A Sparse Direct Solver for GPUs

Jonathan Hogg,
Evgueni Ovtchinnikov,
Jennifer Scott*

STFC Rutherford Appleton Laboratory

12 February 2014
Oxford e-Research Centre
Many-core Seminar

* Thanks also to Jeremy Appleyard of NVIDIA
Aims

Sparse $Ax = b.$

Fast.

Direct methods  Factorize matrix $A = LU$ then triangular solves.

- MATLAB backslash easy.
- Black box - works 99.999% of the time
- GPU libraries: few/none

Iterative methods  CG and friends.

- Expertise required to pick correct method
- Often requires preconditioning
- Doesn’t work for all matrices
- GPU libraries: many
Factorization

Factorize as:

\[ A = L D L^T \]

- Sparse
- Symmetric: \( A = A^T \)
- Non-singular (for simplicity!)
Modern direct solver design

**Four phases**

**Ordering** Find fill-reducing permutation

**Analyse** Find dense submatrix structure.
Setup data representation.

**Factor** Perform factorization with pivoting.

**Solve** Use factorization to solve $Ax = b$. 

GPU Challenges

▶ Thousands of small dense subproblems (e.g. $8 \times 1$)
▶ Pivoting on large dense subproblems (e.g. $4000 \times 2000$)
▶ Substantial sparse scatter/gather
▶ Complicated kernels (register pressure)
Modern direct solver design

Four phases

Ordering  Find fill-reducing permutation

Analyse  Find dense submatrix structure.
           Setup data representation.

Factor  Perform factorization with pivoting.

Solve  Use factorization to solve \( Ax = b \).

GPU Challenges

- Thousands of small dense subproblems (e.g. \( 8 \times 1 \))
- Pivoting on large dense subproblems (e.g. \( 4000 \times 2000 \))
- Substantial sparse scatter/gather
- Complicated kernels (register pressure)
Previous work

Pre-existing work

- A few codes go beyond this.
  None publicly available?
  No pivoting: potentially unstable
  Fairly modest speedups: CPU ↔ GPU bottleneck

Our implementation

- Puts entire factorization and solve phases on GPU
- Open source, including all auxiliary codes
- Delivers over $5 \times$ speedup vs 2 CPU sockets on large problems
Tree parallelism

Operations in first two block columns are independent.
Data flow graph called Assembly Tree
Real world assembly tree: PARSEC/SiNa
Node parallelism

For an individual block, in order:

Assemble contributions from children
(sparse gather)

Factor $m \times k$ matrix with threshold pivoting
(partial dense $LDL^T$)

Contribution given by Schur complement
(dgemm)

Each task itself can be parallelized (some better than others!)
First challenge: Exploit both tree and node parallelism

**Note:** CUBLAS only supports multiple BLAS on *same* dimensions. ⇒ Have to write our own routines.

- CPU populates a data structure of tasks
- Assigns an appropriate number of blocks to each task
- Launches a kernel on \( \sum \) blocks
- Costs several registers to do this (can’t use constant cache)
Enforcing task ordering

Need to enforce assembly tree ordering

- Ideally would do so via global memory with single kernel
- Want to support Fermi, insufficient registers
- Use level based approach instead

```
level 3          7
    /          /          /
level 2        3        6        6
        /    /          /
level 1  1  2  4  5
```
Enforcing task ordering

Need to enforce assembly tree ordering

- Ideally would do so via global memory with single kernel
- Want to support Fermi, insufficient registers
- Use level based approach instead

Outstanding Issues

Load balance:

- Disparate node sizes
- Freedom of assignment
Factorization: basics

**Basic Algorithm**

1. Factor \( A_{11} = L_{11} D_1 L_{11}^T \)
2. Divide \( L_{21} = A_{21} L_{11}^{-T} \)
3. Form \( C = L_{21} D_1 L_{21}^T \)

Stability

- All entries in \( L_{21} \) < \( u^{-1} \)
- Entries of \( D_1 \) calculated in stable fashion
- Typically \( u = 0.01 \).
Factorization: basics

**Basic Algorithm**

1. Factor $A_{11} = L_{11}D_1L_{11}^T$
2. Divide $L_{21} = A_{21}L_{11}^{-T}$
3. Form $C = L_{21}D_1L_{21}^T$

**Stability**

- **All** entries in $L_{21} < u^{-1}$
- Entries of $D_1$ calculated in stable fashion

Typically $u = 0.01$. 
Factorization: parallel pivoting I

**Traditional algorithm**

- Work column by column
- Bring column up-to-date
- Find maximum element $\alpha$ in column of $A_{21}$
- Pivot test $\alpha/a_{11} < u^{-1}$. Accept/reject pivot
Factorization: parallel pivoting 1

**Traditional algorithm**
- Work column by column
- Bring column up-to-date
- Find maximum element \( \alpha \) in column of \( A_{21} \)
- Pivot test \( \alpha/a_{11} < u^{-1} \). Accept/reject pivot

**Problems**
- Very stop-start (one column at a time)
- All-to-all communication for every column
Sparse Direct Solver for GPUs

Hogg, Ovtchinnikov and Scott

Size distributions

- Wide range of sizes
- Often $m \gg k$
Solution

- Try-it-and-see pivoting (\textit{a posteriori pivoting})

New algorithm

- Work by blocks of $L_{21}$
- Every block factorizes copy of $A_{11}$
- Every block checks $\max |l_{21}| < u^{-1}$
- All-to-all communication when all blocks are done
- Discard columns that have failed on \textit{any} block

We use a block size of $32 \times 8$. 
Implementation Issues

- Inefficient if lots of rejected pivots
- Still quite stop-start
- High register pressure (especially on Fermi)
Factorization: parallel pivoting III

Implementation Issues

- Inefficient if lots of rejected pivots
- Still quite stop-start
- High register pressure (especially on Fermi)

Future work

- Implement Subset pivoting or other CA technique as fall back
- Move to DAG-based implementation (Kepler only) (Significant performance improvement expected)
Assembly: Sparse gather/scatter

Can be framed as *either* sparse gather *or* sparse scatter.

- Need to enforce ordering: prefer sparse gather
- Launch one kernel per child
  (i.e. all first children, then all second, ...)
Auxiliary codes

Many auxiliary routines are required that are still CPU-based:

- Ordering (Nested Dissection)
- Analyse (Assorted Graph Algorithms)
- Scaling (MC64 or SpMv)

... but only run once for a sequence of problems

Auction-based scaling: alternative to MC64

For some problems, serial MC64 scaling takes $> 75\%$ of time

- 95\% of the quality
- 10\% of the time
- Parallelizable
Results

Comparison

- C2050 GPU (Fermi) [515GFlops, 238 TDP]
- $2 \times$ Xeon E5620 = 8 cores (Westmere-EP) [76.8GFlops, 160W TDP]
- Flops ratio about $7 \times$

Test Problems

- $4 \times$ Optimization (IPM)
- $4 \times$ Finite Element
- $4 \times$ Finite Difference
# Times(s) and Speedup: Factor+Solve

<table>
<thead>
<tr>
<th>Problem</th>
<th>CPU</th>
<th>GPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>GHS_indef/c-72</td>
<td>0.48</td>
<td>0.35</td>
<td>1.37</td>
</tr>
<tr>
<td>GHS_indef/c-71</td>
<td>2.98</td>
<td>0.64</td>
<td>4.66</td>
</tr>
<tr>
<td>GHS_indef/ncvxqp3</td>
<td>10.65</td>
<td>2.03</td>
<td>5.25</td>
</tr>
<tr>
<td>Schenk_IBMNA/c-big</td>
<td>12.37</td>
<td>2.64</td>
<td>4.69</td>
</tr>
<tr>
<td>Nasa/nasasrb</td>
<td>0.88</td>
<td>0.17</td>
<td>5.18</td>
</tr>
<tr>
<td>DNVS/shipsec1</td>
<td>4.18</td>
<td>0.90</td>
<td>4.64</td>
</tr>
<tr>
<td>GHS_psdef/bmwcra_1</td>
<td>4.45</td>
<td>0.93</td>
<td>4.78</td>
</tr>
<tr>
<td>DNVS/ship_003</td>
<td>9.52</td>
<td>2.16</td>
<td>4.41</td>
</tr>
<tr>
<td>McRae/ecology1</td>
<td>1.64</td>
<td>0.94</td>
<td>1.75</td>
</tr>
<tr>
<td>AMD/G3_circuit</td>
<td>4.54</td>
<td>2.13</td>
<td>2.13</td>
</tr>
<tr>
<td>GHS_psdef/apache2</td>
<td>11.50</td>
<td>2.64</td>
<td>4.36</td>
</tr>
<tr>
<td>Lin/Lin</td>
<td>17.89</td>
<td>2.97</td>
<td>6.02</td>
</tr>
</tbody>
</table>
## Code hot-spots

<table>
<thead>
<tr>
<th></th>
<th>c-72</th>
<th>c-big</th>
<th>shipsec1</th>
<th>Lin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.37</td>
<td>4.69</td>
<td>2.33</td>
<td>6.02</td>
</tr>
<tr>
<td>Contrib</td>
<td>19</td>
<td>780</td>
<td>1607</td>
<td>1568</td>
</tr>
<tr>
<td>Assembly</td>
<td>27</td>
<td>446</td>
<td>38</td>
<td>302</td>
</tr>
<tr>
<td>Factor</td>
<td>82</td>
<td>481</td>
<td>850</td>
<td>666</td>
</tr>
<tr>
<td>Waiting</td>
<td>143</td>
<td>525</td>
<td>405</td>
<td>352</td>
</tr>
</tbody>
</table>

Times are in ms.
Waiting = time not in kernels.
Factor is poor
Conclusions and Future Work

**Story so far**
- New open source sparse direct solver in CUDA
  - Will be released with a little more tidying
- Speedups over host of around 5 on large problems
- Needed to both:
  - Handle peculiarities of device
  - Use new algorithms for massive parallelism

**Near Future**
- Multi-GPU

**Long-term**
- DAG-based factor
- GPU-based scaling
- Auto-generation from stencil?
Thanks for listening!

Questions?
A Supplementary slide

Some supplementary text.
(Note numbering of supplementary slides is outside that of normal slides.)