MANCHESTER 1824

The University of Manchester

ARUP

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

$$Y = Y + AX = A \begin{bmatrix} x_1 & x_2 & \dots & x_\ell \end{bmatrix}$$

$$A \in \mathbb{R}^{n \times n},\ X \in \mathbb{R}^{n \times \ell}$$

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

$$M = X^T A X$$

SpMM

$$Y = Y + AX = A \begin{bmatrix} x_1 & x_2 & \dots & x_\ell \end{bmatrix}$$

$$A \in \mathbb{R}^{n \times n},\ X \in \mathbb{R}^{n \times \ell}$$

Compressed Sparse Row (CSR)

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

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

$$\texttt{col\_idx} = (0, 1, 2, 3, 0, 1, 2, 3, 2)$$

$$\texttt{row\_start} = (0, 4, 6, 8, 9)$$

Compressed Sparse Row (CSR)

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

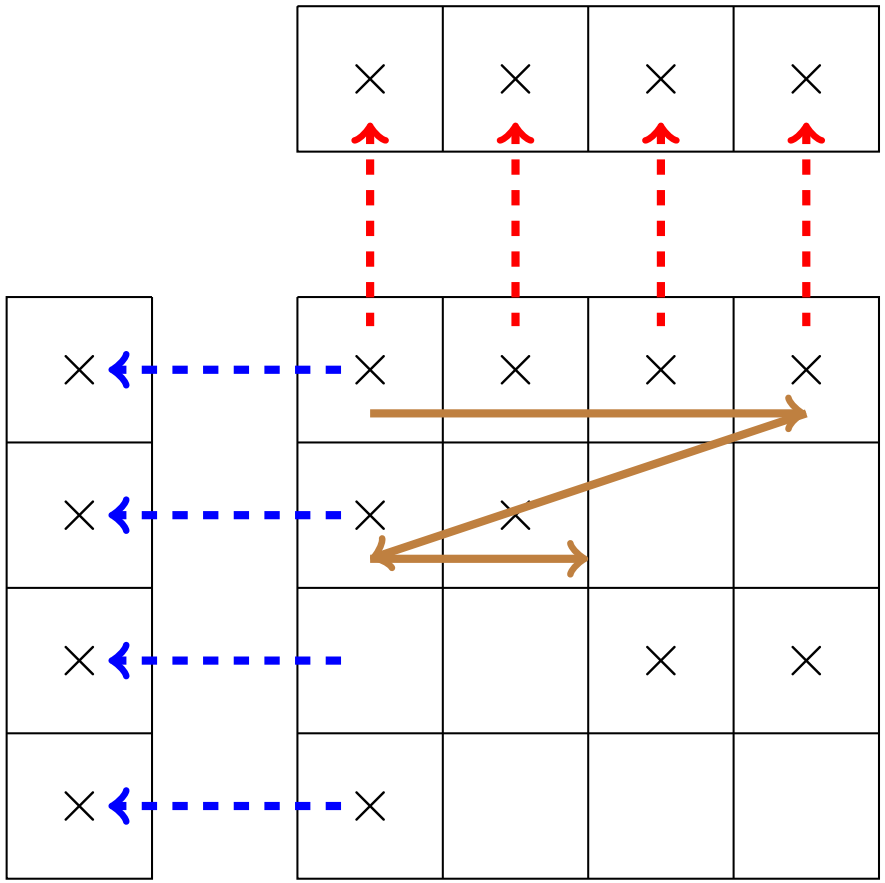
$$\texttt{val} = (a_{00}, a_{01}, a_{02}, a_{03}, a_{10}, a_{11}, a_{22}, a_{23}, a_{32})$$
$$\texttt{col\_idx} = (0, 1, 2, 3, 0, 1, 2, 3, 2)$$
$$\texttt{row\_start} = (0, 4, 6, 8, 9)$$

Storage $\approxeq 2z + z + n$ words where $z =$ number of non zeros.

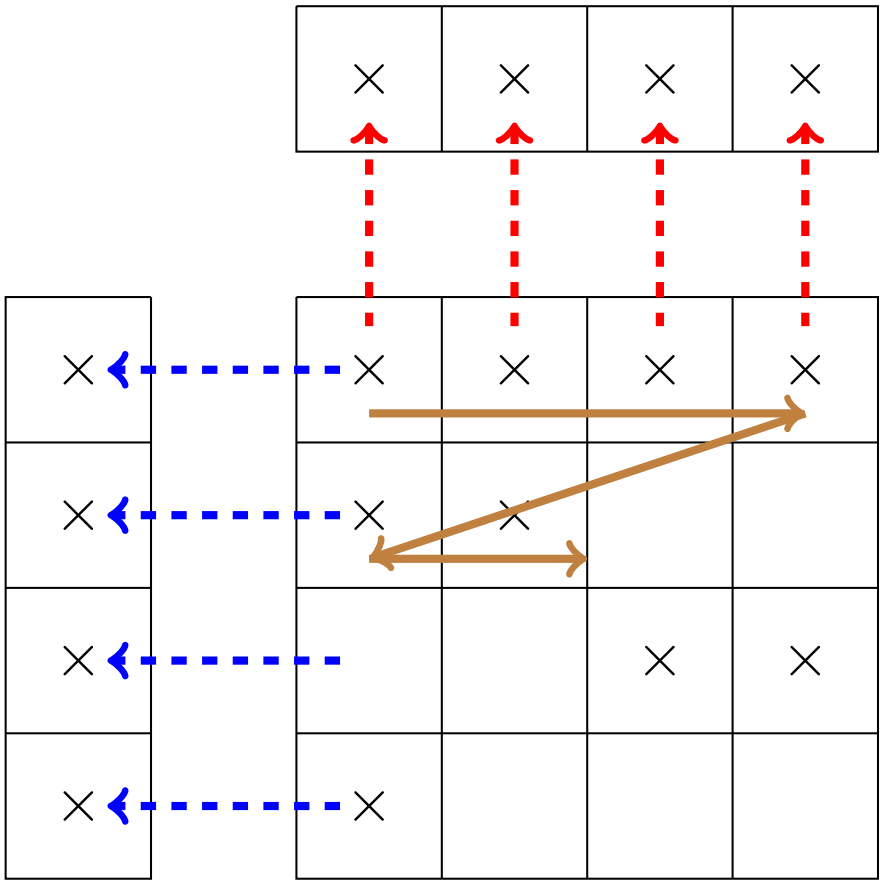
CSR – SpMV

```
1  int i, j, k;
2  double yi;
3  for(i=0; i<n; ++i)
4  {
5    yi = 0.;
6    for(j = row[i]; j<row[i+1]; ++j)
7    {
8      yi += val[j] * x[col[j]];
9    }
10   y[i] = yi;
11 }
```

Memory Access Pattern



Memory Access Pattern



Disadvantage

- Poor cache and register reuse

Block Compressed Row (BCSR)

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

$$\mathtt{r} = 2, \mathtt{c} = 2$$

$$\mathtt{val} = (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, 0, 0, a_{22}, a_{23}, a_{32}, 0)$$

$$\mathtt{col\_idx} = (0, 1, 1)$$

$$\mathtt{row\_start} = (0, 2, 3)$$

Block Compressed Row (BCSR)

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

$$\begin{aligned} \texttt{r} &= 2, \texttt{c} = 2 \\ \texttt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, 0, 0, a_{22}, a_{23}, a_{32}, 0) \\ \texttt{col\_idx} &= (0, 1, 1) \\ \texttt{row\_start} &= (0, 2, 3) \end{aligned}$$

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

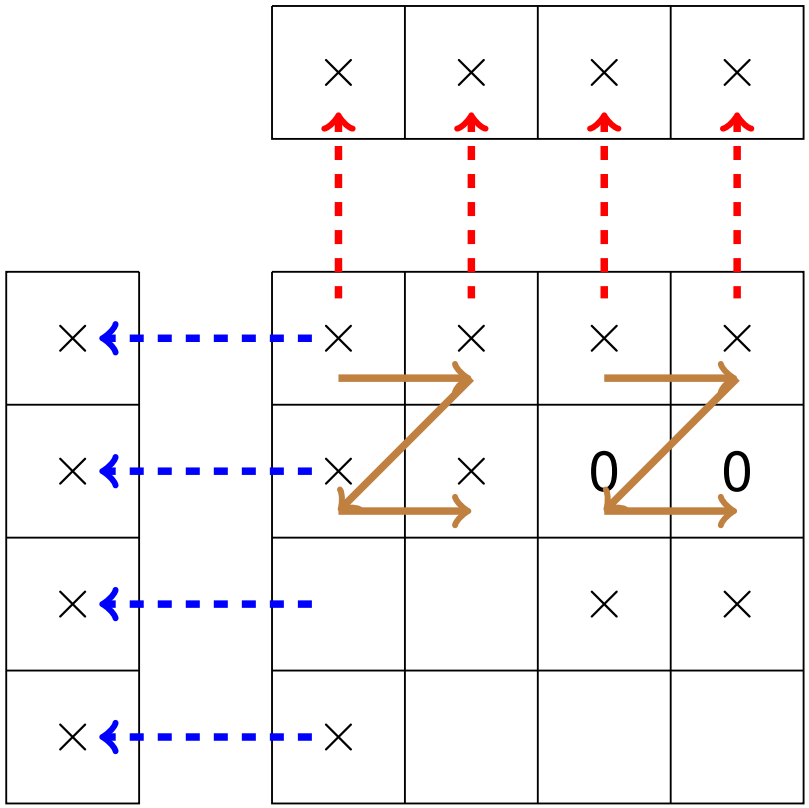
$$f_{rc} := \frac{nnb \times r \times c}{z}.$$

SpMV With BCSR

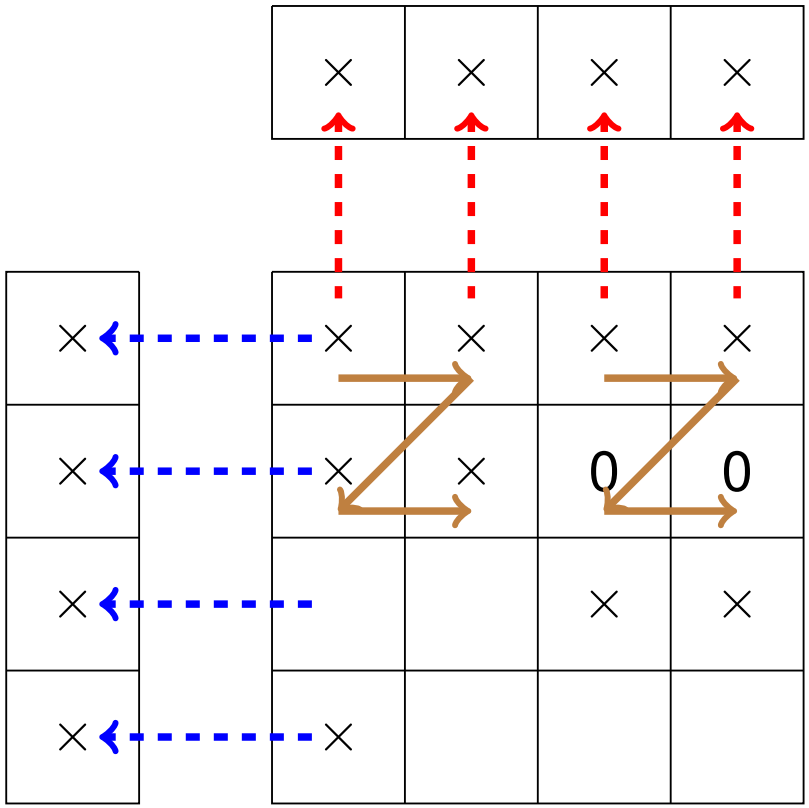
2×2 blocks

```
1  void bsmvm_2x2_1 (int bm,
2                    int *row_start, int *col_idx, double *value,
3                    double *x, double *y)
4  {
5    int i, j;
6
7    for (i=0; i<bm; i++,dest+=2)
8    {
9      register double d0, d1;
10     d0 = y[0];
11     d1 = y[1];
12     for (j=row_start[i]; j<row_start[i+1]; j++,col_idx++,value+=4)
13     {
14       d0 += value[0] * x[*col_idx+0];
15       d1 += value[2] * x[*col_idx+0];
16       d0 += value[1] * x[*col_idx+1];
17       d1 += value[3] * x[*col_idx+1];
18     }
19     y[0] = d0;
20     y[1] = d1;
21   }
22 }
```

Memory access pattern



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?

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

$$a_{02}\ a_{03} + \begin{matrix} 1 & 1 \\ 0 & 0 \end{matrix}$$

Storing blocks as sparse blocks?

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

$$a_{02}\ a_{03} \quad + \quad \begin{matrix} 1 & 1 \\ 0 & 0 \end{matrix}$$

$$\begin{matrix} 1 & 1 \\ 0 & 0 \end{matrix} \rightarrow 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 = 3$$

Mapped Blocked Sparse Row (MBR)

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

$$\begin{aligned} \texttt{r} &= 2, \texttt{c} = 2 \\ \texttt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \\ \texttt{col\_idx} &= (0, 1, 1) \\ \texttt{b\_map} &= (15, 3, 7) \\ \texttt{row\_start} &= (0, 2, 3) \end{aligned}$$

Mapped Blocked Sparse Row (MBR)

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

$$\begin{aligned} \texttt{r} &= 2, \texttt{c} = 2 \\ \texttt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \\ \texttt{col\_idx} &= (0, 1, 1) \\ \texttt{b\_map} &= (15, 3, 7) \\ \texttt{row\_start} &= (0, 2, 3) \end{aligned}$$

Motivation

Mapped Blocked Sparse Row (MBR)

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

$$\begin{aligned} \mathtt{r} &= 2, \mathtt{c} = 2 \\ \mathtt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \\ \mathtt{col\_idx} &= (0, 1, 1) \\ \mathtt{b\_map} &= (15, 3, 7) \\ \mathtt{row\_start} &= (0, 2, 3) \end{aligned}$$

Motivation

- Storage $\leqslant 2z + \frac{z}{r}\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$ words where $\delta = \frac{\mathtt{sizeof(int)}}{\mathtt{sizeof(maptype)}}$

Mapped Blocked Sparse Row (MBR)

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

$$\begin{aligned} \mathtt{r} &= 2, \mathtt{c} = 2 \\ \mathtt{val} &= (a_{00}, a_{01}, a_{10}, a_{11}, a_{02}, a_{03}, a_{22}, a_{23}, a_{32}) \\ \mathtt{col\_idx} &= (0, 1, 1) \\ \mathtt{b\_map} &= (15, 3, 7) \\ \mathtt{row\_start} &= (0, 2, 3) \end{aligned}$$

Motivation

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

Storage

CSR	BSR	MBR
$3z + n$	$2zf + \frac{zf}{rc} + \frac{n}{r}$	$2z + \frac{z}{r}\left(1 + \frac{1}{\delta}\right) + \frac{n}{r}$

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 ∈ 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 × 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 $\mathcal{O}(n_{\text{set bits}})$ time

```
1  Given block dimensions (r, c) and x, y ∈ 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 × 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:
  $y = (x)\&(-x) = 00001000$.

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

$h(x)$	Index
000	0
001	1
010	6
011	2
100	7
101	5
110	4
111	3

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

$h(x)$	Index
000	0
001	1
010	6
011	2
100	7
101	5
110	4
111	3

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
- Right shift by $8 - \log_2 8 = 5$ to get 110

$h(x)$	Index
000	0
001	1
010	6
011	2
100	7
101	5
110	4
111	3

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
- Right shift by $8 - \log_2 8 = 5$ to get 110
- Lookup table to retrieve 4

$h(x)$	Index
000	0
001	1
010	6
011	2
100	7
101	5
110	4
111	3

Optimizations

Amortizing decoding costs

```
1  Given ... and x, y ∈ 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 × 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 *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 $y_1(i) += \texttt{*val} \times x_1(j)$
8 $\vdots$
9 $y_\ell(i) += \texttt{*val} \times x_\ell(j)$
10 increment `val`
11 end
12 end
13 end

Optimizations

Template meta-programming

- Template–based loop unroller

```
1  for(;bmap;bmap &= bmap-1) {
2    r = lastTrailingBit(bmap);
3    i = r>>2; j = r&3; val = *b_value;
4    y0[i] += val * x0[j];
5    y1[i] += val * x1[j];
6    ++b_value;
7  }
```

Optimizations

Template meta-programming

- Template–based loop unroller

```
__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

Optimizations

Template meta-programming

- Generating compile–time bound kernels from a single codebase

```
1 template<int nrhs, typename MapType>
2 void mbsmvm_8x8_x(int m, int *b_row_start, int *b_col_idx,
3                    MapType* b_map, double *b_value,
4                    double *src, double *dest)
```

Results

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).

Results

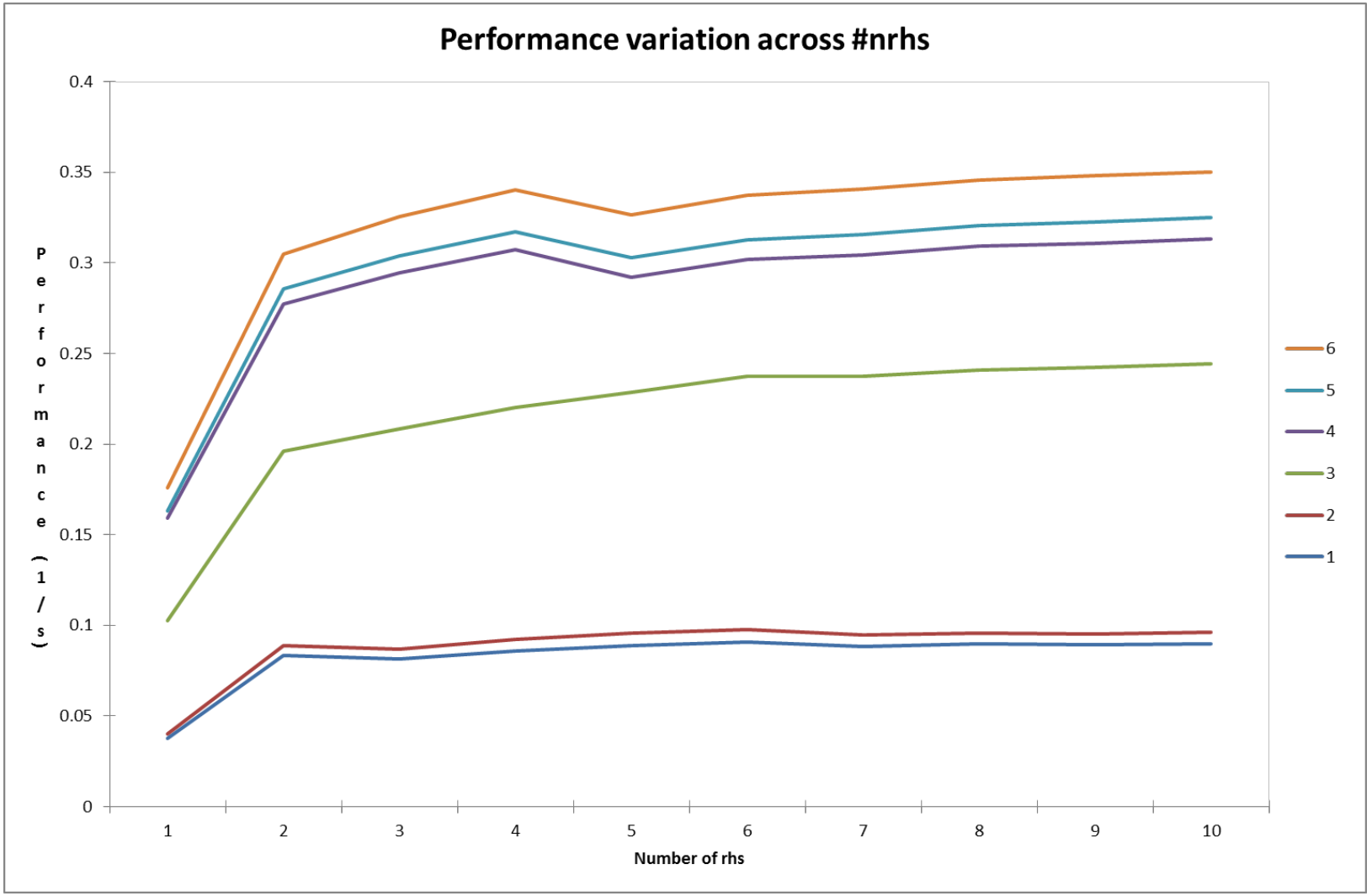
- Range of matrices from University of Florida Sparse Matrix collection and structural stiffness matrices
- Performance compared with Intel MKL `dcsrgemv`, CSR SpMV and BCSR SpMV (wherever applicable).
- Performance over Intel and AMD using ICC
  - MS VC vs ICC – very little difference

Test matrices

	Matrix	Source	Dimension	Application
1	ASIC_680k	U.Florida	682862	Circuit simulation
2	dielfilterV2real	U.Florida	1157456	Electromagnetics
3	dielfilterV3real	U.Florida	1102824	Electromagnetics
4	nlpkkt80	U.Florida	1062400	Optimization
5	ecology1	U.Florida	1000000	
6	random matrix	MATLAB	10000	
7	hamrle3	U.Florida	1447360	Circuit simulation
8	rajat29	U.Florida	643994	Circuit simulation
9	Sports hub	Arup	143460	FEA
10	Water cube	Arup	68598	FEA

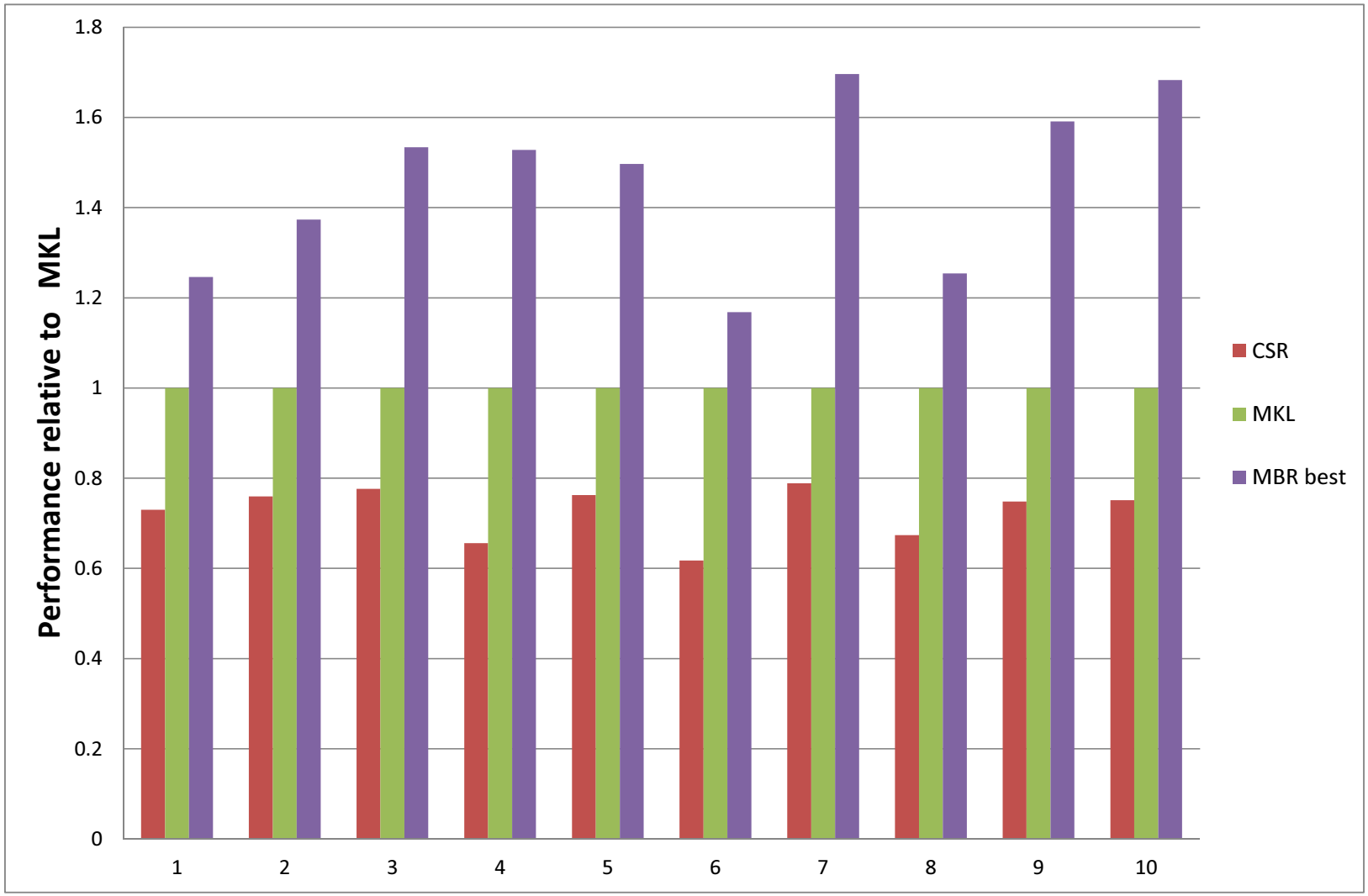
Performance variation with number of rhs

4×4 blocks, across problems



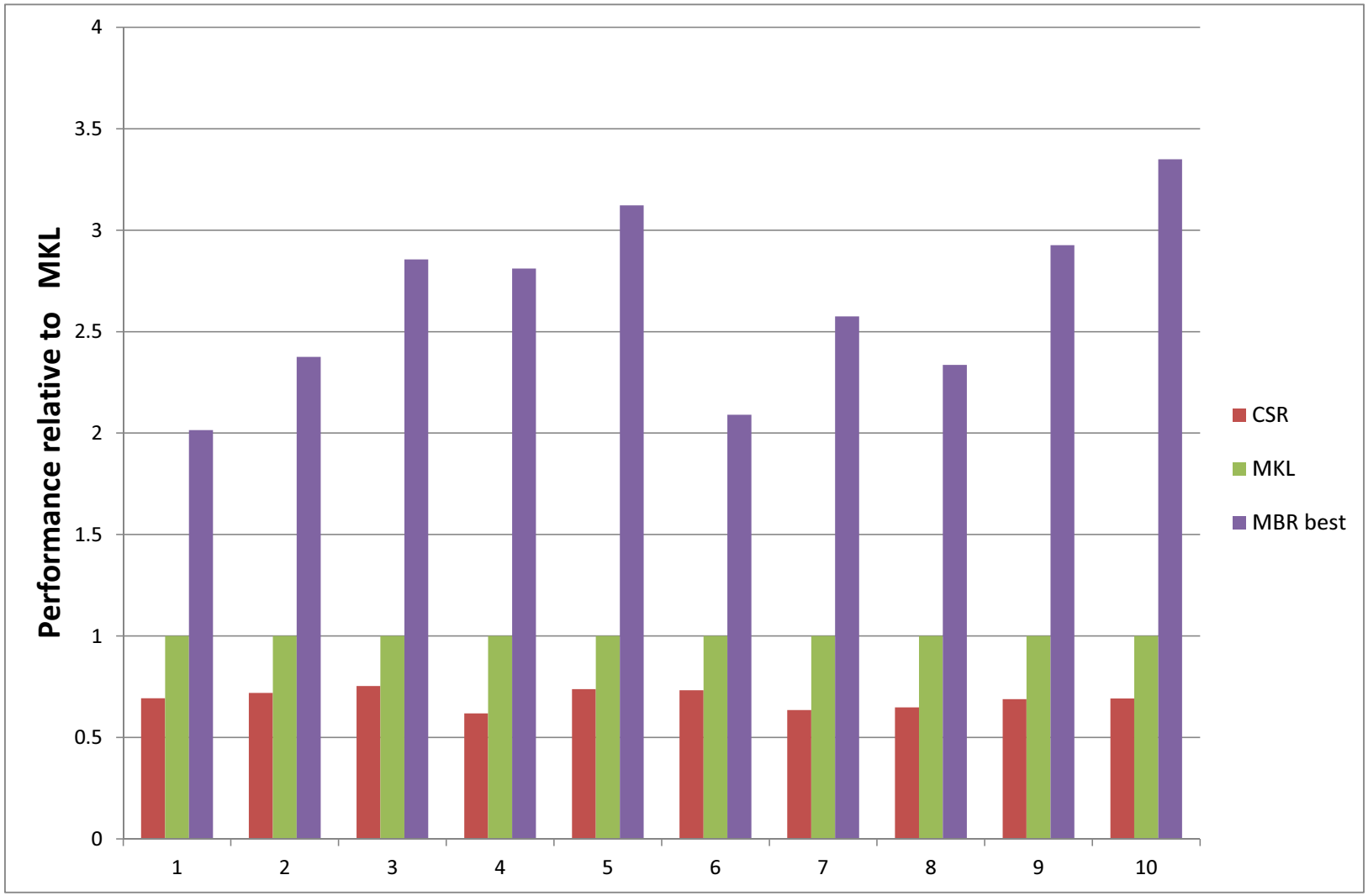
Performance comparison

Intel Xeon E5450



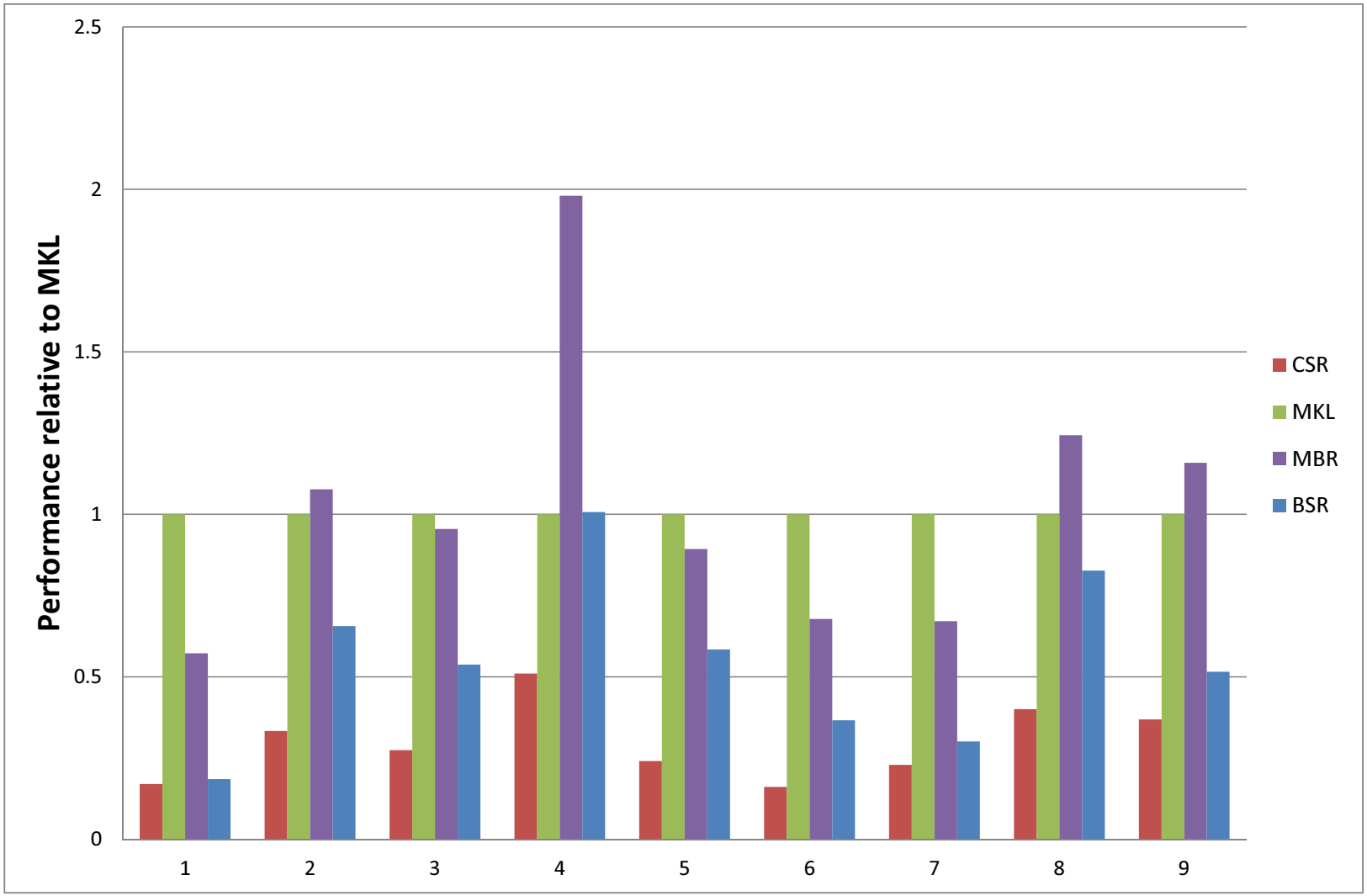
Performance comparison

AMD Opteron 6220



Performance comparison – with BSR

Intel Core i7–2720QM



Summary

Summary

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

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

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 upto 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.