Yelick, JD)
"Tall-Skinny" QR (Grigori, Hoemmen, Langou, JD)



Tuning space for Krylov Methods

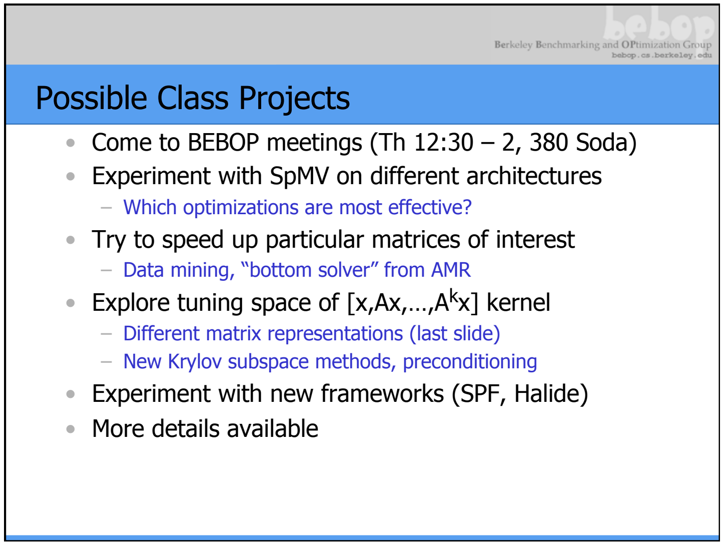
•Many different algorithms (GMRES, BiCGStab, CG, Lanczos,…), polynomials, preconditioning

•Classifications of sparse operators for avoiding communication

- Explicit indices or nonzero entries cause most communication, along with vectors
- Ex: With stencils (all implicit) all communication for vectors

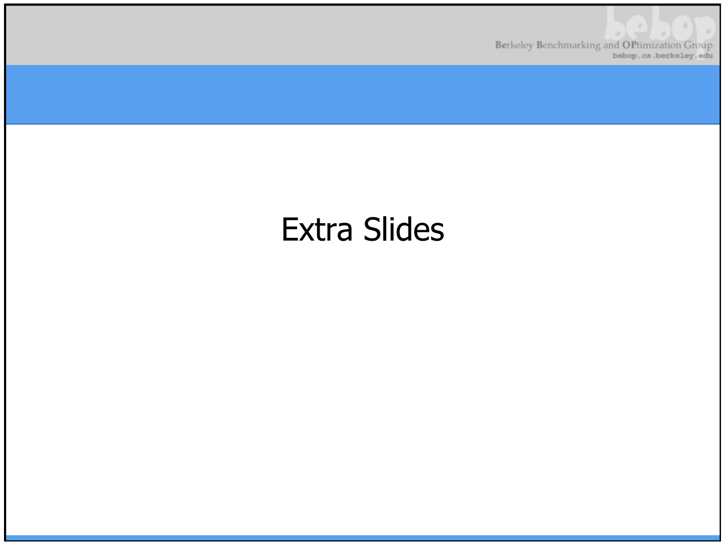
		Indices	
		Explicit (O(nnz))	Implicit (o(nnz))
Nonzero entries	Explicit (O(nnz))	CSR and variations	Vision, climate, AMR,…
	Implicit (o(nnz))	Graph Laplacian	Stencils

- Operations
  - $[x, Ax, A^2x,\ldots, A^kx]$ or $[x, p_1(A)x, p_2(A)x, \ldots, p_k(A)x]$
  - Number of columns in x
  - $[x, Ax, A^2x,\ldots, A^kx]$ and $[y, A^Ty, (A^T)^2y,\ldots, (A^T)^ky]$, or $[y, A^TAy, (A^TA)^2y,\ldots, (A^TA)^ky]$,
  - return all vectors or just last one
- Cotuning and/or interleaving
  - $W = [x, Ax, A^2x,\ldots, A^kx]$ and $\{TSQR(W)$ or W^TW or … $\}$
  - Ditto, but throw away W



Possible Class Projects

- Come to BEBOP meetings (Th 12:30 – 2, 380 Soda)
- Experiment with SpMV on different architectures
  - Which optimizations are most effective?
- Try to speed up particular matrices of interest
  - Data mining, “bottom solver” from AMR
- Explore tuning space of $[x,Ax,\ldots,A^kx]$ kernel
  - Different matrix representations (last slide)
  - New Krylov subspace methods, preconditioning
- Experiment with new frameworks (SPF, Halide)
- More details available



Extra Slides