How to Write Fast Numerical Code
Spring 2017
Lecture: Memory bound computation, sparse linear algebra, OSKI

Instructor: Markus Püschel
TA: Alen Stojanov, Georg Ofenbeck, Gagandeep Singh

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ATLAS

Model-Based ATLAS

source: Pingali, Yotov, Cornell U.
Principles

- **Optimization for memory hierarchy**
  - Blocking for cache
  - Blocking for registers

- **Basic block optimizations**
  - Loop order for ILP
  - Unrolling + scalar replacement
  - Scheduling & software pipelining

- **Optimizations for virtual memory**
  - Buffering (copying spread-out data into contiguous memory)

- **Autotuning**
  - Search over parameters (ATLAS)
  - Model to estimate parameters (Model-based ATLAS)

- *All high performance MMM libraries do some of these (but possibly in a different way)*

Today

- Memory bound computations
- Sparse linear algebra, OSKI
Memory Bound Computation

- Data movement, not computation, is the bottleneck
- Typically: Computations with operational intensity $I(n) = O(1)$

Memory Bound Or Not? Depends On ...

- The computer
  - Memory bandwidth
  - Peak performance
- How it is implemented
  - Good/bad locality
  - SIMD or not
- How the measurement is done
  - Cold or warm cache
  - In which cache data resides
  - See next slide
Example: BLAS 1, Warm Data & Code

\[ z = x + y \text{ on Core i7 (Nehalem, one core, no SSE), } \text{icc } 12.0 \ /O2 \ /fp:fast \ /Qipo \]

<table>
<thead>
<tr>
<th>Percentage peak performance (peak = 1 add/cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Graph" /></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Sparse Linear Algebra

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

References:

- [Sparsity/Bebop](#) website
Sparse Linear Algebra

- Very different characteristics from dense linear algebra (LAPACK etc.)

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

- Core building block: Sparse MVM


Sparse MVM (SMVM)

- \[ y = y + Ax \]

- Typically executed many times for fixed \( A \)
- What is reused (temporal locality)?
- Upper bound on operational intensity?
Storage of Sparse Matrices

- Standard storage is obviously inefficient: Many zeros are stored
  - Unnecessary operations
  - Unnecessary data movement
  - Bad operational intensity
- Several sparse storage formats are available
- Most popular: Compressed sparse row (CSR) format
  - blackboard

CSR

- Assumptions:
  - A is m x n
  - K nonzero entries

A as matrix

<table>
<thead>
<tr>
<th>b</th>
<th>c</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A in CSR:

- values: b c c a b b c
- col_idx: 0 1 3 1 2 3 2
- row_start: 0 3 4 6 7

- Storage:
  - K doubles + (K+m+1) ints = \( \Theta(\max(K, m)) \)
  - Typically: \( \Theta(K) \)
Sparse MVM Using CSR

\[ y = y + Ax \]

```c
void smvm(int m, const double* values, const int* col_idx,
          const int* row_start, double* x, double* y)
{
    int i, j;
    double d;

    /* loop over m rows */
    for (i = 0; i < m; i++) {
        d = y[i]; /* scalar replacement since reused */

        /* loop over non-zero elements in row i */
        for (j = row_start[i]; j < row_start[i+1]; j++)
            d += values[j] * x[col_idx[j]];
        y[i] = d;
    }
}
```

CSR + sparse MVM: Advantages?

- **Advantages:**
  - Only nonzero values are stored
  - All three arrays for A (values, col_idx, row_start) accessed consecutively in MVM (good spatial locality)
  - Good temporal locality with respect to y

- **Disadvantages:**
  - Insertion into A is costly
  - Poor temporal locality with respect to x
Impact of Matrix Sparsity on Performance

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

- **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

- **Idea:** Blocking for registers
- **Reason:** Reuse x to reduce memory traffic
- **Execution:** Block SMVM $y = y + Ax$ into micro MVMs
  - Block size $r \times c$ becomes a parameter
  - Consequence: Change A from CSR to $r \times c$ block-CSR (BCSR)
- **BCSR:** Blackboard
BCSR (Blocks of Size $r \times c$)

- **Assumptions:**
  - $A$ is $m \times n$
  - Block size $r \times c$
  - $K_{r,c}$ nonzero blocks

$A$ as matrix ($r = c = 2$)

$A$ in BCSR ($r = c = 2$):

- **Storage:**
  - $rcK_{r,c}$ doubles + $(K_{r,c}+m/r+1)$ ints = $\Theta(rcK_{r,c})$
  - $rcK_{r,c} \geq K$

Sparse MVM Using 2 x 2 BCSR

```c
void smvm_2x2(int bm, const int *b_row_start, const int *b_col_idx,
               const double *b_values, double *x, double *y)
{
    int i, j;
    double d0, d1, c0, c1;

    /* loop over bm block rows */
    for (i = 0; i < bm; i++) {
        d0 = y[2*i];  /* scalar replacement since reused */
        d1 = y[2*i+1];

        /* dense micro MVM */
        for (j = b_row_start[i]; j < b_row_start[i+1]; j++, b_values += 2) {
            c0 = x[2*b_col_idx[j]+0];  /* scalar replacement since reused */
            c1 = x[2*b_col_idx[j]+1];
            d0 += b_values[0] * c0;
            d1 += b_values[1] * c1;
        }
        y[2*i] = d0;
        y[2*i+1] = d1;
    }
}
```
**BCSR**

- **Advantages:**
  - Temporal locality with respect to x and y
  - Reduced storage for indexes

- **Disadvantages:**
  - Storage for values of A increased (zeros added)
  - Computational overhead (also due to zeros)

- **Main factors (since memory bound):**
  - **Plus:** increased temporal locality on x + reduced index storage = reduced memory traffic
  - **Minus:** more zeros = increased memory traffic

---

**Which Block Size \((r \times c)\) is Optimal?**

**Example:**

- 20,000 \(\times\) 20,000 matrix (only part shown)
- Perfect 8 \(\times\) 8 block structure
- No overhead when blocked \(r \times c,\) with \(r, c\) divides 8

*source: R. Vuduc, LLENL*
How to Find the Best Blocking for given A?

- Best block size is hard to predict (see previous slide)

- **Solution 1**: Searching over all $r \times c$ within a range, e.g., $1 \leq r, c \leq 12$
  - Conversion of $A$ in CSR to BCSR roughly as expensive as 10 SMVMs
  - Total cost: 1440 SMVMs
  - Too expensive

- **Solution 2**: Model
  - Estimate the gain through blocking
  - Estimate the loss through blocking
  - Pick best ratio
Model: Example

Gain by blocking (dense MVM)

Overhead (average) by blocking

\[
\frac{16}{9} = 1.77
\]

\[
\frac{1.4}{1.77} = 0.79 \text{ (no gain)}
\]

Model: Doing that for all \( r \) and \( c \) and picking best

Model

- **Goal**: find best \( r \times c \) for \( y = y + Ax \)
- **Gain** through \( r \times c \) blocking (estimation):
  \[
  G_{r,c} = \frac{\text{dense MVM performance in } r \times c \text{ BCSR}}{\text{dense MVM performance in CSR}}
  \]
  dependent on machine, independent of sparse matrix
- **Overhead** through \( r \times c \) blocking (estimation)
  scan part of matrix \( A \)
  \[
  O_{r,c} = \frac{\text{number of matrix values in } r \times c \text{ BCSR}}{\text{number of matrix values in CSR}}
  \]
  independent of machine, dependent on sparse matrix
- **Expected gain**: \( G_{r,c} / O_{r,c} \)
Gain from Blocking (Dense Matrix in BCSR)

Pentium III

• machine dependent
• hard to predict

Itanium 2


Typical Result

Performance Summary — [pentium3-linux-icc]

CSR
BCSR model
BCSR exhaustive search

Analytical upper bound
how obtained?
(blackboard)

Principles in Bebop/Sparsity Optimization

- Optimization for memory hierarchy = increasing locality
  - Blocking for registers (micro-MVMs)
  - Requires change of data structure for \( A \)
  - Optimizations are input dependent (on sparse structure of \( A \))

- Fast basic blocks for small sizes (micro-MVM):
  - Unrolling + scalar replacement

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

Different from ATLAS

SMVM: Other Ideas

- Cache blocking
- Value compression
- Index compression
- Pattern-based compression
- Special scenario: Multiple inputs
**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 2.2x, only for few matrices, with 1 x 1 BCSR

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

---

**Value Compression**

- **Situation:** Matrix A contains many duplicate values

- **Idea:** Store only unique ones plus index information

*Figure: Kourtis, Goumas, and Koziris, Improving the Performance of Multithreaded Sparse Matrix-Vector Multiplication using Index and Value Compression, pp. 511-519, ICPP 2008*
Index Compression

- **Situation:** Matrix A contains sequences of nonzero entries
- **Idea:** Use special byte code to jointly compress col_idx and row_start

**Coding**

**Decoding**

0: \( \text{acc} = \text{acc} + 256 + \text{arg} \); \( \text{col} = \text{col} + 1 \); \( \text{emit}._{\text{element}}(\text{row}, \text{col}) \);
1: \( \text{col} = \text{col} + 256 + \text{arg} \); \( \text{acc} = 0 \);
2: \( \text{emit}._{\text{element}}(\text{row}, \text{col}) \);
3: \( \text{col} = \text{col} + 256 + \text{arg} \); \( \text{acc} = 0 \);
4: \( \text{emit}._{\text{element}}(\text{row}, \text{col}) \);
5: \( \text{row} = \text{row} + 1 \); \( \text{col} = 0 \);

Pattern-Based Compression

- **Situation:** After blocking A, many blocks have the same nonzero pattern
- **Idea:** Use special BCSR format to avoid storing zeros; needs specialized micro-MVM kernel for each pattern

**A as matrix**

\[
\begin{bmatrix}
\text{b} & \text{c} & \text{c} \\
\text{a} & & \\
\text{b} & \text{b} & \\
\text{c} & & \\
\end{bmatrix}
\]

**Values in 2 x 2 BCSR**

\[
\begin{bmatrix}
\text{b} & \text{c} & 0 & \text{a} & 0 & \text{c} & 0 & 0 & \text{b} & \text{b} & \text{c} & 0 \\
\end{bmatrix}
\]

**Values in 2 x 2 PBR**

\[
\begin{bmatrix}
\text{b} & \text{c} & \text{a} & \text{c} & \text{b} & \text{b} & \text{c} \\
\end{bmatrix}
\]

+ bit string: 1101 0100 1110
Special scenario: Multiple inputs

- Situation: Compute SMVM $y = y + Ax$ for several independent $x$
- Blackboard
- Experiments: up to 9x speedup for 9 vectors