How to Write Fast Code
18-645, spring 2008
11th Lecture, Feb. 20th

Instructor: Markus Püschel
TAs: Srinivas Chellappa (Vas) and Frédéric de Mesmay (Fred)
Technicalities

■ HW 2
  ▪ Grades: $\mu = 82$, $\sigma = 16$, max = 106, min = 39
  ▪ Time: $\mu = 11$, $\sigma = 5$
  ▪ Grades are now in blackboard: please double check

■ HW 2 feedback
About Plots (and Tables)

■ Above all they have to be readable
  ▪ If you print out black & white, don’t use color (different marker shapes, line styles)
  ▪ Always label axes and put a title
  ▪ Large enough font
  ▪ Proper number format (no 10s of zeros, no 10 digits after decimal point, no 2.345E09)
  ▪ Always discuss and analyze plots

Not good:
Research Projects

- **Projects and supervisors**
- **Start thinking about optimization**
  - If your problem is a numerical kernel: try techniques you learned in class, first focus is memory hierarchy
  - If your problem has several steps: determine bottleneck, then start optimizing bottleneck

Example:
Profiling JPEG 2000
# Meetings next Monday

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Markus</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 – 11:45</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>11:45 – 12:30</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>1:30 – 2:15</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2:15 – 3</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3 – 3:45</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>4:30 – 5:15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5:15 – 6</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td><strong>Fred</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:30 – 5:15</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5:15 – 6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6 – 6:45</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td><strong>Vas</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:45 – 4:30</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4:30 – 5:15</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5:15 – 6</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td><strong>Franz</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 – 1:45</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>2 – 2:45</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>4:30 – 5:15</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Today

- Sparse matrix-vector multiplication (MVM)
- Sparsity/Bebop
Sparse MVM

- \( y = y + Ax \), A sparse but known

- **Important routine in:**
  - finite element methods
  - PDE solving
  - physical/chemical simulation (e.g., fluid dynamics)
  - linear programming
  - scheduling
  - signal processing (e.g., filters)
  - ...

- **In these applications,** \( y = y + Ax \) is performed many times
  - justifies one-time tuning effort

- **Fundamental difference between MVM and MMM**
  - blackboard
Storage of Sparse Matrices

- Standard storage (as 2-D array) inefficient (many zeros are stored)

- Several sparse storage formats are available

- Explain compressed sparse row (CSR) format (blackboard)
  - advantage: arrays are accessed consecutively for $y = y + Ax$
  - disadvantage: inserting elements is costly, no reuse of $x$
Direct Implementation $y = Ax$, A in CSR

```c
void smvm_1x1( int m, const double* value, const int* col_idx,
               const int* row_start, const double* x, double* y )
{
    int i, jj;

    /* loop over rows */
    for( i = 0; i < m; i++ ) {
        double y_i = y[i];

        /* loop over non-zero elements in row i */
        for( jj = row_start[i]; jj < row_start[i+1]; jj++, col_idx++, value++ ) {
            y_i += value[0] * x[col_idx[0]];
        }
        y[i] = y_i;
    }
}
```

**scalar replacement** (only $y$ is reused)

**indirect array addressing** (problem for compiler opt.)
Optimizing Sparse MVM

- **Sparsity/Bebop**

Impact of Matrix Sparsity on Performance

- **Adressing overhead (dense MVM vs. dense MVM in CSR):**
  - ~ 2x slower (Mflop/s, example only)

- **Irregular structure**
  - ~ 5x slower (Mflop/s, example only) for “random” sparse matrices

- **Fundamental difference between MVM and sparse MVM (SMVM):**
  - sparse MVM is input **dependent** (sparsity pattern of A)
  - changing the order of computation (blocking) requires changing the data structure (CSR)
Bebop/Sparsity: SMVM Optimizations

- Register blocking
- Cache blocking
Register Blocking

- **Idea:** divide SMVM $y = y + Ax$ into micro (dense) MVMs of matrix size $r \times c$
  - store $A$ in $r \times c$ block CSR ($r \times c$ BCSR)

- **Explain on blackboard**
  - **Advantages:**
    - reuse of $x$ and $y$ (as for dense MVM)
    - reduces index overhead
  - **Disadvantages:**
    - computational overhead (zeros added)
    - storage overhead (for $A$)
Example: \( y = Ax \) in 2 x 2 BCSR

```c
void smvm_2x2( int bm, const int *b_row_start, const int *b_col_idx,
               const double *b_value, const double *x, double *y )
{
    int i, jj;

    /* loop over block rows */
    for( i = 0; i < bm; i++, y += 2 ) {
        register double d0 = y[0];
        register double d1 = y[1];

        /* dense micro MVM */
        for( jj = b_row_start[i]; jj < b_row_start[i+1]; jj++, b_col_idx++, b_value += 2*2 ) {
            d0  += b_value[0] * x[b_col_idx[0]+0];
            d1  += b_value[2] * x[b_col_idx[0]+0];
            d0  += b_value[1] * x[b_col_idx[0]+1];
            d1  += b_value[3] * x[b_col_idx[0]+1];
        }
        y[0] = d0;
        y[1] = d1;
    }
}
```

Which Block Size (r x c) is Optimal?

- **Example:** ~ 20,000 x 20,000 matrix with perfect 8 x 8 block structure, 0.33% non-zero entries
- **In this case:** no overhead when blocked r x c, with r,c divides 8

source: R. Vuduc, LLNL
Speed-up through r x c Blocking

- machine dependent
- hard to predict

How to Find the Best Blocking for given A?

- Best blocksize hard to predict (see previous slide)

- Searching over all $r \times c$ (within a range, say 1..12) BCSR expensive
  - But: conversion of A in CSR to BCSR roughly as expensive as 10 SMVMs

- **Solution:** Performance model for given A
  - blackboard
Gain from Blocking (Dense Matrix in BCSR)

Pentium III

Itanium 2

- machine dependence
- hard to predict

Register Blocking: Experimental results

- Paper applies method to a large set of sparse matrices

- Performance gains between 1x (no gain) for very unstructured matrices and 4x

Cache Blocking

- **Idea**: divide sparse matrix into blocks of sparse matrices

![Block Diagram]

- **Experiments**:
  - Requires very large matrices (x and y do not fit into cache)
  - Speed-up up to 2.2x, speed-up only for few matrices, with 1 x 1 BCSR

Multiple Vector Optimization

- **Blackboard**

- **Experiments:** up to 9x speedup for 9 vectors

Principles in Bebop/Sparsity Optimization

- Optimization for memory hierarchy = increasing locality
  - Blocking for registers (micro-MMMs) + change of data structure for A
  - Less important: blocking for cache
  - Optimizations are input dependent (on sparse structure of A)

- Fast basic blocks for small sizes (micro-MMM):
  - Loop unrolling (reduce loop overhead)
  - Some scalar replacement (enables better compiler optimization)

- Search for the fastest over a relevant set of algorithm/implementations alternatives (= r, c)
  - Use of performance model (versus measuring runtime) to evaluate expected gain

red = different from ATLAS