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
Context: Performance Tuning in the Sparse Case

- 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?
  - Sparse matrix-vector multiply (SpM\times V) performance: less than 10% of machine peak
  - Sparse code has . . .
    - high bandwidth requirements (extra storage)
    - poor locality (indirect, irregular memory access)
    - poor instruction mix (data structure manipulation)
  - Performance depends on architecture, kernel, matrix
Example: Matrix olafu

Spy Plot: 03-olafu.rua

N = 16146
nnz = 1.0M
Kernel = SpM×V
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 $r, c \in \{1, 2, 3, 6\}$. 
### Blocking Performance (Mflop/s) [03–olafu.rua; itanium–linux–ecc]

<table>
<thead>
<tr>
<th>row block size \ column block size</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>0.86</td>
<td>0.99</td>
<td>0.93</td>
</tr>
<tr>
<td>2</td>
<td>1.21</td>
<td>1.27</td>
<td>0.97</td>
<td>0.78</td>
</tr>
<tr>
<td>3</td>
<td>1.44</td>
<td>0.99</td>
<td>1.07</td>
<td>0.64</td>
</tr>
<tr>
<td>6</td>
<td>1.20</td>
<td>0.81</td>
<td>0.75</td>
<td>0.95</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 \( \text{SpM} \times V \)?
  - Derive performance upper bounds for blocking
    - Performance is memory-bound: reducing data structure size is critical
  - 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
  - Identify new kernels and opportunities for reducing memory traffic
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.)
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)

3 x 3 Register Blocking Example

688 true non−zeros
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

3 x 3 Register Blocking Example

688 true non-zeros
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
  - This example: 50% fill led to 1.5x speedup on Pentium III
Top 10 codes labeled by speedup over unblocked \((1 \times 1)\) code. Max speedup = 2.03 \((8 \times 5)\).
<table>
<thead>
<tr>
<th>Processor Type</th>
<th>Configuration</th>
<th>Clock Speed</th>
<th>MFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>333 MHz Sun Ultra 2i</td>
<td>2.03</td>
<td>36</td>
<td>72</td>
</tr>
<tr>
<td>500 MHz Intel Pentium III</td>
<td>2.54</td>
<td>36</td>
<td>107</td>
</tr>
<tr>
<td>375 MHz IBM Power3</td>
<td>1.22</td>
<td>174</td>
<td>256</td>
</tr>
<tr>
<td>800 MHz Intel Itanium</td>
<td>1.55</td>
<td>86</td>
<td>109</td>
</tr>
</tbody>
</table>
**Choosing the Block Size**

- **Off-line benchmarking (once per architecture)**
  - **Measure** **Dense Performance** \((r,c)\)
    Performance (Mflop/s) of dense matrix in sparse \(r \times c\) blocked format, \(\forall r, c\)

- **At run-time, when matrix is known:**
  - **Estimate** **Fill Ratio** \((r,c)\), \(\forall r, c\)
    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**

- **Total run-time cost:** \(\approx 40 \text{ SpM} \times \text{Vs}\)
  - (Re)building the matrix: \(\approx 35 \text{ SpM} \times \text{Vs}\)
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

- 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
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)

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

Platform

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

L1  L2  L3  Mem
Overview of Performance Results

- Experimental setup
  - Four machines: Ultra 2i, Pentium III, 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: Sun Ultra 2i

Performance Summary [ultra-solaris]

DGEMV (n=1000): 58 Mflop/s
Performance Results: Sun Ultra 2i

Performance Summary [ultra-solaris]

DGEMV (n=1000): 58 Mflop/s
Performance Results: Sun Ultra 2i

Performance Summary [ultra–solaris]

- DGEMV (n=1000): 58 Mflop/s
Performance Results: Sun Ultra 2i

Performance Summary [ultra–solaris]

- DGEMV (n=1000): 58 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 Pentium III

Performance Summary [pentium3–linux–icc]

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

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

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

### Performance Summary [itanium–linux–ecc]

<table>
<thead>
<tr>
<th>Analytic upper bound</th>
<th>Upper bound (PAPI)</th>
<th>Sparsity (exhaustive)</th>
<th>Sparsity (heuristic)</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGEMV (n=1000): 310 Mflop/s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing performance summary](image-url)
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>
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 (1.8x)

- **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>1</th>
<th>2</th>
<th>3</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1.40</td>
<td>1.39</td>
<td>1.42</td>
<td>1.53</td>
</tr>
<tr>
<td>3</td>
<td>1.31</td>
<td>1.34</td>
<td>1.41</td>
<td>1.39</td>
</tr>
<tr>
<td>2</td>
<td>1.17</td>
<td>1.25</td>
<td>1.30</td>
<td>1.30</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.09</td>
<td>1.19</td>
<td>1.21</td>
</tr>
</tbody>
</table>

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.24/41
Define *fill ratio* and *dense performance*

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

- 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)
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)

L1 Misses -- [ultra-solaris]

- Upper bound
- PAPI
- Lower Bound

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.26/41
Cache Miss Bound Verification: Sun Ultra 2i (L2)

L2 Misses -- [ultra-solaris]

- Upper bound
- PAPI
- Lower Bound

Matrix no. of misses (millions)

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.27/41
Cache Miss Bound Verification: Intel Pentium III (L1)
Cache Miss Bound Verification: Intel Pentium III (L2)

L2 Misses -- [pentium3-linux-icc]

matrix

no. of misses (millions)

10^{-1} 10^{0} 10^{1}

Upper bound
PAPI
Lower Bound

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.29/41
Cache Miss Bound Verification: IBM Power3 (L1)

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.30/41
Cache Miss Bound Verification: IBM Power3 (L2)

L2 Misses -- [power3-aix]

Matrix (no. of misses (millions))

10^1

Upper bound
PAPI
Lower Bound

10^0

10^-1

1 4 5 7 8 9 10 12 13 15 40

matrix

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.31/41
Cache Miss Bound Verification: Intel Itanium (L2)
Cache Miss Bound Verification: Intel Itanium (L3)

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

- Upper bound
- PAPI
- Lower Bound

Matrix
- Number of misses (millions)
- Logarithmic scale

Matrix 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.33/41
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>2</td>
<td>0.74</td>
</tr>
<tr>
<td>3</td>
<td>0.74</td>
</tr>
<tr>
<td>4</td>
<td>0.71</td>
</tr>
<tr>
<td>5</td>
<td>0.86</td>
</tr>
<tr>
<td>6</td>
<td>0.85</td>
</tr>
<tr>
<td>7</td>
<td>0.80</td>
</tr>
<tr>
<td>8</td>
<td>0.84</td>
</tr>
<tr>
<td>9</td>
<td>0.81</td>
</tr>
<tr>
<td>10</td>
<td>0.84</td>
</tr>
<tr>
<td>11</td>
<td>0.72</td>
</tr>
<tr>
<td>12</td>
<td>0.76</td>
</tr>
<tr>
<td>13</td>
<td>0.76</td>
</tr>
<tr>
<td>14</td>
<td>0.79</td>
</tr>
<tr>
<td>15</td>
<td>0.76</td>
</tr>
<tr>
<td>16</td>
<td>0.76</td>
</tr>
<tr>
<td>17</td>
<td>0.79</td>
</tr>
<tr>
<td>18</td>
<td>0.79</td>
</tr>
<tr>
<td>19</td>
<td>0.79</td>
</tr>
<tr>
<td>20</td>
<td>0.79</td>
</tr>
<tr>
<td>21</td>
<td>0.79</td>
</tr>
<tr>
<td>22</td>
<td>0.79</td>
</tr>
<tr>
<td>23</td>
<td>0.79</td>
</tr>
<tr>
<td>24</td>
<td>0.79</td>
</tr>
<tr>
<td>25</td>
<td>0.79</td>
</tr>
<tr>
<td>26</td>
<td>0.79</td>
</tr>
<tr>
<td>27</td>
<td>0.79</td>
</tr>
<tr>
<td>28</td>
<td>0.79</td>
</tr>
<tr>
<td>29</td>
<td>0.79</td>
</tr>
<tr>
<td>30</td>
<td>0.79</td>
</tr>
<tr>
<td>31</td>
<td>0.79</td>
</tr>
<tr>
<td>32</td>
<td>0.79</td>
</tr>
<tr>
<td>33</td>
<td>0.79</td>
</tr>
<tr>
<td>34</td>
<td>0.79</td>
</tr>
<tr>
<td>35</td>
<td>0.79</td>
</tr>
<tr>
<td>36</td>
<td>0.79</td>
</tr>
<tr>
<td>37</td>
<td>0.79</td>
</tr>
<tr>
<td>38</td>
<td>0.79</td>
</tr>
<tr>
<td>39</td>
<td>0.79</td>
</tr>
<tr>
<td>40</td>
<td>0.79</td>
</tr>
<tr>
<td>41</td>
<td>0.79</td>
</tr>
<tr>
<td>42</td>
<td>0.79</td>
</tr>
<tr>
<td>43</td>
<td>0.79</td>
</tr>
<tr>
<td>44</td>
<td>0.79</td>
</tr>
</tbody>
</table>

- **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)

Platform

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

L1 | L2 | L3

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.37/41
Performance Results: Ultra Solaris

Performance Summary [ultra−solaris]

**DGEMV (n=1000): 58 Mflop/s**
Performance Results: Intel Pentium III

Performance Summary [pentium3–linux–icc]

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

Performance Summary [power3-aix]

- DGEMV (n=2000): 260 Mflop/s

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


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Conference</th>
<th>Volume</th>
<th>Pages</th>
<th>Location</th>
<th>Publisher</th>
</tr>
</thead>
</table>

41-1