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, PETSc...) 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 (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 a few parameters
    - Matrix dimension, size of FFT, etc.

Tuning Register Tile Sizes (Dense Matrix Multiply)

Example: Select a Matmul Implementation

Example: Support Vector Classification
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

References

- Statistical Models for Empirical Search-Based Performance Tuning
- Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning

- More references
  - Machine Learning for Predictive Autotuning with Boosted Regression Trees,
  - 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
Linear Programming Matrix

A Sparse Matrix You Encounter Every Day

Matrix-vector multiply kernel: \[ y(i) \leftarrow y(i) + A(i,j)x(j) \]
for each row \( i \)
for \( k = \text{ptr}[i] \) to \( \text{ptr}[i+1]-1 \) do
\[ y[i] = y[i] + \text{val}[k] \times \text{ind}[k] \]

SpMV with Compressed Sparse Row (CSR) Storage

Example: The Difficulty of Tuning

- \( n = 21200 \)
- \( \text{nnz} = 1.5 \text{ M} \)
- kernel: SpMV
- Source: NASA structural analysis problem
Example: The Difficulty of Tuning

- $n = 21200$
- $\text{nnz} = 1.5 \text{ M}$
- 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 $rc \gg 1$, 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

Register Profile: Itanium 2
SpMV Performance (Matrix #2): Generation 1

<table>
<thead>
<tr>
<th>Processor</th>
<th>Power3 - 13%</th>
<th>Power4 - 14%</th>
<th>Itanium 2 - 31%</th>
<th>Itanium 1 - 7%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power3</td>
<td>1.07</td>
<td>1.15</td>
<td>1.52</td>
<td>1.00</td>
</tr>
<tr>
<td>Power4</td>
<td>1.11</td>
<td>1.22</td>
<td>1.57</td>
<td>1.00</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>1.08</td>
<td>1.06</td>
<td>1.43</td>
<td>1.00</td>
</tr>
<tr>
<td>Itanium 1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

195 Mflop/s
100 Mflop/s
703 Mflop/s
469 Mflop/s
225 Mflop/s
103 Mflop/s
1.1 Gflop/s
276 Mflop/s

Register Profiles: IBM and Intel IA-64

<table>
<thead>
<tr>
<th>Processor</th>
<th>Power3 - 17%</th>
<th>Power4 - 16%</th>
<th>Itanium 2 - 33%</th>
<th>Itanium 1 - 8%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power3</td>
<td>1.52</td>
<td>1.61</td>
<td>1.55</td>
<td>1.11</td>
</tr>
<tr>
<td>Power4</td>
<td>1.57</td>
<td>1.34</td>
<td>1.97</td>
<td>1.08</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>1.43</td>
<td>3.54</td>
<td>2.58</td>
<td>1.22</td>
</tr>
<tr>
<td>Itanium 1</td>
<td>1.00</td>
<td>1.61</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

252 Mflop/s
122 Mflop/s
820 Mflop/s
459 Mflop/s
247 Mflop/s
107 Mflop/s
1.2 Gflop/s
190 Mflop/s

SpMV Performance (Matrix #2): Generation 2

<table>
<thead>
<tr>
<th>Processor</th>
<th>Ultra 2i - 9%</th>
<th>Ultra 3 - 5%</th>
<th>Pentium III-M - 15%</th>
<th>Pentium III - 19%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>1.52</td>
<td>1.48</td>
<td>1.73</td>
<td>1.00</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>1.46</td>
<td>1.51</td>
<td>1.64</td>
<td>1.00</td>
</tr>
<tr>
<td>Pentium III-M</td>
<td>1.29</td>
<td>1.32</td>
<td>2.05</td>
<td>1.00</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

63 Mflop/s
35 Mflop/s
109 Mflop/s
53 Mflop/s
96 Mflop/s
42 Mflop/s
120 Mflop/s
58 Mflop/s

Register Profiles: Sun and Intel x86

<table>
<thead>
<tr>
<th>Processor</th>
<th>Ultra 2i - 11%</th>
<th>Ultra 3 - 5%</th>
<th>Pentium III-M - 15%</th>
<th>Pentium III - 21%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>1.58</td>
<td>1.55</td>
<td>1.73</td>
<td>1.00</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>1.48</td>
<td>1.51</td>
<td>1.64</td>
<td>1.00</td>
</tr>
<tr>
<td>Pentium III-M</td>
<td>1.29</td>
<td>1.32</td>
<td>2.05</td>
<td>1.00</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

72 Mflop/s
35 Mflop/s
90 Mflop/s
50 Mflop/s
108 Mflop/s
42 Mflop/s
122 Mflop/s
58 Mflop/s
Another example of tuning challenges

- More complicated non-zero structure in general
- \( N = 16614 \)
- \( \text{NNZ} = 1.1M \)

Zoom in to top corner

- More complicated non-zero structure in general
- \( N = 16614 \)
- \( \text{NNZ} = 1.1M \)

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!

- 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 $r \times c$ block size
  - **Off-line benchmark**
    - Precompute $\text{Mflops}(r,c)$ using dense $A$ for each $r \times c$
    - Once per machine/architecture
  - **Run-time "search"**
    - Sample $A$ to estimate $\text{Fill}(r,c)$ for each $r \times c$
  - **Run-time heuristic model**
    - Choose $r$, $c$ to minimize $\text{time} \sim \text{Fill}(r,c) / \text{Mflops}(r,c)$

Accurate and Efficient Adaptive Fill Estimation

- **Idea:** Sample matrix
  - Fraction of matrix to sample: $s \in [0,1]$
  - Cost $\sim O(s \cdot \text{nnz})$
  - 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

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

See p. 375 of Vuduc's thesis for matrices

Accuracy of the Tuning Heuristics (Matrix No. 1)

- Performance (Mflops)
- Matrix No.: D, FBM, FBM (var), Other, LP
- 1 to 44

Accuracy of the Tuning Heuristics (Matrix No. 2)

- Performance (Mflops)
- Matrix No.: D, FBM, FBM (var), Other, LP
- 1 to 44
Upper Bounds on Performance for blocked SpMV

- \( P = \frac{\text{flops}}{\text{time}} \)
  - Flops = \( 2 \cdot \text{nnz}(A) \)
- 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” \( \alpha \) at \( L_c \) cache & \( \alpha_{\text{mem}} \)
  - e.g., Saavedra-Barrera and PMaC MAPS benchmarks

\[
\text{Time} \geq \sum_{i} \alpha_{i} \cdot \text{Hits}_{i} + \alpha_{\text{max}} \cdot \text{Misses}_{\text{mem}}
\]

\[
= \alpha_{L} \cdot \text{Loads} + \sum_{i} (\alpha_{i} - \alpha_{L}) \cdot \text{Misses}_{i} + (\alpha_{\text{max}} - \alpha_{L}) \cdot \text{Misses}_{\text{mem}}
\]
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
  - $A^TAx, A^TA^TAx$: 4x over CSR, 1.8x over RB
  - $A^TA$: 2x over CSR, 1.5x over RB
  - $[A^TA, A^TA^TA, ..., A^TA^TA^TA]$
Cache Optimizations for $AA^T \times x$

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

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

- **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, $Ax=\lambda x$, SVD
    - Used when SpMV only affordable operation on $A$
      - Krylov Subspace Methods
      - How to choose
        - For $Ax=b$: www.netlib.org/templates/templates.html
        - For $Ax=\lambda x$: 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?**
  - Yes: Try CG
  - No: Try MINRES
- **A available?**
  - Yes: Try GMRES
  - No: Try CGS or Bi-CGStab or GMRES(k)
- **Largest and smallest eigenvalues known?**
  - Yes: Try CG on normal equations
  - No: Try GMRES
- **A definite?**
  - Yes: Try CG
  - No: Try CG with Chebyshev Accel.

All methods (GMRES, CGS, CG…) depend on SpMV (or variations…)
See [www.netlib.org/templates/Templates.html](http://www.netlib.org/templates/Templates.html) for details

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

Before: Green + Red
After: Green + Blue

Before: Green + Red
After: Green + Blue
How do permutations affect algorithms?

- $A = \text{original matrix, } A^P = A \text{ with permuted rows, columns}$
- Naive approach: permute $x$, multiply $y = A^P x$, permute $y$
- Faster way to solve $Ax=b$
  - Write $A^P = P^T A P$ where $P$ is a permutation matrix
  - Solve $A^P x^P = P^T b$ for $x^P$, then let $x = P x^P$
  - 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!
  - $A^P x^P = \lambda x^P$ implies $Ax = \lambda x$ where $x = P x^P$
- 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)
- IBM QS20 Cell Blade
- Sun T2+ T5140 (Victoria Falls)

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

8 threads
8 threads

Source: Sam Williams

Multicore SMPs Used
(threads)

- Intel Xeon E5345 (Clovertown)
- AMD Opteron 2356 (Barcelona)
- Sun T2+ T5140 (Victoria Falls)
- IBM QS20 Cell Blade

8 threads
8 threads

128 threads
16 threads

Source: Sam Williams

*SPEs only*
Multicore SMPs Used (Non-Uniform Memory Access - NUMA)

- AMD Opteron 2356 (Barcelona)
- Intel Xeon E5345 (Clovertown)
- IBM QS20 Cell Blade
- Sun T2+ T5140 (Victoria Falls)

Multicore SMPs Used (peak double precision flops)

- AMD Opteron 2356 (Barcelona)
- Intel Xeon E5345 (Clovertown)
- IBM QS20 Cell Blade
- Sun T2+ T5140 (Victoria Falls)

Multicore SMPs Used (Total DRAM bandwidth)

- AMD Opteron 2356 (Barcelona)
- Intel Xeon E5345 (Clovertown)
- IBM QS20 Cell Blade
- Sun T2+ T5140 (Victoria Falls)

Source: Sam Williams

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

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

Well Structured (sorted by nonzeros/row)

Poorly Structured: hodgepodge

Extremely Aspect Ratio: (linear programming)

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

- 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?

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

Source: Sam Williams

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)**

- 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)

**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

**Prefetch for SpMV**

- 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;
}

**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 footprint.**

Source: Sam Williams

Matrix Compression Strategies

- Where possible we may encode indices with less than 32 bits
- We may also select different matrix formats
  - 1x1
  - 1x2
  - 2x1
  - 3x2

- 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)
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

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?

Source: Sam Williams
Auto-tuned SpMV Performance
(max speedup)

- 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?

Optimized Sparse Kernel Interface - pOSKI
bebop.cs.berkeley.edu/poski
- Provides sparse kernels automatically tuned for user’s matrix & machine
  - BLAS-style functionality: SpMV, Ax & ATy
  - Hides complexity of run-time tuning
- Based on OSKI – bebop.cs.berkeley.edu/oski
  - Autotuner for sequential sparse matrix operations:
    - SpMV (Ax and ATx), solve sparse triangular systems, ...
  - So far pOSKI only does multicore optimizations of SpMV
    - Class projects!
  - Up to 4.5x faster SpMV (Ax) on Intel Sandy Bridge E
- On-going work by the Berkeley Benchmarking and Optimization (BeBop) group

Optimizations in pOSKI, so far
- Fully automatic heuristics for
  - Sparse matrix-vector multiply (Ax, ATx)
    - 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 under development
  - Cache-level blocking, Reordering (RCM, TSP), variable block structure, index compressing, Symmetric storage, etc.

How the pOSKI Tunes (Overview)

Library Install-Time (offline) → Application Run-Time

- User’s hints
- Workload from program monitoring
- History
- Empirical & Heuristic Search
- Submatrix
- Data Struct & Code
- Selected Code Variants
- Benchmark
- Generated Code Variants
- Sample Dense Matrix

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

To user: Matrix handle for kernel calls
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

```c
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,
                                           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++ )
  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

```c
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

```c
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);
```

```c
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

```c
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

```c
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
    - #partitions = k×#threads
  - Partitioning model (OneD, SemiOneD, TwoD)
    - Default: OneD, #partitions = #threads

**Matrix:**

```c
poski_partitionarg_t *pmat = poski_PartitionMatHints(SemiOneD, 16);
poski_mat_t A_tunable = poski_CreateMatCSR( ... , pmat, ... );
```

**Vector:**

```c
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 float-point
- MKL Sparse BLAS Level 2: mkl_dcsrmv()

Performance in GFlops

<table>
<thead>
<tr>
<th></th>
<th>OSLK</th>
<th>MKL</th>
<th>pOSKI</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense</td>
<td>1.0x</td>
<td>4.8x</td>
<td>3.2x</td>
</tr>
<tr>
<td>kkt_power</td>
<td>1.0x</td>
<td>3.2x</td>
<td>2.2x</td>
</tr>
<tr>
<td>bone</td>
<td>1.8x</td>
<td>4.5x</td>
<td>4.1x</td>
</tr>
<tr>
<td>largebasis</td>
<td>1.0x</td>
<td>2.9x</td>
<td>4.5x</td>
</tr>
<tr>
<td>tsopf</td>
<td>1.0x</td>
<td>1.0x</td>
<td>4.7x</td>
</tr>
<tr>
<td>ldoor</td>
<td>1.0x</td>
<td>1.0x</td>
<td>4.7x</td>
</tr>
<tr>
<td>wiki</td>
<td>1.0x</td>
<td>1.0x</td>
<td>4.7x</td>
</tr>
</tbody>
</table>

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 [x₁, ..., xₖ]
    - Find best solution x as linear combination of [x₁, ..., xₖ]
- Main cost is k SpMs
- 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: \([Ax, A^2x, ..., A^kx]\)

- Replace k iterations of \(y = Ax\) with \([Ax, A^2x, ..., A^kx]\)

\[
\begin{align*}
A^3x & \cdot \cdot \cdot \\
A^2x & \cdot \cdot \cdot \\
A\cdot x & \cdot \cdot \cdot \\
x & \cdot \cdot \cdot \\
1 & 2 & 3 & 4 & ... & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 32
\end{align*}
\]

- 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 $y = A\cdot x$ 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 \([Ax, A^2x, \ldots, A^kx]\)

- 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 \(y = Ax\) 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 \(y = Ax\) 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: \([Ax, A^2x, ..., A^kx]\)

- Replace \(k\) iterations of \(y = Ax\) with \([Ax, A^2x, ..., A^kx]\)
- 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 $\rightarrow$ (hyper)graph partitioning
- Top-to-bottom processing $\rightarrow$ Traveling Salesman Problem

Parallel Algorithm

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

Locally Dependent Entries for $[x, Ax, \ldots, A^kx]$, A tridiagonal

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

Remotely Dependent Entries for $[x, Ax, \ldots, A^kx]$, A tridiagonal

- 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

Reduce redundant work by half

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

Remote 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=λx
  - Does k SpMV 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 Mark Hoemmen’s PhD thesis, other papers at bebop.cs.berkeley.edu

Minimizing Communication of GMRES to solve Ax=b

GMRES: find x in span{b,Ab,…,A^k b} minimizing || Ax-b ||^2

Cost of k steps of standard GMRES vs new GMRES

<table>
<thead>
<tr>
<th></th>
<th>Standard GMRES</th>
<th>Communication-avoiding GMRES</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i=1 to k</td>
<td>w = A · v(i-1)</td>
<td>W = [ v, Av, A^2 v, …, A^k v ]</td>
</tr>
<tr>
<td>MGS(w, v(0),…,v(i-1))</td>
<td></td>
<td>[Q,R] = TSQR(W) “Tall Skinny QR”</td>
</tr>
<tr>
<td>update v(i), H</td>
<td>Build H from R, solve LSQ problem</td>
<td></td>
</tr>
<tr>
<td>endfor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>solve LSQ problem with H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequential: #words_moved = O(nnnz) from SpMV + O(k^2 n) from MGS
Parallel: #messages = O(1) from SpMV + O(k log p) from MGS

Oops – W from power method, precision lost!
Speed ups of GMRES on 8-core Intel Clovertown
Requires co-tuning kernels [MHDY09]

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

- Matrix powers
- kernel
- QSOR
- Block Gram-Schmidt
- Small dense operations
- Sparse matrix-vector product
- Modified Gram-Schmidt

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.”


CA-BiCGStab

```
For j = 0 to m - 1, Do
    r <- b - A w
    Compute the (m + j) Gram vector.
    For i = 0 to m - j - 1, Do
        H <- [H, (b_i, H_{m+j-i})]
    EndDo
    For i = 0 to m - j, Do
        d_i <- H_{m+i} * r
    EndDo
    For i = 0 to m - j, Do
        F_i <- d_i /
    EndDo
    Computed the (m + j) Gram matrix.
    y <- Q' F'
    Compute the (m + j) Gram vector.
    For i = 0 to m - j - 1, Do
        H <- [H, (b_i, H_{m+j-i})]
    EndDo
    For i = 0 to m - j, Do
        d_i <- H_{m+i} * r
    EndDo
    For i = 0 to m - j, Do
        F_i <- d_i /
    EndDo
```

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

<table>
<thead>
<tr>
<th>Indices</th>
<th>Explicit (O(mn2))</th>
<th>Implicit (o(mn2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonzero entries</td>
<td>CSR and variations</td>
<td>Vision, climate, AFR,...</td>
</tr>
<tr>
<td>Operations</td>
<td>Explicit (O(mn2))</td>
<td>Implicit (o(mn2))</td>
</tr>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td>Graph Laplacian</td>
<td>Stencils</td>
</tr>
</tbody>
</table>

33
Possible Class Projects

- Come to BEBOP meetings (W 12 – 1:30, 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 \([x, Ax, ..., A^k x]\) kernel
  - Different matrix representations (last slide)
  - New Krylov subspace methods, preconditioning
- Experiment with new frameworks (SPF, Halide)
- More details available

Extra Slides