GPU-accelerated sparse triangular solver

Longxiang Chen
Yongjian Hu
Chi Gou

Outline

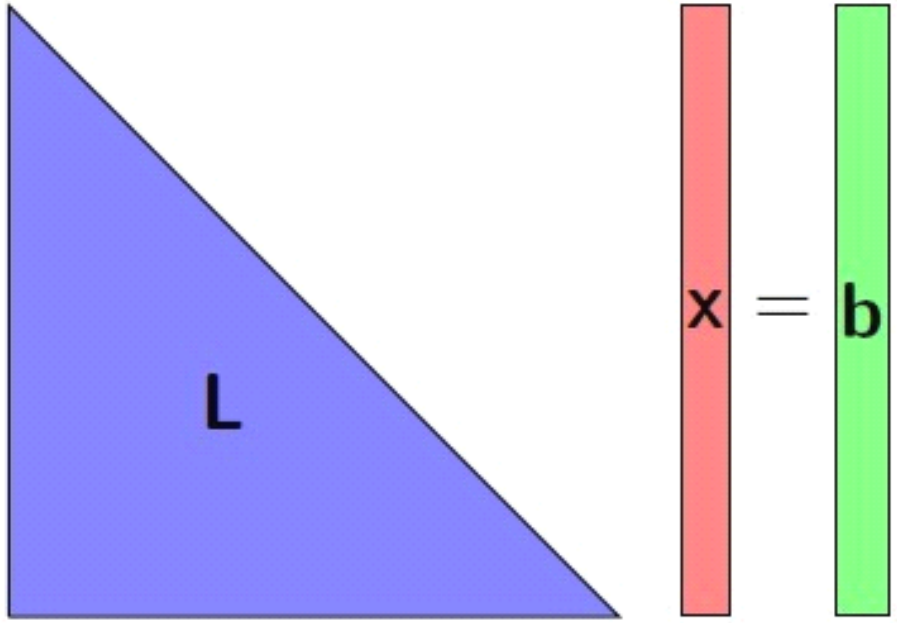
- Introduction
- Matrix Storing Format
- Analysis and Solver
- Implement
- Results
- Conclusion

Outline

- Introduction
- Matrix Storing Format
- Analysis and Solver
- Implement
- Results
- Conclusion

Triangular Solver

- Solve $L\vec{x} = \vec{b}$, L is lower triangular matrix.



Basic Algorithm

Input: Lower-triangular $n \times n$ matrix L, right-hand-side vector x.
for $i = 1, n$ **do**

$$x(i+1:n) = x(i+1:n) - L(i+1:n, i) * x(i)$$

end for
Output: solution vector x.

$$\begin{pmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & 1 & \\ l_{41} & l_{42} & l_{43} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

Science & Technology Facilities Council

Pictures from STFC Rutherford Appleton Laboratory

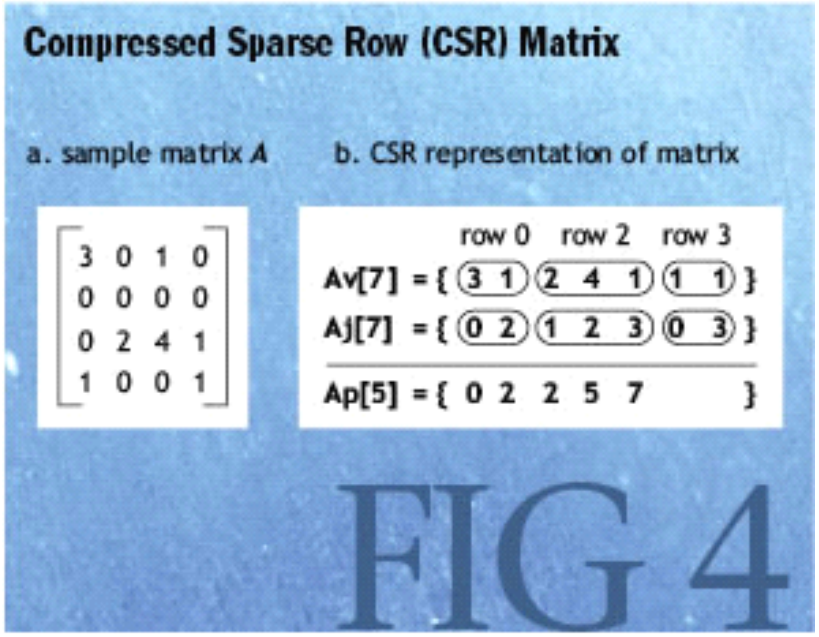
Outline

- Introduction
- Matrix Storing Format
- Analysis and Solver
- Implement
- Results
- Conclusion

Compressed sparse row(CSR)

- CSR is (val, col_ind, row_ptr)

1. val: an array of the non-zero values
2. col_ind: the column indices corresponding to the values
3. row_ptr: the list of value indexes where each row starts.



Compressed sparse column(CSC)

- CSR is (val, row_ind, col_ptr)

1. val: an array of the non-zero values
2. row_ind: the row indices corresponding to the values
3. col_ptr: the list of value indexes where each column starts.

CSR and CSC

- Use both to improve the efficiency of solving
- CPU:

  CSC for topology sorting to choose the nodes that can put in the same level
- GPU:

  CSR for equation solving row by row

Outline

- Introduction
- Matrix Storing Format
- Analysis and Solver
- Implement
- Results
- Conclusion

Analysis

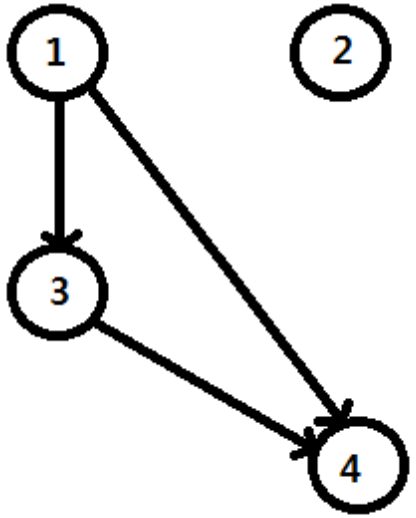
- Choose the rows that are independent to each other
- Use Directed Acyclic Graph(DAG)
- Topology sorting to choose nodes in one level

How to leveling?

- Topology sorting all nodes
- Push the node with indegree = 0 into queue, then decrease its children's indegree by 1
- Recursive until all nodes in queue

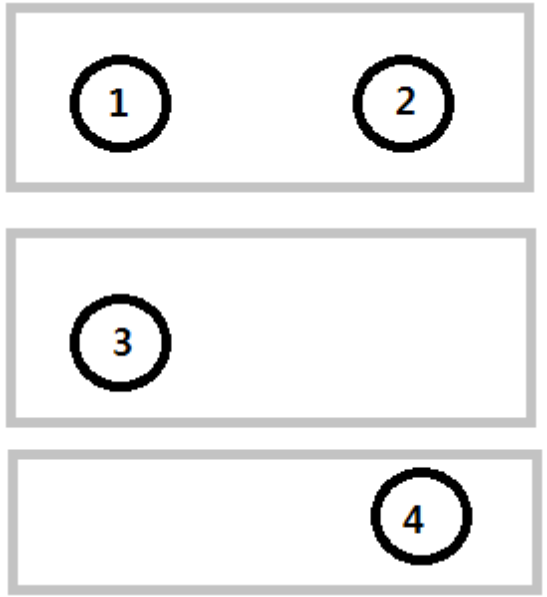
example

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$



example

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{bmatrix}$$



Solver

- Naive
- CSC format with single block
- CSR format with multiblock

Naive

- One block, multithreads
- solve order: row by row
- pro: easy to implement
- con: efficiency is bad, not parallelized

CSC with single block

- Nodes in same level executing at the same time, others wait
- Each thread calculate one column
- One thread works for one column, but all nodes in this level.

```
Gpu_kernel() {
  foreach level {
    foreach ele in this level {
      if (threadID == ele) {
          sum this row;
          x[i] = (b[i] – sum) / A[i][i];
      }
    }
    _syncthreads();
  }
}
```

CSR with Multiblock

- CPU works with GPU to acclerate
- CPU does the analysis phase, and GPU finishes the solve phase.

Outline

- Introduction
- Matrix Storing Format
- Analysis and Solver
- Implement
- Results
- Conclusion

What CPU does

1. Choose the nodes that are independent with each other.

   Set dimGrid = independent nodes number

   Set dimBlock = max number of non-zero elements in these nodes

1. Call the GPU_Kernel to execute these rows only

Example:

$$\begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix}$$

Example

- CSC
- row_ind = [0, 3, 4, 1, 3, 2, 3, 4]
- col_ptr = [0, 3, 4, 5, 6, 7]
- indeg = [0, 0, 0, 2, 1]

$$\begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix}$$

Example

- CSC
- row_ind = [0, 3, 4, 1, 3, 2, 3, 4]
- col_ptr = [0, 3, 4, 5, 6, 7]
- indeg = [0, 0, 0, 2, 1]

level 0 = [0, 1, 2] (children indeg-1)

indeg = [-1, -1, -1, 0, 0]

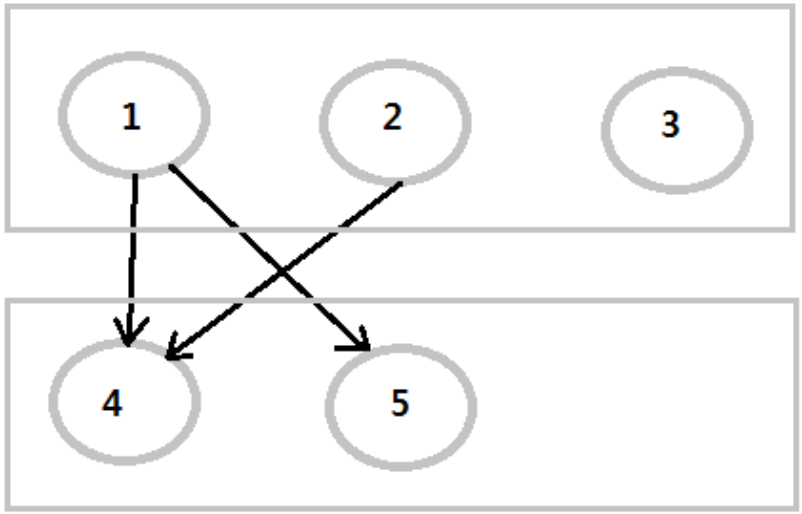
level 1 = [3, 4]

Done.

$$\begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix}$$

Example:

$$\begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix}$$



What GPU does

1. each block handles one row
2. each thread handles one nozero elements
3. reduction to sum all x calculated so far
4. diagnal element of thead solves

   x[i] = (b[i] - sum) / val[i][i]

   in this row

Example

- topo = [0, 1, 2, 3, 4]
- lv_ptr = [0, 3, 5]

$$\begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix}$$

Example

- topo = [0, 1, 2, 3, 4]
- lv_ptr = [0, 3, 5]
- GPU_Kernel:

level 0:

#block = 3, #thread = 1

topo = [0, 1, 2]

solve $x_{0,}$ x_1 and x_2

$$\left[\begin{array}{ccccc} * & 0 & 0 & 0 & 0 \\ \hline 0 & * & 0 & 0 & 0 \\ \hline 0 & 0 & * & 0 & 0 \\ \hline * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{array}\right]$$

Example

- topo = [0, 1, 2, 3, 4]
- lv_ptr = [0, 3, 5]
- GPU_Kernel:

level 1:

#block = 2, #thread = 3

topo = [3, 4]

$$\left[\begin{array}{ccccc} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ \hline * & 0 & 0 & 0 & * \end{array}\right]$$

Example

level 1:

#block = 2, #thread = 3

row 3: (num = $\lfloor \log_2 nz \rfloor$ = 2)

1. shared S[2] = [$tmpval_0 * x_0$, $tmpval_1 * x_1$]
2. reduction to sum them up
3. $x_3 = (b_3 - S[0]) / val[3][3]$

$$\begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix}$$

Example

level 1:

#block = 2, #thread = 3

row 4: (num = $\lfloor \log_2 nz \rfloor$ = 0)

1. shared S[0] = [$tmpval_0 * x_{0,}$]
2. reduction to sum them up
3. x_4= (b_4 - S[0]) / val[4][4]

$$\begin{bmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ * & * & 0 & * & 0 \\ * & 0 & 0 & 0 & * \end{bmatrix}$$

Pseudocode

```
CPU CODE:
for each level
   dimgrid = number of rows in this level
   dimblock = max number of nonzero
                  elements in these rows
   call GPU_kernel;
End
```

```
GPU_kernel() {
   determin rows in the same level
   shared array to store [
nonzero element * x solved in previous level]

   reduction to sum the results in shared array
   solve x = b - sum of array
}
```

Outline

- Introduction
- Matrix Storing Format
- Analysis
- Solver
- Results
- Conclusion

Results

	Analysis (usec)	Solving (usec)	Total Time (usec)
Naïve CPU		1798	1798
CUBLAS		1744	1744
CSR CPU		38	38
GPU Single Block	80	1140	1220
GPU Multi Block	84	183	267
CUSPARSE	1021	65	1086

- newton.engr.ucr.edu
  - CPU 32-core Intel Xeon ES-2670 @ 2.6GHz
  - GPU devices has 14 Multi-Processors

Results

	Analysis (usec)	Solving (usec)	Total Time (usec)	Speedup
Naïve CPU		1798	1798	6.7341
CUBLAS		1744	1744	6.5318
CSR CPU		38	38	0.1423
GPU Single Block	80	1140	1220	4.5683
GPU Multi Block	84	183	267	
CUSPARSE	1021	65	1086	4.0674

- newton.engr.ucr.edu
  - CPU 32-core Intel Xeon ES-2670 @ 2.6GHz
  - GPU devices has 14 Multi-Processors

Outline

- Introduction
- Matrix Storing Format
- Analysis
- Solver
- Results
- Conclusion

Conclusions

- Combine CPU and GPU together to improve the efficiency
- Use CPU for topology sorting and leveling, speedup is (1021/80) = 12.7625 to cusparse.
- Implment a faster sparse triangular solver than cusparse, speedup is 4.0674
- Find our code in https://github.com/suifengls/cs217_trsv

Thank you!