AER E 361: Computational Techniques for Aerospace Design

The Matrix

Iowa State University

http://temporallogic.org/courses/AERE361/

Dense Matrices

```
double matrix[10][10];
```

OR

```
double *matrix = (double *)malloc(M * N * sizeof(double));
```

Number of rows: M
Number of columns: N
Cost of storing the dense matrix: $M \times N \times sizeof(double)$ bytes

For M = N = 1,000,000 and $sizeof(double) = 8$ bytes, we have 8 TB of storage

Dense Matrix Struct

```
typedef struct dense_matrix_ {
  int nrows;
  int ncolumns;
  double values[nrows][ncolumns];
} dense_matrix;
```

Real Life: Many *Sparse* Matrices

Sparse matrices contain many more 0s than non-0s.

By storing matrices smartly, we can save:

- access time (including cache hits!)
- memory

Where Do Sparse Matrices Occur? Everywhere!

- Sparsity is a consequence of **local** correlation in **PDE** discretization (current in a circuit, temperature, or voltage).
- Linear Algebraic Equations: matrices represent systems of linear equations
- Finite Element Models

Matrix Solution of Linear Systems

When solving systems of linear equations, we can represent a linear system of equations by an **augmented matrix**, a matrix which stores the coefficients and constants of the linear system and then manipulate the augmented matrix to obtain the solution of the system.

Example:

$$x + 3y = 5$$
$$2x - y = 3$$

The augmented matrix associated with the above system is

$$\left[\begin{array}{cc|c} 1 & 3 & 5 \\ 2 & -1 & 3 \end{array}\right]$$

Coordinate Format (COO)

- rows array
- columns array
- values array

For every non-zero value in the original sparse matrix, there is an entry at index *i* in the *rows* array, *columns* array, and *values* array that stores the row, column, and value of that non-zero item

COO Example

```
                    columns
                      0   1
row 0 [ 1.0 2.0 ]
row 1 [ 0.0 4.0 ]
```

is

```
unsigned rows[3] =    { 0,   0,   1   };
unsigned columns[3] = { 0,   1,   1   };
double values[3] =    { 1.0, 2.0, 4.0 };
```

- at cell (0, 0) the value 1.0 is stored,
- at cell (0, 1) the value 2.0 is stored,
- at cell (1, 1) the value 4.0 is stored.
- nothing is stored for (1, 0), so its value is implicitly set to 0.0.

Finding a Value in COO

Example: we want the value stored at cell (1,1):

```
for (i = 0; i < M*N; i++) {
  if (rows[i] == 1 and columns[i] == 1} {
    result = values[i];
  } /*end if*/
  else { /*no exact match is found*/
    result = 0;
  } /*end else*/
} /*end for*/
```

Finding a Value in COO

Example: we want the value stored at cell (1,1):

```
for (i = 0; i < M*N; i++) {
  if (rows[i] == 1 and columns[i] == 1} {
    result = values[i];
  } /*end if*/
  else { /*no exact match is found*/
    result = 0;
  } /*end else*/
} /*end for*/
```

This is not a very intelligent algorithm...

Algorithms Asside: Finding a Value in COO

How can we search on average half the space?
Can we set result to 0 only once?

```
for (i = 0; i < M*N; i++) {
  if (rows[i] == 1 and columns[i] == 1} {
    result = values[i];
  } /*end if*/
  else { /*no exact match is found*/
    result = 0;
  } /*end else*/
} /*end for*/
```

COO Storage Savings Example

Example:

- $1,000,000 \times 1,000,000$ matrix
- 5% of cells have non-zero values
- 50,000,000,000 non-zero values in total
- assuming sizeof(unsigned) $==$ 4)

Storage cost:

```
(sizeof(unsigned) + sizeof(unsigned) + sizeof(double)) * 5E10
 => 16 bytes * 5E10
 => 800 GB ... WAY LESS than 8TB!
```

Note: memory savings is *not* linear in terms of sparsity: 5% non-zeros is stored in 10% of the size of the dense matrix!

Sparse vs Dense, Defined by COO

Where is the storage tipping point?
When storing 16 bytes (per non-zero value) is cheaper than storing 8 bytes for every cell:

```
16 * nnz < 8 * nrows * ncols
```

where nnz = "number of non-zeros"

```
16 * (sparsity * nrows * ncols) < 8 * nrows * ncols
  => sparsity * nrows * ncols < 0.5 * nrows * ncols
  => sparsity < 0.5
```

So, if sparsity $< 50\%$, we save!

$$\text{sparsity} < \frac{\#\text{ of bytes per value in dense}}{\#\text{ of bytes per value in COO}}$$

COO Matrix Struct

```
typedef struct coo_matrix_ {
  int nnz;
  unsigned rows[nnz];
      /* row index for each non-zero value */
  unsigned columns[nnz];
      /* column index for each non-zero value */
  double values[nnz];
      /* each non-zero value */
} coo_matrix;
```

Can We Do Better?

- the space to store each element is double (16 vs 8 bytes)
- some repetitive data is stored
  - $\forall$ values i in the *rows* array: $0 \leq i < M$
  - $\forall$ values j in the *columns* array: $0 \leq i < N$
- linearly searching for values is inefficient
  - why search the whole array for the value at cell (980, 1020)?
  - better to just jump to offset 980 * ncols + 1020 in your array ...

Compressed Row (CSR)

COO:

```
unsigned rows[5]    = { 0,   0,   0,   1,   2   };
unsigned columns[5] = { 0,   3,   5,   0,   3   };
double   values[5]  = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

There's a lot of redundant data in `rows`!

Compressed Row (CSR)

COO:

```
unsigned rows[5]    = { 0,   0,   0,   1,   2   };
unsigned columns[5] = { 0,   3,   5,   0,   3   };
double   values[5]  = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

There's a lot of redundant data in `rows`!

- using 5 integers to differentiate between just three values: row 0, 1, or 2

Compressed Row (CSR)

COO:

```
unsigned rows[5]    = { 0,   0,   0,   1,   2   };
unsigned columns[5] = { 0,   3,   5,   0,   3   };
double   values[5]  = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

There's a lot of redundant data in `rows`!

- using 5 integers to differentiate between just three values: row 0, 1, or 2
- why do we need more items in `rows` than there are rows?

Compressed Row (CSR)

COO:

```
unsigned rows[5]    = { 0,   0,   0,   1,   2   };
unsigned columns[5] = { 0,   3,   5,   0,   3   };
double   values[5]  = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

There's a lot of redundant data in `rows`!

- using 5 integers to differentiate between just three values: row 0, 1, or 2
- why do we need more items in `rows` than there are rows?
- we can still keep `columns` and `values` sorted by row for efficiency, even if we improve `rows`!

Compressed Row (CSR)

COO:

```
unsigned rows[5]    = { 0,   0,   0,   1,   2   };
unsigned columns[5] = { 0,   3,   5,   0,   3   };
double   values[5]  = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

There's a lot of redundant data in `rows`!

- using 5 integers to differentiate between just three values: row 0, 1, or 2
- why do we need more items in `rows` than there are rows?
- we can still keep `columns` and `values` sorted by row for efficiency, even if we improve `rows`!
- It is more efficient to store the number of items in each row, than the row index for each item!

Compressed Row (CSR)

COO:

```
unsigned rows[5]    = { 0,   0,   0,   1,   2   };
unsigned columns[5] = { 0,   3,   5,   0,   3   };
double   values[5]  = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Improvement:

- Store *number of items* in each row instead of row index for each item
- `row_counts[i]` tells how many non-zero values in row i
- `row_counts` is *nrows* long instead of *nnz*
- save space if average nnz per row is > 1

```
unsigned row_counts[3] = { 3,               1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

This saves *two integers*!

Compressed Row (CSR)

- To do a look-up: have to scan all of `row_counts` to figure out the offset of values from the target row in the *columns* and *values* arrays
- Looking for (2,3) means checking `row_counts[2]` to find the offset of row 2 in *columns* and *values*
- Note: values belonging to `row i` in `columns` and `values` start where the values of row i-1 end, i.e., at offset `sum(row_counts[0:i-1])`
- Scanning is expensive!

```
unsigned row_counts[3] = { 3,             1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Let's try this ...

Compressed Row (CSR)

```
unsigned row_counts[3] = { 3,                 1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Compressed Row (CSR)

```
unsigned row_counts[3] = { 3,                 1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Fetch the value at (2,3):

- To find the start of row 2, sum *row_counts[0]* and *row_counts[1]* to get 4.
- Then go to offset 4 in *columns* and check the value there.
- It matches the column we're searching for (3), so fetch *values[4]* and return that.

Compressed Row (CSR)

Where is this inefficient?

```
unsigned row_counts[3] = { 3,                 1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Compressed Row (CSR)

Where is this inefficient?

```
unsigned row_counts[3] = { 3,                 1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

- We had to scan through all of *row_counts* to figure out the offset of values from the target row in the `columns` and `values` arrays!

Compressed Row (CSR)

Where is this inefficient?

```
unsigned row_counts[3] = { 3,                 1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

- We had to scan through all of *row_counts* to figure out the offset of values from the target row in the `columns` and `values` arrays!
- What if we had a matrix with 1,000,000 rows?

Compressed Row (CSR)

Where is this inefficient?

```
unsigned row_counts[3] = { 3,                 1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

- We had to scan through all of *row_counts* to figure out the offset of values from the target row in the `columns` and `values` arrays!
- What if we had a matrix with 1,000,000 rows?
- How can we store *one value* to make this more efficient?

Compressed Row (CSR)

Where is this inefficient?

```
unsigned row_counts[3] = { 3,                 1,   1 };
unsigned columns[5]    = { 0,   3,   5,   0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

- We had to scan through all of *row_counts* to figure out the offset of values from the target row in the `columns` and `values` arrays!
- What if we had a matrix with 1,000,000 rows?
- How can we store *one value* to make this more efficient?
- We can just store the row offsets directly!

Compressed Row (CSR)

```
unsigned row_counts[3] = { 3,              1,   1 };
unsigned columns[5]    = { 0,   3,   5,    0,   3 };
double   values[5]     = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Like this . . .

```
unsigned row_offsets[3] = { 0,              3,   4 };
unsigned columns[5]     = { 0,   3,   5,    0,   3 };
double   values[5]      = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

- Calculating the offset of row i in `columns` or `values` just requires checking `row_offsets[i]`
- Now `row_offsets[i] == sum(row_counts[0:i-1])`

Compressed Row (CSR)

```
unsigned row_offsets[3] = { 0,                3,   4 };
unsigned columns[5]     = { 0,   3,   5,   0,   3 };
double   values[5]      = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Fetch the value at (2,3):

- Check `row_offsets[2]` to find the offset of row 2 in `columns` and `values`
- Then go to offset 4 in *columns* and check the value there.
- It matches the column we're searching for (3), so fetch *values[4]* and return that.

Compressed Row (CSR)

```
unsigned row_offsets[3] = { 0,             3,   4 };
unsigned columns[5]     = { 0,   3,   5,   0,   3 };
double   values[5]      = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

Advantages of CSR:

- Same advantages as COO for storing columns and values
- stores offsets of each row in `columns` and `values` rather than the actual row index for each non-zero item (as in COO)
- Limits the length of `row_offsets` to just `nrows`
  - (saving space)
- Allows us to quickly find all of the values belonging to a row i by just jumping to offset `row_offsets[i]` in `columns` and `values`
  - (more efficient than scanning)

Compressed Row (CSR)

```
unsigned row_offsets[3] = { 0,              3,   4 };
unsigned columns[5]     = { 0,   3,   5,    0,   3 };
double   values[5]      = { 3.0, 4.0, 1.0, 1.0, 2.0 };
```

One more thing...
Add one element to the end of `row_offsets`: the total number of non-zero entries in the matrix

```
unsigned row_offsets[4] = { 0,              3,   4, 5 };
```

- Can calculate the length of row i using $row[i+1] - row[i]$ without special-casing for the last row
- Need to store the number of non-zero entries in the matrix anyway
- Simply store *nnz* in `row_offsets[nrows]` rather than in a separate scalar variable

CSR Matrix Struct

```
typedef struct csr_matrix_ {
  int nrows; /*the number of rows*/
  /* The offset in columns,
     values of each row's representation
     nnz = rows[nrows]*/
  unsigned row_offsets[nrows + 1];
  unsigned columns[nnz]; /* same as COO */
  double values[nnz]; /* same as COO */
} csr_matrix;
```

CSR Matrix Struct

```
typedef struct csr_matrix_ {
  int nrows; /*the number of rows*/
  /* The offset in columns,
     values of each row's representation
     nnz = rows[nrows]*/
  unsigned row_offsets[nrows + 1];
  unsigned columns[nnz]; /* same as COO */
  double values[nnz]; /* same as COO */
} csr_matrix;
```

How do we access this in a cache-efficient way?

CSR Matrix Struct

```
typedef struct csr_matrix_ {
  int nrows; /*the number of rows*/
  /* The offset in columns,
     values of each row's representation
     nnz = rows[nrows]*/
  unsigned row_offsets[nrows + 1];
  unsigned columns[nnz]; /* same as COO */
  double values[nnz]; /* same as COO */
} csr_matrix;
```

How do we access this in a cache-efficient way?

Can we do better?

CSR Matrix Struct

```
typedef struct csr_matrix_ {
  int nrows; /*the number of rows*/
  /* The offset in columns,
     values of each row's representation
     nnz = rows[nrows]*/
  unsigned row_offsets[nrows + 1];
  unsigned columns[nnz]; /* same as COO */
  double values[nnz]; /* same as COO */
} csr_matrix;
```

How do we access this in a cache-efficient way?
Can we do better? WHEN do we want to do better?