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

Richard Vuduc
Eun-Jin Im
James Demmel
Katherine Yelick
Attila Gyulassy
Chris Hsu
Shoaib Kamil
Benjamin Lee
Hyun-Jin Moon
Rajesh Nishtala

Automatic Performance Tuning of Sparse Matrix Kernels

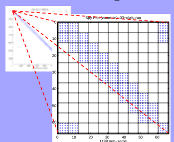
Problem Context

Sparse kernel performance depends on both the matrix and hardware platform.

- Challenges in tuning sparse code
  - Typical uniprocessor performance < 10% peak
    - *Indirect, irregular memory accesses*
    - *High bandwidth requirements, poor instruction mix*
  - Hardware complexity is increasing
    - *Microprocessor performance difficult to model*
    - *Widening processor-memory gap; deep memory hierarchies*
  - Performance depends on **architecture, kernel, and matrix**
- Goal: Automatic tuning of sparse kernels
  - Choose best data structure and implementation for given kernel, sparse matrix, and machine
    - *Matrix known only at run-time (in general)*
  - Evaluate code against architecture-specific upper bounds

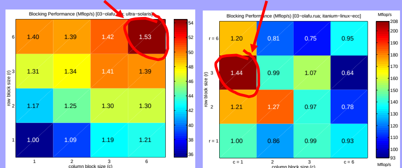
Observations

Performance depends *strongly* on both the matrix and hardware platform.



Sparse matrix example—A 6x6 blocked storage format appears to be the most natural choice for this matrix for a sparse matrix-vector multiply (SpMxV) implementation…

Ultra 2i: 6x6 Itanium: 3x1



Architecture dependence—Sixteen *r* x *c* blocked compressed sparse row implementations of SpMxV, each color coded by performance (Mflop/s) and labeled by speedup over the unblocked (1 x 1) code for the sparse matrix above on two platforms: 333 MHz Ultra 2i (left) and 800 MHz Itanium (right). *The best block size is not always 6x6!*

Approach to Automatic Tuning

For each kernel, *identify and generate* a space of implementations, and *search* for the best one.

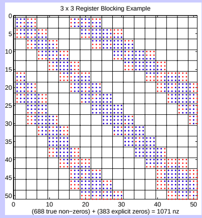
- Implementation space
  - Conceptually, the set of "interesting" implementations
  - Depends on kernel and input
  - May vary: instruction mix and order, memory access patterns, data structures and precisions, mathematical formulation, …
- Search using models and experiments
  - Either off-line, on-line, or combination
- Successful examples
  - Dense linear algebra: ATLAS/PHiPAC
  - Signal processing: FFTW; SPIRAL
  - MPI collective operations (Vadhiyar & Dongarra, 2001)

Example: Choosing a Block Size

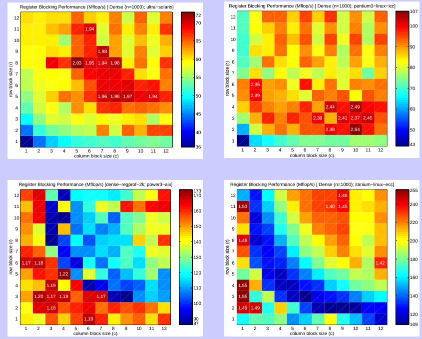
The SPARSITY system (Im & Yelick, 1999) applies the methodology to *y=Ax*.

- Consider sparse matrix-vector multiply (SpMxV)
- Implementation space
  - Set of r x c block sizes
  - Fill in explicit zeros
- Search
  - **Off-line benchmarking** (once per architecture)
    - Measure *Dense Performance (r,c)*, in Mflop/s, of dense matrix in sparse *r x c* format
  - **Run-time estimation** (when matrix is known)
    - Estimate *Fill Ratio (r,c)*: (# stored non-zeros) / (# true non-zeros)
    - Choose *r x c* to maximize

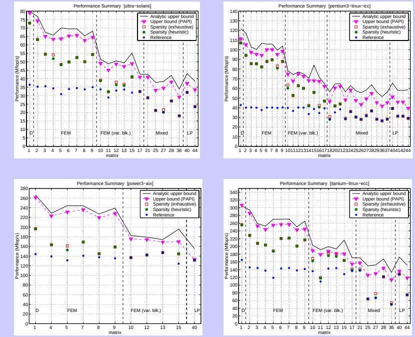
$$\text{Estimated Performance}(r,c) = \frac{\text{Dense Performance}(r,c)}{\text{Fill Ratio}(r,c)}$$



Filling in zeros—True non-zeros (•) and explicit zeros (•); fill ratio=1.5.



Off-line benchmarking—Performance (Mflop/s) for a dense matrix in sparse format on four architectures (clockwise from upper-left): Ultra 2i-333, Pentium III-500, Power3-375, Itanium-800. *Performance is a strong function of the hardware platform.*

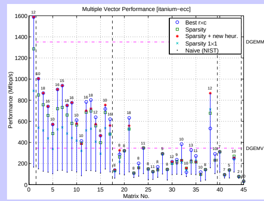


Experimental results—Performance (Mflop/s) on a set of 44 benchmark matrices from a variety of applications. *Speedups of 2.5x are possible.* [SC'02]

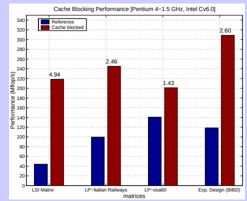
Exploiting Matrix Structure

Additional techniques for *y=Ax*, sparse triangular solve, and A^TAx.

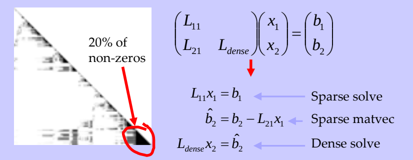
- Sparse matrix-vector multiply
  - Register-level blocking (up to 2.5x speedups)
  - Symmetry (up to 2x speedup)
  - Diagonals, bands (up to 2.2x)
  - Splitting for variable block structure (1.3x—1.7x)
  - Reordering to create dense blocks + splitting (up to 2x)
  - Cache blocking (1.5x—5x)
  - Multiple vectors (2—7x)
  - And combinations…
- Sparse triangular solve
  - Hybrid sparse/dense data structure (1.2x—1.8x)
- Higher-level sparse kernels
  - AA^Tx, A^TAx (1.2—4.2x)
  - RAR^T, A^kx, …

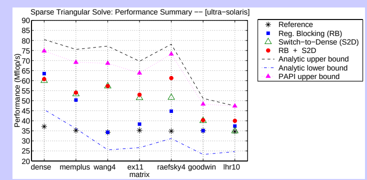


Multiple vectors—Significant speedups are possible when multiplying by several vectors (800 MHz Itanium; DGEMM n=k=2000, m=32).

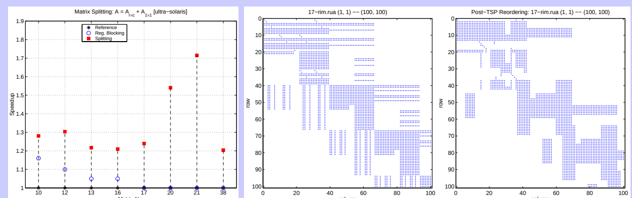


Cache blocking—Performance on a Pentium 4 for information retrieval and linear programming matrices, with up to *5x speedups*.





Sparse triangular solve—(*Top-left*) Triangular factors from sparse LU often have a large dense trailing triangle. (*Top-right*) The matrix can be partitioned into sparse (L_{11}, L_{21}) and dense (L_{dense}) parts. (*Bottom*) Performance improvements from register blocking the sparse part, and calling a tuned vendor BLAS routine (TRSM) for the dense solve step. [ICS/POHLL '02]

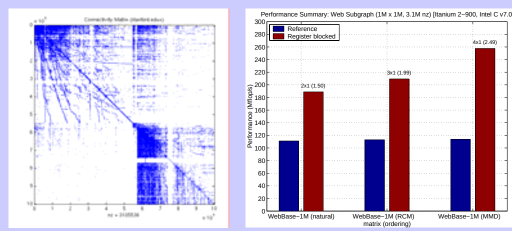


Splitting and reordering—(*Left*) Speedup after splitting a matrix (possibly after reordering) into a blocked part and an unblocked part to avoid fill. (*Middle, right*) Dense blocks can be created by a judicious reordering—on matrix 17, we used a traveling salesman problem formulation due to Pinar (1997).

BeBOP: Current and Future Work

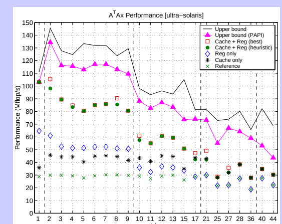
Understanding the impact on higher-level kernels, algorithms, and applications.

- Design and implementation of a library based on the Sparse BLAS; new heuristics for efficiently choosing optimizations.
- Study of performance implications for higher-level algorithms (*e.g.*, block Lanczos)
- New sparse kernels (*e.g.*, powers A^k, triple product RAR^T)
- Integrating with applications (*e.g.*, DOE SciDAC codes)
- Further automation: generating implementation generators
- Using bounds to evaluate current and future architectures



Application matrices: Web connectivity matrix—Speeding up SpMxV for a web subgraph (*left*) using register blocking and reordering (*right*) on a 900 MHz Itanium 2.

$$y = AA^Tx = (a_1 \cdots a_m)\begin{pmatrix} a_1^T \\ \vdots \\ a_m^T \end{pmatrix} x = \sum_{k=1}^{m} a_k (a_k^T x)$$



Sparse AA^Tx, A^TAx—(*Left*) A can be brought through the memory hierarchy only once: for each column a_k of A, compute a dot product followed by a vector scale ("axpy"). (*Right*) This cache optimized implementation can be naturally combined with register blocking.