Algorithms and Computation in Signal Processing

special topic course 18-799B
spring 2005
9th Lecture Feb. 8, 2005

Instructor: Markus Pueschel
TA: Srinivas Chellappa
MMM versus MVM
Matrix-Vector Multiplication (MVM)

- **MMM:**
  - BLAS3
  - $O(n^2)$ data (input), $O(n^3)$ computation, implies $O(n)$ reuse per number
    (More precise on blackboard)

- **MVM:** $y = Ax$
  - BLAS2
  - $O(n^2)$ data, $O(n^2)$ computation
  - explain which optimizations remain useful (partially blackboard)
    - cache blocking?
    - register blocking?
    - unrolling?
    - scalar replacement?
    - add/mult interleaving, skewing?
Matrix-Vector Multiplication (MVM)

- **MMM:**
  - BLAS3
  - $O(n^2)$ data (input), $O(n^3)$ computation, implies $O(n)$ reuse per number

- **MVM:** $y = Ax$
  - BLAS2
  - $O(n^2)$ data, $O(n^2)$ computation
  - explain which optimizations remain useful (partially blackboard)
    - cache blocking? yes, but reuse of $x$ and $y$ only
    - register blocking? yes, but reuse of $x$ and $y$ only
    - unrolling? yes
    - scalar replacement? $x$ and $y$ only
    - add/mult interleaving, skewing? yes
    - expected gains smaller
MMM vs. MVM: Performance

- Performance for 2000 x 2000 matrices
- Best code out of ATLAS, vendor lib., Goto

<table>
<thead>
<tr>
<th>Processor and compiler</th>
<th>Clock (MHz)</th>
<th>Data cache sizes</th>
<th>DGEMV (MFLOPS)</th>
<th>DGEMM (MFLOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun UltraSPARC III Sun C v6.0</td>
<td>333</td>
<td>L1: 16 KB, L2: 2 MB</td>
<td>58</td>
<td>425</td>
</tr>
<tr>
<td>Intel Pentium III Mobile (Coppermine) Intel C v6.0</td>
<td>800</td>
<td>L1: 16 KB, L2: 256 KB</td>
<td>147</td>
<td>590</td>
</tr>
<tr>
<td>IBM Power4 IBM xlc v6</td>
<td>1300</td>
<td>L1: 64 KB, L2: 1.5 MB, L3: 32 MB</td>
<td>915</td>
<td>3500</td>
</tr>
<tr>
<td>Intel Itanium 2 Intel C v7.0</td>
<td>900</td>
<td>L1: 16 KB, L2: 256 KB, L3: 3 MB</td>
<td>1330</td>
<td>3500</td>
</tr>
</tbody>
</table>

Sparse Matrix-Vector Multiplication (Sparsity, Bebop)
Sparse MVM

- $y = Ax$, $A$ 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, $y = Ax$ is performed many times
  - justifies one-time tuning effort
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 $y = Ax$
  - disadvantage: no reuse of $x$ and $y$, inserting elements costly
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.)
Code Generation/Tuning for Sparse MVM

- Sparsity/Bebop  

Impact of Matrix-Sparsity on Performance

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

- **Irregular structure**
  - ~ 5x slower (mflops, 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 change of data structure (CSR)
Bebop/Sparsity: SMVM Optimizations

- Register blocking
- Cache blocking
Register Blocking

- **Idea:** divide SMVM $y = Ax$ 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 $r \times c$ Blocking

- **Ultra 2i**
  - Row block size $r$
  - Col. block size $c$

- **Itanium 2**
  - Row block size $r$
  - Col. block size $c$

- Machine dependence
- Hard to predict

How to Find the Best Register Blocking for given A?

- Best blocksize hard to predict (see previous slide)

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

- Solution: Performance model for given A
Performance Model for given A

- Model for given A built from
  - Gain of blocking:
    \[ G_{r,c} = \text{Performance } r \times c \text{ BCSR/Performance CSR for dense MVM} \]
    machine dependent, independent of matrix A
  
  - Computational overhead:
    \[ O_{r,c} = \text{size of } A \text{ in } r \times c \text{ BCSR/size of } A \text{ in CSR} \]
    machine independent, dependent on A
    computed by scanning only a fraction of the matrix
    (blackboard example)

- Model: Performance gain from \( r \times c \) blocking of \( A \):
  \[ P_{r,c} = G_{r,c} / O_{r,c} \]

- For given \( A \), use this model to search over all \( r, c \) in \( \{1, \ldots, 12\} \)
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

- Experiments:
  - requires very large matrices (x and y do not fit into cache)
  - speed-up up to 80%, speed-up only for few matrices, with 1 x 1 BCSR

**Figure Source:** Eun-Jin Im, Katherine A. Yelick, Richard Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels, Int’l Journal of High Performance Comp. App., 18(1), pp. 135-158, 2004
Multiple Vector Optimization

- Blackboard

- Experiments: up to 9x speedup for 9 vectors

Principles in Bebop/Sparsity Code Generation

- **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/implementation alternatives (= r, c)**
  - Use of performance model (versus measuring runtime) to evaluate expected gain

red = different from ATLAS