Transient Stability of Power System

Programming Massively Parallel Graphics Multiprocessors using CUDA
Final Project

Amirhassan Asgari Kamiabad
996620802
Electric Power System

- Largest infrastructure made by man
- Prone to many kinds of disturbance: Blackouts
- Transient stability analysis: dynamic behavior of power system few seconds following a disturbance

(Electric Power Network)
Voltage, Current, Power angle....

(String and Mass Network)
Force, Speed, Acceleration....
Transient Stability Computation

- Differential and Algebraic Equations
  \[ \frac{dx}{dt} = f(x, y) \]
  \[ 0 = g(x, y) \]
  \[ F(y_{n+1}, x_{n+1}) = 0 \]

- Discretize the differential part and use Backward Euler

- Newton Method solve nonlinear system

- Solve Linear System

\[ Y e = S \]
\[ A x = b \]
Jacobian Matrix Properties

- Large and Sparse: $10k \times 10k$ with 50k non-zeros
- Stiff: Eigen values are widespread
- Save in sparse compressed row format
  - Data, indices and pointer vectors

<table>
<thead>
<tr>
<th>Column</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

\[
\begin{bmatrix}
1 & 7 & 0 & 0 \\
0 & 2 & 8 & 0 \\
5 & 0 & 3 & 9 \\
0 & 6 & 0 & 4 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
0 & 1/ & 1 & 2/ & 0 & 2 & 3/ & 1 & 3 \\
1 & 7/ & 2 & 8/ & 5 & 3 & 9/ & 6 & 4 \\
0 & 2 & 4 & 7 & 9
\end{bmatrix}
\]

indices  data  pointer
Direct: LU factorization

- LU factorization: \( Ax = (L \, U)x = L(Ux) = b \)
  \[
  \begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
  \end{bmatrix}
  =
  \begin{bmatrix}
  1 & 0 & 0 \\
  L_{21} & 1 & 0 \\
  L_{31} & L_{32} & 1
  \end{bmatrix}
  \begin{bmatrix}
  U_{11} & U_{12} & U_{13} \\
  0 & U_{22} & U_{23} \\
  0 & 0 & U_{33}
  \end{bmatrix}
  \]

- Backward Substitution \( Ly = b \)
- Forward Substitution \( Ux = y \)

- Number of active threads reduce during run time
- Backward and forward substitution are serial algorithms
Indirect: Conjugate Gradient

- Iterative method which use only matrix multiplication and addition
- Necessarily converges in N iteration if the matrix is symmetric positive definite
- Suffer from slow rate of convergence
  - Transforming system to an equivalent with same solution but easier to solve
  - Preconditioning can improve robustness and efficiency
Chebychev Preconditioner

- Iterative method which estimate inverse of matrix
- Main GPU kernels:
  - Sparse Matrix-Matrix Multiply
  - Dense to Sparse transfer

\[ A^{-1} = \frac{c_0}{2} I + \sum_{k=1}^{r} c_k T_k(Z) \]

\[ T_k(Z) = 2Z(T_{k-1}(Z)) - T_k(Z) \]
Sparse Matrix-Matrix Multiplication

- One thread per row: Multiply each element in the row
  - for (int jj = row_start; jj < row_end; jj++)
- Find corresponding element:
  - while (kk < col_end) && (indice[jj] >= vec1_indice[kk])
- If found, perform the multiplication
  - if (indice[jj] == vec1_indice[kk])
  - sum += data[jj] * vec1_data[kk];

Thread Id = 2:

<table>
<thead>
<tr>
<th>Indices</th>
<th>[0 1/ 1 2/ 0 2 3/ 1 3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>[1 7/ 2 8/ 5 3 9/ 6 4]</td>
</tr>
<tr>
<td>pointer</td>
<td>[0 2 4 7 9]</td>
</tr>
</tbody>
</table>
Sparse Matrix-Matrix Multiplication Code

```c
const int row = (blockDim.x * blockIdx.x + threadIdx.x);

if(row < num_rows){
    float sum = 0;
    int dense_flag = 0;

    int row_start = ptr[row];
    int row_end   = ptr[row+1];
    int col_start = vec1_ptr[col_num];
    int col_end   = vec1_ptr[col_num+1];

    for (int jj = row_start; jj < row_end; jj++){
        int kk = col_start; int not_found = 1;
        while( (kk < col_end) && (indice[jj] >= vec1_indice[kk]) && not_found){
            if (indice[jj] == vec1_indice[kk]){
                sum += data[jj] * vec1_data[kk];
                not_found = 0;
            }
            kk++;
        }
    }
```
Dense to Sparse Conversion

- Save the result of multiplication in dense format:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- If thread holds zero, write zero, otherwise write 1:

  | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

- Perform scan over the pointer vector:

  | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 |

```c
if (dense_ptr[tid] - dense_ptr[tid-1]){
  data[dense_ptr[tid] -1 + data_num] = dense_data[tid];
  indice[dense_ptr[tid] -1 + data_num] = tid;
}
```
Conjugate Gradient Algorithm

- Main GPU Kernels:
  - Vector Norm
  - Vector inner product \((a,b)\)
  - Update vectors

[Diagram showing the algorithm steps]
Vector Norm and Inner Product

- Vector norm adds square of all elements and output square root of result:
  - Load elements and multiply by itself
  - Perform scan and compute square root
- Inner product multiply corresponding elements of two vector and add up the results:
  - Load both vectors and multiply corresponding elements
  - Perform scan and report the results
Kernels Optimization:

- Optimizations on Matrix-Matrix vector multiply
  - Load both matrices in sparse
  - Copy vector in shared memory
  - Optimized search
  - Implement add in same kernel to avoid global memory

- Challenges:
  - Link lists and sparse formats
  - Varying matrix size in each kernel
Kernels Evaluation:

Main kernels Execution Time
Preconditioner Efficiency

- Methodology:
  - 1- Effect on Eigen values spectrum
  - 2- Effect on number of conjugate gradient iterations

Preconditioning Effect on IEEE-57 Test System
### CG Efficiency

<table>
<thead>
<tr>
<th>Matrix Name</th>
<th>Matrix Size</th>
<th>Total Nonzeros</th>
<th>Condition Number</th>
<th>Total Time Per Iteration (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E05R</td>
<td>236</td>
<td>5856</td>
<td>6e+04</td>
<td>1.13</td>
</tr>
<tr>
<td>E20R</td>
<td>4241</td>
<td>131556</td>
<td>5045e+10</td>
<td>11.33</td>
</tr>
<tr>
<td>E30R</td>
<td>9661</td>
<td>306356</td>
<td>3.47e+11</td>
<td>45.7</td>
</tr>
</tbody>
</table>

Matrices Structural Plot (source: Matrix Market)
Discussion

- Challenges, problems and ideas
  - Work with all matrices: matrix market form
  - New kernel ideas
    - SpMMul
    - Dens2Sp
- Future work and updates
  - Other iterative methods: GMRES, BiCG
Thank you!