Optimising SpMV and SpMM
Reducing communication using a bit-mapped format

Ramaseshan Kannan
rkannan@ maths. man. ac. uk

School of Maths, University of Manchester and Arup UK

Sparse Days 2012, Toulouse
Sparse Matrix:

A matrix that contains enough zero entries to be worth taking advantage of them to reduce computation and storage.
Sparse Matrix:

A matrix that contains enough zero entries to be worth taking advantage of them to reduce computation and storage.

**SpMV**

\[ y = y + Ax \]
\[ A \in \mathbb{R}^{n \times n}, \ x \in \mathbb{R}^n \]

**SpMM**

\[ Y = Y + AX = A \begin{bmatrix} x_1 & x_2 & \ldots & x_\ell \end{bmatrix} \]
\[ A \in \mathbb{R}^{n \times n}, \ X \in \mathbb{R}^{n \times \ell} \]
Sparse Matrix:
A matrix that contains enough zero entries to be worth taking advantage of them to reduce computation and storage.

**SpMV**

\[ y = y + Ax \]
\[ A \in \mathbb{R}^{n \times n}, \; x \in \mathbb{R}^{n} \]

**Motivation**

\[ M = X^T AX \]

**SpMM**

\[ Y = Y + AX = A \begin{bmatrix} x_1 & x_2 & \ldots & x_\ell \end{bmatrix} \]
\[ A \in \mathbb{R}^{n \times n}, \; X \in \mathbb{R}^{n \times \ell} \]
Compressed Sparse Row (CSR)

\[
A = \begin{pmatrix}
 a_{00} & a_{01} & a_{02} & a_{03} \\
 a_{10} & a_{11} & 0 & 0 \\
 0 & 0 & a_{22} & a_{23} \\
 0 & 0 & a_{32} & 0
\end{pmatrix}
\]

\[
\text{val} = (a_{00}, a_{01}, a_{02}, a_{03}, a_{10}, a_{11}, a_{22}, a_{23}, a_{32})
\]

\[
\text{col_idx} = (0, 1, 2, 3, 0, 1, 2, 3, 2)
\]

\[
\text{row_start} = (0, 4, 6, 8, 9)
\]
Compressed Sparse Row (CSR)

\[ A = \begin{pmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & 0 & 0 \\
  0 & 0 & a_{22} & a_{23} \\
  0 & 0 & a_{32} & 0
\end{pmatrix} \]

val = \( (a_{00}, a_{01}, a_{02}, a_{03}, a_{10}, a_{11}, a_{22}, a_{23}, a_{32}) \)

col_idx = \( (0, 1, 2, 3, 0, 1, 2, 3, 2) \)

row_start = \( (0, 4, 6, 8, 9) \)

Storage \( \approx 2z + z + n \) words where \( z = \) number of non-zeros.
int i, j, k;
double yi;
for(i=0; i<n; ++i)
{
    yi = 0.;
    for(j = row[i]; j<row[i+1]; ++j)
    {
        yi += val[j] * x[col[j]];
    }
    y[i] = yi;
}
Memory Access Pattern

Disadvantage
- Poor cache and register reuse
Memory Access Pattern

Disadvantage

▶ Poor cache and register reuse
Block Compressed Row (BCSR)

\[ A = \begin{pmatrix}
    a_{00} & a_{01} & a_{02} & a_{03} \\
    a_{10} & a_{11} & 0 & 0 \\
    0 & 0 & a_{22} & a_{23} \\
    0 & 0 & a_{32} & 0 \\
\end{pmatrix} \]

\( r = 2, c = 2 \)

\[ \text{val} = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, 0, 0, a_{22}, a_{23}, a_{32}, 0) \]

\[ \text{col_idx} = (0, 1, 1) \]

\[ \text{row_start} = (0, 2, 3) \]
Block Compressed Row (BCSR)

\[ A = \begin{pmatrix}
    a_{00} & a_{01} & a_{02} & a_{03} \\
    a_{10} & a_{11} & 0 & 0 \\
    0 & 0 & a_{22} & a_{23} \\
    0 & 0 & a_{32} & 0 \\
\end{pmatrix} \]

\[ r = 2, \ c = 2 \]

\[ \text{val} = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, 0, 0, a_{22}, a_{23}, a_{32}, 0) \]

\[ \text{col_idx} = (0, 1, 1) \]

\[ \text{row_start} = (0, 2, 3) \]

Storage \( \approx 2zf_{rc} + \frac{zf_{rc}}{rc} + \frac{n}{r} \) words where

\[ f_{rc} := \frac{nnb \times r \times c}{z}. \]
void bsmvm_2x2_1 (int bm,
        int *row_start, int *col_idx, double *value,
        double *x, double *y)
{
    int i, j;

    for (i=0; i<bm; i++, dest+=2)
    {
        register double d0, d1;
        d0 = y[0];
        d1 = y[1];
        for (j=row_start[i]; j<row_start[i+1]; j++, col_idx++, value+=4)
        {
            d0 += value[0] * x[*col_idx+0];
            d1 += value[2] * x[*col_idx+0];
            d0 += value[1] * x[*col_idx+1];
            d1 += value[3] * x[*col_idx+1];
        }
        y[0] = d0;
        y[1] = d1;
    }
}
Memory access pattern

Disadvantages

▶ Involves storing extra zeros
▶ Performance sensitive to matrix structure
▶ Awkward to handle dimensions that don’t divide by block size
Memory access pattern

Disadvantages

- Involves storing extra zeros
- Performance sensitive to matrix structure
- Awkward to handle dimensions that don’t divide by block size
Storing blocks as sparse blocks?

\[ A = \begin{pmatrix} 
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & 0 & 0 \\
  0 & 0 & a_{22} & a_{23} \\
  0 & 0 & a_{32} & 0 
\end{pmatrix} \]
Storing blocks as sparse blocks?

\[
A = \begin{pmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & 0 & 0 \\
  0 & 0 & a_{22} & a_{23} \\
  0 & 0 & a_{32} & 0
\end{pmatrix}
\]

\[a_{02} a_{03} + \begin{pmatrix}
  1 & 1 \\
  0 & 0
\end{pmatrix}\]
Storing blocks as sparse blocks?

\[ A = \begin{pmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & 0 & 0 \\
  0 & 0 & a_{22} & a_{23} \\
  0 & 0 & a_{32} & 0
\end{pmatrix} \]

\[ a_{02} a_{03} + \begin{pmatrix}
  1 & 1 \\
  0 & 0
\end{pmatrix} \]

\[ \begin{pmatrix}
  1 & 1 \\
  0 & 0
\end{pmatrix} \rightarrow 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 = 3 \]
Mmapped Blocked Sparse Row (MBR)

\[
A = \begin{pmatrix}
  a_{00} & a_{01} & a_{02} & a_{03} \\
  a_{10} & a_{11} & 0 & 0 \\
  0 & 0 & a_{22} & a_{23} \\
  0 & 0 & a_{32} & 0
\end{pmatrix}
\]

\[r = 2, c = 2\]
\[val = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32})\]
\[col_idx = (0, 1, 1)\]
\[b_map = (15, 3, 7)\]
\[row_start = (0, 2, 3)\]
Mapped Blocked Sparse Row (MBR)

\[ A = \begin{pmatrix}
    a_{00} & a_{01} & a_{02} & a_{03} \\
    a_{10} & a_{11} & 0 & 0 \\
    0 & 0 & a_{22} & a_{23} \\
    0 & 0 & a_{32} & 0
\end{pmatrix} \]

- \( r = 2, c = 2 \)
- \( \text{val} = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \)
- \( \text{col_idx} = (0, 1, 1) \)
- \( \text{b_map} = (15, 3, 7) \)
- \( \text{row_start} = (0, 2, 3) \)

**Motivation**
Mapped Blocked Sparse Row (MBR)

\[
A = \begin{pmatrix}
 a_{00} & a_{01} & a_{02} & a_{03} \\
 a_{10} & a_{11} & 0 & 0 \\
 0 & 0 & a_{22} & a_{23} \\
 0 & 0 & a_{32} & 0 \\
\end{pmatrix}
\]

\[
r = 2, c = 2
\]

\[
\text{val} = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32})
\]

\[
\text{col_idx} = (0, 1, 1)
\]

\[
\text{b_map} = (15, 3, 7)
\]

\[
\text{row_start} = (0, 2, 3)
\]

Motivation

- Storage \(\leq 2z + \frac{z}{r} \left(1 + \frac{1}{\delta}\right) + \frac{n}{r} \) words where \(\delta = \frac{\text{sizeof(int)}}{\text{sizeof(maptype)}}\)
Mapped Blocked Sparse Row (MBR)

\[ A = \begin{pmatrix}
    a_{00} & a_{01} & a_{02} & a_{03} \\
    a_{10} & a_{11} & 0 & 0 \\
    0 & 0 & a_{22} & a_{23} \\
    0 & 0 & a_{32} & 0
\end{pmatrix} \]

\[ r = 2, \ c = 2 \]

\[ \text{val} = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \]

\[ \text{col_idx} = (0, 1, 1) \]

\[ \text{b_map} = (15, 3, 7) \]

\[ \text{row_start} = (0, 2, 3) \]

Motivation

- Storage \( \leq 2z + \frac{z}{r} \left(1 + \frac{1}{\delta}\right) + \frac{n}{r} \) words where \( \delta = \frac{\text{sizeof(int)}}{\text{sizeof(maptype)}} \)
- Bitwise operations highly optimized
### Storage

<table>
<thead>
<tr>
<th>CSR</th>
<th>BSR</th>
<th>MBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3z + n$</td>
<td>$2zf + \frac{zf}{rc} + \frac{n}{r}$</td>
<td>$2z + \frac{z}{r} \left( 1 + \frac{1}{\delta} \right) + \frac{n}{r}$</td>
</tr>
</tbody>
</table>
SpMV for MBR
Compute $y = y + Ax$ for $A$ in MBR format with block dimensions $(r, c)$

1. Given block dimensions $(r, c)$ and $x, y \in \mathbb{R}^n$
2. for each block row $bi$
   3. for each block column $bj$ in block row $bi$
      4. $map = b_{map}_{bj}$
      5. for each bit position $p$ in $map$
         6. if bit($map_p$) = 1
            7. $i = p \% r, j = p \% c$
            8. $y(i) += val \times x(j)$
            9. increment $val$
      10. end
   11. end
12. end
13. end
Challenges

Decoding blocks implies iterating through all set bits

- Compiler cannot optimize loops with conditionals.
- Poor performance because processor cannot predict branches accurately
- Work done in decoding blocks not amortized
Optimizations
Decoding in $O(n_{\text{set bits}})$ time

1. Given block dimensions $(r, c)$ and $x, y \in \mathbb{R}^n$
2. for each block row $bi$
3.     for each block column $bj$ in row $bi$
4.         $map = b_{\_map_{bj}}$
5.         for each set bit $p$ in $map$
6.             $i := p \% r, j := p \% c$
7.             $y(i) += *val \times x(j)$
8.         increment val
9.     end
10. end
11. end
Optimizations

Use de Bruijn sequences to find index of a bit

Given an 8 bit number $x = 01101000$, find index of the last 1.
Optimizations
Use de Bruijn sequences to find index of a bit

Given an 8 bit number $x = 01101000$, find index of the last 1.
Optimizations
Use de Bruijn sequences to find index of a bit

Given an 8 bit number \( x = 01101000 \), find index of the last 1.

- Isolate trailing bit by using 2’s complement of \( x \):
  \[ y = (x) \& \neg x = 00001000. \]
Optimizations

Use de Bruijn sequences to find index of a bit

Given an 8 bit number $x = 01101000$, find index of the last 1.

- Isolate trailing bit by using 2’s complement of $x$:
  $$y = (x)\&(-x) = 00001000.$$

- Pick a *de Bruijn* sequence, e.g. 00011101 and generate its hash table

<table>
<thead>
<tr>
<th>$h(x)$</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>6</td>
</tr>
<tr>
<td>011</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>3</td>
</tr>
</tbody>
</table>
Optimizations

Use de Bruijn sequences to find index of a bit

Given an 8 bit number \( x = 01101000 \), find index of the last 1.

- Isolate trailing bit by using 2’s complement of \( x \):
  \[ y = (x) & (-x) = 00001000. \]
- Pick a *de Bruijn* sequence, e.g. 00011101 and generate its hash table
- Multiply *de Bruijn* by \( y \) to get 11010000

<table>
<thead>
<tr>
<th>( h(x) )</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>6</td>
</tr>
<tr>
<td>011</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>3</td>
</tr>
</tbody>
</table>
Optimizations

Use de Bruijn sequences to find index of a bit

Given an 8 bit number \( x = 01101000 \), find index of the last 1.

- Isolate trailing bit by using 2’s complement of \( x \):
  \[
  y = (x) \& (\neg x) = 00001000.
  \]
- Pick a de Bruijn sequence, e.g. 00011101 and generate its hash table
- Multiply de Bruijn by \( y \) to get 11010000
- Right shift by \( 8 - \log_2 8 = 5 \) to get 110

<table>
<thead>
<tr>
<th>( h(x) )</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>6</td>
</tr>
<tr>
<td>011</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>3</td>
</tr>
</tbody>
</table>
Optimizations

Use de Bruijn sequences to find index of a bit

Given an 8 bit number $x = 01101000$, find index of the last 1.

- Isolate trailing bit by using 2’s complement of $x$: $y = (x) \& (-x) = 00001000$.
- Pick a de Bruijn sequence, e.g. 00011101 and generate its hash table

<table>
<thead>
<tr>
<th>$h(x)$</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>6</td>
</tr>
<tr>
<td>011</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
</tr>
<tr>
<td>110</td>
<td>4</td>
</tr>
<tr>
<td>111</td>
<td>3</td>
</tr>
</tbody>
</table>

- Multiply de Bruijn by $y$ to get 11010000
- Right shift by $8 - \log_2 8 = 5$ to get 110
- Lookup table to retrieve 4
Amortizing decoding costs

1. Given ... and $x, y \in \mathbb{R}^n$
2. for each block row $bi$
3. for each block column $bj$ in row $bi$
4. $map = b\_map_{bj}$
5. for each set bit $p$ in $map$
6. $i := p \% r, j := p \% c$
7. $y(i) += *val \times x(j)$
8. increment $val$
9. end
10. end
11. end
Optimizations

Amortizing decoding costs

1. Given ... and $x_1 \cdots x_\ell$, $Y \in \mathbb{R}^{n \times \ell}$
2. for each block row $b_i$
3.   for each block column $b_j$ in row $b_i$
4.       $map = b_{map_{bj}}$
5.   for each set bit $p$ in $map$
6.       $i := p \% r, j := p \% c$
7.       $y_1(i) + = *val \times x_1(j)$
8.       
9.       $y_\ell(i) + = *val \times x_\ell(j)$
10.      increment $val$
11.     end
12.  end
13. end
Optimizations
Template meta-programming

- Template–based loop unroller

```cpp
for(; bmap; bmap &= bmap-1) {
    r = lastTrailingBit(bmap);
    i = r>>2; j = r&3; val = *b_value;
    y0[i] += val * x0[j];
    y1[i] += val * x1[j];
    ++b_value;
}
```
Optimizations

Template meta-programming

- Template–based loop unroller

```cpp
__forceinline void multiplier_impl(double d[], double s[],
const double& val) {
    d[0] += val * s[0];
}

template<int nrhs, int rhs> __forceinline void multiplier(double d[],
double s[], const double& val) {
    multiplier_impl(d + 4*rhs, s + 4*rhs, val);
    multiplier<nrhs, rhs+1>(d, s, val);
}

for(;bmap; bmap &= bmap-1) {
    r = lastTrailingBit(bmap);
    i = r>>2; j = r&3; val = *b_value;
    multiplier<nrhs, 0>(d + idx_d, s + idx_s, val);
    ++b_value;
}
```
Optimizations

Template meta-programming

- Generating compile–time bound kernels from a single codebase

```cpp
template<int nrhs, typename MapType>
void mbsmvm_8x8_x(int m, int *b_row_start, int *b_col_idx,
MapType* b_map, double *b_value,
double *src, double *dest)
```
Optimizations

Template meta-programming

- Generating compile–time bound kernels from a single codebase

```cpp
template<int nrhs, typename MapType>
void mbsmvm_8x8_x(int m, int *b_row_start, int *b_col_idx,
                   MapType* b_map, double *b_value,
                   double *src, double *dest)
```
Results

- Range of matrices from University of Florida Sparse Matrix collection and structural stiffness matrices
- Performance compared with Intel MKL dcsrgemv, CSR SpMV and BCSR SpMV (wherever applicable)
- Performance over Intel and AMD using ICC
- MS VC vs ICC – very little difference
Results

- Range of matrices from University of Florida Sparse Matrix collection and structural stiffness matrices
Results

- Range of matrices from University of Florida Sparse Matrix collection and structural stiffness matrices
- Performance compared with Intel MKL `dcsrgemv`, CSR SpMV and BCSR SpMV (wherever applicable).

Ramaseshan Kannan

Optimising SpMV and SpMM
Results

- Range of matrices from University of Florida Sparse Matrix collection and structural stiffness matrices
- Performance compared with Intel MKL `dcsrgemv`, CSR SpMV and BCSR SpMV (wherever applicable).
- Performance over Intel and AMD using ICC
  - MS VC vs ICC – very little difference
## Test matrices

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Source</th>
<th>Dimension</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC_680k</td>
<td>U.Florida</td>
<td>682862</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>diefilterV2real</td>
<td>U.Florida</td>
<td>1157456</td>
<td>Electromagnetics</td>
</tr>
<tr>
<td>diefilterV3real</td>
<td>U.Florida</td>
<td>1102824</td>
<td>Electromagnetics</td>
</tr>
<tr>
<td>nlpkkt80</td>
<td>U.Florida</td>
<td>1062400</td>
<td>Optimization</td>
</tr>
<tr>
<td>ecology1</td>
<td>U.Florida</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>random matrix</td>
<td>MATLAB</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>hamrle3</td>
<td>U.Florida</td>
<td>1447360</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>rajat29</td>
<td>U.Florida</td>
<td>643994</td>
<td>Circuit simulation</td>
</tr>
<tr>
<td>Sports hub</td>
<td>Arup</td>
<td>143460</td>
<td>FEA</td>
</tr>
<tr>
<td>Water cube</td>
<td>Arup</td>
<td>68598</td>
<td>FEA</td>
</tr>
</tbody>
</table>
Performance variation with number of rhs

$4 \times 4$ blocks, across problems
Performance comparison

Intel Xeon E5450

![Graph showing performance comparison between CSR, MKL, and MBR best for Intel Xeon E5450.]
Performance comparison
AMD Opteron 6220

[Chart showing performance comparison between different methods relative to MKL.]
Performance comparison – with BSR
Intel Core i7–2720QM

![Graph showing performance comparison with BSR]
Summary

Mapped Blocked Sparse is a sparse-blocked format that provides superior reuse of CPU registers for SpMV and SpMM.

Performance is less sensitive to the non–zero structure of the matrix in comparison to BSR.

An efficient algorithm for decoding the blocks avoids delays due to conditionals and branch prediction.

For SpMM, performance gains of up to 3.1x over MKL dcsrgemv and 4.7x over naive CSR are obtained.
Summary

- Mapped Blocked Sparse is a sparse-blocked format that provides superior reuse of CPU registers for SpMV and SpMM

- Performance is less sensitive to the non–zero structure of the matrix in comparison to BSR

- An efficient algorithm for decoding the blocks avoids delays due to conditionals and branch prediction.

- For SpMM, performance gains of upto 3.1x over MKL dcsrgemv and 4.7x over naive CSR are obtained.
Summary

- Mapped Blocked Sparse is a sparse-blocked format that provides superior reuse of CPU registers for SpMV and SpMM.
- Performance is less sensitive to the non–zero structure of the matrix in comparison to BSR.
- An efficient algorithm for decoding the blocks avoids delays due to conditionals and branch prediction.
- For SpMM, performance gains of up to 3.1x over MKL dcsrgemv and 4.7x over naive CSR are obtained.
Summary

- Mapped Blocked Sparse is a sparse-blocked format that provides superior reuse of CPU registers for SpMV and SpMM
- Performance is less sensitive to the non–zero structure of the matrix in comparison to BSR
- An efficient algorithm for decoding the blocks avoids delays due to conditionals and branch prediction.
Summary

- Mapped Blocked Sparse is a sparse-blocked format that provides superior reuse of CPU registers for SpMV and SpMM.
- Performance is less sensitive to the non–zero structure of the matrix in comparison to BSR.
- An efficient algorithm for decoding the blocks avoids delays due to conditionals and branch prediction.
- For SpMM, performance gains of up to 3.1x over MKL dcsrgemv and 4.7x over naive CSR are obtained.
Further work

- Test suitability for other blocked operations like sparse $LDL^T$ (?)
- Examine performance in SMP case
- Need a suite of tools like sparse Cholesky, backward substitution etc.