High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 16

Sparse matrix data structures

- Only nonzero elements are stored in sparse matrix data structures, which makes possible the storage of sparse matrices of large dimension.
- Sometimes some zeros are stored (explicit zeros) to maintain block or symmetric sparsity patterns, for example.
- Formats are generally optimized for sparse matrix-vector multiplication (SpMV).
- Conversion cost to an efficient format may be important.

Coordinate format (COO)

Example:

$$\begin{bmatrix} 10 & 11 & \\ & 12 & 13 \\ & & 14 \end{bmatrix}$$

COO format uses three arrays for the above matrix:

rowind	2	1	3	1	2
colind	2	2	3	1	3
a	12	11	14	10	13

with N=3 and NNZ=5.

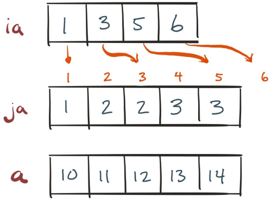
Nonzeros can be in any order in general.

Compressed sparse row format (CSR)

Example:

$$\begin{bmatrix} 10 & 11 & \\ & 12 & 13 \\ & & 14 \end{bmatrix}$$

CSR format uses three arrays for the above matrix:



with N=3.

Rows are stored contiguously in memory. This is useful if row-wise access should be effcient. (Within a row, entries may not be in order.)

A simple variation is compressed sparse row format (CSC).

Data access patterns for SpMV

In straightforward implementations of $y = Ax$ for matrices in COO and CSR formats, the arrays are traversed in order. Memory access of data in these arrays is predictable and efficient.

However, x is accessed in irregular order in general, and may use caches poorly.

Example:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} x & & & & x & & x & \\ & x & & x & & & x & \\ & & x & & x & & & x \\ & x & & x & & x & & \\ x & & x & & x & & & \\ & & & x & & x & & \\ x & x & & & & & x & \\ & & x & & & & & x \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix}$$

Data access patterns for SpMV

If "cache size" for x is 3, this SpMV has bad cache behavior:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} x & & & & x & & x & \\ & x & & x & & & x & \\ & & x & & x & & & x \\ & x & & x & & x & & \\ x & & x & & x & & & \\ & & & x & & x & & \\ x & x & & & & & x & \\ & & x & & & & & x \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix}$$

The matrix can be reordered to be banded:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \end{bmatrix} = \begin{bmatrix} x & x & & & & & & \\ x & x & x & & & & & \\ & x & x & x & & & & \\ & & x & x & x & & & \\ & & & x & x & x & & \\ & & & & x & x & x & \\ & & & & & x & x & x \\ & & & & & & x & x \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix}$$

so that it has perfect cache behavior.

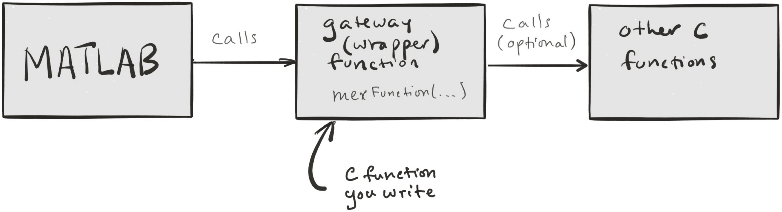
Viewing Matlab's internal sparse matrix data structure

For sparse matrices, Matlab uses compressed sparse column format.

We can use Matlab's **mex** interface to view the raw sparse matrix data structure.

Mex files – calling C codes from Matlab

- C codes are usually more efficient than Matlab programs.
- Some types of algorithms are easier to to write in C than in Matlab.
- You may want to use Matlab to call functions in an existing C library.



Mex gateway function

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]);
```

`nlhs` – number of objects to return

`plhs` – array of objects to be returned

`nrhs` – number of inputs

`prhs` – array of input objects

Example: `a = add_mex(b,c);`

```
nlhs = 1        nrhs = 2
plhs = [a]      prhs = [b, c]
```

Compile mex program: `mex add_mex.c` from Matlab prompt.
Compile with `-largeArrayDims` flag if sparse matrices are used.

add_mex.c

```c
#include <stdio.h>
#include "mex.h"

// Usage: a = add_mex(b,c), where a,b,c are scalars

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    printf("sizeof nlhs: %d\n", nlhs);
    printf("sizeof nrhs: %d\n", nrhs);

    double b = *mxGetPr(prhs[0]);
    double c = *mxGetPr(prhs[1]);

    printf("b: %f\n", b);
    printf("c: %f\n", c);

    double a = b+c;

    plhs[0] = mxCreateDoubleScalar(a);
}
```

dump_matrix_mex.c

```c
// Usage: dump_matrix_mex(A) where A is a sparse matrix.
// Matlab sparse matrices are CSC format with 0-based indexing.

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    int n;
    const mwIndex *ia, *ja;
    const double *a;

    n  = mxGetM (prhs[0]);
    ia = mxGetJc(prhs[0]); // column pointers
    ja = mxGetIr(prhs[0]); // row indices
    a  = mxGetPr(prhs[0]); // values

    int i, j;
    for (i=0; i<n; i++)
        for (j=ia[i]; j<ia[i+1]; j++)
            printf("%5d %5d   %f\n", ja[j]+1, i+1, a[j]);
}
```

matvec_mex.c

```c
static void Matvec(int n, const mwIndex *ia, const mwIndex *ja,
                   const double *a, const double *x, double *y)
{
    int i, j;
    double t;

    for (i=0; i<n; i++) {
        t = 0.;
        for (j=ia[i]; j<ia[i+1]; j++)
            t += a[j]*x[ja[j]];
        y[i] = t;
    }
}

// Usage: y = matvec_mex(a, x);
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    int n = mxGetN(prhs[0]);
    plhs[0] = mxCreateDoubleMatrix(n, 1, mxREAL); // solution vector

    Matvec(n, mxGetJc(prhs[0]), mxGetIr(prhs[0]), mxGetPr(prhs[0]),
        mxGetPr(prhs[1]), mxGetPr(plhs[0]));
}
```

Advanced sparse matrix data structures

Reference:
M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: A unified sparse matrix data format for modern processors with wide SIMD units, 2014.

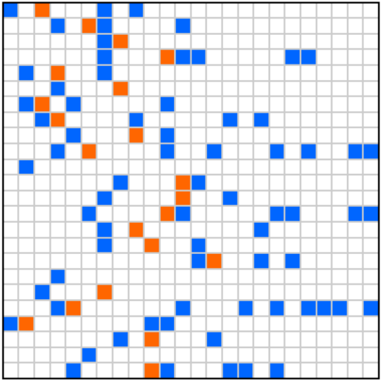
Some figures below are taken from the above reference.

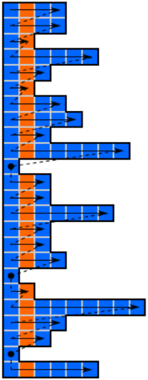
Advanced sparse matrix data structures

Computational considerations:

- SpMV is generally viewed as being limited by memory bandwidth
- On accelerators and coprocessors, memory bandwith may not be the limiting factor
- SIMD (single instruction, multiple data) must be used to increase the flop rate
- It is desirable to use long loops (rather than short loops) to reduce overheads
- Efficient use of SIMD may result in bandwidth being saturated when using a smaller number of cores (saving energy)

CSR format





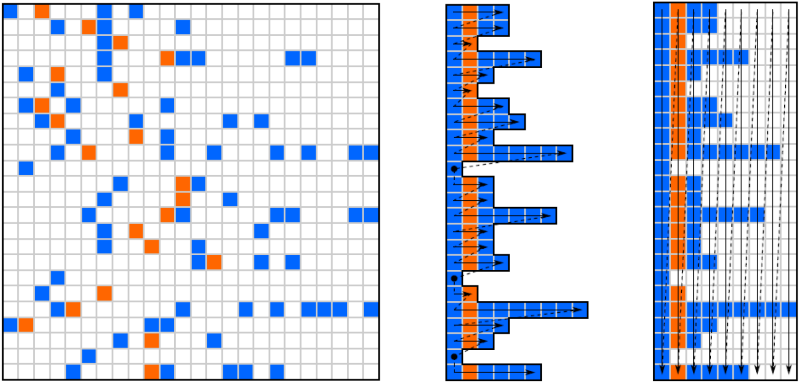
SpMV code using CSR format (SIMD illustration)

```c
for(i = 0; i < N; ++i) {
  for(j = rpt[i]; j < rpt[i+1]; ++j) {
      y[i] += val[j] * x[col[j]];
  }
}
```

```c
1 for(i = 0; i < N; ++i)
2 {
3   tmp0 = tmp1 = tmp2 = tmp3 = 0.;
4   for(j = rpt[i]; j < rpt[i+1]; j+=4)
5   {
6     tmp0 += val[j+0] * x[col[j+0]];
7     tmp1 += val[j+1] * x[col[j+1]];
8     tmp2 += val[j+2] * x[col[j+2]];
9     tmp3 += val[j+3] * x[col[j+3]];
10  }
11  y[i] += tmp0+tmp1+tmp2+tmp3;
12  // remainder loop
13  for(j = j-4; j < rpt[i+1]; j++)
14    y[i] += val[j] * x[col[j]];
15 }
```

If rows are short, then SIMD is not effectively utilized, and “overhead” of the remainder loop and the reduction (line 11) is relatively large.

ELLPACK format



ELLPACK format:

- Entries are stored in a dense array in column major order, resulting in long columns, good for efficient computation.
- Explicit zeros are stored if necessary (zero padding).
- Little zero padding if all rows are about the same length.
- Not efficient if have short and long rows.

Potential solutions for the zero-padding problem

Potential solutions for the zero-padding problem

- Hybrid format (ELL+COO) used on GPUs

Potential solutions for the zero-padding problem

- Hybrid format (ELL+COO) used on GPUs
- Jagged diagonal (JDS) format used on old vector supercomputers

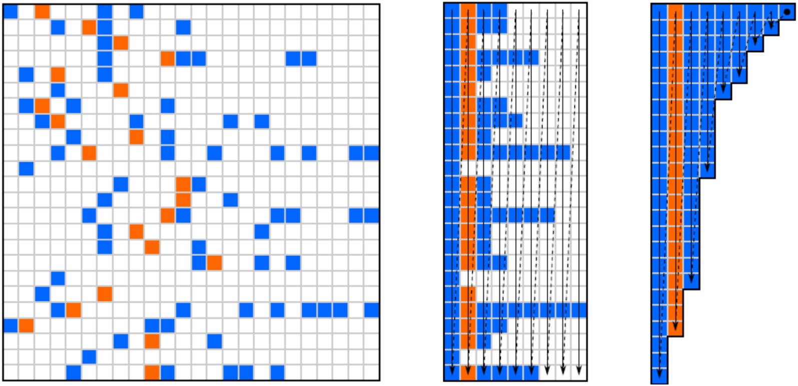
Potential solutions for the zero-padding problem

- Hybrid format (ELL+COO) used on GPUs
- Jagged diagonal (JDS) format used on old vector supercomputers
- Sliced ELLPACK (SELL) format

Potential solutions for the zero-padding problem

- Hybrid format (ELL+COO) used on GPUs
- Jagged diagonal (JDS) format used on old vector supercomputers
- Sliced ELLPACK (SELL) format
- A combination of SELL and JDS: SELL-C-σ

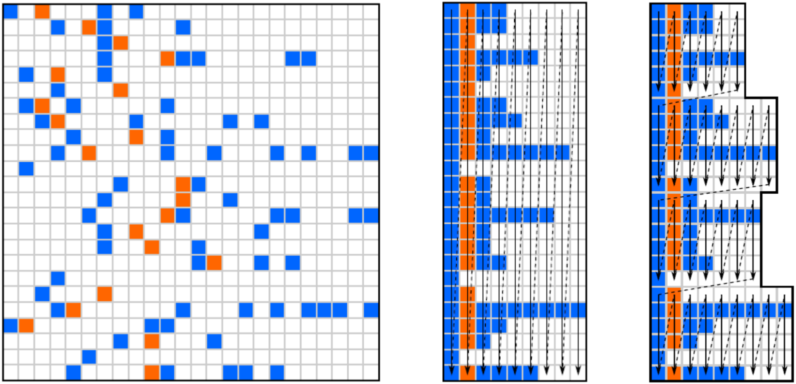
Jagged diagonal format



JDS format sorts the rows by length.

A disadvantage of JDS format is that access to x (in $y = Ax$) may be irregular, leading to poor cache usage.

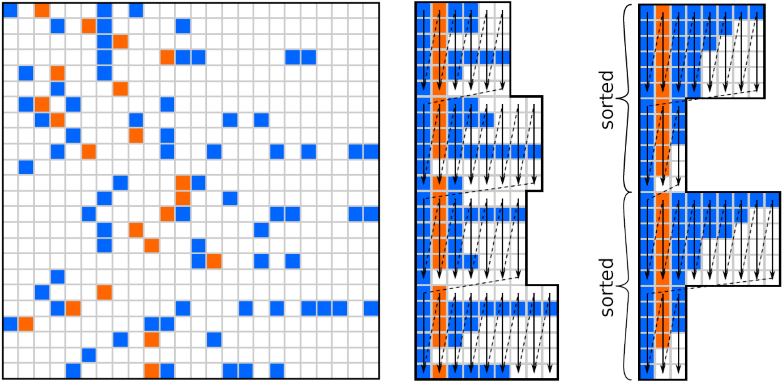
Sliced ELLPACK format



Dense matrix is “sliced” row-wise into chunks.

Avoids problem of irregular access of x since the given ordering can be used in the SpMV computation.

SELL-C-σ format

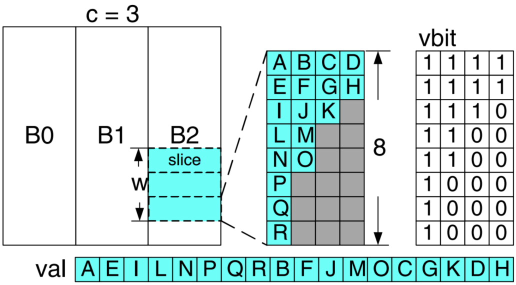


C = chunk size (like in SELL); 6 in above example.
σ = sorting window size; 12 in above example. This parameter helps preserve locality in accesses in x (e.g., if the matrix is banded).

Block formats

A more explicit way to ensure locality in accesses to x is to partition the matrix by block columns.

The ELLPACK Sparse Block (ESB) format uses both partitioning by block rows (like Sliced ELLPACK) and by block columns (for x locality), giving sparse blocks that are stored in an ELLPACK-like format.



In this figure, $c = 3$ block columns are used. Rows are sorted within windows of size w. Instead of column lengths, bit vectors

Some references

- Jagged diagonal format: Saad, Krylov subspace methods on supercomputers, 1989.
- Hybrid ELL+COO format: Bell and Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, 2009.
- Sliced ELLPACK format: Monakov, Lokhmotov, and Avetisyan, Automatically tuning sparse matrix-vector multiplication for GPU architectures, 2010.
- ELLPACK Sparse Block (ESB) format: Liu, Smelyanskiy, Chow, and Dubey, Efficient sparse matrix-vector multiplication on x86-based many-core processors, 2013.
- SELL-C-σ format: Kreutzer, Hager, Wellein, Fehske, and Bishop, A unified sparse matrix data format for modern processors with wide SIMD units, 2014.