Overview

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
- Cache size
- Peak performance

The algorithm
- Dependencies

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>L1 cache</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

Guess L2 cache size

Guess the read bandwidth from L1 cache

2 doubles/cycle

1 double/cycle

1/2 double/cycle

sum of vector lengths (working set)

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)

\[ y = y + Ax, \text{ } A \text{ sparse but known (below } A \text{ is square)} \]

Typically executed many times for fixed A

What is reused (possible temporal locality)?

Upper bound on operational intensity? \[ I(n) \leq \frac{2K}{8(K+3n)} \leq \frac{1}{4} \]
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

Popular for performance: Compressed sparse row (CSR) format

CSR

Assumptions:

- A is m x n
- K nonzero entries

A as matrix

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

A in CSR:

<table>
<thead>
<tr>
<th>values</th>
<th>col_idx</th>
<th>row_start</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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?

CSR

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 (e.g., when 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 x c becomes a parameter
- Consequence: Change A from CSR to r x c block_CSR (BCSR)

BCSR: Next slide
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 ($r = c = 2$)

$$\begin{array}{ccc}
    b & c & c \\
    a & & \\
    b & b & \\
    c & & \\
\end{array}$$

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

$$\begin{array}{cccc}
    b_{values} & b & c & 0 \\
    b_{col_idx} & 0 & 1 & 1 \\
    b_{row_start} & 0 & 2 & 3 \\
\end{array}$$

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

Sparse MVM Using $2 \times 2$ 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++) {
            c0 = x[2*b_col_idx[j]+0]; /* scalar replacement since reused */
            c1 = x[2*b_col_idx[j]+1];
            d0 += b_values[j] * c0;
            d1 += b_values[j] * 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)

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

source: R. Vuduc, Georgia Tech
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
- So 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

16/9 = 1.77

1.4/1.77 = 0.79 (no gain)

Model: Doing that for all r and c and picking best

Model

Goal: find best r x c for y = y + Ax

Gain through r x 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 x 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)

- machine dependent
- hard to predict

Typical Result (assumes cold cache)

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

Value compression
Index compression
Pattern-based compression
Multiple inputs
Value Compression

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

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

![Matrix A in CSR and CSR-VI formats]

Kourtis, Goumas, and Kaziris, 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 and Decoding diagrams]

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

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

Values in 2 x 2 BCSR

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

Values in 2 x 2 PBR

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

+ bit string: 1101 0100 1110

Source: Belgin, Back, and Ribbens, Pattern-based Sparse Matrix Representation for Memory-Efficient SMVM Kernels, pp. 100-109, ICS 2009

Multiple Inputs

Situation: Compute SMVM \( y = y + Ax \) for several independent \( x \)

Experiments: up to 9x speedup for 9 vectors

Multiple Vector Performance: Ultra 2