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

In machine learning and HPC applications
Matrix Factorization
Link prediction
Vertices clustering
Latent semantic model
Word embedding as input to DNN
Recommender systems
Complex network
Deep learning
Web search
Tensor decomposition
Model compression
Embedding layer
Deep learning
Ratings \( R \)
\( n \) items
\( m \) users
\( \approx \)

Sparse Matrix-Vector Multiplication (SpMV)

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)

Science Area | Number of Teams | Codes |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Climate and Weather</td>
<td>3</td>
<td>CESM, GRAM, CM1/ARF, HOMME</td>
</tr>
<tr>
<td>Planetary Magnetospheres</td>
<td>2</td>
<td>HSDS, VPC, CM1/10, MAGMA</td>
</tr>
<tr>
<td>Stellar Atmospheres and Supernovae</td>
<td>5</td>
<td>PINN, MAESTRO, CASTRO, SIDOM, ChaRTX, XSF, LIMES</td>
</tr>
<tr>
<td>Cosmology</td>
<td>2</td>
<td>CoCo, AGATET</td>
</tr>
<tr>
<td>Combustion Turbulence</td>
<td>2</td>
<td>FSDS, DISTR</td>
</tr>
<tr>
<td>General Relativity</td>
<td>2</td>
<td>Cactus, Ham3D, LibEV</td>
</tr>
<tr>
<td>Molecular Dynamics</td>
<td>4</td>
<td>AMBER, GROMACS, NAMD, LAMMPS</td>
</tr>
<tr>
<td>Quantum Chemistry</td>
<td>2</td>
<td>EAGLE, QM, QM</td>
</tr>
<tr>
<td>Material Science</td>
<td>3</td>
<td>MSe, QM, QM</td>
</tr>
<tr>
<td>Earthquakes Seismology</td>
<td>2</td>
<td>APW-ODC, Hse, LQ, MDD</td>
</tr>
<tr>
<td>Quantum Chromodynamics</td>
<td>1</td>
<td>Chrona, Mpc, USQCD</td>
</tr>
<tr>
<td>Social Networks</td>
<td>1</td>
<td>EPISIMDEMS</td>
</tr>
<tr>
<td>Evolution</td>
<td>1</td>
<td>Eve</td>
</tr>
<tr>
<td>Engineering/Systems of Systems</td>
<td>1</td>
<td>GRIPS, Revis</td>
</tr>
</tbody>
</table>

©Wen-mei W. Hwu and David Kirk/NVIDIA, 2010-2018
A Simple Parallel SpMV

- Each thread processes one row

<table>
<thead>
<tr>
<th>Row</th>
<th>3 0 1 0</th>
<th>Thread 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>0 0 0 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>Row</td>
<td>0 2 4 1</td>
<td>Thread 2</td>
</tr>
<tr>
<td>Row</td>
<td>1 0 0 1</td>
<td>Thread 3</td>
</tr>
</tbody>
</table>

Compressed Sparse Row (CSR) Format

CSR Representation

- Nonzero values: \{3, 1, 2, 4, 1, 1, 1\}
- Column indices: \{0, 2, 1, 2, 3, 0, 3\}
- Row Pointers: \{0, 2, 2, 5, 7\}

Dense representation

<table>
<thead>
<tr>
<th>Row</th>
<th>3 0 1 0</th>
<th>Thread 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>0 0 0 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>Row</td>
<td>0 2 4 1</td>
<td>Thread 2</td>
</tr>
<tr>
<td>Row</td>
<td>1 0 0 1</td>
<td>Thread 3</td>
</tr>
</tbody>
</table>

CSR Data Layout

<table>
<thead>
<tr>
<th>row_ptr</th>
<th>0 2 2 5 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>3 1 2 4 1 1 1</td>
</tr>
<tr>
<td>col_index</td>
<td>0 2 1 2 3 0 3</td>
</tr>
</tbody>
</table>

CSR Kernel Design

Dot product With vector
A Parallel SpMV/CSR Kernel (CUDA)

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
A parallel SpMV/ELL kernel

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.   }
}

Coordinate (COO) format

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

<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>0, 2, 1, 2, 3, 0, 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column indices col_index[7]</td>
<td>0, 2, 1, 2, 3, 0, 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row indices row_index[7]</td>
<td>0, 2, 1, 2, 3, 0, 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
COO Allows Reordering of Elements

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

a sequential loop that implements SpMV/COO

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

READ CHAPTER 10