GPU Accelerated Algebraic Multigrid

Gundolf Haase and Manfred Liebmann

Institute for Mathematics and Scientific Computing
University of Graz

in cooperation with G. Plank [University of Oxford / Med-Uni Graz]

Petropolis, Aug 15, 2008
Contents

- Graz - Projects - Motivation
- Multigrid
- Algebraic Multigrid
- Parallel Algebra
- GPU programming
- AMG on (multiple) GPUs
- Conclusions
Graz

Universities:
- Karl-Franzens-Universität
- University of Technology
- Medical University
- University of Art and Music

City:
- 250,000 inhabitants including 35,000 students
- Old city - beautiful city center
- South of the Alps - north of the Balkan
Ongoing Projects

- **SFB** (special research group) on
  
  **Mathematical Optimization and Applications in Biomedical Sciences**
  
  at the KFU, TUG, MUG since May 1, 2007.

  FEMBEM: Fast Finite Element and Boundary Element Methods for Optimality Systems
  
  http://math.uni-graz.at/mobis

- **Doktoratskolleg** (graduate school) on

  **Numerical Simulations in Technical Sciences**


  Parallel Algorithms in Poroelasticity
  Fast Multilevel Solvers in Electromagnetics
  Sparse Eigenvalue-/Vector Solvers and Clustering
  Multi–Scale CFD–Reaction Simulations
  
  http://www.DK-NumSim.tugraz.at

- Austrian Grid: WP **Numerical Simulations in the GRID**

  The works is supported by the bm:bwk (Federal Ministry for Education, Science and Culture)
  
  http://www.austriangrid.at

- NSF-Project, NFN-project
Motivation

- We have developed parallel codes for 15 years, incl. Multigrid, Multilevel, AMG, Krylov methods, . . . .

- We have a dozen of applications from potential problems, elasticity problems, Maxwell’s equations.

- Approx. 25 licenses for the parallel code PEBBLES.

- We have written a book on parallelization [Douglas/Haase/Langer].
Motivation

- We have developed parallel codes for 15 years, incl. Multigrid, Multilevel, AMG, Krylov methods, ... 

- We have a dozen of applications from potential problems, elasticity problems, Maxwell’s equations.

- Approx. 25 licenses for the parallel code PEBBLES.

- We have written a book on parallelization [Douglas/Haase/Langer].

What’s wrong with our available codes?  

⇒  Let’s have a look at an example.
Motivation

- We have developed parallel codes for 15 years, incl. Multigrid, Multi-level, AMG, Krylov methods, . . . .

- We have a dozen of applications from potential problems, elasticity problems, Maxwell’s equations.

- Approx. 25 licenses for the parallel code PEBBLES.

- We have written a book on parallelization [Douglas/Haase/Langer].

What’s wrong with our available codes?

⇒ Let’s have a look at an example.
Rabbit Heart [G. Planck, M. Liebmann, G. Haase]

Reentry Induction in a Rabbit Ventricular Model

- time-dependent electrical potential, anisotropic coefficients
- 5.082.272 tetrahedrons with 862.525 FEM-nodes
- Goal: 150 Mill. tetrahedrons using parallel mesh generator Spider by F. Kicking
PEBBLES as AMG–preconditioner in the heart problem ($\epsilon = 10^{-6}$)

- sequentially, 111.589 nodes, Pentium4 3GHz:

<table>
<thead>
<tr>
<th>solver</th>
<th>solution [sec.]</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILU/CG</td>
<td>12.0</td>
<td>211</td>
</tr>
<tr>
<td>Hypre</td>
<td>1.9</td>
<td>5</td>
</tr>
<tr>
<td>SuperLU</td>
<td>1.2</td>
<td>(but 70 sec. in setup)</td>
</tr>
<tr>
<td>PEBBLES/CG</td>
<td>0.8</td>
<td>10</td>
</tr>
</tbody>
</table>

- parallel, 862.515 nodes, Opteron nodes, PEBBLES:

<table>
<thead>
<tr>
<th>processors</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>solver iterations</td>
<td>13</td>
<td>12</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>coarse grid</td>
<td>3059</td>
<td>4008</td>
<td>4850</td>
<td>3070</td>
</tr>
<tr>
<td>solver in sec.</td>
<td>10.3</td>
<td>9.0</td>
<td>5.0</td>
<td>3.2</td>
</tr>
<tr>
<td>setup in sec.</td>
<td>[0.3]</td>
<td>[0.6]</td>
<td>[0.7]</td>
<td>[0.4]</td>
</tr>
<tr>
<td></td>
<td>37.1</td>
<td>22.4</td>
<td>17.9</td>
<td>11.3</td>
</tr>
<tr>
<td></td>
<td>[3.7]</td>
<td>[7.0]</td>
<td>[9.7]</td>
<td>[5.4]</td>
</tr>
</tbody>
</table>

- Some obscurities wrt. reconstruction of PEBBLES.
- Our parallel data structures didn’t fit into global code.
- Conclusion: It’s worth to continue the development of the code. But we have to re–write the code.
Motivation for a new code

- PETSc [B. Smith] is used by cooperation partners but we need more/other information for an efficient parallel AMG.

⇒ We have to provide similar functionality as PETSc to convince partners to use our code.

- Handling of old code is too complicated, i.e., one developer has to be always available.

- The professional code cannot be used in education (too much overhead from data setup)

- Code developers left for jobs in industry [M. Kuhn, S. Reitzinger]

⇒ Redesign of interfaces, data structures and functionality.
Motivation for a new code

- PETSc [B. Smith] is used by cooperation partners but we need more/other information for an efficient parallel AMG.

⇒ We have to provide similar functionality as PETSc to convince partners to use our code.

- Handling of old code is too complicated, i.e., one developer has to be always available.

- The professional code cannot be used in education (too much overhead from data setup)

- Code developers left for jobs in industry [M. Kuhn, S. Reitzinger]

⇒ Redesign of interfaces, data structures and functionality.

**Goal:** Toolbox provides all needed **basic** routines for **parallel** functionality.

**Goal:** Write your own parallel code by **re-using sequential** code.
Technical Background

- Technological shift towards many processors and distributed memory
  - Multiple cores per processor: Dualcore, Quadcore
  - Multiple processors per machine
  - Multiple machines per computing cluster
  - Multiple clusters: Grid computing

- Single computation cores hit a speed limit: 3-4 GHz
  - Solution: More cores per processor
  - Cell processor: 8 Synergistic Processing Elements (SPE) with 3.2 GHz, and theoretical 25.6 GFLOPS for each SPE
  - Intel 80 core research chip with 6.26 GHz and 2 TFlops

- Software design process is still largely sequentially oriented
  - Parallelization: An afterthought
  - Tools: Message passing protocol (MPI)
  - Thread based parallelization

- Parallelization will soon be a *necessity* not an option!
  - Expect 1000s of processors in near future
  - Algorithms have to be designed with parallelization in mind.
  - Failure to do so will lead to very bad performance
Introduction into Multigrid

- Problem Classes
- Classical Multigrid
- Algebraic Multigrid
- Contents
Considered Problem Classes

Find $u$ such that

\[ Lu(x) = f(x) \quad \forall x \in \Omega \]

\[ lu(x) = g(x) \quad \forall x \in \partial \Omega \]

variational \downarrow \text{ formulation}

Find $u \in V :$ \[ a(u,v) = \langle F,v \rangle \quad \forall v \in V \]

FEM, FDM \downarrow \text{ FVM, FIT}

\[
\text{Solve} \quad \mathbf{K}_h \cdot \mathbf{u}_h = \mathbf{f}_h \quad \mathbf{u}_h \in \mathbb{R}^{N_h}
\]

- (linear) 2\textsuperscript{nd} order problem.
  - Poisson equation (temperature)
  - Lamé equation (deformation)
  - Maxwell’s equations (magnetic field)

- Matrix $\mathbf{K}_h$ is sparse, positive definite
  (symmetric, large dimension)

- non-linear and time-dependent problems.
The Principle of Multigrid

Solve the system of equations

\[ K_h u_h = f_h \]

by a sequence of different discretizations and

Combination of different accuracy levels.

\[ K_{h} u_{h} = f_{h} \]
\[ K_{2h} u_{2h} = f_{2h} \]
\[ K_{4h} u_{4h} = f_{4h} \]
Two grid method

Gain: optimal iterative solver (memory, CPU-time).

Remark: A recursive application leads to a multigrid cycle.
(Classical) Multigrid $V(\nu_F, \nu_B)$-cycle: $\text{MG}(u_l, f_l, l)$

if $l = \text{COARSELEVEL}$ then
  $u_l = (K_l)^{-1}f_l$ is solved by a direct solver
else
  Smooth $\nu_F$-times with $K_lu_l = f_l$
  Calculate defect $d_l = f_l - K_lu_l$
  Restrict defect $d_{l+1} = P_l^T d_l$
  Set $w_{l+1} \equiv 0$
  $\text{MG}(w_{l+1}, d_{l+1}, l+1)$
  Interpolate correction $w_l = P_l w_{l+1}$
  Update solution $u_l = u_l + w_l$
  Smooth $\nu_B$-time with $K_lu_l = f_l$
end if
Strategies in Multigrid

Each MG iteration requires:
Matrices $K_l$, smoother, interpolation $P_l$, (restriction $R_l = P_l^T$) for each discretization level.
Strategies in Multigrid

Each MG iteration requires:
Matrices $K_l$, smoother, interpolation $P_l$, (restriction $R_l = P_l^T$) for each discretization level.

- **Classical MG:**

Meshes/grids, Matrices, interpolation are given for all levels.
Strategies in Multigrid

Each MG iteration requires:
Matrices $K_l$, smoother, interpolation $P_l$, (restriction $R_l = P_l^T$) for each discretization level.

- **Classical MG:**

Meshes/grids, Matrices, interpolation are given for all levels.

- **Full MG - adaptive MG:**

Coarse mesh, coarse matrix and interpolation are given.
Strategies in Multigrid

Each MG iteration requires:
Matrices $K_l$, smoother, interpolation $P_l$, (restriction $R_l = P_l^T$) for each discretization level.

- **Classical MG:**

Mesures/grids, Matrices, interpolation are given for all levels.

- **Full MG - adaptive MG:**

Coarse mesh, coarse matrix and interpolation are given.

- **Algebraic MG (AMG):**

Only the fine mesh matrix is known (+ fine mesh).
Main ingredients of AMG

(i) coarsening:

\[ \omega^h = \omega_C^h \cup \omega_F^h . \]

(ii) interpolation weights:

\[ P = \{ \alpha_{ij} \}_{i \in \omega^h, j \in \omega_C^h} : \mathbb{R}^H \mapsto \mathbb{R}^h . \]

(iii) coarse mesh matrix:

\[ K^H = P^T \cdot K \cdot P \]

(iv) (i)-(iii) have to be applied recursively.

(v) apply the standard MG-procedure.
Coarsening
Prolongation $P : \mathbb{R}^{NH}(\omega_C) \mapsto \mathbb{R}^{Nh}(\omega_C \cup \omega_F)$

Factorization of $K_h = \begin{pmatrix} K_C & K_{CF} \\ K_{FC} & K_F \end{pmatrix}$ and inversion of the factors:

$$K_h^{-1} = \begin{pmatrix} I_C & 0 \\ -K_F^{-1}K_{FC} & I_F \end{pmatrix} \cdot \begin{pmatrix} S_C^{-1} & 0 \\ 0 & K_F^{-1} \end{pmatrix} \cdot \begin{pmatrix} I_C & -K_{CF}K_F^{-1} \\ 0 & I_F \end{pmatrix}$$

$\approx P$

The Twogrid method is
**Prolongation** \( P : \mathbb{R}^{N_H}(\omega_C) \mapsto \mathbb{R}^{N_h}(\omega_C \cup \omega_F) \)

Factorization of \( K_h = \begin{pmatrix} K_C & K_{CF} \\ K_{FC} & K_F \end{pmatrix} \) and inversion of the factors:

\[
K_h^{-1} = \begin{pmatrix} I_C & 0 \\ -K_F^{-1}K_{FC} & I_F \end{pmatrix} \cdot \begin{pmatrix} S_C^{-1} & 0 \\ 0 & K_F^{-1} \end{pmatrix} \cdot \begin{pmatrix} I_C & -K_{CF}K_F^{-1} \\ 0 & I_F \end{pmatrix}
\]

\[ \approx P \]

The Twogrid method is an approximation of the factors:

\[
C_h^{-1} = P \cdot \begin{pmatrix} (P^T K_h P)^{-1} & 0 \\ 0 & (I_F - M_F)K_F^{-1} \end{pmatrix} \cdot P^T
\]

The prolongation is
Prolongation $P : \mathbb{R}^{N_H}(\omega_C) \mapsto \mathbb{R}^{N_h}(\omega_C \cup \omega_F)$

Factorization of $K_h = \begin{pmatrix} K_C & K_{CF} \\ K_{FC} & K_F \end{pmatrix}$ and inversion of the factors:

$$K_h^{-1} = \begin{pmatrix} I_C & 0 \\ -K_F^{-1}K_{FC} & I_F \end{pmatrix} \cdot \begin{pmatrix} S_C^{-1} & 0 \\ 0 & K_F^{-1} \end{pmatrix} \cdot \begin{pmatrix} I_C & -K_{CF}K_F^{-1} \\ 0 & I_F \end{pmatrix}$$

$$\approx P$$

The Twogrid method is an approximation of the factors:

$$C_h^{-1} = P \cdot \begin{pmatrix} (P^T K_h P)^{-1} & 0 \\ 0 & (I_F - M_F)K_F^{-1} \end{pmatrix} \cdot P^T$$

The prolongation is the local approximation of the global discrete harmonic extension from the coarse space into the fine space:

$$u_h = P \cdot u_H \approx$$ Find $u_h \in \mathbb{V}_h$ such that $L_h u_h(x) = 0 \quad \forall x \in \omega^h \setminus \omega^H$

$$u_h(x) = u_H(x) \quad \forall x \in \omega^H$$
**Prolongation** \( P : \mathbb{R}^{N_H}(\omega_C) \mapsto \mathbb{R}^{N_h}(\omega_C \cup \omega_F) \)

Factorization of \( K_h = \begin{pmatrix} K_C & K_{CF} \\ K_{FC} & K_F \end{pmatrix} \) and inversion of the factors:

\[
K_h^{-1} = \begin{pmatrix} \frac{1}{\omega} & 0 \\ -K_C^{-1}K_{FC} & I_F \end{pmatrix} \cdot \begin{pmatrix} S_C & 0 \\ 0 & K_F^{-1} \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{\omega} & 0 \\ 0 & -K_{CF}K_F^{-1} \end{pmatrix}
\]

\[ \approx P \]

The Trilu grid method is an approximation of the factors:

\[
C_h^{-1} = P \cdot \left( (P^T K_h P)^{-1} \begin{pmatrix} 0 \\ (I_F - M_F)K_F^{-1} \end{pmatrix} \right) \cdot P^T
\]

The prolongation is the local approximation of the global discrete harmonic extension from the coarse space into the fine space:

\[
u_h = P \cdot u_H \approx \begin{cases} 0 & \text{if } x \in \omega^h \setminus \omega^H \\ u_H(x) & \text{if } x \in \omega^H \end{cases}
\]

Find \( u_h \in \mathbb{V}_h \) such that

\[
L_h u_h(x) = 0 \quad \forall x \in \omega^h \setminus \omega^H
\]

\[
u_h(x) = u_H(x) \quad \forall x \in \omega^H
\]
Parallel Algebra

- Concept for Parallelization
- Extended Parallelization Concept
- Parallel Multigrid
- Parallel factorization

- Contents
Parallelization Concepts in the Domain Decomposition Context

Assume: Finite element simulation over a complicated domain using an unstructured mesh.

- Main Strategy: Domain decomposition of the mesh.
  - Elements are distributed onto different processors
  - Graph partitioning problem: METIS/PARMETIS software
  - Communication setup requires global node numbers and local element connectivity

![Diagram](image)

Figure 1: Finite element mesh distributed to four processing nodes with global element and node numbers.
Non-overlapping Data Decomposition

accumulated

\[ u_s = A_s u \]
\[ M_s = A_s M A_s^T \]

distributed

\[ r = \sum_{s=1}^{P} A_s^T r_s \]
\[ K = \sum_{s=1}^{P} A_s^T K_s^{\text{FEM}} A_s \]
Basic Operations

without communication

\[ v \leftarrow K \cdot s \]
\[ r \leftarrow f + \alpha \cdot v \]
\[ w \leftarrow u + \alpha \cdot s \]
\[ r \leftarrow R^{-1} \cdot w \]

global reduce

\[ \alpha \leftarrow \langle w, r \rangle = \sum_{s=1}^{P} \langle w_s, r_s \rangle \]
\[ w \leftarrow r = \sum_{s=1}^{P} A_s^T r_s \]
\[ M \leftarrow \sum_{s=1}^{P} A_s K_s A_s^T \]

next neighbor comm.

with \( R = \text{diag}\{R_{ii}\}_{i=1}^{N} = \text{diag}\{\text{# subdomains } x_i \text{ is associated with}\} = \sum_{s=1}^{P} A_s^T \cdot A_s \)

and \( R^{-1} \equiv I = \sum_{s=1}^{P} A_s^T \cdot I_s \cdot A_s \) (partition of unity)
Parallel CG: \( \text{PCG}(K, u, f) \)

\[
\begin{align*}
\text{repeat} \\
\quad v & \leftarrow K \cdot s \\
\quad \alpha & \leftarrow \sigma / \langle s, v \rangle \\
\quad u & \leftarrow u + \alpha s \\
\quad r & \leftarrow r - \alpha v \\
\quad w & \leftarrow C^{-1} \cdot r \\
\quad \sigma & \leftarrow \langle w, r \rangle \\
\quad \beta & \leftarrow \sigma / \sigma_{\text{old}} \\
\quad s & \leftarrow w + \beta s \\
\text{until termination}
\end{align*}
\]
Matrix Patterns and their Application

The following operations can be performed in parallel without any communication:

\[ f = K \cdot u \]

Matrix \( M \) fulfills the pattern condition:

\[ \forall i, j \in \omega : \sigma^i \not\subseteq \sigma^j \implies M^{i,j} = 0 \quad i \not\leftarrow j \]

\[ u = M \cdot w \]
\[ f = M^T \cdot r \]
\[ K^H = M^T \cdot K \cdot M \]

Theorems/Proofs [Haase]

Ex.:  
\[ \sigma^{[11]} \not\subseteq \sigma^{[27]} \implies M^{[11,27]} = 0 \quad 11 \not\leftarrow 27 \]
\[ \sigma^{[27]} \not\subseteq \sigma^{[11]} \implies M^{[27,11]} = 0 \quad 27 \not\leftarrow 11 \]
\[ \sigma^{[11]} \not\subseteq \sigma^{[14]} \implies M^{[11,14]} = 0 \quad 11 \not\leftarrow 14 \]
\[ \sigma^{[27]} \not\subseteq \sigma^{[14]} \implies M^{[27,14]} = 0 \]
Admissible Matrix Operations

Vertex, Edge, Inner nodes

\[
\mathbf{M} = \begin{pmatrix}
\mathbf{M}_V & 0 & 0 \\
\mathbf{M}_{EV} & \mathbf{M}_E & 0 \\
\mathbf{M}_{IV} & \mathbf{M}_{IE} & \mathbf{M}_I
\end{pmatrix} = \mathbf{M}_L + \mathbf{M}_D \implies \mathbf{u} = \mathbf{M} \cdot \mathbf{w}
\]

Pattern condition \(\sigma^{[i]} \not\subseteq \sigma^{[j]} \implies \mathbf{M}^{[i,j]} = 0\) has to be fulfilled in all submatrices!!

Allows operations as (Parallel ADI [DouHaa])

\[
\mathbf{w} = \mathbf{M} \cdot \mathbf{u} := (\mathbf{M}_L + \mathbf{M}_D) \cdot \mathbf{u} + \sum_{s=1}^{P} A_s^T \mathbf{M}_{U,s} R_s^{-1} \cdot \mathbf{u}_s
\]

or, for \(\mathbf{M} = \mathbf{L}^{-1} \cdot \mathbf{U}^{-1}\) ([Haase])

\[
\mathbf{w} = \mathbf{L}^{-1} \mathbf{U}^{-1} \cdot \mathbf{r} := \mathbf{L}^{-1} \sum_{s=1}^{P} A_s^T \mathbf{U}_s^{-1} \cdot \mathbf{r}_s
\]
Parallel Iteration Schemes to Solve $K \cdot u = f$

- Richardson iteration:
  \[
  u_{s}^{k+1} := u_{s}^{k} + \tau \sum_{q=1}^{P} (f - K \cdot u^{k})_{q}
  \]

- Jacobi iteration with $D = \sum_{s=1}^{P} \text{diag} \{K_s\}$:
  \[
  u_{s}^{k+1} := u_{s}^{k} + \omega D_{s}^{-1} \sum_{q=1}^{P} (f - K \cdot u^{k})_{q}
  \]

- Incomplete factorization $R = \mathcal{L} \cdot \mathcal{L}^{-1} + R$:
  \[
  u_{s}^{k+1} := u_{s}^{k} + \mathcal{L}_{s}^{-1} \sum_{q=1}^{P} \mathcal{L}_{q}^{-1} (f - K \cdot u^{k})_{q}
  \]
Parallel Multigrid : PMG($K, u, f, \ell$)

if $\ell == 1$ then
L"Solve $\sum_{s=1}^{P} A_s^T K A_s \cdot u = f$
else
    $\tilde{u} \leftarrow \text{SMOOTH}(K, u, f, \nu)$
    $d \leftarrow f - K \cdot u$
    $d^H \leftarrow \mathcal{P}^T \cdot d$
    $w^H \leftarrow 0$
    PMG$(K^H, w^H, d^H, \ell - 1)$
    $w \leftarrow \mathcal{P} \cdot w^H$
    $\hat{u} \leftarrow \tilde{u} + w$
    $u \leftarrow \text{SMOOTH}(K, \hat{u}, f, \nu)$
end if
Communicator Class

The Parallel Toolbox provides a communicator class to handle the core functions.

- Main features of communicator class
  - Efficient algorithms for the communication setup
  - In place accumulation and distribution on vectors
  - Single MPI call for the complete accumulation process
  - Efficient all-to-all communication pattern

Figure 2: All-to-all communication pattern for the accumulation process using four processing nodes.
Real World Examples and Benchmarks

Parallel conjugate gradient solver with AMG preconditioner on Kepler and Boltzmann cluster.

- Testcase: 3D bunny heart simulation (G.Plank/M.Liebmann/G.Haase)

<table>
<thead>
<tr>
<th>NP</th>
<th>Solver</th>
<th>Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>31.37</td>
<td>5.42</td>
</tr>
<tr>
<td>2</td>
<td>14.83</td>
<td>2.57</td>
</tr>
<tr>
<td>4</td>
<td>7.59</td>
<td>1.64</td>
</tr>
<tr>
<td>8</td>
<td>3.95</td>
<td>1.11</td>
</tr>
<tr>
<td>16</td>
<td>2.13</td>
<td>1.05</td>
</tr>
<tr>
<td>32</td>
<td>1.35</td>
<td>1.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NP</th>
<th>Solver</th>
<th>Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.58</td>
<td>5.78</td>
</tr>
<tr>
<td>2</td>
<td>14.75</td>
<td>2.85</td>
</tr>
<tr>
<td>4</td>
<td>8.51</td>
<td>2.13</td>
</tr>
<tr>
<td>8</td>
<td>5.25</td>
<td>1.72</td>
</tr>
<tr>
<td>16</td>
<td>3.72</td>
<td>1.88</td>
</tr>
<tr>
<td>32</td>
<td>4.33</td>
<td>3.96</td>
</tr>
</tbody>
</table>

Table 1: Solver and setup times in seconds for Kepler (Infiniband) and Boltzmann (Gigabit) cluster

<table>
<thead>
<tr>
<th>Cluster</th>
<th>CPU</th>
<th>RAM</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kepler</td>
<td>2x Opteron 248 2,2GHz</td>
<td>4GB DDR</td>
<td>1x Infiniband</td>
</tr>
<tr>
<td>Boltzmann</td>
<td>2x Opteron 248 2,2GHz</td>
<td>4GB DDR</td>
<td>1x Gigabit Ethernet</td>
</tr>
</tbody>
</table>

Table 2: Cluster node configuration
Performance Analysis and Conclusions

- Domain decomposition based parallelization is a viable approach
  - Network switch is the critical component
  - Very good performance scaling with Infiniband network
  - Reasonable performance with Gigabit network
Cooperations

- U. Langer/H. Yang (JKU Linz)
  - Parallel Schur complement based Stokes solver
  - Complete parallel solver implemented in three days

- H. Steiner/D. Heidorn (TU Graz)
  - Parallel AMG solver component for multi-phase flow simulation
GPU–accelerated AMG

- Motivation for Using Graphics Hardware
- Parallelization Concepts: Extreme Multithreading
- Code Sample: Sparse Matrix-Vector Multiplication
- Sample: Sparse Matrix-Vector Multiplication
- Sample: AMG on GPU
- Double Precision Arithmetics
(1) Motivation for Using Graphics Hardware

Technological shift towards multi-core and many-core processors!

- **CPU**
  - Dualcore, Quadcore processors up to 3.2 GHz
  - Soon: Octocore processors + Simultaneous Multithreading (SMT), Larrebee
  - Memory bandwidth: \( \approx 5 \text{ GB/s} \)

- **GPU (I/2008)**
  - Each multiprocessor: 8 scalar processors + Shared memory
  - Up to 768 threads per multiprocessor
  - Up to 16 multiprocessors per graphics chip

- **NVIDIA Geforce 8800 GT**
  - 1 GB RAM + 112 scalar processors at 1.6 GHz
  - Memory bandwidth \( \approx 50 \text{ GB/s} \)
  - Peak performance: 364 GFLOPS (single precision)
  - PCI Express Card: **150 EUR**
Floating-Point Performance

Figure 3: Floating-Point Operations per Second for the CPU and GPU
GPU–architecture III/2008

- **NVidia**
  - IEEE 754 single and **double precision** floating point support
  - Up to 30 multiprocessors per graphics chip
  - Multiprocessor: 8 processor cores + 16384 registers + 64KB shared memory
  - 3 x 8 FP32 (MAD + MUL) + 2 x FP64 (MAD) per clock
  - Up to 1024 threads per multiprocessor
  - Up to 4GB RAM per graphics board

- **NVIDIA GTX 280 (III/2008)**
  - 1.4 billion transistors
  - 240 processor cores @ 1.3 GHz + 1 GB RAM
  - Memory bandwidth $\approx 150$ GB/s
  - Peak performance: $\approx 1$ TFLOPS (single) + $\approx 100$ GFLOPS (double)
  - PCI Express Card: **360 EUR**

![Image of NVIDIA GTX 280](image_url)

Figure 4: NVIDIA GTX 280
What are the differences?

- **Difference between GPU and CPU**
  - **GPU** has *no* coherent processor cache only small on-chip shared memory
  - Simple scalar processors supporting many threads for latency hiding
  - **CPU** has big L1 + L2 processor caches
  - Complicated deeply pipelined processor to hide memory latency

![CPU and GPU Diagram]

Figure 5: The GPU Devotes More Transistors to Data Processing
(2) Parallelization Concepts: Extreme Multithreading

Reading data from main memory has 400-600 cycles latency. Solution: *Schedule hundreds of threads to hide the latency!*

- **Grids and Thread Blocks**
  - Threads are organized in 1D/2D/3D **thread blocks** of up to 256 threads
  - Thread blocks are organized in a 1D/2D/3D **grids**
  - Thread blocks are scheduled for execution on the multiprocessors

- **No global thread synchronization**
  - Synchronization only per multiprocessor i.e. thread block

Thread Batching on the GPU
Data Caching

- There is no coherent data cache on the GPU!
  - Scalar processors directly access the DRAM memory
  - Only a small read-only texture cache is available

![Diagram showing memory access patterns: Gather and Scatter]

Figure 6: Memory Access Patterns
### Shared Memory

- Multiprocessors share a small on-chip memory region

![Diagram of shared memory](image)

**Without shared memory**

**With shared memory**

*Figure 7: Shared Memory Brings Data Closer to the ALUs*
Coalesced Memory Access

- Coalesced memory access is mandatory!
  - Every thread must access the correct memory bank!
  - Coalescing over a range of 16 threads
  - Extreme performance penalty for non-coalesced memory access

Figure 8: Coalesced Memory Access
What are the Implications?

- **Algorithms must be designed to fit within all these restrictions!**
  - Failure to do so results in very bad performance
  - Challenging algorithm design
  - Design for 1000s of threads per computational kernel
  - Excellent performance if done right!
(3) Sample: Sparse Matrix-Vector Multiplication

**Nvidia CUDA Toolkit**: C/C++ compiler and libraries for GPU programming

![CUDA Software Stack Diagram](image-url)

Figure 9: Compute Unified Device Architecture Software Stack
Nvidia CUDA Toolkit

- C/C++ library/driver for GPU
- GPU specific code can be written in C/C++
- Reasonably easy integration with standard compilers.
- Allows incremental approach to GPU programming
- Time consuming kernels can be executed at the GPU
- Separation of host and device code
- Uses GPU as co-processor
The Algorithm

Let $A \in \mathbb{R}^{N \times N}$ be a matrix in compressed row storage format and $u, b \in \mathbb{R}^N$.

- **CRS Sparse Matrix-Vector Kernel**
  - Schedule a thread for every sparse scalar product!
  - Thread $i$ calculates $u_i = \sum_{j=1}^{N} A_{ij} b_j$

  **Looks nice! Performs very badly!**
Problems and Solutions

- **Non-coalesced memory access!**
  - Rearrange CRS data structure for coalesced access
  - Interleave the sparse matrix rows for at least 16 consecutive rows
  - Holes in the data structure: Not critical! Typical 5-10% increase in storage

- **Random access to $b$ vector!**
  - Use texture unit of the GPU for random access to $b$ vector
  - Texturing is optimized for spacial locality: Small read-only cache
Example: Sparse Matrix-Vector Multiplication

```c
#define N 256
extern "C" void _device_linear_operator
    (int *cnt, int *dsp, int *col, float *ele, int m, int n, int size, float *u, float *v)
{
    cudaBindTexture(0, tex_u, u, sizeof(float) * m);

    struct linear_operator_params parms;
    parms.cnt = cnt;
    parms.dsp = dsp;
    parms.col = col;
    parms.ele = ele;
    parms.u = u;
    parms.v = v;
    parms.n = n;

    __device_linear_operator<<<(n + N - 1)/N, N>>>(parms); //Asynchronous kernel launch on the GPU.

    cudaUnbindTexture(tex_u);
}
```
Example: Sparse Matrix-Vector Multiplication

```c
#define N 256
texture<float> tex_u;
__global__ void __device_linear_operator(struct linear_operator_params parms)
{
  unsigned int j = N * blockIdx.x + threadIdx.x;
  if(j < parms.n)
  {
    unsigned int blkStart = parms.dsp[j];
    unsigned int blkStop = blkStart + N * parms.cnt[j];

    float s = 0.0;
    for(unsigned int i = blkStart; i < blkStop; i += N)
    {
      unsigned int q = parms.col[i];
      float a = parms.ele[i];
      float b = tex1Dfetch(tex_u, q);
      s += a * b;
    }
    parms.v[j] = s;
  }
}
```
Performance of matrix vector product (single precision)

- Finite element fluid dynamics simulation test matrices [Steiner, Brenn]
  - (a) $N = 720.000$, $\# A = 5,020.800$
  - (b) $N = 274.625$, $\# A = 7,189.057$

<table>
<thead>
<tr>
<th></th>
<th>AMD Opteron 8347</th>
<th>Intel C2D E6850</th>
<th>Nvidia Geforce 8800 GT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>0.25</td>
<td>0.97</td>
<td>10.1</td>
</tr>
<tr>
<td>(b)</td>
<td>0.58</td>
<td>1.17</td>
<td>10.8</td>
</tr>
</tbody>
</table>

Table 3: Performance of sparse matrix-vector multiplication in GFLOPS
(4) Sample: AMG on GPU

- **Current Work: Advanced solver components**
  - CPU/GPU parallel algebraic multigrid solver component
  - Integration with the Parallel Toolbox framework
  - Support for multi-GPU computing node configurations

- **Key performance indicator: Sparse matrix-vector multiplication**
  - Multigrid prolongation and restriction operators and Jacobi smoother
  - Vector operations work just fine
  - Naive Gauss-Seidel does not work!
  - High algorithmic complexity in the AMG setup \(\rightarrow\) CPU-computing
Rabbit Heart [G. Planck, M. Liebmann, G. Haase]

Reentry Induction in a Rabbit Ventricular Model

- time-dependent electrical potential, anisotropic coefficients
- 5.082.272 tetrahedrons with 862.525 FEM-nodes
- Goal: 150 Mill. tetrahedrons using parallel mesh generator Spider by F. Kickinger
PCG-AMG Benchmark (single precision)

- PCG-AMG Performance with Parallel Toolbox
  - 6x – 8x over Intel C2D E6850 3.0 GHz
  - 16x – 24x over AMD Opteron 8347 1.9 GHz
  - Performance of 250.000 EUR Infiniband 32x Opteron cluster computer!
  - 1000x price/performance advantage!

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>A</th>
<th>Intel E6850</th>
<th>Nvidia 8800 GT</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>720.000</td>
<td>5,020.800</td>
<td>0.089777</td>
<td>0.0122688</td>
<td>7.3</td>
</tr>
<tr>
<td>(b)</td>
<td>274.625</td>
<td>7,189.057</td>
<td>0.050223</td>
<td>0.0078146</td>
<td>6.4</td>
</tr>
<tr>
<td>(c)</td>
<td>862.515</td>
<td>12,795.209</td>
<td>0.158694</td>
<td>0.0219833</td>
<td>7.2</td>
</tr>
<tr>
<td>(d)</td>
<td>111.589</td>
<td>1,445.373</td>
<td>0.017278</td>
<td>0.0038916</td>
<td>4.4</td>
</tr>
</tbody>
</table>

Table 4: Timing of PCG-AMG iteration in seconds
Early Engineering Sample: Double Precision Tesla Board (II/2008)

Figure 10: Nvidia Tesla GPU Computing Solutions

- **Tesla T10P Double Precision Board**
  - Prerelease hardware with early driver version
  - Double precision performance looks good
  - Only limited testing due to board instability!

<table>
<thead>
<tr>
<th></th>
<th>Nvidia 8800 GT</th>
<th>Tesla T10P</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>0.0038916 (FP32)</td>
<td>0.0050266 (FP64)</td>
</tr>
<tr>
<td>#A</td>
<td>1, 445.373</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Timing of PCG-AMG iteration in seconds
GPU4U - 4 GTX 280 (1GB) [III/2008]
GPU4U - Timings

AMG Benchmark with GNU compiler gcc -O3 -funroll-all-loops. Relative accuracy: $\epsilon_{CG} < 10^{-6}$. Test matrix: TBunnyC_KieSym.bin.

<table>
<thead>
<tr>
<th></th>
<th>$N$</th>
<th>#A</th>
<th>Opteron 8347</th>
<th>Phenom 9950</th>
<th>Intel E6850</th>
<th>Nvidia GTX 280</th>
<th>gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>720.000</td>
<td>5,020.800</td>
<td>0.295789</td>
<td>0.146261</td>
<td>0.13993</td>
<td>0.0109677</td>
<td>12.7</td>
</tr>
<tr>
<td>(b)</td>
<td>274.625</td>
<td>7,189.057</td>
<td>0.163788</td>
<td>0.083934</td>
<td>0.0754385</td>
<td>0.0108806</td>
<td>6.9</td>
</tr>
<tr>
<td>(c)</td>
<td>862.515</td>
<td>12,795.209</td>
<td>0.517548</td>
<td>0.269558</td>
<td>0.222496</td>
<td>0.0236462</td>
<td>9.4</td>
</tr>
<tr>
<td>(d)</td>
<td>111.589</td>
<td>1,445.373</td>
<td>0.049779</td>
<td>0.028357</td>
<td>0.0240076</td>
<td>0.00360722</td>
<td>6.6</td>
</tr>
</tbody>
</table>

Table 6: Timing of PCG-AMG iteration in seconds (double precision)

Parallel Computer vs. GPU

<table>
<thead>
<tr>
<th></th>
<th>$N$</th>
<th>#A</th>
<th>Kepler 32P (IB)</th>
<th>Boltzmann 16P (GBE)</th>
<th>Liebmann 16P (SHM)</th>
<th>Nvidia GTX 280</th>
<th>price</th>
<th>EUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>862.515</td>
<td>12,795.209</td>
<td>0.02109375</td>
<td>0.058125</td>
<td>0.05375</td>
<td>0.0236462</td>
<td>230.000</td>
<td>150.000</td>
</tr>
</tbody>
</table>

Table 7: Timing of PCG-AMG iteration in seconds (double precision)

<table>
<thead>
<tr>
<th></th>
<th>$N$</th>
<th>#A</th>
<th>Liebmann 16P (SHM)</th>
<th>Nvidia GTX 280</th>
<th>4 x GTX 280</th>
<th>Speedup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>862.515</td>
<td>12,795.209</td>
<td>0.05375</td>
<td>0.0236462</td>
<td>.....</td>
<td>—</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8: Timing of PCG-AMG iteration in seconds (double precision) [Aug 15, 2008]
Conclusions

- **GPU computing looks very promising!**
  - Attractive price / performance ratio
  - Tesla clusters for HPC applications
  - Incremental software development supporting C/C++
  - Reasonable porting effort using the free CUDA toolkit
  - Website: [www.nvidia.com/cuda/](http://www.nvidia.com/cuda/)

- **Mathematically:**
  - Fast Double Precision on GPU is available
  - Use GPUs as Co-processor
  - GPU programming pays off – even for sparse scalar matrices.
  - ASM methods better suited.
  - There are two new initiatives (Berkely, Stanford) to unify code development on many–core processors (GPU, Cell, Larrabee).
Thank you for your attention!