Mixed-Precision GPU-Multigrid Solvers with Strong Smoothers and Applications in CFD and CSM

Dominik Göddeke

Institut für Angewandte Mathematik (LS3)
TU Dortmund
dominik.goeddeke@math.tu-dortmund.de

SIMTECH 2011
International Conference on Simulation Technology
University of Stuttgart, June 16 2011
Motivation

**Hardware isn’t our friend any more**
- Paradigm shift towards parallelism and heterogeneity
  - In a single chip: Multicores, GPUs, . . .
  - In a workstation, cluster node, . . .
  - In a big cluster, supercomputer, . . .
- Data movement cost gets prohibitively expensive
- Technical reason: Power wall + memory wall + ILP wall = brick wall

**Challenges in numerical HPC**
- Existing codes don’t run faster automatically any more
- Compilers can’t solve these problems, libraries are limited
- Traditional numerics is often contrary to these hardware trends
- *We (the numerics people) have to take action*
Hardware-oriented numerics

**Conflicting situations**
- Existing methods no longer hardware-compatible
- Neither want less numerical efficiency, nor less hardware efficiency

**Challenge: New algorithmic way of thinking**
- Balance these conflicting goals

**Consider short-term hardware details in actual implementations, but long-term hardware trends in the design of numerical schemes**
- Locality, locality, locality
- Communication-avoiding (-delaying) algorithms between all flavours of parallelism
- Multilevel methods, hardware-aware preconditioning
Grid and Matrix Structures

Flexibility ↔ Performance
Grid and matrix structures

**General sparse matrices (unstructured grids)**
- CSR (and variants): General data structure for arbitrary grids
- Maximum flexibility, but during SpMV
  - Indirect, irregular memory accesses
  - Index overhead reduces already low arithm. intensity further
- Performance depends on nonzero pattern (grid numbering)

**Structured sparse matrices**
- Example: Structured grids, suitable numbering ⇒ band matrices
- Important: No stencils, fully variable coefficients
- Direct regular memory accesses, fast independent of mesh
- Exploitation in the design of strong MG components
Example: Poisson on unstructured mesh

- Nehalem vs. GT200, ≈ 2M bilinear FE, MG-JAC solver
- Unstructured formats highly numbering-dependent
- Multicore 2–3x over singlecore, GPU 8–12x over multicore
- Banded format (here: 8 ‘blocks’) 2–3x faster than best unstructured layout and predictably on par with multicore
Discretisation and Solver Structures in FEAST

Scalable, Locality-preserving Parallel Multilevel Solvers
Approach in FEAST

**Combination of structured and unstructured advantages**
- Global macro-mesh: Unstructured, flexible, complex domains
- Local micro-meshes: Structured (logical TP-structure), fast
- Important: Structured ≠ simple meshes!

**Solver approach ScaRC exploits data layout**
- Parallel efficiency: Strong and weak scalability
- Numerical scalability: Convergence rates independent of problem size and partitioning (multigrid!)
- Robustness: Mesh and operator anisotropies (strong smoothers!)
**ScaRC: Concepts**

**ScaRC for scalar systems**
- Hybrid multilevel domain decomposition method
- Minimal overlap by extended Dirichlet BCs
- Inspired by parallel MG (‘best of both worlds’)
  - Multiplicative between levels, global coarse grid problem (MG-like)
  - Additive horizontally: block-Jacobi / Schwarz smoother (DD-like)
- Schwarz smoother encapsulates local irregularities
  - Robust and fast multigrid (‘gain a digit’), strong smoothers
  - Maximum exploitation of local structure

---

**global BiCGStab**
preconditioned by
**global multilevel (V 1+1)**
additively smoothed by
for all $\Omega_i$: **local multigrid**
coarse grid solver: UMFPACK
ScaRC for multivariate problems

**Block-structured systems**

- Guiding idea: Tune scalar case once per architecture instead of over and over again per application
- Blocks correspond to scalar subequations, coupling via special preconditioners
- Block-wise treatment enables *multivariate ScaRC solvers*

\[
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\begin{pmatrix}
u_1 \\
u_2
\end{pmatrix}
= f,
\]

\[
\begin{pmatrix}
A_{11} & 0 & B_1 \\
0 & A_{22} & B_2 \\
B_1^T & B_2^T & 0
\end{pmatrix}
\begin{pmatrix}
v_1 \\
v_2 \\
p
\end{pmatrix}
= f,
\]

\[
\begin{pmatrix}
A_{11} & A_{12} & B_1 \\
A_{21} & A_{22} & B_2 \\
B_1^T & B_2^T & C_C
\end{pmatrix}
\begin{pmatrix}
v_1 \\
v_2 \\
p
\end{pmatrix}
= f
\]

$A_{11}$ and $A_{22}$ correspond to scalar (elliptic) operators

$\Rightarrow$ Tuned linear algebra and tuned solvers
Minimal invasive accelerator integration

Bandwidth distribution in a hybrid CPU/GPU node

- GPU
  - SIMT-MP
  - SIMT-MP
  - SIMT-MP
  - SIMT-MP

- CPU-Socket
  - Cores
  - Caches
  - Cores
  - Caches
  - Shared caches and MCs

- Device Memory
  - 1-8 GB/s
  - PCIe

- Host Memory
  - 1-2 GB/s
  - Infiniband (PCIe)

- Bandwidths:
  - 20-180 GB/s
  - 6-35 GB/s
  - 1-2 GB/s
Minimally invasive accelerator integration

**Guiding concept: locality**

- Accelerators: Most time-consuming inner component
- CPUs: Outer MLDD solver (only hardware capable of MPI anyway)
- Block-structured approach inside MPI rank allows double-buffering and PCIe communication overlap
- Employ mixed precision approach

<table>
<thead>
<tr>
<th>global BiCGStab</th>
<th>preconditioned by</th>
</tr>
</thead>
<tbody>
<tr>
<td>global multilevel (V 1+1)</td>
<td>additively smoothed by</td>
</tr>
<tr>
<td>for all $\Omega_i$: local multigrid</td>
<td>coarse grid solver: UMFPACK</td>
</tr>
<tr>
<td>ScaRC: MG/DD Löser</td>
<td>Schwarz Smoother Tasks + Data Scheduler</td>
</tr>
<tr>
<td>GPU Backend CUDA / OpenCL OpenGL</td>
<td></td>
</tr>
<tr>
<td>Heterogeneous Accelerator Hardware</td>
<td></td>
</tr>
</tbody>
</table>
Minimally invasive accelerator integration

Benefits and challenges

- Balance acceleration potential and integration effort
- Accelerate many different applications built on top of one central FE and solver toolkit
- Diverge code paths as late as possible
- Develop on a single GPU and scale out later
- Retain all functionality
- Do not sacrifice accuracy
- No changes to application code!

Challenges

- Heterogeneous task assignment to maximise throughput
- Overlapping CPU and GPU computations with transfers
Strong Smoothers

Parallelising Inherently Sequential Operations
Motivation: Why strong smoothers?

Test case: Generalised Poisson problem with anisotropic diffusion

- $-\nabla \cdot (G \nabla u) = f$ on unit square (one FEAST patch)
- $G = I$: standard Poisson problem, $G \neq I$: arbitrarily challenging
- Example: $G$ introduces anisotropic diffusion along some vector field

Only multigrid with a strong smoother is competitive
Gauß-Seidel smoother

Disclaimer: Not necessarily a good smoother, but a good didactical example.

**Sequential algorithm**

- Forward elimination, sequential dependencies between matrix rows
- Illustrative: Coupling to the left and bottom

1st idea: Classical wavefront-parallelisation (exact)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Pro: Always works to resolve *explicit* dependencies
- Con: Irregular parallelism and access patterns, implementable?
Gauß-Seidel smoother

2nd idea: Decouple dependencies via multicolouring (inexact)

- Jacobi (red) – coupling to left (green) – coupling to bottom (blue) – coupling to left and bottom (yellow)

Analysis

- Parallel efficiency: 4 sweeps with $\approx N/4$ parallel work each
- Regular data access, but checkerboard pattern challenging for SIMD/GPUs due to strided access
- Numerical efficiency: Sequential coupling only in last sweep
Gauß-Seidel smoother

3rd idea: Multicolouring = renumbering

- After decoupling: ‘Standard’ update (left + bottom) is suboptimal
- Does not include all already available results

- Recoupling: Jacobi (red) – coupling to left and right (green) – top and bottom (blue) – all 8 neighbours (yellow)
- More computations than standard decoupling
- Experiments: Convergence rates of sequential variant recovered (in absence of preferred direction)
**Tridiagonal smoother (line relaxation)**

**Starting point**
- Good for ‘line-wise’ anisotropies
- ‘*Alternating Direction Implicit (ADI)*’
  technique alternates rows and columns
- CPU implementation: Thomas-Algorithm (inherently sequential)

**Observations**
- One independent tridiagonal system per mesh row
- ⇒ top-level parallelisation across mesh rows
- Implicit coupling: Wavefront and colouring techniques not applicable
Tridiagonal smoother (line relaxation)

**Cyclic reduction for tridiagonal systems**

- Exact, stable (w/o pivoting) and cost-efficient
- Problem: Classical formulation parallelises computation but not memory accesses on GPUs (bank conflicts in shared memory)
- Developed a better formulation, 2-4x faster
- Index challenge, general idea: Recursive padding between odd and even indices on all levels
Combined GS and TRIDI

Starting point

- CPU implementation: Shift previous row to RHS and solve remaining tridiagonal system with Thomas-Algorithm
- Combined with ADI, this is the best general smoother (we know) for this matrix structure

Observations and implementation

- Difference to tridiagonal solvers: Mesh rows depend sequentially on each other
- Use colouring ($\#c \geq 2$) to decouple the dependencies between rows (more colours = more similar to sequential variant)
Test problem: Generalised Poisson with anisotropic diffusion

- Total efficiency: Time per unknown per digit (\(\mu s\))
- Mixed precision iterative refinement multigrid solver
- Intel Westmere vs. NVIDIA Fermi
Summary: Smoother parallelisation

- Factor 10-30 (dep. on precision and smoother selection) speedup over already highly tuned CPU implementation
- Same functionality on CPU and GPU
- Balancing of numerical and parallel efficiency (hardware-oriented numerics)
Cluster Results
Linearised elasticity

\[
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\begin{pmatrix}
u_1 \\
u_2
\end{pmatrix} = f
\]

\[
\begin{pmatrix}
(2\mu + \lambda)\partial_{xx} + \mu\partial_{yy} \\
(\mu + \lambda)\partial_{yx} \\
(\mu + \lambda)\partial_{yx} \\
\mu\partial_{xx} + (2\mu + \lambda)\partial_{yy}
\end{pmatrix}
\]

Global multivariate BiCGStab
block-preconditioned by
Global multivariate multilevel \((V 1+1)\)
additively smoothed (block GS) by
for all \(\Omega_i\): solve \(A_{11}c_1 = d_1\)
by
local scalar multigrid
update RHS: \(d_2 = d_2 - A_{21}c_1\)
for all \(\Omega_i\): solve \(A_{22}c_2 = d_2\)
by
local scalar multigrid
coarse grid solver: UMFPACK
USC cluster in Los Alamos, 16 dualcore nodes (Opteron Santa Rosa, Quadro FX5600)

Problem size 128 M DOF

Dualcore 1.6x faster than singlecore (memory wall)

GPU 2.6x faster than singlecore, 1.6x than dualcore
Speedup analysis

Theoretical model of expected speedup

- Integration of GPUs increases resources
- Correct model: Strong scaling within each node
- Acceleration potential of the elasticity solver: \( R_{\text{acc}} = \frac{2}{3} \) (remaining time in MPI and the outer solver)
- \( S_{\text{max}} = \frac{1}{1 - R_{\text{acc}}} \)
- \( S_{\text{model}} = \frac{1}{(1 - R_{\text{acc}}) + \left(\frac{R_{\text{acc}}}{S_{\text{local}}}\right)} \)

This example

- Accelerable fraction \( R_{\text{acc}} \): 66%
- Local speedup \( S_{\text{local}} \): 9x
- Modeled speedup \( S_{\text{model}} \): 2.5x
- Measured speedup \( S_{\text{total}} \): 2.6x
- Upper bound \( S_{\text{max}} \): 3x

![Graph showing speedup analysis](image-url)
Weak scalability

Simultaneous doubling of problem size and resources

- Left: Poisson, 160 dual Xeon / FX1400 nodes, max. 1.3 B DOF
- Right: Linearised elasticity, 64 nodes, max. 0.5 B DOF

Results

- No loss of weak scalability despite local acceleration
- 1.3 billion unknowns (no stencil!) on 160 GPUs in less than 50 s
Stationary laminar flow (Navier-Stokes)

\[
\begin{pmatrix}
A_{11} & A_{12} & B_1 \\
A_{21} & A_{22} & B_2 \\
B_1^T & B_2^T & C
\end{pmatrix}
\begin{pmatrix}
u_1 \\
u_2 \\
p
\end{pmatrix}
= 
\begin{pmatrix}
f_1 \\
f_2 \\
g
\end{pmatrix}
\]

fixed point iteration
assemble linearised subproblems and solve with
**global BiCGStab** (reduce initial residual by 1 digit)
Block-Schurcomplement preconditioner
1) approx. solve for velocities with
**global MG** \((V1+0)\), additively smoothed by
for all \(\Omega_i\): solve for \(u_1\) with
**local MG**
2) update RHS: \(d_3 = -d_3 + B^T(c_1, c_2)^T\)
3) scale \(c_3 = (M_p^L)^{-1}d_3\)
Stationary laminar flow (Navier-Stokes)

**Solver configuration**
- Driven cavity: Jacobi smoother sufficient
- Channel flow: ADI-TRIDI smoother required

**Speedup analysis**

<table>
<thead>
<tr>
<th></th>
<th>( R_{\text{acc}} )</th>
<th>( S_{\text{local}} )</th>
<th>( S_{\text{total}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>L9</td>
<td>L10</td>
<td>L9</td>
<td>L10</td>
</tr>
<tr>
<td>DC Re250</td>
<td>52%</td>
<td>62%</td>
<td>9.1\times</td>
</tr>
<tr>
<td>Channel flow</td>
<td>48%</td>
<td>–</td>
<td>12.5\times</td>
</tr>
</tbody>
</table>

**Shift away from domination by linear solver**

- Fraction of FE assembly and linear solver of total time, max. problem size

<table>
<thead>
<tr>
<th></th>
<th>DC Re250</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>12:88</td>
<td>38:59</td>
</tr>
<tr>
<td>GPU</td>
<td>31:67</td>
<td>68:28</td>
</tr>
</tbody>
</table>
Summary
Summary

**ScaRC solver scheme**
- Globally-unstructured-locally-structured
- Tight co-design of discretisation (grid and finite elements) with multilevel solver
- Beneficial on CPUs and GPUs
- Numerically and computationally future-proof (some odd ends still to be resolved)

**GPU computing**
- Parallelising strong recursive smoothers
- Minimally invasive acceleration with legacy codes

**Significant speedups**
- On a single device: one order of magnitude
- On the application level: Reduced due to Amdahl’s Law
Acknowledgements

**Collaborative work with**

- Robert Strzodka (Max Planck Institut Informatik)
- Jamaludin Mohd-Yusof, Patrick McCormick (Los Alamos National Laboratory)

**Supported by**

- DFG: TU 102/22-1, TU 102/22-2
- BMBF: *HPC Software für skalierbare Parallelrechner: SKALB* project 01IH08003D

[http://www.mathematik.tu-dortmund.de/~goeddeke](http://www.mathematik.tu-dortmund.de/~goeddeke)