Performance Comparisons of Parallel Power Flow Solvers on GPU System

Chunhui Guo\textsuperscript{1}, Baochen Jiang\textsuperscript{1}, Hao Yuan\textsuperscript{1}, Zhiqiang Yang\textsuperscript{1}, Li Wang\textsuperscript{2}, Shangping Ren\textsuperscript{2}

\textsuperscript{1}Shandong University at Weihai, China
\textsuperscript{2}Illinois Institute of Technology, USA
Outline

1. Background
2. Power Flow Model
3. Power Flow Solver
4. Parallelization
5. Performance Evaluation
6. Conclusion & Future Work
Describe steady state of a power system

Importance
- optimize real-time control of running power systems
- provide essential information for designing new power systems
- provide basics for other power system analysis

Calculation
- involve thousands of equations

Goal
- increase computation speed
Parallel Computing

- Common approaches
  - multi-threading
  - parallel machines
  - distributed systems

- Disadvantages of these approaches
  - special hardware support
  - high cost
  - limited speed improvement
Parallel Computing on GPU

- **GPU (Graphics Processing Unit)**
  - high computing efficiency
  - low price
  - widely used in many fields
  - CUDA (Compute Unified Device Architecture)

- **Current parallel power solvers on GPU**
  - Newton method, Jacobi method

- **What’s missing**
  - comparison among different parallel solvers

- **Our work**
  - parallelize and compare three common power flow solvers
For a power system with \( n \) independent buses, the power equations of bus \( i \) are:

\[
P_i = \sum_{k=1}^{n} |V_i V_k Y_{ik}| \cos(\theta_{ik} + \delta_k + \delta_i) \quad (1)
\]

\[
Q_i = -\sum_{k=1}^{n} |V_i V_k Y_{ik}| \sin(\theta_{ik} + \delta_k + \delta_i) \quad (2)
\]

- \( i, k \): bus number
- \( P \): real power
- \( Q \): reactive power
- \( |V| \): voltage magnitude
- \( \delta \): voltage angle
- \( |Y_{ik}| \): magnitude of admittance between bus \( i \) and bus \( k \)
- \( \theta_{ik} \): angle of admittance between bus \( i \) and bus \( k \)
Equation (1) and (2)

- non-linear
- both $|Y_{ik}|$ and $\theta_{ik}$ are known
- in $P$, $Q$, $|V|$ and $\delta$, two variables are known
- solvable

In order to calculate power flow, we need to solve the non-linear equations which consist of equation (1) and (2).
Power Flow Solver

- Calculation method
  - Gauss-Seidel solver
  - Newton-Raphson solver
  - P-Q decoupled solver

- Calculate steps
  - Input
  - Rearrange buses
  - Admittance matrix
  - Initialization
  - Iteration
  - Output
Power Flow Solver

- Gauss-Seidel solver
  - use the latest iteration value

- Newton-Raphson solver
  - transform non-linear equations to linear equations by Taylor series
  - coefficient matrix of linear equations (Jacobian matrix) needs to be recalculated in each iteration
  - polar form and rectangular form

- P-Q decoupled solver
  - simplified version of Newton-Raphson solver
  - use imaginary part of bus admittance to replace Jacobian matrix
  - coefficient matrix of linear equations remains unchanged
Speedup Analysis

- We use the multiplication number to estimate the computation cost and does not consider the communication cost between CPU and GPU.
- The speedup is sequential computation cost divided by parallel computation cost.
- For a power system with \( n \) buses, theoretical speedups are

<table>
<thead>
<tr>
<th>Power Flow Solver</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss-Seidel Solver</td>
<td>0.2n</td>
</tr>
<tr>
<td>Newton-Raphson Solver</td>
<td>2n</td>
</tr>
<tr>
<td>P-Q Decoupled Solver</td>
<td>0.4n</td>
</tr>
</tbody>
</table>
Parallelization

- Two problems
  - Which operations to parallelize?
  - How to parallelize?
- Parallelization operations
  - Bus admittance matrix computation
  - Iteration process
- Parallelization steps
  - Allocate GPU memory
  - Copy original data from CPU to GPU
  - Call kernel to process data
  - Copy result data from GPU to CPU
  - Release GPU memory
Gauss-Seidel Iteration

- **Gauss-Seidel iterative format**

\[
V_{i}^{(k+1)} = \frac{1}{Y_{ii}} \left( \frac{P_{i} - jQ_{i}}{V_{i}^{(k)}} \right) - \sum_{j=1}^{i-1} Y_{ij}V_{j}^{(k+1)} - \sum_{j=i+1}^{n} Y_{ij}V_{j}^{(k)}
\]  \hspace{1cm} (3)

\[
Q_{i}^{(k)} = - \text{Im} \left[ V_{i}^{(k)} \left( \sum_{j=1}^{i-1} Y_{ij}V_{j}^{(k+1)} + \sum_{j=i}^{n} Y_{ij}V_{j}^{(k)} \right) \right]
\]  \hspace{1cm} (4)

- **Parallelization operations**
  - summation operations in equation (3) and (4)
• Parallelization operations
  - Jacobian matrix computation
  - **linear equations solver**

• Jacobian matrix computation

\[
J = \begin{bmatrix}
H & N \\
K & L \\
\end{bmatrix} \tag{5}
\]
P-Q Decoupled Iteration

- Parallelization operations
  - linear equations solver
Linear Equations Solver

- Gaussian elimination method
  - forward elimination
  - back substitution

- Augmented matrix

\[
A = \begin{bmatrix}
a_{11} & \cdots & a_{1k} & \cdots & a_{1n} & a_{1,n+1} \\
\vdots & & \vdots & & \vdots & \vdots \\
a_{k1} & a_{kk} & a_{kn} & a_{k,n+1} \\
\vdots & & \vdots & & \vdots & \vdots \\
a_{n1} & \cdots & a_{nk} & \cdots & a_{nn} & a_{n,n+1}
\end{bmatrix}
\]  

(6)

- kth forward elimination step

\[
a_{kj} = \frac{a_{kj}}{a_{kk}}, \quad (j = k + 1 \sim n+1)
\]  

(7)

\[
a_{ij} = a_{ij} - a_{ik} \cdot a_{kj}, \quad (i = k + 1 \sim n, \ j = k + 1 \sim n+1)
\]  

(8)
Gaussian Forward Elimination (1)

- Kernel to process equation (7)

\[ a_{kj} = a_{kj} / a_{kk}, (j = k + 1 \sim n + 1) \]  \hspace{1cm} (7)

**Algorithm 1 GAUSS ELIMINATION CUDA KERNEL A**

**Input:** Augmented matrix in GPU memory: `augMatrixGPU`, number of rows in matrix `augMatrixGPU`: \( n \), the Gauss forward elimination step: \( k \).

1. \( i \leftarrow \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \)
2. \( j \leftarrow \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} \)
3. **if** \( i = k \) and \( j > k \) and \( j < n + 1 \) and \( \text{augMatrixGPU}[k \times (n + 1) + k] \neq 0.0 \) **then**
4. \( \text{augMatrixGPU}[k \times (n + 1) + j] \leftarrow \text{augMatrixGPU}[k \times (n + 1) + j] / \text{augMatrixGPU}[k \times (n + 1) + k] \)
5. **end if**
Kernel to process equation (8)

\[ a_{ij} = a_{ij} - a_{ik} \ast a_{kj}, (i = k + 1 \sim n, j = k + 1 \sim n + 1) \]  

(8)

**Algorithm 2 GAUSS ELIMINATION CUDA KERNEL B**

**Input:** Augmented matrix in GPU memory:
- \( \text{augMatrixGPU} \), number of rows in matrix \( \text{augMatrixGPU} \): \( n \), the Gauss forward elimination step: \( k \).

1. \( i \leftarrow \text{blockIdx.x} \ast \text{blockDim.x} + \text{threadIdx.x} \)
2. \( j \leftarrow \text{blockIdx.y} \ast \text{blockDim.y} + \text{threadIdx.y} \)
3. **if** \( i > k \) and \( i < n \) and \( j > k \) and \( j < n + 1 \) and \( \text{augMatrixGPU}[k \times (n + 1) + k] \neq 0.0 \) **then**
4. \( \text{augMatrixGPU}[i \times (n + 1) + j] \leftarrow \text{augMatrixGPU}[i \times (n + 1) + j] - \text{augMatrixGPU}[i \times (n + 1) + k] \times \text{augMatrixGPU}[k \times (n + 1) + j] \)
5. **end if**
Algorithm 3 GAUSS FORWARD ELIMINATION

Input: Augmented matrix in GPU memory:

- augmentMatrix, number of rows in matrix
- augmentMatrix: n.

1: cudaMalloc((void**)&aguMatrixGPU, sizeof(float) \times n \times (n + 1))
2: cudaMemcpy2D(aguMatrixGPU, sizeof(float) \times (n + 1), aguMatrix, sizeof(float) \times (n + 1), sizeof(float) \times (n + 1), n, cudaMemcpyHostToDevice)
3: dim3 blockDim(22, 22)
4: dim3 gridDim((n+blockDim.x−1)/blockDim.x, (n+1+blockDim.y−1)/blockDim.y)
5: for k ← 0 to n − 1 do
6: GaussKernelA <<< gridDim, blockDim >>> (aguMatrixGPU, n, k);
7: GaussKernelB <<< gridDim, blockDim >>> (aguMatrixGPU, n, k);
8: end for
9: cudaMemcpy2D(aguMatrix, sizeof(float) \times (n + 1), aguMatrixGPU, sizeof(float) \times (n + 1), sizeof(float) \times (n + 1), n, cudaMemcpyDeviceToHost)
10: cudaFree(aguMatrixGPU)
Performance Evaluation

- **Experiment platform**
  - host: Intel i3-2100 CPU(3.10GHz) & 2G RAM
  - device: Nvidia GeForce GTS450 GPU(192 CUDA cores & 1G RAM)
  - software: Windows 7, CUDA 4.0

- **Experiment power systems**

<table>
<thead>
<tr>
<th>System</th>
<th>Bus Count</th>
<th>Branch Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>IEEE30</td>
<td>30</td>
<td>41</td>
</tr>
<tr>
<td>IEEE118</td>
<td>118</td>
<td>186</td>
</tr>
<tr>
<td>IEEE300</td>
<td>300</td>
<td>357</td>
</tr>
<tr>
<td>Shandong</td>
<td>974</td>
<td>1449</td>
</tr>
</tbody>
</table>
Experiment Result (1)

- Gauss-Seidel solver

<table>
<thead>
<tr>
<th>System</th>
<th>CPU Runtime (s)</th>
<th>GPU Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE9</td>
<td>0.0001</td>
<td>0.3276</td>
<td>0.0003</td>
</tr>
<tr>
<td>IEEE30</td>
<td>0.002</td>
<td>0.7051</td>
<td>0.0028</td>
</tr>
<tr>
<td>IEEE118</td>
<td>0.023</td>
<td>3.2963</td>
<td>0.007</td>
</tr>
<tr>
<td>IEEE300</td>
<td>0.3428</td>
<td>7.2992</td>
<td>0.047</td>
</tr>
<tr>
<td>Shandong</td>
<td>1.2147</td>
<td>19.603</td>
<td>0.062</td>
</tr>
</tbody>
</table>
Newton-Raphson solver

<table>
<thead>
<tr>
<th>System</th>
<th>CPU Runtime (s)</th>
<th>GPU Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE9</td>
<td>0.0015</td>
<td>0.0094</td>
<td>0.1596</td>
</tr>
<tr>
<td>IEEE30</td>
<td>0.0098</td>
<td>0.0094</td>
<td>1.0426</td>
</tr>
<tr>
<td>IEEE118</td>
<td>0.3132</td>
<td>0.1997</td>
<td>1.5684</td>
</tr>
<tr>
<td>IEEE300</td>
<td>4.689</td>
<td>2.6848</td>
<td>1.7465</td>
</tr>
<tr>
<td>Shandong</td>
<td>583.831</td>
<td>10.881</td>
<td>53.656</td>
</tr>
</tbody>
</table>
Experiment Result (3)

- P-Q decoupled solver

<table>
<thead>
<tr>
<th>System</th>
<th>CPU Runtime (s)</th>
<th>GPU Runtime (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE9</td>
<td>0.0047</td>
<td>0.0047</td>
<td>1.0</td>
</tr>
<tr>
<td>IEEE30</td>
<td>0.0081</td>
<td>0.0125</td>
<td>0.648</td>
</tr>
<tr>
<td>IEEE118</td>
<td>0.1137</td>
<td>0.117</td>
<td>0.9718</td>
</tr>
<tr>
<td>IEEE300</td>
<td>1.5107</td>
<td>1.1606</td>
<td>1.3017</td>
</tr>
<tr>
<td>Shandong</td>
<td>148.974</td>
<td>5.5068</td>
<td>27.0527</td>
</tr>
</tbody>
</table>
Result Analysis

![Graph showing speedup vs system size for different solvers]

- Gauss-Seidel solver
- Newton-Raphson solver
- P-Q decoupled solver

The graph illustrates the speedup achieved with increasing system size for each solver.
Conclusion

- Parallelize three power flow solvers on GPU
  - bus admittance matrix computation
  - iteration process

- Compare speedup of three parallel power flow solvers
  - Newton-Raphson solver: best
  - P-Q decoupled solver: middle
  - Gauss-Seidel solver: worst
Future Work

- Improve speedup
- Reduce computation time
- Study different applications
- …
Thank You!

Q & A

Chunhui Guo  chunhui.guo@hotmail.com