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?
- Sparse matrix-vector multiply (SpM×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 \texttt{olafu}

```
<table>
<thead>
<tr>
<th>\texttt{olafu}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>2400</td>
</tr>
<tr>
<td>4800</td>
</tr>
<tr>
<td>7200</td>
</tr>
<tr>
<td>9600</td>
</tr>
<tr>
<td>12000</td>
</tr>
<tr>
<td>14400</td>
</tr>
</tbody>
</table>
```

- \texttt{Spy Plot: 03-olafu.rua}
- \texttt{N = 16146}
- \texttt{nnz = 1.0M}
- \texttt{Kernel = SpM \times V}

Example: Matrix $\text{olafu}$

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

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

![Heatmap showing blocking performance](image)

(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 $\text{SpM} \times \text{V}$?
  - 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
  - Identify new kernels and opportunities for reuse
## Related Work

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

### Code generation
- Sparse compilers (Bik \([\text{BW99}]\), Bernoulli \([\text{Sto97}]\))
- Generic programming (Blitz++ \([\text{Vel98}]\), MTL \([\text{SL98}]\), GMCL \([\text{Neu98}]\), \ldots)
- FLAME \([\text{GGHvdG01}]\)

### Sparse performance modeling and tuning
- Temam and Jalby \([\text{TJ92}]\)
- Toledo \([\text{Tol97}]\), White and Sadayappan \([\text{WS97}]\), Pinar \([\text{PH99}]\)
- Navarro \([\text{NGLPJ96}]\), Heras \([\text{HPDR99}]\), Fraguela \([\text{FDZ99}]\)
- Gropp, \textit{et al.} \([\text{GKKS99}]\), Geus \([\text{GR99}]\)

### Compilers (analysis and models)
- CROPS (UCSD/Carter, Ferrante, \textit{et al.})
- TUNE (Chatterjee, \textit{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 2 Register Blocking Example

- Store dense blocks
- BCSR with uniform blocks
- Reduce storage and bandwidth requirements
- Fully unroll block multiplies
- Improves register reuse, scheduling
- Fill-in zeros: trade-off extra ops for better efficiency
- 1.3–2.5x speedup on FEM matrices
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
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

3 x 2 Register Blocking Example

(598 true non-zeros) + (434 explicit zeros) = 1032 nz
Top 10 codes labeled by speedup over unblocked \((1 \times 1)\) code. Max speedup = 2.03 \((8 \times 5)\).
### 333 MHz Sun Ultra 2i (2.03)

<table>
<thead>
<tr>
<th>Column Block Size (c)</th>
<th>Row Block Size (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.94</td>
</tr>
<tr>
<td>2</td>
<td>1.95</td>
</tr>
<tr>
<td>3</td>
<td>1.94</td>
</tr>
<tr>
<td>4</td>
<td>1.98</td>
</tr>
<tr>
<td>5</td>
<td>1.96</td>
</tr>
<tr>
<td>6</td>
<td>1.98</td>
</tr>
<tr>
<td>7</td>
<td>1.97</td>
</tr>
<tr>
<td>8</td>
<td>1.94</td>
</tr>
<tr>
<td>9</td>
<td>1.94</td>
</tr>
<tr>
<td>10</td>
<td>1.95</td>
</tr>
<tr>
<td>11</td>
<td>1.96</td>
</tr>
<tr>
<td>12</td>
<td>2.03</td>
</tr>
</tbody>
</table>

### 500 MHz Intel Pentium III (2.54)

<table>
<thead>
<tr>
<th>Column Block Size (c)</th>
<th>Row Block Size (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.36</td>
</tr>
<tr>
<td>2</td>
<td>2.39</td>
</tr>
<tr>
<td>3</td>
<td>2.44</td>
</tr>
<tr>
<td>4</td>
<td>2.49</td>
</tr>
<tr>
<td>5</td>
<td>2.39</td>
</tr>
<tr>
<td>6</td>
<td>2.38</td>
</tr>
<tr>
<td>7</td>
<td>2.54</td>
</tr>
<tr>
<td>8</td>
<td>2.38</td>
</tr>
<tr>
<td>9</td>
<td>2.39</td>
</tr>
<tr>
<td>10</td>
<td>2.37</td>
</tr>
<tr>
<td>11</td>
<td>2.36</td>
</tr>
<tr>
<td>12</td>
<td>2.35</td>
</tr>
</tbody>
</table>

### 375 MHz IBM Power3 (1.22)

<table>
<thead>
<tr>
<th>Column Block Size (c)</th>
<th>Row Block Size (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.53</td>
</tr>
<tr>
<td>2</td>
<td>1.55</td>
</tr>
<tr>
<td>3</td>
<td>1.55</td>
</tr>
<tr>
<td>4</td>
<td>1.55</td>
</tr>
<tr>
<td>5</td>
<td>1.48</td>
</tr>
<tr>
<td>6</td>
<td>1.46</td>
</tr>
<tr>
<td>7</td>
<td>1.45</td>
</tr>
<tr>
<td>8</td>
<td>1.46</td>
</tr>
<tr>
<td>9</td>
<td>1.45</td>
</tr>
<tr>
<td>10</td>
<td>1.46</td>
</tr>
<tr>
<td>11</td>
<td>1.45</td>
</tr>
<tr>
<td>12</td>
<td>1.46</td>
</tr>
</tbody>
</table>

### 800 MHz Intel Itanium (1.55)

<table>
<thead>
<tr>
<th>Column Block Size (c)</th>
<th>Row Block Size (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1.42</td>
</tr>
<tr>
<td>2</td>
<td>1.40</td>
</tr>
<tr>
<td>3</td>
<td>1.41</td>
</tr>
<tr>
<td>4</td>
<td>1.42</td>
</tr>
<tr>
<td>5</td>
<td>1.45</td>
</tr>
<tr>
<td>6</td>
<td>1.45</td>
</tr>
<tr>
<td>7</td>
<td>1.46</td>
</tr>
<tr>
<td>8</td>
<td>1.46</td>
</tr>
<tr>
<td>9</td>
<td>1.45</td>
</tr>
<tr>
<td>10</td>
<td>1.45</td>
</tr>
<tr>
<td>11</td>
<td>1.46</td>
</tr>
<tr>
<td>12</td>
<td>1.46</td>
</tr>
</tbody>
</table>
Search: 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 \((r,c) = \text{(number of stored values)} / \text{(number of true non-zeros)}\)
  - Choose \(r, c\) that maximizes
    \[
    \text{Dense Performance} \ (r, c) = \text{Fill Ratio} \ (r, c)
    \]
  - Convert from input format to \(r \times c\) BCSR

- Total run-time cost: \(\approx 40\) SpM\(\times\)Vs
  - (Re)building the matrix: \(\approx 35\) SpM\(\times\)Vs
Performance Bounds for Register Blocking

- How close are we to the speed limit?
- **Upper-bounds** on performance: \( \frac{(\text{flops})}{(\text{time})} \) [Mflop/s]
  - Flops \( \approx 2 \times \) (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.,*

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

- 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?

Platform

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

Fraction of Cycles (exhaustive best; average over matrices)

0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
Overview of Performance Results

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

- 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)

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

Performance Summary [ultrasolaris]

- **DGEMV (n=1000): 58 Mflop/s**

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.15/37
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]

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

Performance Summary [pentium3-linux-icc]

- DGEMV (n=1000): 96 Mflop/s

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.16/37
**Performance Results: Intel Pentium III**

### Performance Summary [pentium3–linux–icc]

- **DGEMV (n=1000):** 96 Mflop/s

---

**Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.16/37**
**Performance Results: Intel Pentium III**

**Performance Summary [pentium3–linux–icc]**

- **DGEMV (n=1000): 96 Mflop/s**

---

Image showing a graph with performance metrics for different operations, including:
- **Analytic upper bound**
- **Upper bound (PAPI)**
- **Sparsity (exhaustive)**
- **Sparsity (heuristic)**

Operations include:
- **D**
- **FEM**
- **FEM (var. blk.)**
- **Mixed**
- **LP**

Matrix performance is plotted against Performance (Mflop/s) with various markers indicating different performance levels.
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

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]

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

DGEMV (n=1000): 310 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>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1.09</td>
<td>1.70</td>
<td>Itanium</td>
</tr>
<tr>
<td>13</td>
<td>1.50</td>
<td>1.52</td>
<td>Pentium 3</td>
</tr>
<tr>
<td>17</td>
<td>1.04</td>
<td>1.59</td>
<td>Itanium</td>
</tr>
<tr>
<td>27</td>
<td>1.16</td>
<td>1.94</td>
<td>Itanium</td>
</tr>
<tr>
<td>27</td>
<td>1.10</td>
<td>1.53</td>
<td>Ultra 2i</td>
</tr>
<tr>
<td>29</td>
<td>1.00</td>
<td>1.98</td>
<td>Pentium 3</td>
</tr>
<tr>
<td>36</td>
<td>1.08</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 (1/2)

- Further performance improvements
  - symmetry (1.5–2x)
  - 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 (up to 1.8x)

- Higher-level kernels that permit reuse
  - $A^T Ax$ (1.5–3x)
  - $Ax$ and $A^T y$ simultaneously, $A^k x$, $RAR^T$, . . .
Directions (2/2)

- An automatically tuned sparse matrix library
  - Code generation via sparse compilers (Bernoulli; Bik)
  - Plan to extend Sparse BLAS 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 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.24/37
Define *fill ratio* and *dense performance*

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

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

- 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

Matrix no. of misses (millions)
Cache Miss Bound Verification: Intel Pentium III (L1)
Cache Miss Bound Verification: Intel Pentium III (L2)
Cache Miss Bound Verification: IBM Power3 (L1)

L1 Misses -- [power3-aix]

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.30/37
Cache Miss Bound Verification: IBM Power3 (L2)

L2 Misses -- [power3-aix]

- Upper bound
- PAPI
- Lower Bound

Matrix

No. of misses (millions)

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

Matrix

1 4 5 7 8 9 10 12 13 15 40
Cache Miss Bound Verification: Intel Itanium (L2)
Cache Miss Bound Verification: Intel Itanium (L3)

![Graph showing L3 Misses -- [itanium-linux-ecc] with Upper bound, PAPI, and Lower Bound markers.]

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.33/37
Latency vs. Bandwidth

Sustainable Memory Bandwidth (STREAM)

Platform

Fraction of Main Memory Bandwidth

Ultra 2i

Pentium III

Power3

Itanium

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.34/37
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]

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

Execution Time Model (PAPI 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>0.5</td>
<td>0.1</td>
<td>0.4</td>
<td>0.05</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>0.25</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>Itanium (L1/L2/L3)</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply – p.37/37
References


37-1
