CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs

Jiya Su‡, Feng Zhang◊, Weifeng Liu★, Bingsheng He+, Ruofan Wu◊, Xiaoyong Du◊, Rujia Wang‡

◊Renmin University of China
★China University of Petroleum
+National University of Singapore
‡Illinois Institute of Technology
Outline

1. Background
2. Motivation
3. Challenges
4. CapelliniSpTRSV
5. Evaluation
6. Source Code at Github
7. Conclusion
Outline

1. Background
2. Motivation
3. Challenges
4. CapelliniSpTRSV
5. Evaluation
6. Source Code at Github
7. Conclusion
1. Background

Sparse Matrix in CSR format

Lower Triangular Matrix $L$

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

(a) Matrix $L$.

 csrRowPtr = (0, 1, 2, 4, 7, 10, 12, 16, 20)

csrColIdx = (0, 1, 1, 2, 1, 2, 3, 0, 1, 4, 2, 5, 0, 2, 5, 6, 0, 1, 2, 7)

csrVal = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)

(b) CSR representation.
1. Background

Sparse Triangular Solve

Example: \( Lx = b \)
1. Background

Sparse Triangular Solve

Example: $Lx = b$
1. Background

Concepts:

- Component
1. Background

Concepts:
- Component
- Element

Lower Triangular Matrix $L$

Matrix $L$

$$
\begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
6 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
7 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{bmatrix}
$$

Element

$$
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
\end{bmatrix}
\times
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
\end{bmatrix}
= 
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
\end{bmatrix}
$$

$x \times b$
1. Background

Concepts:
· Component
· Element
· Dependency

Lower Triangular Matrix $L$

Matrix $L$

$$
\begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & & & & & & \\
1 & 1 & 1 & & & & & \\
2 & & 1 & 1 & & & & \\
3 & & & 1 & 1 & 1 & & \\
4 & & & & 1 & 1 & & \\
5 & & & & & 1 & 1 & \\
6 & & & & & & 1 & 1 \\
7 & & & & & & & 1
\end{bmatrix}
$$

Dependency

$$
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1
\end{bmatrix}
\times
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1
\end{bmatrix}
=
\begin{bmatrix}
1 \\
1 \\
3 \\
3 \\
2 \\
4
\end{bmatrix}
$$
1. Background

Concepts:
- Component
- Element
- Dependency
- Level

Lower Triangular Matrix $L$

Matrix $L$

Level set

$\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{bmatrix}$

$x$

$b$
1. Background

Level-set SpTRSV

The level-set method has two phases: (1) grouping nodes (rows or columns) that can be consumed in parallel, and (2) solving nodes group by group with barriers between.

(a) Matrix $L$.

(b) Components $x$ in the level-sets.
1. Background

Level-set SpTRSV

The level-set method has two phases: (1) grouping nodes (rows or columns) that can be consumed in parallel, and (2) solving nodes group by group with barriers between.

(a) Matrix $L$

(b) Components $x$ in the level-sets.
1. Background

Level-set SpTRSV

The level-set method has two phases: (1) grouping nodes (rows or columns) that can be consumed in parallel, and (2) solving nodes group by group with barriers between.

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

(a) Matrix $L$.

(b) Components $x$ in the level-sets.
1. Background

Synchronization-Free SpTRSV (warp-level)

The algorithm computes components $x$ in the original row order of the input matrix and uses one warp to compute one row.

It uses a new flag array `in_degree` to show whether the component $x$ is solved, which avoids the synchronization and greatly reduces the processing time.

1. Background

Synchronization-Free SpTRSV (warp-level)

The algorithm computes components $x$ in the original row order of the input matrix and uses one warp to compute one row.

It uses a new flag array `in_degree` to show whether the component $x$ is solved, which avoids the synchronization and greatly reduces the processing time.

1. Background

Synchronization-Free SpTRSV (warp-level)

The algorithm computes components $x$ in the original row order of the input matrix and uses one warp to compute one row.

It uses a new flag array $\text{in\_degree}$ to show whether the component $x$ is solved, which avoids the synchronization and greatly reduces the processing time.

1. Background

Case study for preprocessing time and execution time of different SpTRSV algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>time (ms)</th>
<th>nlpkkt160</th>
<th>wiki-Talk</th>
<th>cant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-Set</td>
<td>preprocessing</td>
<td>310.07</td>
<td>31.09</td>
<td>4.81</td>
</tr>
<tr>
<td></td>
<td>execution</td>
<td>28.07</td>
<td>12.89</td>
<td>28.79</td>
</tr>
<tr>
<td>cuSPARSE</td>
<td>preprocessing</td>
<td>16.24</td>
<td>1.99</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>execution</td>
<td>37.98</td>
<td>11.88</td>
<td>7.69</td>
</tr>
<tr>
<td>Sync-Free</td>
<td>preprocessing</td>
<td>8.07</td>
<td>0.42</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>execution</td>
<td>27.73</td>
<td>10.02</td>
<td>5.02</td>
</tr>
</tbody>
</table>
2. Motivation

Performance trend of warp-level synchronization-free SpTRSV.

\[
\text{parallel\_granularity} = \log_{c_1}\left(\frac{\log_{c_2}(n_{\text{level}})}{\log_{c_3}(nnz_{\text{row}} + b_1)} + b_2\right)
\]
2. Motivation

Performance trend of warp-level synchronization-free SpTRSV.

$$parallel\_granularity = \log_{c_1}\left(\frac{\log_{c_2}(n_{level})}{\log_{c_3}(nnz_{row} + b_1)} + b_2\right)$$

The performance declines after reaching the peak state.
2. Motivation

(a) Level-Set SpTRSV.

(b) Warp-Level Synchronization-Free SpTRSV.

(c) Thread-Level Synchronization-Free SpTRSV (CapelliniSpTRSV).
2. Motivation

• Observation: Warp-level synchronization-free SpTRSV algorithm cannot fully utilize GPU resources when parallel granularity is large.

• Insight:
3. Challenges

- Challenge 1: avoiding deadlocks
  - In thread-level design, the threads in one warp may have dependencies.

![Diagram showing thread and warp levels with data transmission](image-url)
3. Challenges

- Challenge 2: last element checking
  - We need to verify whether the processed element is on the diagonal, which causes time overhead.
3. Challenges

• Challenge 3: thread execution model
  • Although we use a thread to handle one component, the GPUs are still executed in the warp execution mode.
4. CapelliniSpTRSV

• Design to avoid deadlocks
  • A two-phase mechanism to avoid the deadlocks in CapelliniSpTRSV
4. CapelliniSpTRSV

main() {  //host code
    InputMatrix(L);  // Rows = L.row_number
    InitiateVector(x, b, get_value);  // x = 0, get_value = 0
    launchKernel(Rows);  // create Rows threads
}

kernel(L, x, b, get_value) {  // GPU kernel
    rowID = globalID;
    sum = 0;
    B = getBoundary(L, rowID);
    processWhileLoop(L, b, B, rowID, sum, get_value);
    processWrtFst(L, b, B, rowID, sum, get_value, x);
}

(a) Two-Phase CapelliniSpTRSV
4. CapelliniSpTRSV

- Design to avoid deadlocks
  - A two-phase mechanism to avoid the deadlocks in CapelliniSpTRSV.

- Efficient last element checking
  - A novel design to reduce the number of last element checking.
processWhileLoop(L, b, B, rowID, sum, get_value){
    For id = L.rowID.start to B{
        While !checkSolve(L, id, get_value);
        recordValue(L, id, b, sum);
    }
}

processWrtFst(L, b, B, rowID, sum, get_value, x){
    id = B;
    While id < L.rowID.end{
        While checkSolve(L, id, get_value){
            recordValue(L, id, b, sum);
            id ++;
        }
        If id == (L.rowID.end -1){
            computeXValue(L, x, b, sum, rowID);
            setValue_get(rowID, get_value);
            id ++;
        }
    }
}
4. CapelliniSpTRSV

processWhileLoop(L, b, B, rowID, sum, get_value){
    For id = L.rowID.start to B{
        While !checkSolve(L, id, get_value);
            recordValue(L, id, b, sum);
    }
}

processWrtFst(L, b, B, rowID, sum, get_value, x){
    id = B;
    While id < L.rowID.end{
        While checkSolve(L, id, get_value){
            recordValue(L, id, b, sum);
            id ++;
        }
        If id == (L.rowID.end -1){
            computeXValue(L, x, b, sum, rowID);
            setValue_get(rowID, get_value);
            id ++;
        }
    }
}
4. CapelliniSpTRSV

• Design to avoid deadlocks
  • A two-phase mechanism to avoid the deadlocks in CapelliniSpTRSV.

• Efficient last element checking
  • A novel design to reduce the number of such last element checkings.

• Adaptation to GPU thread execution
  • A Writing-First optimization that threads can compute the elements and write the partial results first without waiting for the other threads.
main() {  //host code
    InputMatrix(L); // Rows = L.row_number
    InitiateVector(x, b, get_value); // x = 0, get_value = 0
    launchKernel(Rows); // create Rows threads
}

kernel(L, x, b, get_value) {  // GPU kernel
    rowID = globalID;
    sum = 0;
    B = getBoundary(L, rowID);
    processWhileLoop(L, b, B, rowID, sum, get_value);
    processWrtFst(L, b, B, rowID, sum, get_value, x);
}

(a) Two-Phase CapelliniSpTRSV
main() {  //host code
    InputMatrix(L); // Rows = L.row_number
    InitiateVector(x, b, get_value); // x = 0, get_value = 0
    launchKernel(Rows); // create Rows threads
}

kernel(L, x, b, get_value) {  // GPU kernel
    rowID = globalID;
    sum = 0;
    B = getBoundary(L, rowID);
    processWhileLoop(L, b, B, rowID, sum, get_value);
    processWrtFst(L, b, B, rowID, sum, get_value, x);
}

(a) Two-Phase CapelliniSpTRSV
4. CapelliniSpTRSV

main() {
  //host code
  InputMatrix(L); // Rows = L.row_number
  InitiateVector(x, b, get_value); // x = 0, get_value = 0
  LaunchKernel(Rows); // create Rows threads
}

kernel(L, x, b, get_value) {
  // GPU kernel
  rowID = globalID;
  sum = 0;
  processWrtFst(L, b, L.rowID.start, rowID, sum, get_value, x);
}

(b)Writing-First CapelliniSpTRSV
4. CapelliniSpTRSV

Features:

• **No preprocessing**
  • Our algorithm can be easily applied to various situations.

• **Strong effectiveness**
  • Our algorithm completes the current synchronization-free SpTRSV design.

• **CSR format**
  • The most popular CSR format.
4. CapelliniSpTRSV

Features:

• No preprocessing
  • Our algorithm can be easily applied to various situations.

• Strong effectiveness
  • Our algorithm completes the current synchronization-free SpTRSV design.

• CSR format
  • The most popular CSR format.
4. CapelliniSpTRSV

Features:

• No preprocessing
  • Our algorithm can be easily applied to various situations.

• Strong effectiveness
  • Our algorithm completes the current synchronization-free SpTRSV design.

• CSR format
  • The most popular CSR format.
4. CapelliniSpTRSV

Features:

• No preprocessing
  • Our algorithm can be easily applied to various situations.

• Strong effectiveness
  • Our algorithm completes the current synchronization-free SpTRSV design.

• CSR format
  • The most popular CSR format.
5. Evaluation

Experimental Setup

• Methods
  • Capellini
  • SyncFree
  • cuSPARSE

• Platforms
  • Pascal: GTX 1080
  • Volta: V100
  • Turing: RTX 2080 ti

• Datasets
  • 245 matrices from University of Florida Sparse Matrix Collection
5. Evaluation

Experimental Setup

• Methods
  • Capellini
  • SyncFree
  • cuSPARSE

• Platforms
  • Pascal: GTX 1080
  • Volta: V100
  • Turing: RTX 2080 ti

• Datasets
  • 245 matrices from University of Florida Sparse Matrix Collection
5. Evaluation

Experimental Setup

• Methods
  • Capellini
  • SyncFree
  • cuSPARSE

• Platforms
  • Pascal: GTX 1080
  • Volta: V100
  • Turing: RTX 2080 ti

• Datasets
  • 245 matrices from University of Florida Sparse Matrix Collection
5. Evaluation

Performance (GFLOPS/s) average:
- cuSPARSE: 1.92 GFLOPS/s
- SyncFree: 1.78 GFLOPS/s
- CapelliniSpTRSV: 6.84 GFLOPS/s

(a) Pascal (GeForce GTX 1080)  (b) Volta (Tesla V100)  (c) Turing (GeForce RTX 2080 Ti)
5. Evaluation

Speedup average:

SyncFree : 4.97x

cuSPARSE: 4.74x
5. Evaluation

Algorithm preference distribution

![Graph showing algorithm preference distribution](image)
5. Evaluation

Detailed Analysis

Bandwidth utilization (sum of read and write bandwidth)
5. Evaluation

- Detailed Analysis

(a) Number of GPU instructions executed.

(b) Percentage of instruction dependency stalls.
6. Source Code at GitHub

- https://github.com/JiyaSu/CapelliniSpTRSV
7. Conclusion

- We show our insights in current SpTRSV algorithms and propose parallel granularity to describe sparse matrices.
- We develop CapelliniSpTRSV to process sparse matrices that previous SpTRSV algorithms cannot handle efficiently.
- We evaluate CapelliniSpTRSV with 245 matrices, and demonstrate its benefits over the state-of-the-art SpTRSV.
7. Conclusion

- We show our insights in current SpTRSV algorithms and propose parallel granularity to describe sparse matrices.

- We develop CapelliniSpTRSV to process sparse matrices that previous SpTRSV algorithms cannot handle efficiently.

- We evaluate CapelliniSpTRSV with 245 matrices, and demonstrate its benefits over the state-of-the-art SpTRSV.
7. Conclusion

• We show our insights in current SpTRSV algorithms and propose parallel granularity to describe sparse matrices.

• We develop CapelliniSpTRSV to process sparse matrices that previous SpTRSV algorithms cannot handle efficiently.

• We evaluate CapelliniSpTRSV with 245 matrices, and demonstrate its benefits over the state-of-the-art SpTRSV.
Thank you!

• Any questions?

CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs

Jiya Su♦, Feng Zhang♦, Weifeng Liu★, Bingsheng He+,
Ruofan Wu♦, Xiaoyong Du♦, Rujia Wang‡

♦Renmin University of China
★ China University of Petroleum
+National University of Singapore
‡ Illinois Institute of Technology

Jiya_Su@ruc.edu.cn, fengzhang@ruc.edu.cn, weifeng.liu@cup.edu.cn, hebs@comp.nus.edu.sg,
2017202106@ruc.edu.cn, duyong@ruc.edu.cn, rwang67@iit.edu