Thinking Parallel:
Sparse Iterative Solvers with CUDA

Jonathan Cohen
NVIDIA Research
Roadmap

This is a “roadmap” talk – some work in progress, but mostly motivational

Sparse linear solvers matter
  => This means parallel preconditioners

Methods should reflect hardware trends
  => This means multilevel & DD methods

(Interesting other approaches to try)
Sparse Iterative Solvers
Sparse iterative solvers

Solve $Ax=b$ where $A$ is
- Huge – up to billions of unknowns
- Sparse – $A$ is mostly zeroes
- $A$ often has structure due to the physics of the problem
- $A$ often symmetric/Hermitian and/or positive definite
- Avoid forming $A^{-1}$, which is usually much denser than $A$

Why not direct solvers?
- Direct solvers work well up to certain sizes, beyond that iterative solvers needed
- Direct methods are very complex
- Sparse methods only require ability to compute ‘$Ax$’
Example – Discrete Laplacian

\[-\Delta u = R\]

=> \( A_{21} u_1 + A_{22} u_2 + A_{23} u_3 + A_{24} u_4 = R_1 \)
Iterative Solvers

Solve \( A \ u = r \), where \( A \) and \( r \) are known

Error is easy to estimate: \( E = A \ u' - r \)

Basic iterative scheme:

Start with a guess for \( u \), call it \( u' \)

Until \( |A \ u' - r| < \) tolerance

\( u' \leq \) Update\( (u', A, r) \)

Return \( u' \)

Update() is the key
Krylov Methods

- Update() involves searching the Krylov space:
  \[ \text{span}\{x, Ax, A^2x, A^3x, A^4x, \ldots\} \]

- Lots of variants: CG, BiCGStab, GMRES, CGNR
  - Some only work for A symmetric, some require more or less intermediate storage, etc.

- Require ability to form product Ax, dot products, basic BLAS operations
  - Communication minimizing – fewer global dot products

- Convergence depends on spectral properties of A
Preconditioning

Idea: improve spectral properties of $A$ without modifying the Krylov method

Choose “preconditioning matrix” $M^{-1}$

Solve

$$M^{-1}A\ u = M^{-1}r$$

In Krylov methods, only need to compute $M^{-1}x$, not represent $M^{-1}$ explicitly

Art: $M^{-1}$ easy to compute, $M^{-1}A$ has better spectrum

- If $M^{-1} = A^{-1}$, problem becomes trivial, but forming $M^{-1}u$ as hard as solving $Au=r$
- If $M^{-1} = I$, forming $M^{-1}u$ is trivial, but spectrum unchanged
Preconditioning

Where the action is at

“Good sparse iterative solver” = “Good preconditioner”

Often preconditioners look like “approximate solvers” to the original equation

- Based on physics/geometry of underlying problem
- Based on analysis of algebraic structure of A
- Based on generic procedure ("black box")
- Based on “approximate factorization” of A

No preconditioner works for all problems

Community needs many options (see, e.g., PETsc)
Manifesto

- Develop a library of good parallel CUDA-accelerated preconditioners
- Interface to existing libraries (PETsc, Trilinos)
- Range of solvers
  - black-box solvers
  - building blocks for problem-specific solvers
  - “semi-black box” solvers (user optionally provides some problem-specific information)

Open source everything:
http://code.google.com/p/cusp-library/
Parallel Preconditioning: Why so Hard?

Good preconditioners “couple” many dimensions together efficiently

- Krylov methods find point along “search direction” that minimizes error
- good preconditioner minimizes error along many directions in a single step
- “direction” = vector element
- “minimize along many directions” => communication between multiple elements
- “coupling together” at odds with “run in parallel”
Other Preconditioning Approaches

- Various parallel smoothers
  - Graph coloring => multicolored Gauss-Seidel
  - Polynomial / Jacobi / SOR

- Approximate Inverse methods
  - AINV – serial (slow!) factorization, good performance
  - SPAI – William Sawyer et al. working on this

- Incomplete Factorizations / Support Tree
  - (parallel) sparse direct solver on subset (e.g. Vaidya)
  - Incomplete Cholesky + (parallel) sparse triangular solve

- Algebraic Multigrid
  - Lots of options, not clear what’s best approach in CUDA
  - Nathan Bell & collaborators at UIUC working on SA
Diagonal (Jacobi) Preconditioning

- Maximum parallelism, minimal coupling
- $M^{-1} = \text{Diag}(A)^{-1} = \text{Diag}(1/A_{11}, 1/A_{22}, \ldots, 1/A_{nn})$
- Asymptotically bad, but hard to beat for easy problems on GPU

```c
__global__ void diagonal_precond(float *u, float *Aii, int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    if (i < n) {
        u[i] /= Aii[i];
    }
}
```
Domain Decomposition

Idea:
- Split variables into $m$ domains ($m \ll n$)
- Solve each domain independently
- Combine per-domain solutions together somehow

$n=7$
$m=2$
\[
\begin{pmatrix}
A_{11} & A_{12} & A_{15} \\
A_{21} & A_{22} & A_{23} \\
A_{32} & A_{33} & \\
A_{51} & A_{55} & \\
\end{pmatrix}
\begin{pmatrix}
u_1 \\
u_2 \\
u_3 \\
u_5 \\
\end{pmatrix}
=
\begin{pmatrix}
R_1 \\
R_2 \\
R_3 \\
R_5 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
A_{44} & A_{46} \\
A_{64} & A_{66} & A_{67} \\
A_{76} & A_{77} \\
\end{pmatrix}
\begin{pmatrix}
u_4 \\
u_6 \\
u_7 \\
\end{pmatrix}
=
\begin{pmatrix}
R_4 \\
R_6 \\
R_7 \\
\end{pmatrix}
\]
Block-Jacobi Preconditioning

\[
M^{-1} = \begin{pmatrix}
A_{\{1\}}^{-1} & 0 & \cdots & 0 \\
0 & A_{\{2\}}^{-1} & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & A_{\{m\}}^{-1}
\end{pmatrix} = \begin{pmatrix}
A_{\{1\}}^{-1} \\
A_{\{2\}}^{-1} \\
\vdots \\
A_{\{m\}}^{-1}
\end{pmatrix}
\]

- When the blocks have size 1, this is diagonal
- Parallelism vs. numerical benefit trade-off
  - Large blocks = less parallelism, better convergence
  - Small blocks = more parallelism, worse convergence
Digression: Architectural Trends
Bottom Line: Power

Report published September 28, 2008:

- Four Major Challenges*
  - Energy and Power challenge
  - Memory and Storage challenge
  - Concurrency and Locality challenge
  - Resiliency challenge

- Number one issue is **power**
  - Extrapolations of current architectures and technology indicate over 100MW for an Exaflop!
  - Power also constrains what we can put on a chip

Available at
www.darpa.mil/ipto/personnel/docs/ExaScale_Study_Initial.pdf
In the past we had constant-field scaling

\[
\begin{align*}
L' &= \frac{L}{2} & \text{(feature length)} \\
V' &= \frac{V}{2} & \text{(voltage)} \\
E' &= \frac{1}{2}CV^2 = \frac{E}{8} & \text{(capacitance \sim L)} \\
f' &= 2f & \text{(frequency \sim 1/L)} \\
A' &= L^2 = \frac{A}{4} & \text{(area)} \\
P' &= P & \text{(power/area = Ef/A)} \\
f'/A' &= 8 \frac{f}{A} & \text{(ops/s/area)}
\end{align*}
\]

Halve L and get 8x the op rate for the same power in a fixed area
Now voltage is held nearly constant

\[
\begin{align*}
L' &= \frac{L}{2} \quad \text{(feature length)} \\
V' &= V \quad \text{(voltage)} \\
E' &= \frac{1}{2}CV^2 = \frac{E}{2} \quad \text{(capacitance } \sim L) \\
f' &= 2f \quad \text{(frequency } \sim 1/L) \\
A' &= \frac{L^2}{4} = \frac{A}{4} \quad \text{(area)} \\
P' &= 4P \quad \text{(power/area } = Ef/A) \\
f'/A' &= 8 \frac{f}{A} \quad \text{(ops/s/area)}
\end{align*}
\]

Halve \(L\) and get 8x the op rate for 4x the power in a fixed area
4,000 64b FPUs fit on a chip

64b FPU
0.1mm²
50pJ/op
1.5GHz
Moving a word across die = 10FMAs
Moving a word off chip = 20FMAs

64b FPU
0.1mm²
50pJ/op
1.5GHz

64b 1mm
Channel
25pJ/word

64b Off-Chip
Channel
1nJ/word

64b Floating Point
Minimal length interconnect?
What does this mean?

- Limiting factor will power
- Die area becomes free ("dark silicon")
- FLOPS cost roughly zero, communication is expensive

Communication topologies to minimize wire-length
- Flat communication => all-to-all [Not going to happen]
- => On-chip communication will exploit locality
- => Deeply hierarchical trees
- => Algorithms that exploit locality will be fast.
- => Algorithms that assume flat memory will be slow.
- => This is fundamental to processor design
Architecturally-Informed Preconditioners
Domain Decomposition, Revisited

2 levels are parallelism exhibited:
1) Each block is independent
2) Solving $A_{(i)}^{-1}u_{(i)} = r_{(i)}$ is locally parallel

As much parallelism as diagonal preconditioning
But less global communication
Cost: higher constant, more local communication
Diameter = \( n^{1/2}/m^{1/2} \)

Total cost of preconditioning

Diagonal:
\[ O(n) \text{ entries} \times O(1) \text{ cost per entry} \times O(n^{1/2}) \text{ iterations} = O(n^{3/2}) \]

Block Jacobi:
\[ O(n/m) \text{ domains} \times O(m^{3/2}) \text{ cost per domain} \times O(n^{1/2}/m^{1/2}) \text{ iterations} = O(n^{3/2}) \]
Schwarz Methods

- Block Jacobi is “least common denominator”
- But good model for Schwarz methods
  - Similar computation and communication patterns
- Schwarz methods
  - Many variants: Additive, Optimized, Harmonic Overlap, Restricted, Multiplicative, etc.
  - “Optimized” methods: modify boundary conditions for local problems to transmit information optimally
  - Algebraic or geometric variants for all
  - Smoothers
  - Add coarsening to get multilevel method
Restricted Additive Schwarz (RAS)

- Default (non-symmetric) preconditioner in PETsc
- Read from neighboring domain, only update to own domain => no write conflicts

1. Read
2. Solve
3. Write
RAS in CUDA

Setup

- Input: Matrix A, per-element domain assignment
  - Grow the domain regions to include overlap
  - Build local matrices, store in local data structures
  - Precompute whatever is needed

Per-iteration

- Input: vector r, Output: \(x = M^{-1}r\)
  - Load local matrix and subset of x into shared memory
  - Solve \(A_{(i)}x_{(i)} = r_{(i)}\)
  - For internal entries, write \(x_{(i)}\) back to x
Per-Iteration Solve

- Launch 1 thread block per domain
- For thread block j:
  - Allocate buffers in shared mem
  - Copy $A_{ij}$, $r_{ij}$ from global mem to shared mem
  - In shared mem: $x_{ij} = 0$
  - __syncthreads()
  - Solve $A_{ij}x_{ij} = r_{ij}$ using polynomial smoothing
  - __syncthreads()
- For interior cells i:
  - $X_{\text{global}[\text{local\_to\_global}[i]]} = x_{\text{local}[i]}$
Tesla C2050

Global Memory
4GB

Shared Memory
48KB

Global Mem Aggregate
137 GB/s
(80 GB/s achieved)

Shared Mem Aggregate
960 GB/sec
(357 GB/s achieved)

SM0

10 GB/sec

68 GB/sec

SM1

SM13
RAS As Smoother

- 1024 x 1024 5-pt Poisson matrix
- 24 x 24 domains, overlap=1 (sized to 48k)
- From global: 99 MB
  - Matrix data, rhs, per domain data, + bookkeeping
- From shared: 153 MB
  - Matrix data, rhs, solution, + other vectors

<table>
<thead>
<tr>
<th>Relaxer</th>
<th>GMEM Data</th>
<th>GMEM BW/s</th>
<th>GMEM time</th>
<th>SHMEM Data</th>
<th>SHMEM BW/s</th>
<th>SHMEM time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAS</td>
<td>99MB</td>
<td>80 GB/s</td>
<td>1.21 ms</td>
<td>153 MB</td>
<td>357 GB/s</td>
<td>0.42 ms</td>
</tr>
<tr>
<td>Jacobi</td>
<td>108MB</td>
<td>96 GB/s*</td>
<td>1.10 ms</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

1 RAS with 10 local polysmooth iterations = 5 global Jacobi Iterations

* Assume 70% of peak is achievable
RAS as Preconditioner

Lose on per-iteration, save on (expensive) global cost by reducing iteration count

<table>
<thead>
<tr>
<th></th>
<th>Diagonal</th>
<th>RAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditioner kernel</td>
<td>0.4 ms</td>
<td>14.7 ms</td>
</tr>
<tr>
<td>+ BiCGStab / iteration</td>
<td>5.1 ms</td>
<td>5.1 ms</td>
</tr>
<tr>
<td>Total per iteration</td>
<td>5.9 ms</td>
<td>34.6 ms (5.7x worse)</td>
</tr>
<tr>
<td>Iterations to convergence</td>
<td>1582</td>
<td>257 (6.2x better)</td>
</tr>
<tr>
<td>Total time</td>
<td>9,362.5 ms</td>
<td>8,901.3 ms</td>
</tr>
</tbody>
</table>
### RAS Scaling vs Diagonal Scaling

<table>
<thead>
<tr>
<th>Problem size</th>
<th>RAS Iterations</th>
<th>RAS Time (ms)</th>
<th>RAS ms/iter/n</th>
<th>Diagonal Iterations</th>
<th>Diagonal Time (ms)</th>
<th>Diagonal ms/iter/n</th>
</tr>
</thead>
<tbody>
<tr>
<td>128²</td>
<td>46</td>
<td>43.40911</td>
<td>5.7597E-05</td>
<td>286</td>
<td>261.7252</td>
<td>5.5855E-05</td>
</tr>
<tr>
<td>256²</td>
<td>82</td>
<td>204.0062</td>
<td>3.7962E-05</td>
<td>516</td>
<td>555.764</td>
<td>1.6435E-05</td>
</tr>
<tr>
<td>512²</td>
<td>157</td>
<td>1,480.164</td>
<td>3.5964E-05</td>
<td>914</td>
<td>1,804.13</td>
<td>7.5298E-06</td>
</tr>
<tr>
<td>1024²</td>
<td>257</td>
<td>8,901.323</td>
<td>3.3031E-05</td>
<td>1582</td>
<td>9,362.599</td>
<td>5.644E-06</td>
</tr>
<tr>
<td>2048²</td>
<td>508</td>
<td>69,589.17</td>
<td>3.266E-05</td>
<td>5292</td>
<td>116,161.7</td>
<td>5.2334E-06</td>
</tr>
</tbody>
</table>
RAS - Local Solver Performance

Polynomial vs. Jacobi (PCG > 4x worse)
Hierarchical Domain Decomposition

Map numerical hierarchy to machine hierarchy
Line smoothing

- Similar computational structure as DD
  - Local matrix is tridiagonal
  - Can think of “1d domains”

- Unstructured meshes => ‘path covering’ problem
  - “Optimal” path covering is NP-Complete, probably overkill
  - Parallel path covering algorithm (joint work with Peter Zaspel) in experimental cusp branch
Per-SM Tridiagonal Solver

PCI-E: CPU-GPU data transfer
MT GE: multi-threaded CPU Gaussian Elimination
GEP: CPU Gaussian Elimination with pivoting (from LAPACK)

Applying Line Smoother

Launch 1 thread block per domain line

For thread block j:
- Allocate buffers in shared mem
- Copy $A_{\{j\}}, r_{\{j\}}$ from global mem to shared mem
- In shared mem: $x_{\{j\}} = 0$
- __syncthreads()
- Solve $A_{\{j\}}x_{\{j\}} = r_{\{j\}}$ using polynomial smoothing cyclic reduction
- __syncthreads()

For interior cells $i$: For all cells:
- $X_{\text{global}}[\text{local_to_global}[i]] = x_{\text{local}}[i]$
Line Smoothing Ideas - WIP

**ADI Preconditioning / Smoothing?**
- Provides good coupling across degrees of freedom
- See Göddeke and Strzodka’s work in FEM
- “Algebraic ADI” - choose directions based on local algebraic information?

**Multilevel methods from “path aggregation”?**
- “Path aggregation multigrid” joint work with Peter Zaspel
- Turns out, this is “graph matching MG” scheme (Kim, Xu, & Zikatanov) (hat tip: James Brannick)
- “AMLI” approach: simple aggregation, but use more sophisticated cycling

**Semi-coarsening?**
- Coarse problems should fill GPU better – more efficient?
What’s next?

- Better local solvers – this is a great research topic
  - What’s the fastest local solver on an SM?
- Implement variety of Schwarz variants
  - Optimized RAS variants
  - Combine with cusp’s graph coloring algorithm => multicolored multiplicative Schwarz
- Multilevel
- Line smoothing methods
  - (Algebraic?) semi-coarsening MG from path coverings
  - Graph-matching / AMLI schemes
  - ADI as smoother
- Integrate with PETSc and Trilinos
  - Do this intelligently – adapt to entire system hierarchy
  - CUDA-accelerated graph partitioning?
- Apply to some real problems…
Conclusion

- Relative bandwidths are following clear trend
  - Local = fast
  - Global = slow
- Find numerical methods with matching communication patterns
  - Numerically: more coupling better
  - Architecturally: coupling ok, must be hierarchical
- Building blocks: Local linear solvers, ways of combining them
- Build more sophisticated methods from there
Thanks

http://code.google.com/p/cusp-library/
http://code.google.com/r/jcohenpersonal-precond/

Peter Zaspel, Nathan Bell, Michael Garland, Yao Zhang
Amik St. Cyr, James Brannick, Rio Yokota
Grow Regions

**Input:** vector<int> domain;
- domain[i] = j, means i in domain j

**tuples = { (i, domain[i]) }**

**for each tuple:**
- count[tuple_id] = #{nbr_id with domain[nbr_id] != j}

**offsets = prefix_sum(count, ‘+’) (thrust::scan)**

**for each tuple:**
- for each in {nbr_row with domain[nbr_row] != j}
  - new_tuple[offsets[tuple_id] + nbr_id] = (nbr_row, j)

**tuples = tuples + new_tuples**

Sort lexicographically (brings dup elements together)

Remove duplicates (thrust::unique_copy)