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)

Tuning sparse linear algebra kernels is hard
- Sparse code has . . .
  - high bandwidth requirements (extra storage)
  - poor locality (indirect, irregular memory access)
  - poor instruction mix (data structure manipulation)
- Sparse matrix-vector multiply (SpM×V) performance:
  less than 10% of machine peak
- Performance depends on kernel, architecture, and matrix
Example: Matrix *olafu*

Spy Plot: 03–olafu.rua

- \( N = 16146 \)
- \( \text{nnz} = 1.0M \)
- Kernel = SpM \( \times \) V

A natural choice: blocked compressed sparse row (BCSR) storage. Is it the best choice?

Experiment: Measure performance of all block sizes for 1015156 non–zeros.
Example: Matrix olafu

N = 16146
nnz = 1.0M
Kernel = SpM×V

A natural choice:
6×6 blocked compressed sparse row (BCSR) storage. Is it the best choice?

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

Blocking Performance (Mflop/s) [03-olafu.rua; itanium-linux-ecc]

(Peak machine speed: 3.2 Gflop/s)
Speedups on Itanium: The Need for Search

Blocking Performance (Mflop/s) [03–olafu.rua; itanium–linux–ecc]

(Peak machine speed: 3.2 Gflop/s)
Speedups on Itanium: The Need for Search

Blocking Performance (Mflop/s) [03-olafu.rua; itanium-linux-ecc]

(Peak machine speed: 3.2 Gflop/s)
Speedups on Itanium: The Need for Search

Blocking Performance (Mflop/s) [03-olafu.rua; itanium-linux-ecc]

<table>
<thead>
<tr>
<th>r = 6</th>
<th>0.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.44</td>
</tr>
<tr>
<td>2</td>
<td>0.64</td>
</tr>
<tr>
<td>r = 1</td>
<td>1.00</td>
</tr>
<tr>
<td>c = 1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>c = 6</td>
<td></td>
</tr>
</tbody>
</table>

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

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

- What are the limits on performance of blocked \( \text{SpM} \times \text{V} \)?
  - Derive performance upper bounds for blocking
  - Often within 20% of upper bound, placing limits on improvement from more “low-level” tuning
  - Performance is memory-bound: reducing data structure size is critical

- Where are the new opportunities (kernels, techniques) for achieving higher performance?
  - Identify cases in which blocking does and does not work
  - Identify new kernels and opportunities for reducing memory traffic
Related Work

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

- Interfaces
  - Sparse BLAS Standard [BCD+01]
  - NIST SparseBLAS [RP96], SPARSKIT [Saa94], PSBLAS [FC00]
  - PETSc, hypre, …
Key Questions and Conclusions

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

- **What are the limits on performance of blocked \( \text{SpM} \times \text{V} \)?**
  - Derive performance upper bounds for blocking
  - Often within 20% of upper bound, placing limits on improvement from more “low-level” tuning
  - Performance is memory-bound: reducing data structure size is critical

- **Where are the new opportunities (kernels, techniques) for achieving higher performance?**
  - Identify cases in which blocking does and does not work
  - Identify new kernels and opportunities for reducing memory traffic
Approach to Automatic Tuning

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

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

- The SPARSITY system for SpM×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, …
  - **Search**
    - Off-line: benchmarking (once per architecture)
    - Run-time: estimate matrix properties (“search”) and predict best tuning parameters
Approach to Automatic Tuning

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

- The SPARSITY system for SpM×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, …
  - Search
    - Off-line: benchmarking (once per architecture)
    - Run-time: estimate matrix properties (“search”) and predict best tuning parameters
Register-Level Blocking (SPARSITY): 3x3 Example

3 x 3 Register Blocking Example

688 true non-zeros
Register-Level Blocking (SPARSITY): 3x3 Example

- BCSR with uniform grid
Register-Level Blocking (SPARSITY): 3x3 Example

- Fill-in zeros: trade-off extra flops for better efficiency
- This example: 50% fill led to 1.5x speedup on Pentium III
Off-line benchmarking (once per architecture)

- Measure **Dense Performance** \((r,c)\)
  
  Performance (Mflop/s) of dense matrix in sparse \(r \times c\) blocked format

At run-time, when matrix is known:

- Estimate **Fill Ratio** \((r,c), \forall r, c\)
  
  \[
  \text{Fill Ratio} (r,c) = \frac{\text{(number of stored values)}}{\text{(number of true non-zeros)}}
  \]

- Choose \(r, c\) that maximizes

  \[
  \text{Est. Performance} (r,c) = \frac{\text{Dense Performance} (r,c)}{\text{Fill Ratio} (r,c)}
  \]

- Convert from input format to \(r \times c\) BCSR
Top 10 codes labeled by speedup over unblocked code. Max speedup = 2.54 ($2 \times 10$).
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)
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 of blocked SpM×V?**
  - Derive performance upper bounds for blocking
  - Often within 20% of upper bound, placing limits on improvement from more “low-level” tuning
  - Performance is memory-bound: reducing data structure size is critical

- **Where are the new opportunities (kernels, techniques) for achieving higher performance?**
  - Identify cases in which blocking does and does not work
  - Identify new kernels and opportunities for reducing memory traffic
Performance Bounds for Register Blocking

- How close are we to the speed limit of blocking?

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

\[
T = (\text{L1 hits})\alpha_1 + (\text{L2 hits})\alpha_2 + (\text{mem hits})\alpha_{\text{mem}} \\
= (\text{Loads})\alpha_1 + (\text{L1 misses})(\alpha_2 - \alpha_1) + (\text{L2 misses})(\alpha_{\text{mem}} - \alpha_2)
\]

- Need *lower bound* on time, *i.e.*, *lower bound* on misses
  - Count only compulsory misses (*i.e.*, ignore conflict misses)
  - Account for line size
  - True miss counts typically 10–20% larger than lower bound
Overview of Performance Results

- Experimental setup
  - Four machines: Pentium III, Ultra 2i, Power3, Itanium
  - 44 matrices: dense, finite element, mixed, 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: Intel Pentium III

Performance Summary [pentium3-linux-icc]

matrix

Performance (Mflop/s)

DGEMV (n=1000): 96 Mflop/s
DGEMV (n=1000): 96 Mflop/s
Performance Results: Intel Pentium III

DGEMV (n=1000): 96 Mflop/s
Performance Results: Intel Pentium III

Performance Summary [pentium3-linux-icc]

- Analytic upper bound
- Upper bound (PAPI)
- Sparsity (exhaustive)
- Sparsity (heuristic)

DGEMV (n=1000): 96 Mflop/s
DGEMV (n=1000): 96 Mflop/s
Performance Results: Sun Ultra 2i

Performance Summary [ultra-solaris]

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

DGEMV (n=1000): 58 Mflop/s
Performance Results: Intel Itanium

Performance Summary [itanium-linux-ecc]

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

DGEMV (n=1000): 310 Mflop/s
Performance Results: IBM Power3

Performance Summary [power3-aix]

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

DGEMV (n=2000): 260 Mflop/s
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.
Current and Future Work (1/2)

- Further performance improvements
  - symmetry (1.5–2x speedups)
  - diagonals, block diagonals, and 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)
  - hybrid sparse/dense structure (1.2–1.8x)

- Higher-level kernels that permit reuse
  - $A^T A x$ (1.5–3x)
  - $A x$ and $A^T y$ simultaneously, $A^k x$, $RAR^T$, . . . (future work)
Current and Future Work (2/2)

- An automatically tuned sparse matrix library
  - Code generation via sparse compilers (Bernoulli; Bik)
  - Plan to extend new Sparse BLAS standard by one routine to support tuning

- Architectural issues
  - Improvements for Power3?
  - Latency vs. bandwidth (see paper)
  - Using models to explore architectural design space
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</td>
</tr>
<tr>
<td></td>
<td>1.39</td>
</tr>
<tr>
<td></td>
<td>1.42</td>
</tr>
<tr>
<td></td>
<td>1.53</td>
</tr>
<tr>
<td>3</td>
<td>1.31</td>
</tr>
<tr>
<td></td>
<td>1.34</td>
</tr>
<tr>
<td></td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>1.39</td>
</tr>
<tr>
<td>2</td>
<td>1.17</td>
</tr>
<tr>
<td></td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>1.30</td>
</tr>
<tr>
<td></td>
<td>1.30</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>1.19</td>
</tr>
<tr>
<td></td>
<td>1.21</td>
</tr>
</tbody>
</table>
Where in Memory is the Time Spent?

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

<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>0.2</td>
<td>0.1</td>
<td>0.4</td>
</tr>
<tr>
<td>Power3 (L1/L2)</td>
<td>0.2</td>
<td>0.3</td>
<td>0.1</td>
<td>0.4</td>
</tr>
<tr>
<td>Itanium (L1/L2/L3)</td>
<td>0.2</td>
<td>0.3</td>
<td>0.1</td>
<td>0.4</td>
</tr>
</tbody>
</table>
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: Sun Ultra 2i (L1)
Cache Miss Bound Verification: Sun Ultra 2i (L2)

L2 Misses -- [ultra-solaris]

- Upper bound
- PAPI
- Lower Bound

<table>
<thead>
<tr>
<th>matrix</th>
<th>no. of misses (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10^{-1}</td>
</tr>
<tr>
<td>2</td>
<td>10^0</td>
</tr>
<tr>
<td>3</td>
<td>10^1</td>
</tr>
</tbody>
</table>

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.28/43
Cache Miss Bound Verification: Intel Pentium III (L1)
Cache Miss Bound Verification: Intel Pentium III (L2)

L2 Misses -- [pentium3-linux-icc]

Upper bound
PAPI
Lower Bound

no. of misses (millions)

matrix

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.30/43
Cache Miss Bound Verification: IBM Power3 (L1)

L1 Misses -- [power3-aix]

- Upper bound
- PAPI
- Lower Bound

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.31/43
Cache Miss Bound Verification: IBM Power3 (L2)
Cache Miss Bound Verification: Intel Itanium (L2)
Latency vs. Bandwidth

Sustainable Memory Bandwidth (STREAM)

- \( a[i] = b[i] \)
- \( a[i] = \alpha \cdot b[i] \)
- \( a[i] = b[i] + c[i] \)
- \( a[i] = b[i] + \alpha \cdot c[i] \)
- \( s += a[i] \)
- \( s += a[i] \cdot b[i] \)
- \( s += a[i]; k += ind[i] \)
- \( s += a[i] \cdot x[ind[i]]; x \in L_{\text{chip}} \)
- \( s += a[i] \cdot x[ind[i]]; x \in L_{\text{ext}} \)
- \( \text{DGEMV} \)
- \( \text{SpM} \times V \) (dense, 1×1)
- \( \text{SpM} \times V \) (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]

<table>
<thead>
<tr>
<th>Matrix #</th>
<th>Cost (no. of reference SpxMVs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>heuristic</td>
</tr>
<tr>
<td></td>
<td>rebuild</td>
</tr>
</tbody>
</table>

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.37/43
Where in Memory is the Time Spent? (PAPI Data)

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

Platform

Fraction of Cycles (exhaustive best; average over matrices)

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

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

L1  L2  L3  Mem
Performance Results: Ultra Solaris

Performance Summary [ultra–solaris]

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

DGEMV (n=1000): 58 Mflop/s
## Fill: Some Surprises!

- Sometimes faster to fill in many zeros

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Speedup</th>
<th>Fill ratio</th>
<th>(Size BCSR) (Size CSR)</th>
<th>Dense (Perf BCSR) (Perf CSR)</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1.09</td>
<td>1.70</td>
<td>1.26</td>
<td>1.55</td>
<td>Itanium</td>
</tr>
<tr>
<td>13</td>
<td>1.50</td>
<td>1.52</td>
<td>1.07</td>
<td>2.30</td>
<td>Pentium 3</td>
</tr>
<tr>
<td>17</td>
<td>1.04</td>
<td>1.59</td>
<td>1.23</td>
<td>1.54</td>
<td>Itanium</td>
</tr>
<tr>
<td>27</td>
<td>1.16</td>
<td>1.94</td>
<td>1.47</td>
<td>1.54</td>
<td>Itanium</td>
</tr>
<tr>
<td>27</td>
<td>1.10</td>
<td>1.53</td>
<td>1.25</td>
<td>1.23</td>
<td>Ultra 2i</td>
</tr>
<tr>
<td>29</td>
<td>1.00</td>
<td>1.98</td>
<td>1.44</td>
<td>1.89</td>
<td>Pentium 3</td>
</tr>
</tbody>
</table>
Performance Results: Intel Pentium III

Performance Summary [pentium3-linux-icc]

- **DGEMV (n=1000): 96 Mflop/s**
Performance Results: IBM Power3

Performance Summary [power3-aix]

- **Analytic upper bound**
- **Upper bound (PAPI)**
- **Sparsity (exhaustive)**
- **Sparsity (heuristic)**
- **Reference**

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

Performance Summary [itanium–linux–ecc]

- DGEMV (n=1000): 310 Mflop/s
References


