Solving Sparse Problems on GPUs

Michael Garland
NVIDIA Research
Heterogeneous Parallel Computing

**Multicore CPU**
Fast Serial Processing

**Manycore GPU**
Scalable Parallel Processing
GPUs are mainstream
GPUs are power efficient
Nov. 2010 Top 500 List

Performance vs. Power

- **Tianhe-1A**: GPU-CPU Supercomputer
- **Jaguar**: CPU only Supercomputer
- **Nebulae**: GPU-CPU Supercomputer
- **Tsubame**: GPU-CPU Supercomputer
- **LBNL**: CPU only Supercomputer

- **Gigaflops**
- **Megawatts**

- **GPU-CPU Supercomputer**
- **CPU only Supercomputer**
- **Power**
GPUs do science

ASUCA Weather Modeling

Blood Flow Simulations

Himeno: Navier Stokes

3990 Tesla GPUs

4000 Tesla GPUs

1024 Tesla GPUs

76.1 Tflops

600 Tflops

7.9 Tflops

Large-scale GPU simulations on Tsubame

© 2011 NVIDIA Corporation
What about problems that are

- Sparse
- Irregular
- Unstructured
with Duane Merrill, University of Virginia

BREADTH-FIRST SEARCH
Breadth-First Search on Graphs

- Pick a source node.
- Rank every vertex by the length of shortest path from source.
- Or label every vertex by predecessor in this traversal order.
Arbitrary locality of reference

3D 7pt Poisson Lattice

Wikipedia, 2007

Sparsity pattern (adjacency matrix)
BFS Algorithm: Quadratic vs. Linear

**Quadratic**
- Inspect every vertex at every BFS iteration to see if it was visited during the previous iteration
- $O(|V|^2 + |E|)$ work
- “Stencil-oriented”
  - Trivially matches GPU machine model
- **Unsuitable for diameter $> O(5)$**

**Linear**
- Each BFS iteration only inspects neighbors of vertices newly-discovered during the last iteration
- $O(|V| + |E|)$ work
- “Allocation-oriented”
  - Considered ill-suited for GPU machine model

3D Poisson Lattice (300³ vertices)
Non-uniform & dynamic workloads

Logical vertex frontier
- The unique vertices discovered during the current iteration

Logical edge frontier
- The neighbors of the previous iteration’s vertex frontier
Substantial dataset variety

- **Wikipedia** (social)
- **3D Poisson grid** (cubic lattice)
- **R-MAT** (random, power-law, small-world)
- **Street map: Europe** (Euclidian space)
- **PDE-constrained optimization** (non-linear KKT)
- **Auto transmission manifold** (tetrahedral mesh)
Implementation issues

- Expose sufficient parallelism
  - globally distribute expansion/contraction in DRAM queue

- Load imbalance between processing elements
  - edge-oriented rather than vertex-oriented algorithm
  - “vectorize” tasks across threads

- Bandwidth inefficiency due to scattered load/store
  - isolate irregular accesses from rest of computation
  - filter lookups with bitmask to better leverage cache
# Experimental corpus

<table>
<thead>
<tr>
<th>Graph</th>
<th>Type</th>
<th>Vertices (millions)</th>
<th>Edges (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>europe.osm</td>
<td>Road network</td>
<td>50.9</td>
<td>108.1</td>
</tr>
<tr>
<td>grid5pt.5000</td>
<td>2D Poisson stencil</td>
<td>25.0</td>
<td>125.0</td>
</tr>
<tr>
<td>hugebubbles-00020</td>
<td>2D mesh</td>
<td>21.2</td>
<td>63.6</td>
</tr>
<tr>
<td>grid7pt.300</td>
<td>3D Poisson stencil</td>
<td>27.0</td>
<td>188.5</td>
</tr>
<tr>
<td>nlpkkt160</td>
<td>Constrained optimization problem</td>
<td>8.3</td>
<td>221.2</td>
</tr>
<tr>
<td>audikw1</td>
<td>Finite element matrix</td>
<td>0.9</td>
<td>76.7</td>
</tr>
<tr>
<td>cage15</td>
<td>Transition prob. matrix</td>
<td>5.2</td>
<td>94.0</td>
</tr>
<tr>
<td>kkt_power</td>
<td>Optimization (KKT)</td>
<td>2.1</td>
<td>13.0</td>
</tr>
<tr>
<td>coPapersCiteseer</td>
<td>Citation network</td>
<td>0.4</td>
<td>32.1</td>
</tr>
<tr>
<td>wikipedia-20070206</td>
<td>Wikipedia page links</td>
<td>3.6</td>
<td>45.0</td>
</tr>
<tr>
<td>kron_g500-logn20</td>
<td>Graph500 random graph</td>
<td>1.0</td>
<td>100.7</td>
</tr>
<tr>
<td>random.2Mv.128Me</td>
<td>Uniform random graph</td>
<td>2.0</td>
<td>128.0</td>
</tr>
<tr>
<td>rmat.2Mv.128Me</td>
<td>RMAT random graph</td>
<td>2.0</td>
<td>128.0</td>
</tr>
</tbody>
</table>
## Single-socket performance comparison

<table>
<thead>
<tr>
<th>Graph</th>
<th>Spy Plot</th>
<th>Avg. Search Depth</th>
<th>Sequential Intel</th>
<th>Sandybridge†</th>
<th>Parallel NVIDIA Tesla C2050</th>
<th>Parallel Intel Nehalem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>10⁹ TE/s</td>
<td>10⁹ TE/s</td>
<td>Parallel speedup</td>
<td></td>
</tr>
<tr>
<td><strong>Europe.osm</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>19314</td>
<td>0.03</td>
<td>0.3</td>
<td>11x</td>
<td>0.12 (4-core††)</td>
</tr>
<tr>
<td><strong>grid5pt.5000</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>7500</td>
<td>0.08</td>
<td>0.6</td>
<td>7.3x</td>
<td>0.47 (4-core††)</td>
</tr>
<tr>
<td><strong>hugebubbles</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>6151</td>
<td>0.03</td>
<td>0.4</td>
<td>15x</td>
<td>0.23 (4-core††)</td>
</tr>
<tr>
<td><strong>grid7pt.300</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>679</td>
<td>0.04</td>
<td>1.1</td>
<td>28x</td>
<td>0.11 (4-core††)</td>
</tr>
<tr>
<td><strong>nlpkk160</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>142</td>
<td>0.26</td>
<td>2.5</td>
<td>10x</td>
<td>0.19 (4-core††)</td>
</tr>
<tr>
<td><strong>audikw1</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>62</td>
<td>0.65</td>
<td>3.0</td>
<td>4.6x</td>
<td></td>
</tr>
<tr>
<td><strong>cage15</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>37</td>
<td>0.13</td>
<td>2.2</td>
<td>18x</td>
<td></td>
</tr>
<tr>
<td><strong>kkt_power</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>37</td>
<td>0.05</td>
<td>1.1</td>
<td>23x</td>
<td></td>
</tr>
<tr>
<td><strong>coPapersCite</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>26</td>
<td>0.50</td>
<td>3.0</td>
<td>5.9x</td>
<td></td>
</tr>
<tr>
<td><strong>wikipedia-2007</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>20</td>
<td>0.07</td>
<td>1.6</td>
<td>25x</td>
<td>0.50 (8-core†††)</td>
</tr>
<tr>
<td><strong>kron_g500-logn20</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>6</td>
<td>0.24</td>
<td>3.1</td>
<td>13x</td>
<td></td>
</tr>
<tr>
<td><strong>random.2Mv.128Me</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>6</td>
<td>0.10</td>
<td>3.0</td>
<td>29x</td>
<td>0.70 (8-core†††)</td>
</tr>
<tr>
<td><strong>rmat.2Mv.128Me</strong></td>
<td><img src="image" alt="Spy Plot" /></td>
<td>6</td>
<td>0.15</td>
<td>3.3</td>
<td>22x</td>
<td></td>
</tr>
</tbody>
</table>

† 3.4GHz Core i7 2600K
†† 2.5 GHz Core i7 4-core, Leiserson et al.
††† 2.7 GHz Xeon X5570 8-core, Agarwal et al.
ALGEBRAIC MULTIGRID

Nathan Bell (NVIDIA), Steven Dalton & Luke Olson (UIUC)
Objective

- Solve (certain) sparse linear systems *very quickly*
- Optimal complexity $O(N)$

$$A \times x = b$$
Multigrid in a Nutshell

Setup Phase
- Construct a hierarchy of grids
- Sequence of coarser versions of the problem

Cycling/Solve Phase
- Reduce high-frequency errors with relaxation
- Restrict remaining low-frequency errors to coarse grid
- Interpolate coarse-grid solution back to fine grid

Applied recursively
- Eliminates all error modes
- Results in V-cycle
- Converges to solution in multiple cycles
Algebraic Multigrid (AMG)

- Constructs “grids” directly from sparse matrix
  - Hierarchy of sparse matrices
  - Requires no geometric knowledge

- Aggregation-based AMG
  - Coarsens clusters of nodes
  - Works for unstructured meshes
AMG on the GPU

- Implemented both Setup and Cycling on GPU
  - Need to identify *fine-grained* parallelism everywhere
  - Distill complex algorithms into *parallel primitives*

**Setup Phase**
- Sparse matrix-matrix multiplication \((C = A \times B)\)
- Sparse matrix transpose
- Matrix format conversions, parallel aggregation, etc.

**Cycling Phase**
- Sparse matrix-vector multiplication \((y = A \times x)\)
- Level 1 BLAS operations (e.g. dot product)
Example: Transpose

- Matrix in coordinate format (COO)
- Sort rows and values by column index
- Implemented with `thrust::sort_by_key`

<table>
<thead>
<tr>
<th>Row indices</th>
<th>Column indices</th>
<th>Nonzero values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 2 4 5</td>
<td>0 1 0 2 0 1</td>
<td>A B C D E F</td>
</tr>
</tbody>
</table>

© 2011 NVIDIA Corporation
Example: Aggregation

- Compute MIS(2) in parallel with extension of Luby’s method
- Create aggregates around each MIS(2) node
- Close analog of standard greedy aggregation scheme
Performance Study

- Eight matrix example
  - Isotropic Poisson Problems
  - Structured and Unstructured

- GPU System
  - Tesla C2050 GPU
  - CUDA 4.0
  - Thrust v1.4
  - Cusp v0.2

- CPU System
  - Intel Core i7 950 CPU
  - MKL v10.3

- Reference Solver
  - Trilinos/ML v5.0

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Notes</th>
<th>Rows</th>
<th>Nonzeros</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>2D FD, 5-point</td>
<td>~1M</td>
<td>~5M</td>
</tr>
<tr>
<td>1b</td>
<td>2D FE, 9-point</td>
<td>~1M</td>
<td>~9M</td>
</tr>
<tr>
<td>2a</td>
<td>3D FD, 7-point</td>
<td>~1M</td>
<td>~7M</td>
</tr>
<tr>
<td>2b</td>
<td>3D FE, 27-point</td>
<td>~1M</td>
<td>~27M</td>
</tr>
<tr>
<td>3a</td>
<td>2D FE, h=0.03</td>
<td>~500K</td>
<td>~4M</td>
</tr>
<tr>
<td>3b</td>
<td>2D FE, h=0.02</td>
<td>~1M</td>
<td>~8M</td>
</tr>
<tr>
<td>3c</td>
<td>2D FE, h=0.015</td>
<td>~2M</td>
<td>~15M</td>
</tr>
<tr>
<td>4</td>
<td>3D FE, h=0.15</td>
<td>~1M</td>
<td>~17M</td>
</tr>
</tbody>
</table>
## Individual Components

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot Product</td>
<td>6.62x</td>
</tr>
<tr>
<td>Vector Addition</td>
<td>6.35x</td>
</tr>
<tr>
<td>Sparse Transpose*</td>
<td>2.90x</td>
</tr>
<tr>
<td>Sparse Matrix-Vector Multiply*</td>
<td>6.00x</td>
</tr>
<tr>
<td>Sparse Matrix-Matrix Multiply*</td>
<td>1.67x</td>
</tr>
</tbody>
</table>

*Average speedup across eight example matrices*
## Setup Phase

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Type</th>
<th>CPU</th>
<th>GPU</th>
<th>Speedup</th>
<th>Trilinos/ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>2D FD</td>
<td>892 ms</td>
<td>518 ms</td>
<td>1.72x</td>
<td>2040 ms</td>
</tr>
<tr>
<td>1b</td>
<td>2D FE</td>
<td>1133 ms</td>
<td>649 ms</td>
<td>1.75x</td>
<td>2298 ms</td>
</tr>
<tr>
<td>2a</td>
<td>3D FD</td>
<td>1639 ms</td>
<td>944 ms</td>
<td>1.74x</td>
<td>2906 ms</td>
</tr>
<tr>
<td>2b</td>
<td>3D FE</td>
<td>2845 ms</td>
<td>2124 ms</td>
<td>1.34x</td>
<td>4420 ms</td>
</tr>
<tr>
<td>3a</td>
<td>2D FE</td>
<td>657 ms</td>
<td>335 ms</td>
<td>1.96x</td>
<td>1324 ms</td>
</tr>
<tr>
<td>3b</td>
<td>2D FE</td>
<td>1484 ms</td>
<td>648 ms</td>
<td>2.29x</td>
<td>2785 ms</td>
</tr>
<tr>
<td>3c</td>
<td>2D FE</td>
<td>2901 ms</td>
<td>1151 ms</td>
<td>2.52x</td>
<td>5236 ms</td>
</tr>
<tr>
<td>4</td>
<td>3D FE</td>
<td>3157 ms</td>
<td>1726 ms</td>
<td>1.83x</td>
<td>4967 ms</td>
</tr>
</tbody>
</table>
## Cycling Phase

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Type</th>
<th>CPU</th>
<th>GPU</th>
<th>Speedup</th>
<th>Trilinos/ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>2D FD</td>
<td>1221 ms (20)</td>
<td>423 ms (51)</td>
<td>7.66x</td>
<td>14,190 ms (33)</td>
</tr>
<tr>
<td>1b</td>
<td>2D FE</td>
<td>1097 ms (16)</td>
<td>461 ms (46)</td>
<td>7.52x</td>
<td>10,590 ms (22)</td>
</tr>
<tr>
<td>2a</td>
<td>3D FD</td>
<td>1760 ms (23)</td>
<td>295 ms (27)</td>
<td>6.76x</td>
<td>14,800 ms (31)</td>
</tr>
<tr>
<td>2b</td>
<td>3D FE</td>
<td>1683 ms (14)</td>
<td>482 ms (24)</td>
<td>5.98x</td>
<td>13,840 ms (20)</td>
</tr>
<tr>
<td>3a</td>
<td>2D FE</td>
<td>1534 ms (42)</td>
<td>286 ms (49)</td>
<td>5.40x</td>
<td>14,020 ms (53)</td>
</tr>
<tr>
<td>3b</td>
<td>2D FE</td>
<td>3704 ms (47)</td>
<td>633 ms (54)</td>
<td>5.77x</td>
<td>34,410 ms (68)</td>
</tr>
<tr>
<td>3c</td>
<td>2D FE</td>
<td>7804 ms (53)</td>
<td>1424 ms (65)</td>
<td>5.75x</td>
<td>44,530 ms (65)</td>
</tr>
<tr>
<td>4</td>
<td>3D FE</td>
<td>4369 ms (43)</td>
<td>1498 ms (50)</td>
<td>2.96x</td>
<td>28,380 ms (47)</td>
</tr>
</tbody>
</table>

**Notes**
- Time to solve $Ax=b$ to $1e-10$ relative tolerance
- AMG as a preconditioner to CG solver
- Iteration counts shown in parentheses
- Reporting per-iteration speedup
Summary

Fully-Parallelized AMG on GPU

- 1.89x Speedup in Setup Phase (average)
- 5.89x Speedup in Cycling Phase (average)

References

"Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods"
Nathan Bell (NVIDIA), Steven Dalton (UIUC), Luke Olson (UIUC),
http://research.nvidia.com/publications

Cusp Library
http://cusp-library.googlecode.com

Thrust Library
http://thrust.googlecode.com
Final Thoughts

- Sparse problems can be solved by
  - exposing sufficient fine-grained parallelism
  - managing load imbalance
  - efficiently using available bandwidth

- Details available in:
  - technical reports
  - open source implementations
  - see [http://research.nvidia.com](http://research.nvidia.com)
Questions?

mgarland@nvidia.com

http://research.nvidia.com