Objective

- To learn the key techniques for compacting input data in parallel sparse methods for reduced consumption of memory bandwidth
  - better utilization of on-chip memory
  - fewer bytes transferred to on-chip memory
  - Better utilization of global memory
  - Challenge: retaining regularity

Sparse Matrix

- Many real-world systems are sparse in nature
  - Linear systems described as sparse matrices
- Solving sparse linear systems
  - Traditional inversion algorithms such as Gaussian elimination can create too many “fill-in” elements and explode the size of the matrix
  - Iterative Conjugate Gradient solvers based on sparse matrix-vector multiplication is preferred
- Solution of PDE systems can be formulated into linear operations expressed as sparse matrix-vector multiplication

Sparse Data

Motivation for Compaction

- Many real-world inputs are sparse/non-uniform
- Signal samples, mesh models, transportation networks, communication networks, etc.
Sparse Matrix in Analytics and AI

- Predict missing ratings
- Group similar users/items
- Match query and document

Matrix Factorization
- Recommender systems
- Natural language processing
- Complex network
- Deep learning
- Web search

Tensor decomposition
- In machine learning and HPC applications

Ratings ($R$) $\approx x^T \theta^T$

Challenges

- Compared to dense matrix multiplication, SpMV
  - Is irregular/unstructured
  - Has little input data reuse
  - Benefits little from compiler transformation tools

- Key to maximal performance
  - Maximize regularity (by reducing divergence and load imbalance)
  - Maximize DRAM burst utilization (layout arrangement)
A Simple Parallel SpMV

- Each thread processes one row

Compressed Sparse Row (CSR) Format

CSR Representation

| Nonzero values | data[7] | {3, 1, 2, 4, 1, 1, 1} |
| Column indices | col_index[7] | {0, 2, 1, 2, 3, 0, 3} |
| Row Pointers | ptr[5] | {0, 2, 5, 7} |

Dense representation

| Row 0 | 3 0 1 0 | Thread 0 |
| Row 1 | 0 0 0 0 | Thread 1 |
| Row 2 | 0 2 4 1 | Thread 2 |
| Row 3 | 1 0 0 1 | Thread 3 |

CSR Kernel Design
A Parallel SpMV/CSR Kernel (CUDA)

```c
__global__ void SpMV_CSR(int num_rows, float *data,
                          int *col_index, int *row_ptr, float *x, float *y) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) {
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row+1];
        for (int elem = row_start; elem < row_end; elem++) {
            dot += data[elem] * x[col_index[elem]];
        }
        y[row] = dot;
    }
}
```

CSR Kernel Control Divergence
- Threads execute different number of iterations in the kernel for-loop

CSR Kernel Memory Divergence (Uncoalesced Accesses)
- Adjacent threads access non-adjacent memory locations
  - Grey elements are accessed by all threads in iteration 0

Regularizing SpMV with ELL(PACK) Format
- Pad all rows to the same length
  - Inefficient if a few rows are much longer than others
- Transpose (Column Major) for DRAM efficiency
- Both data and col_index padded/transposed
1. `__global__ void SpMV_ELL(int num_rows, float *data, int *col_index, int num_elem, float *x, float *y) {
2.   int row = blockIdx.x * blockDim.x + threadIdx.x;
3.   if (row < num_rows) {
4.     float dot = 0;
5.     for (int i = 0; i < num_elem; i++) {
6.       dot += data[row+i*num_rows]*x[col_index[row+i*num_rows]];
7.     }
8.     y[row] = dot;
9.   }
10. }

A parallel SpMV/ELL kernel

Memory Coalescing with ELL

Coordinate (COO) format

• Explicitly list the column and row indices for every non-zero element

Nonzero values `data[7]`:

Row 0: `{3, 1, 1, 1, 4, 1, 1}`
Row 2: `{2, 4, 1, 3, 0, 0}`
Row 3: `{1, 1, 0, 3, 2, 2}`

Column indices `col_index[7]`:

Row 0: `{0, 1, 2, 1, 3}`
Row 2: `{1, 2, 3, 0, 0}`
Row 3: `{2, 2, 2, 3, 3}`

Row indices `row_index[7]`:

Row 0: `{0, 2, 1, 3}`
Row 2: `{2, 1, 3, 3}`
Row 3: `{3, 3, 3}`
### COO Allows Reordering of Elements

<table>
<thead>
<tr>
<th>Nonzero values data[7]</th>
<th>Row 0</th>
<th>Row 2</th>
<th>Row 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3, 1, 2, 4, 1, 1, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Column indices col_index[7] | 0, 2, 1, 2, 0, 3, 3 |
| Row indices row_index[7]   | 0, 0, 2, 2, 2, 3, 3 |

- Nonzero values data[7] { 1, 1, 2, 4, 3, 1, 1 }
- Column indices col_index[7] { 0, 2, 1, 2, 0, 3, 3 }
- Row indices row_index[7] { 3, 0, 2, 2, 0, 2, 3 }

```c
1. for (int i = 0; i < num_elem; row++)
2. y[row_index[i]] += data[i] * x[col_index[i]];
```

*a sequential loop that implements SpMV/COO*