Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply

Richard Vuduc, James Demmel, Katherine Yelick
Shoaib Kamil, Rajesh Nishtala, Benjamin Lee
November 20, 2002

Berkeley Benchmarking and OPtimization (BeBOP) Project
www.cs.berkeley.edu/~richie/bebop
Computer Science Division, U.C. Berkeley
Application performance dominated by a few computational kernels

Performance tuning today
- Vendor-tuned libraries (e.g., BLAS) or user hand-tunes
- Automatic tuning (e.g., PHiPAC/ATLAS, FFTW/SPIRAL/UHFFT)

Why is tuning hard in the sparse case?
- Typical sparse matrix-vector multiply (SpM×V) performance: less than 10% of machine peak
- Compared to dense code, sparse code has ...
  - higher bandwidth requirements (extra storage)
  - poor locality (indirect, irregular memory access)
  - poor instruction mix (data structure manipulation)
- Performance depends on architecture, kernel, matrix
Example: Matrix $\text{olafu}$

Spy Plot (1, 1): [03–olafu.rua]

$N = 16146$

nnz = 1.0M

Kernel = SpM $\times V$

A natural choice:

$6 \times 6$ blocked CSR storage. Is it the best choice?

Experiment:

Measure performance of all $r \times c$ block sizes for $r, c \in \{1, 2, 3, 6\}$.
The Need for Search: Performance on Itanium

<table>
<thead>
<tr>
<th>row block size (r)</th>
<th>column block size (c)</th>
<th>Blocking Performance (Mflop/s) [03-olafu.rua; itanium-linux-ecc]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.86</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.93</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.27</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.78</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1.44</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0.64</td>
</tr>
</tbody>
</table>

(Peak machine speed: 3.2 Gflop/s)
Key Questions and Conclusions

- **How do we choose the best tuning parameters automatically?**
  - New heuristic for choosing optimal (or near-optimal) block sizes

- **What are the limits on performance for SpM×V?**
  - Derive performance upper and lower bounds for blocking
  - Show we can get within 20% of upper bound, placing limits on improvement from more “low-level” tuning

- **Where are the new opportunities (kernels, techniques) for achieving higher performance?**
  - Identify cases in which blocking does and does not work
  - Discuss opportunities for exploiting other structures and reuse in additional kernels
Approach to Automatic Tuning

- For each kernel,
  - Identify and generate a space of implementations
  - Search to find the fastest one (using models, experiments)
Approach to Automatic Tuning

- For each kernel,
  - Identify and generate a space of implementations
  - Search to find the fastest one (using models, experiments)

- The SPARSITY system for SpM\times V [Im & Yelick ’99]
  - Interface
    - Input: Your sparse matrix (CSR)
    - Output: Data structure + routine tuned to your matrix & machine
  - Implementation space
    - register level blocking \((r \times c)\)
    - cache blocking, multiple vectors, reordering, splitting, …
  - Search
    - Off-line: experiments/benchmarking
    - Run-time: estimate matrix properties (“search”) and evaluate model
Approach to Automatic Tuning

- For each kernel,
  - *Identify* and *generate* a space of implementations
  - *Search* to find the fastest one (using models, experiments)

- The SPARSITY system for SpMV [Im & Yelick ’99]
  - Interface
    - Input: Your sparse matrix (CSR)
    - Output: Data structure + routine tuned to your matrix & machine
  - Implementation space
    - register level blocking \((r \times c)\)
    - cache blocking, multiple vectors, reordering, splitting, …
  - Search
    - Off-line: experiments/benchmarking
    - Run-time: estimate matrix properties (“search”) and evaluate model
Register-Level Blocking (SPARSITY)

- Store dense $r \times c$ blocks
  - BCSR with uniform blocks
  - Reduce storage and bandwidth requirements

- Fully unroll block multiplies
  - Improves register reuse, scheduling

4x3 Register Blocking Example

nz = 598
Register-Level Blocking (SPARSITY)

- Store dense $r \times c$ blocks
  - BCSR with uniform blocks
  - Reduce storage and bandwidth requirements
- Fully unroll block multiplies
  - Improves register reuse, scheduling
- Fill-in zeros: trade-off extra flops for better efficiency
  - 1.3–2.5x speedup on FEM matrices
Register Profile: Sun Ultra 2i

Register Blocking Performance (Mflop/s) [ Dense (n=1000); ultra−solaris]

Top 10 codes labeled by speedup over unblocked (1×1) code. Max speedup = 2.03 (8×5).
333 MHz Sun Ultra 2i (2.03)

500 MHz Intel Pentium III (2.54)

375 MHz IBM Power3 (1.22)

800 MHz Intel Itanium (1.55)

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.9/36
Define **fill ratio**: \( \text{number of stored values} / \text{number of true non-zeros} \)

Once per architecture:

- For all \( r, c \), measure performance (Mflop/s) of dense matrix in sparse \( r \times c \) blocked format

At **run-time**, when matrix is known:

- Estimate fill ratio for all \( r, c \)
- Choose \( r, c \) that maximizes
  \[
  \frac{\text{Dense performance (Mflop/s)}}{\text{Fill ratio}}
  \]
- Convert from input format to \( r_{\text{max}} \times c_{\text{max}} \) BCSR

In practice, total run-time cost is 10–40 SpM×Vs
- \((\text{Re})\text{building the matrix}: \approx 70–85\% \text{ of cost}\)
Performance Bounds

- How close are we to the speed limit?

- *Upper-bounds* on performance: \((\text{flops}) / (\text{time})\) [Mflop/s]
  - Flops \(\approx 2 \times (\text{number of true non-zeros})\)
  - Model of execution time
    - Assume memory limited: ignore cost of non-memory ops.
    - Assume known number of hits at each cache/memory level
    - Charge full latency \((\alpha_i)\) for hits at each cache level \(i\), *e.g.*

\[
T = (L1 \text{ hits})\alpha_1 + (L2 \text{ hits})\alpha_2 + \ldots
\]

- Need *lower bound* on time, *i.e.*, *lower bound* on misses
  - Ignore conflict misses on vectors
  - Compulsory miss on each read of matrix non-zero and index
    (accounting for \(r, c\) and fill)
  - Compulsory miss on each vector element
  - Account for line size
Cache Miss Bound Verification: Sun Ultra 2i (L1)

L1 Misses -- [ultra-solaris]

- Upper bound
- PAPI
- Lower Bound

Matrix:

- no. of misses (millions)
- matrix
Where in Memory is the Time Spent?

Execution Time Model (Model Hits/Misses) --- Where is the Time Spent?

Fraction of Cycles (exhaustive best; average over matrices)

Ultra 2i (L1/L2)
Pentium III (L1/L2)
Power3 (L1/L2)
Itanium (L1/L2/L3)

Platform

L1
L2
L3
Mem
Overview of Performance Results

- **Experimental setup**
  - Four machines: Ultra 2i, Pentium III, Power3, Itanium
  - 44 matrices (MatrixMarket and UFL): dense, FEM, assorted, linear programming
  - Measured misses and cycles using PAPI
  - Reference: unblocked (1×1)

- **Main observations**
  - SPARSITY vs. reference: up to 2.5x faster, especially on FEM
  - Block size selection: chooses within 10% of best
  - SPARSITY performance typically within 20% of upper-bound
  - SPARSITY least effective on Power3
Performance Results: Sun Ultra 2i

Performance Summary --- [ultra-solaris]

Reference Sparsity (heuristic) Sparsity (exhaustive) Analytic upper bound Upper bound (PAPI) Analytic lower bound

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.16/36
Performance Results: IBM Power3

Performance Summary -- [power3−aix]

DGEMV (n=2000): 260 Mflop/s
Performance Results: Intel Itanium

Performance Summary -- [itanium-linux-ecc]

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.19/36
Sometimes faster to fill in many zeros

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Reference (Mflop/s)</th>
<th>Best Perf. (Mflop/s)</th>
<th>Block size</th>
<th>Fill ratio</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>109</td>
<td>119</td>
<td>4×1</td>
<td>1.70</td>
<td>Itanium</td>
</tr>
<tr>
<td>13</td>
<td>40</td>
<td>60</td>
<td>3×3</td>
<td>1.52</td>
<td>Pentium 3</td>
</tr>
<tr>
<td>17</td>
<td>136</td>
<td>141</td>
<td>3×1</td>
<td>1.59</td>
<td>Itanium</td>
</tr>
<tr>
<td>27</td>
<td>67</td>
<td>78</td>
<td>3×1</td>
<td>1.94</td>
<td>Itanium</td>
</tr>
<tr>
<td>27</td>
<td>20</td>
<td>22</td>
<td>2×1</td>
<td>1.53</td>
<td>Ultra 2i</td>
</tr>
<tr>
<td>29</td>
<td>28</td>
<td>28</td>
<td>2×2</td>
<td>1.98</td>
<td>Pentium 3</td>
</tr>
<tr>
<td>36</td>
<td>50</td>
<td>54</td>
<td>3×1</td>
<td>2.31</td>
<td>Itanium</td>
</tr>
</tbody>
</table>
Conclusions

- Tuning can be difficult, even when matrix structure is known
  - Performance is a complicated function of architecture and matrix

- New heuristic for choosing block size selects optimal implementation, or near-optimal (performance within 5–10%)

- Limits of low-level tuning for blocking are near
  - Performance is often within 20% of upper-bound, particularly with FEM matrices
  - Unresolved: closing the gap on the Power3
Directions

- Further performance improvements
  - Symmetry (1.5–2x), diagonals/bands (1.2–2x), splitting for variable block structure (1.3–1.7x), reordering to create dense structure (1.7x), cache blocking (1.5–4x), multiple vectors (2–7x), and combinations . . .
  - How to choose optimizations & tuning parameters?

- Sparse triangular solve (ICS’02/POHLL)

- Higher-level kernels that permit reuse
  - $A^T Ax$ (1.5–3x), $Ax$ and $A^T y$ simultaneously, $A^k x$, $RAR^T$, . . .

- Implications for architecture
  - Improvements for Power3?
  - Latency vs. bandwidth (see paper)

- Toward an automatically tuned sparse matrix library
  - Code generation via sparse compilers (Bernoulli; Bik)
  - Can extend Sparse BLAS by one routine to support tuning
Related Work

- **Automatic tuning systems**
  - PHiPAC [BACD97], ATLAS [WPD01], SPARSITY [Im00]
  - FFTW [FJ98], SPIRAL [PSVM01], UHFFT [MMJ00]
  - MPI collective ops (Vadhiyar, et al. [VFD01])

- **Code generation**
  - Sparse compilers (Bik [BW99], Bernoulli [Sto97])
  - Generic programming (Blitz++ [Vel98], MTL [SL98], GMCL [Neu98], ...)
  - FLAME [GGHvdG01]

- **Sparse performance modeling and tuning**
  - Temam and Jalby [TJ92]
  - Toledo [Tol97], White and Sadayappan [WS97], Pinar [PH99]
  - Navarro [NGLPJ96], Heras [HPDR99], Fraguela [FDZ99]
  - Gropp, et al. [GKKS99], Geus [GR99]

- **Compilers (analysis and models)**
  - CROPS (UCSD/Carter, Ferrante, et al.)
  - TUNE (Chatterjee, et al.)
Example: No Big Surprises on Sun Ultra 2i

Blocking Performance (Mflop/s) [03–olafu.rua; ultra–solaris]

<table>
<thead>
<tr>
<th>row block size (r)</th>
<th>column block size (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1.40 1.39 1.42 1.53</td>
</tr>
<tr>
<td>3</td>
<td>1.31 1.34 1.41 1.39</td>
</tr>
<tr>
<td>2</td>
<td>1.17 1.25 1.30 1.30</td>
</tr>
<tr>
<td>1</td>
<td>1.00 1.09 1.19 1.21</td>
</tr>
</tbody>
</table>

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.25/36
Define *fill ratio* and *dense performance*

\[ f_A (r, c) = \frac{\text{# of stored nonzeros using } r \times c \text{ blocks}}{\text{# of true nonzeros}} \]

\[ P_{\text{dense}} (r, c) = \text{Performance (Mflop/s) for dense matrix in sparse } r \times c \text{ format} \]
Define **fill ratio** and **dense performance**

\[
\begin{align*}
    f_A (r, c) &= \frac{\text{# of stored nonzeros using } r \times c \text{ blocks}}{\text{# of true nonzeros}} \\
    P_{\text{dense}}(r, c) &= \text{Performance (Mflop/s) for dense matrix in sparse } r \times c \text{ format}
\end{align*}
\]

**Off-line**: For all \( r \times c \), measure \( P_{\text{dense}}(r, c) \) (*register profile*)
Define *fill ratio* and *dense performance*

\[ f_A (r, c) = \frac{\text{# of stored nonzeros using } r \times c \text{ blocks}}{\text{# of true nonzeros}} \]

\[ P_{\text{dense}}(r, c) = \text{Performance (Mflop/s) for dense matrix in sparse } r \times c \text{ format} \]

**Off-line**: For all \( r \times c \), measure \( P_{\text{dense}}(r, c) \) (*register profile*)

**Run-time**:
- Compute \( f_A (r, c) \)
- Choose \( r, c \) that maximizes

\[ \frac{P_{\text{dense}}(r, c)}{f_A (r, c)} \]

In practice, total run-time cost (incl. reorg.) is 10–30 SpM × Vs
Cache Miss Bound Verification: Intel Pentium III (L1)

L1 Misses -- [pentium3–linux–icc]

Matrix no. of misses (millions)

Upper bound
PAPI
Lower Bound
Cache Miss Bound Verification: Intel Pentium III (L2)

L2 Misses -- [pentium3–linux–icc]
Cache Miss Bound Verification: IBM Power3 (L2)

L2 Misses -- [power3–aix]

- Upper bound
- PAPI
- Lower Bound

no. of misses (millions)

matrix

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.30/36
Cache Miss Bound Verification: Intel Itanium (L2)

L2 Misses --- [itanium-linux-ecc]

- Upper bound
- PAPI
- Lower Bound

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.31/36
Cache Miss Bound Verification: Intel Itanium (L3)

L3 Misses --- [itanium-linux-ecc]

- Upper bound
- PAPI
- Lower Bound

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.32/36
Latency vs. Bandwidth

Sustainable Memory Bandwidth (STREAM)

- $a[i] = b[i]$
- $a[i] = \alpha b[i]$
- $a[i] = b[i] + c[i]$
- $a[i] = b[i] + \alpha c[i]$
- $s += a[i]$
- $s += a[i]b[i]$
- $s += a[i]; k += ind[i]$
- $s += a[i]x[ind[i]]; x \in L_{chip}$
- $s += a[i]x[ind[i]]; x \in L_{ext}$
- Dgemv
- SpMV (dense, 1x1)
- SpMV (dense, best)

- Full-latency model
Related Work (2/2)

- Compilers (analysis and models); run-time selection
  - CROPS (UCSD/Carter, Ferrante, et al.)
  - TUNE (Chatterjee, et al.)
  - Iterative compilation (O’Boyle, et al., 1998)
  - Broadway (Guyer and Lin, ’99)
  - Brewer (’95); ADAPT (Voss, 2000)

- Interfaces: Sparse BLAS; PSBLAS; PETSc

- Sparse triangular solve
  - SuperLU/MUMPS/SPOOLES/UMFPACK/PSPASES…
  - Approximation: Alvarado (’93); Raghavan (’98)
  - Scalability: Rothberg (’92;’95); Gupta (’95); Li, Coleman (’88)
What is the Cost of Search?

Block Size Selection Overhead [itanium-linux-ecc]

Cost (no. of reference Sp×MVs)

Matrix #

heuristic
rebuild
Where in Memory is the Time Spent? (PAPI Data)

Execution Time Model (PAPI Hits/Misses) — Where is the Time Spent?

Fraction of Cycles (exhaustive best; average over matrices)

<table>
<thead>
<tr>
<th>Platform</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i (L1/L2)</td>
<td>0.2</td>
<td>0.3</td>
<td>0.1</td>
<td>0.4</td>
</tr>
<tr>
<td>Pentium III (L1/L2)</td>
<td>0.3</td>
<td>0.3</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Power3 (L1/L2)</td>
<td>0.3</td>
<td>0.3</td>
<td>0.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Itanium (L1/L2/L3)</td>
<td>0.3</td>
<td>0.3</td>
<td>0.1</td>
<td>0.3</td>
</tr>
</tbody>
</table>
References


