Sparse LU Factorization on GPUs for Accelerating SPICE Simulation

Xiaoming Chen
PhD Candidate
Department of Electronic Engineering
Tsinghua University, China
Email: chenxm05@mails.tsinghua.edu.cn
Outline

- Introduction
- Our Solution
- Results
- Conclusions
Outline

- **Introduction**
  - What is SPICE?
  - Why do we need to accelerate SPICE?
  - Related work
  - Left-looking algorithm

- **Our Solution**

- **Results**

- **Conclusions**
What is SPICE?

- Simulation Program with Integrated Circuit Emphasis
- Transistor-level integrated circuit simulator
- SPICE kernel was developed by University of California, Berkeley in 1975
- Commercial software: Hspice (Synopsys), Pspice (Cadence), Spectre (Cadence), Eldo (Monter), …
What is SPICE?

SPICE workflow example

- **Input**
  - Circuit diagram with elements labeled:
    - $e_1$, $e_2$, $v_3$, $k$, $I_0$
  - Resistors $R_1$, $R_2$, $R_3$, $R_4$
  - Source $I_0$

- **Equations**
  - \[
    \begin{bmatrix}
      G_1 & -G_1 - k & k \\
      -G_1 & G_1 + G_2 + G_3 & -G_3 \\
      0 & -G_3 + k & G_3 + G_4 - k
    \end{bmatrix}
    \begin{bmatrix}
      e_1 \\
      e_2 \\
      e_3
    \end{bmatrix}
    =
    \begin{bmatrix}
      I_0 \\
      0 \\
      0
    \end{bmatrix}
  \]

- **Solve**
  - DC simulation output
    - \[
      \begin{bmatrix}
      e_1 = ? \\
      e_2 = ? \\
      e_3 = ?
    \end{bmatrix}
    \]
  - Transient simulation output
    - Graph showing voltage $e_3$ over time (msec)

- **Output**
Why do we need to accelerate SPICE?

SPICE is very time-consuming!
Reason: iterations!
Several days (weeks) to run a pre-layout (post-layout) simulation for a modern analog-to-digital/digital-to-analog convertor
Why do we need to accelerate SPICE?

Transient simulation flow

- Read netlist, build matrix
- Matrix pre-processing
-Transient iterations
-Newton-Raphson iterations
-Iteration update
- Model evaluation

- Sparse LU factorization ($A=LU$)
- Right-hand-solving ($Ly=b, Ux=y$)

- Solving $Ax=b$

Matrix features
- Unsymmetric
- Highly sparse
- Irregular nonzero pattern
- Ill-conditioned

Solving the linear system costs 1/4 to 3/4 of the total time

Bottleneck of SPICE!

Models are independent. Parallelization is straightforward on both CPUs and GPUs.
Related work

**SPARSE (NOT dense) DIRECT (NOT iterative) solvers on GPUs**

<table>
<thead>
<tr>
<th>reference</th>
<th>matrix type</th>
<th>precision</th>
<th>average speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Christen, GPPGPU’07]</td>
<td>unsymmetric</td>
<td>single</td>
<td>about 2X vs. sequential PARDISO</td>
</tr>
<tr>
<td>[Krawezik, SAAHPC’09]</td>
<td>symmetric</td>
<td>double</td>
<td>2.9X vs. 2-threaded ANSYS</td>
</tr>
<tr>
<td>[Yu, Para. Comp. 2011]</td>
<td>unsymmetric</td>
<td>double</td>
<td>about 2.5X vs. sequential UMFPACK</td>
</tr>
<tr>
<td>[George, IPDPS’11]</td>
<td>symmetric</td>
<td>single</td>
<td>7X vs. sequential (double-precision) WSMP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>15X (2 GPUs) vs. sequential (double-precision) WSMP</td>
</tr>
<tr>
<td>[Lucas, HPCCS’11]</td>
<td>symmetric</td>
<td>single</td>
<td>5.91X vs. sequential CPU code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.34X vs. 8-threaded CPU code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3-4X (2 GPUs) vs. CPU code</td>
</tr>
<tr>
<td>[Hogg, Slides 2014]</td>
<td>symmetric</td>
<td>unknown</td>
<td>4.1X vs. CPU code</td>
</tr>
<tr>
<td>[Kim, IPDPSW’13]</td>
<td>unsymmetric</td>
<td>double</td>
<td>15.31X (2 GPUs) vs. sequential CPU code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.69X (2 GPUs) vs. 12-threaded code</td>
</tr>
</tbody>
</table>
Related work

- All these methods use supernodal or multifrontal algorithms which use CUDA BLAS or other dense kernels.
- Circuit matrices are so sparse that BLAS-based methods are usually inefficient.
  - For circuit matrices, $\frac{\text{flops}}{\text{NZ} (L + U - I)}$ is usually less than 200.
Left-looking algorithm

- Left-looking: using left columns to perform numeric update

\[ \vec{a} = \vec{a} - c \times \vec{b} \]

Vector multiplication-and-add (MAD)
Left-looking algorithm

\[ \vec{a} = \vec{a} - c \times \vec{b} \]

- **Read**
- **Write**

Update
Left-looking algorithm

\[ \vec{a} = \vec{a} - c \times \vec{b} \]

Update

read
write
Left-looking algorithm

\[ \vec{a} = \vec{a} - c \times \vec{b} \]

read
write

Update
Algorithm: Left-looking algorithm

1  for k=1:N do
2      x = A(k, :);
3      for j=1:k-1 do
4          x(j+1:N) = x(j+1: N) - x(j)*L(j+1:N, j);
5      end for
6      U(1:k, k) = x(1:k);
7      L(k:N, k) = x(k:N) / U(k, k);
8  end for

\[ \vec{a} = \vec{a} - c \times \vec{b} \]
Outline

- Introduction
- Our Solution
  - Overview
  - Parallelization strategies of LU factorization
  - Workflow
  - Optimization
- Results
- Conclusions
Overview

GPU repeats LU factorization without pivoting for circuit simulation iterations
- The nonzero patterns of L and U are fixed
- The pivoting order is also fixed
- One warp computes one column in a SIMD fashion

CPU computes an LU factorization with pivoting ahead of GPU iterations
- Obtain the nonzero patterns of L and U
- Obtain the pivoting order
- Allocate the GPU memory for L and U
Parallelization strategies of LU factorization

- Step 1: construct the task graph (data dependence graph)
  - Column-level data dependence is determined by the nonzero pattern of $U$ in left-looking algorithm
Parallelization strategies of LU factorization

- Step 2: partition the graph into levels
  - Level definition: the longest distance from any source node to each node
  - Tasks nodes in the same level are completely independent!

<table>
<thead>
<tr>
<th>level</th>
<th>Task nodes (columns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 7</td>
</tr>
<tr>
<td>1</td>
<td>4 5 6</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>
Parallelization strategies of LU factorization

- Step 3: compute each level that has many task nodes in parallel
  - cluster mode parallel algorithm
  - Equally assign workloads to warps
  - Level synchronization

<table>
<thead>
<tr>
<th>level</th>
<th>Task nodes (columns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[1 2] [3 7]</td>
</tr>
<tr>
<td>1</td>
<td>[4 5] [6]</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

cluster mode

warp 1
warp 2
Parallelization strategies of LU factorization

- Step 4: compute other levels in parallel in a pipeline-like fashion

  - Example: 2 warps are computing node 8 and 9 simultaneously. **Warp 2 first uses node 6, 4 and 7 to update node 9, but warp 2 must wait until node 8 finishes.**
Parallelization strategies of LU factorization

- Step 4: compute other levels in parallel in a pipeline-like fashion
  - When warp 1 finishes node 8, it begins to compute node 10. Warp 1 first uses node 3 and 8 to update node 10, at the same, warp 2 uses node 8 to update node 9. Warp 1 must wait until node 9 finishes.
Parallelization strategies of LU factorization

- Pipeline mode parallel algorithm

```
1    while not finished
2        get a new column, say k;
3        x = A(:, k);
4    for j<k where U(j, k)!=0, in topological order do
5        wait until column j is finished;
6        x(j+1:N) = x(j+1:N) - x(j)*L(j+1:N, j);
7    end for
8        U(1:k, k) = x(1:k);
9        L(k:N, k) = x(k:N) / U(k, k);
10   end while
```
Workflow

A hybrid CPU-GPU workflow

Pre-processing on CPU;
A factorization with pivoting on CPU;
Sort the nonzeros in L and U on CPU;
Write locations of nonzeros in A, L and U to GPU;
Write values of nonzeros in A to GPU;
for each level in cluster mode do

  GPU kernel: factorize all columns in this level
  using cluster mode algorithm on GPU;
end for

GPU kernel: factorize the rest of columns using
pipeline mode algorithm on GPU;
Read values of nonzeros in L and U from GPU;
Solve the triangular equations on CPU;

CPU execution/scheduling

GPU execution
Optimization

Memory access coalescing

- Sort the nonzeros in A, L and U by their row indices to improve the data locality
- > 2X performance gain
Outline

- Introduction
- Our Solution
- Results
  - GPU performance
  - Comparison with KLU
  - Comparison with PARDISO
- Conclusions
Results

- Experimental environment
  - Host: Intel Xeon E5-2690 & 128GB main memory
  - Device: NVIDIA Tesla K40 (ECC off)
  - KLU: a sparse solver specially designed for circuit simulation problems, has only sequential version
  - PARDISO: a parallel sparse solver for generic sparse matrices, from Intel MKL, uses BLAS and LAPACK
  - Benchmarks are from University of Florida Sparse Matrix Collection
Results

- K40 peak performance: 1.43 Tflop/s (double-precision)
- Sparse LU factorization is a memory-intensive problem

![Graph showing performance comparison]
Results

- Global memory bandwidth utilization
  - K40 peak bandwidth (ECC off): 288 GB/s
Results

Average speedup vs. KLU: 8.4X (3.0X in geometric-mean)

- GPU time: LU factorization + substitution + data transfer
- CPU time: LU factorization + substitution
Results

- Average speedup vs. KLU: 9.4X (3.6X in geometric-mean)
  - GPU time: LU factorization
  - CPU time: LU factorization
Results

- Average speedup vs. PARDISO (T=1): 10.6X (4.2 in geometric-mean)
  - GPU time: LU factorization + substitution + data transfer
  - CPU time: LU factorization + substitution
Results

Average speedup vs. PARDISO (T=8): 2.7X (1.3X in geometric-mean)

- GPU time: LU factorization + substitution + data transfer
- CPU time: LU factorization + substitution
Outline

- Introduction
- Our Solution
- Results
- Conclusions
Conclusions

- GPU-accelerated sparse LU factorization for circuit simulation problems
  - Nonzero patterns of L and U are obtained by CPU
  - GPU repeats LU factorizations with the same nonzero pattern but different element values
  - Hybrid two-mode scheduling to fit different data dependence
  - No dense kernels used

- Performance
  - 8.4X vs. KLU
  - 10.6X vs. PARDISO (T=1)
  - 2.7X vs. PARDISO (T=8)
Publications


Acknowledgement
We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU
Thanks for your attention!
Questions?

Visit [http://nicslu.weebly.com](http://nicslu.weebly.com) for more information about our work on sparse LU factorization for circuit simulation.