Algebraic Multigrid Methods on GPU-Accelerated Hybrid Architectures

Manfred Liebmann
Institute for Mathematics and Scientific Computing
University of Graz
manfred.liebmann@uni-graz.at

June 7, 2011

Part I

Algebraic Multigrid Methods on GPU-Accelerated Hybrid Architectures

Overview

- Model Problem: Virtual Heart CARP Project
- Parallel PCG-AMG Solver Performance
- Parallel Toolbox Software
- Sequential Algebraic Multigrid Algorithm
- Parallel Algebraic Multigrid Algorithm
- Parallelization on GPU-Accelerated Hybrid Architectures

People and Projects

- **Collaborations**
  - Gundolf Haase, **University of Graz**, Austria (SFB MOBIS)
  - Gernot Plank, **Medical University of Graz**, Austria (SFB MOBIS)
  - Craig C. Douglas, **University of Wyoming**, USA (GPU Cluster)
  - Charles Hirsch, **NUMECA International S.A**, Belgium (E-CFD-GPU Project)
  - Mike Giles, **University of Oxford**, UK (OP2 Project)
  - Zoltán Horváth, **Széchenyi István University**, Hungary (TAMOP Project)

(1) Model Problem: Virtual Heart CARP Project

The virtual heart model is based on the bidomain equations, a set of coupled partial differential equations, which describe the current flow in the myocardium. The bidomain equations can be written as follows:

$$-\nabla \cdot (\bar{\sigma}_i \nabla \phi_i) = -\beta I_m, \quad -\nabla \cdot (\bar{\sigma}_e \nabla \phi_e) = \beta I_m, \quad -\nabla \cdot (\bar{\sigma}_b \nabla \phi_e) = I_e$$

$$I_m = C_m \frac{\partial V_m}{\partial t} + I_{ion}(V_m, \vec{\eta}) - I_{tr}, \quad \frac{d\vec{\eta}}{dt} = g(V_m, \vec{\eta}), \quad V_m = \phi_i - \phi_e$$

The bidomain equations decouple into an **elliptic PDE**

$$(A_i + A_e)\Phi_e^{k+1} = A_i V^{k+1} + I_e$$

a **parabolic PDE**

$$\begin{cases} V^{k*} = (1 - \Delta t A_i)\, V^k - \Delta t A_e \phi_e^k, & \Delta x > 100\mu m \\ \left[1 + \frac{1}{2}\Delta t A_i\right] V^{k*} = \left[1 - \frac{1}{2}\Delta t A_i\right] V^k - \Delta t A_e \phi_e^k, & \Delta x < 100\mu m \end{cases}$$

and a **set of ODEs**

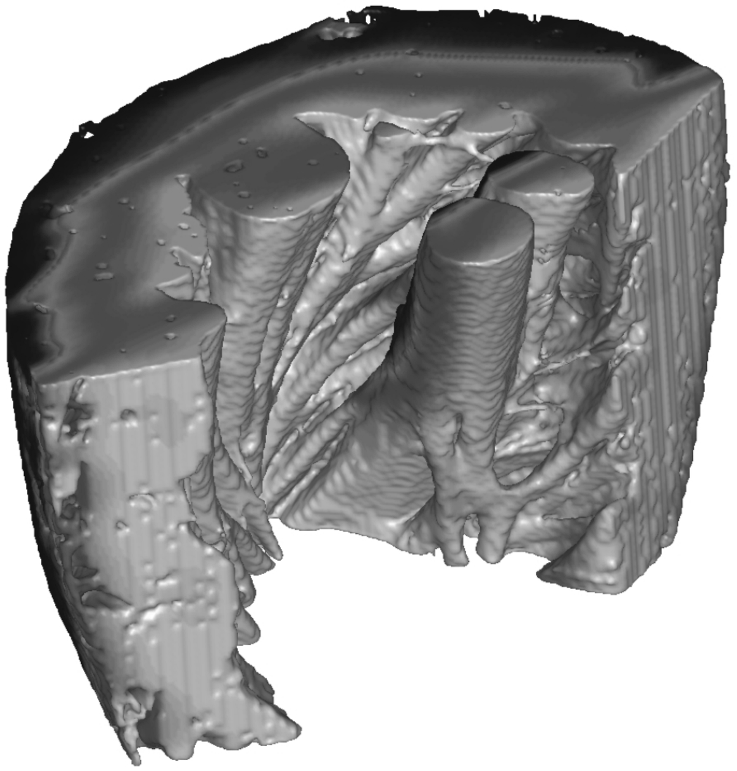
$$V^{k+1} = V^{k*} + \frac{\Delta t}{C_m} i_{ion}\left(V^{k*}, \vec{\eta}^{\,k}\right)$$

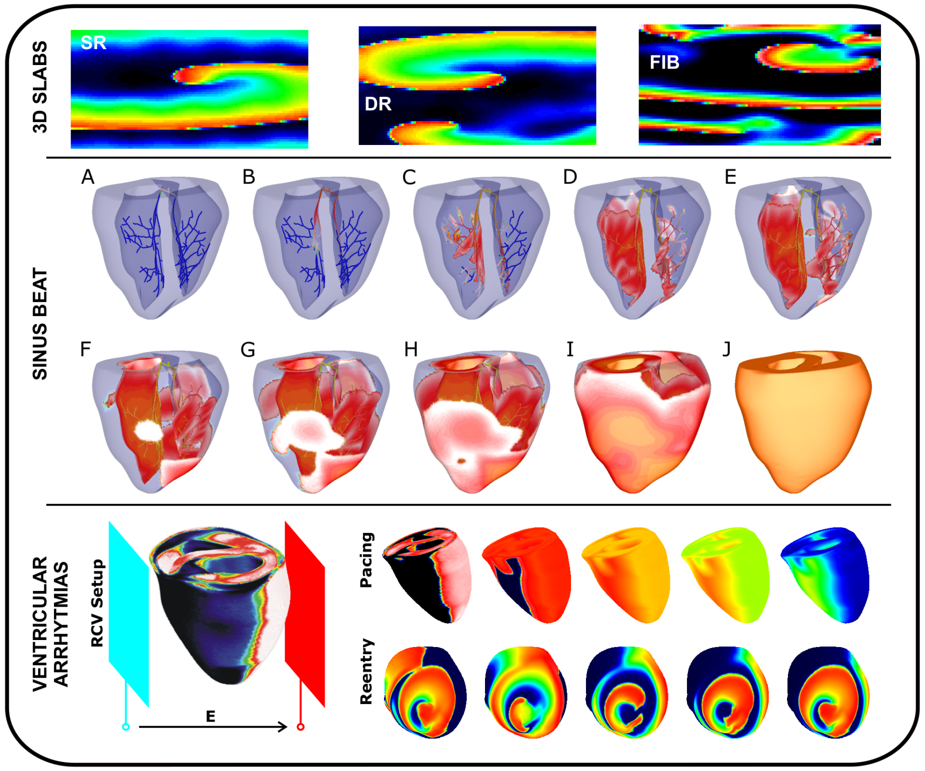
$$\vec{\eta}^{\,k+1} = \vec{\eta}^{\,k} + \Delta t\, g(V^{k+1}, \vec{\eta}^{\,k})$$

with

$$A_i = -\frac{\nabla \cdot (\bar{\sigma}_i \nabla)}{\beta C_m}, \quad A_e = -\frac{\nabla \cdot (\bar{\sigma}_i \nabla)}{\beta C_m}, \quad t = k\Delta t$$

- **Virtual Heart Simulator**
  - CARP project for electrophysiological simulation of cardiac tissue (G. Plank, et al.)
  - Parallel PCG-AMG solver for elliptic subproblem of a virtual heart simulation
  - Bidomain equations on a 3D unstructured FEM mesh
  - Up to 25 million unknowns

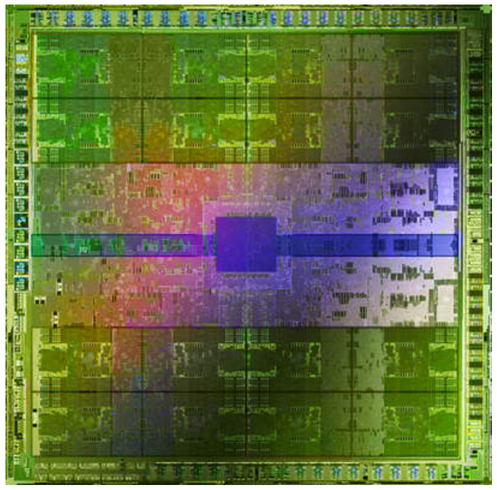




(2) Parallel PCG-AMG Solver Performance

- **CPU / GPU Hardware for the Benchmarks**
  - *kepler*: 16x AMD Opteron 248 @ 2.2 GHz with 32 GB RAM Infiniband
  - *quad2*: 4x AMD Opteron 8347 @ 1.9 GHz with 32 GB RAM
  - *mgser1*: 2x Intel Xeon E5405 @ 2.0 GHz with 8 GB RAM and 1x Nvidia Tesla C1060
  - *gtx*: AMD Phenom 9950 @ 2.6 GHz with 8 GB RAM and 4x Nvidia GTX 280
  - *gpusrv1*: Intel Core i7 965 @ 3.2 GHz with 12 GB RAM and 4x Nvidia GTX 295
  - *fermi*: Intel Core i7 920 @ 2.66 GHz with 12 GB RAM and 2x Nvidia GTX 480

- **GPU Computing Hardware**
  - *mgser1*: 1x Nvidia Tesla C1060 (240 cores / 4 GB on-board RAM)
  - *gtx*: 4x Nvidia Geforce GTX 280 (960 cores / 4 GB on-board RAM)
  - *gpusrv1*: 4x Nvidia Geforce GTX 295 (1,920 cores / 7 GB on-board RAM)
  - *fermi*: 2x Nvidia Geforce GTX 480 (960 cores / 3 GB on-board RAM)

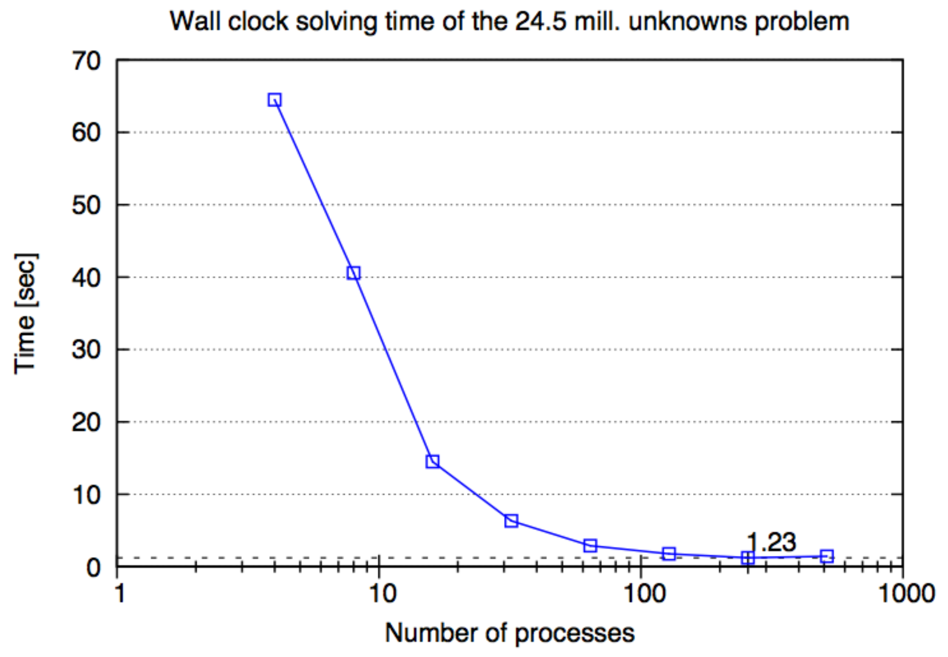


PCG-AMG Solver Performance: Strong Scaling

#cores	kepler	quad2	mgser1 cpu	gtx cpu	gpusrv1 cpu	mgser1 gpu	gtx gpu	gpusrv1 gpu	fermi gpu
1	29.239	30.253	22.615	17.026	9.607	**1.217**	1.016	1.238	0.691
2	14.428	15.954	11.999	9.709	5.662		0.612	0.726	**0.411**
4	7.305	7.544	8.490	**6.562**	**3.885**		**0.367**	0.409	
8	3.607	4.054	**8.226**		4.105			**0.284**	
16	1.909	**3.493**							
32	**1.167**								
Speedup	**25.05**	8.66	2.75	2.59	2.47	1.00	2.77	**4.36**	1.68
Efficiency	**0.78**	0.54	0.34	0.65	0.62	1.00	0.69	0.54	**0.84**
All/1 gpu	1.69	5.05	**11.90**	9.50	5.62	1.76	0.53	0.41	0.59
1/1 gpu	42.31	**43.78**	32.73	24.64	13.90	1.76	1.47	1.79	1.00
All/All gpu	4.11	12.30	**28.96**	23.11	13.68	4.29	1.29	1.00	1.45
1/All gpu	102.95	**106.53**	79.63	59.95	33.83	4.29	3.58	4.36	2.43

Table 1: Parallel PCG-AMG solver: Strong scaling with 1 million unknowns

CPU Virtual Heart CARP Benchmark



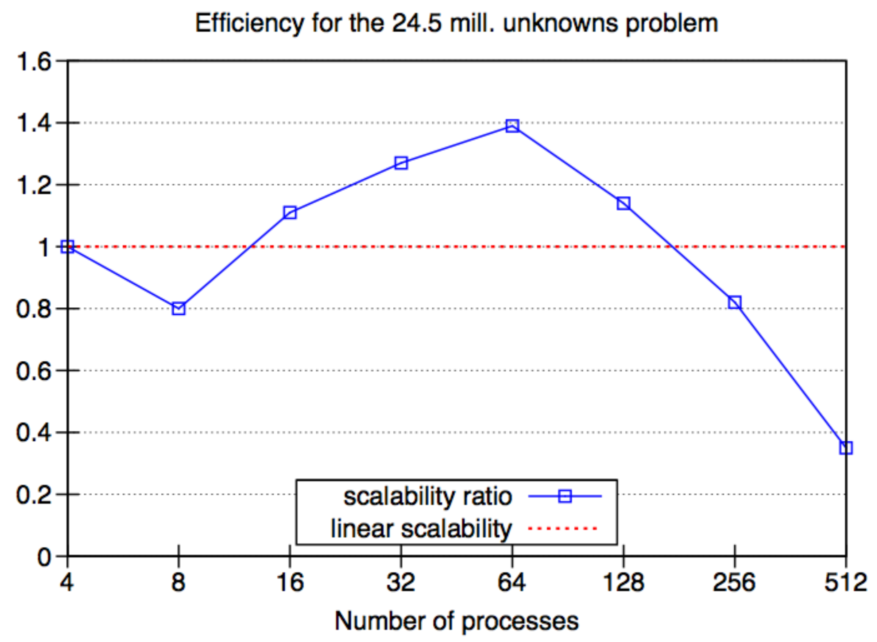
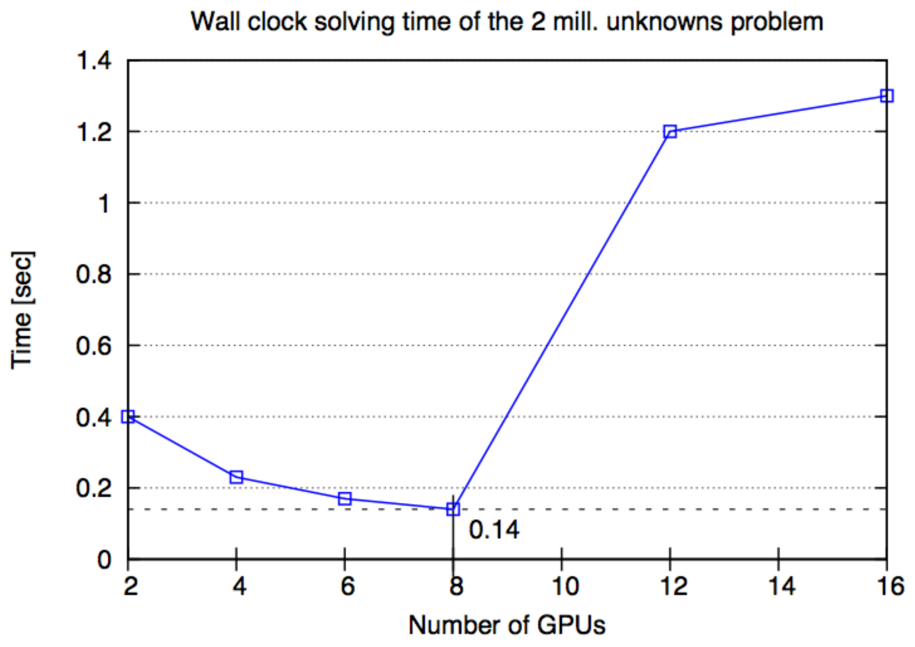


Figure 1: CARP simulator: Strong scaling with 25 million unknowns with up to 512 IBM Power6 CPU cores. Best time: **1.23 sec** [256 CPU cores] (21 iterations)

GPU Virtual Heart CARP Benchmark



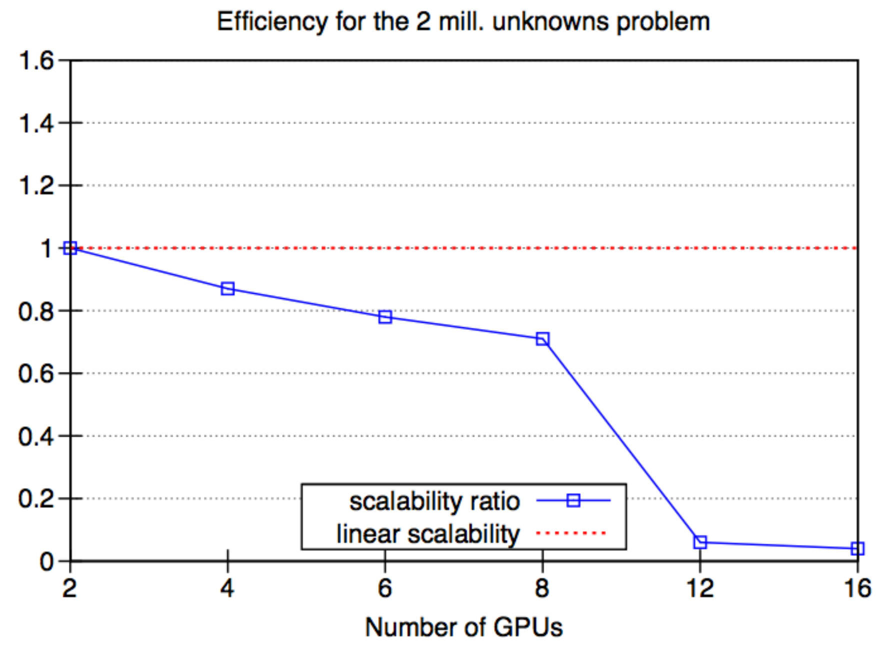


Figure 2: CARP simulator: Strong scaling with 2 million unknowns with up to 8 Nvidia GTX 295 dual GPU boards. Best time: **0.14 sec** [8 GPUs]. 2 Intel Core i7 965 @ 3.2GHz. Best time: **3.60 sec** [8 CPU cores] (20 iterations)

(3) Parallel Toolbox Software

- **Parallel Toolbox**
  - `http://paralleltoolbox.sourceforge.net/`
  - Object oriented C++ code
  - Communicator class handles all data exchange for parallel linear algebra kernels
  - Optimized parallel CPU/GPU solver components: PCG, AMG
  - Flexible and modular design for building complex parallel solvers

Communicator Class

The communicator is derived from a domain decomposition based parallelization approach.

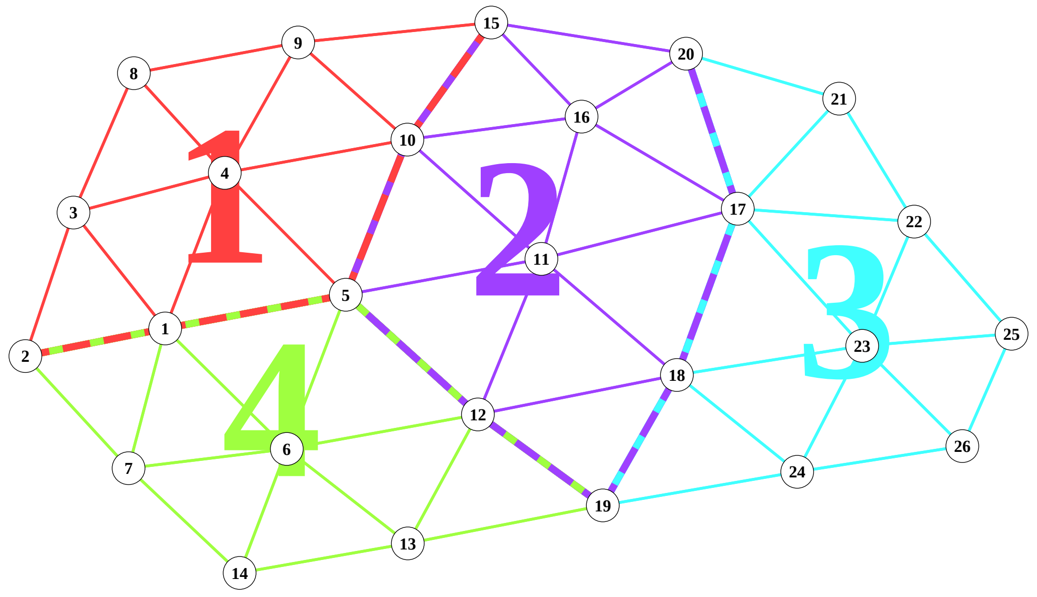


Figure 3: Simple finite element mesh distributed to four processors with global node numbers and color-coded processor domains.

Parallel communication is handled by a communicator object using MPI all-to-all communication patterns. Basic parallel linear algebra routines can be build with the sequential routines and the communicator object.

- **Parallel linear algebra basics**
  - Accumulated vector: $\mathfrak{r}, \mathfrak{s}$ (fraktur font)
  - Distributed vector: r, s (sans-serif font)
  - Accumulated matrix: $\mathfrak{A}, \mathfrak{B}$
  - Distributed matrix: A, B

- **Matrix-vector multiplication and scalar product**
  - Multiplication: $\mathsf{r} \leftarrow \mathsf{A}\mathfrak{s}$, $\mathsf{s} \leftarrow \mathfrak{B}\mathsf{r}$
  - Scalar product: $\sigma \leftarrow \mathfrak{S}(\mathfrak{r}, \mathsf{s}) \equiv \mathfrak{S}(\mathsf{r}, \mathfrak{s})$

- **Accumulation and distribution**
  - Accumulation: $\mathfrak{r} \Leftarrow \mathsf{r}$ *Communication!*
  - Distribution: $\mathsf{r} \Leftarrow \mathfrak{r}$

Essential Communication Routines

Accumulation $\mathfrak{r} \Leftarrow \mathsf{r}$ is the most important communication routine in the Parallel Toolbox. This is the only place where MPI all-to-all communication takes place within linear algebra calculations. The accumulation routine provides a single point to optimize communication performance.

Furthermore, distribution of a vector $\mathsf{r} \Leftarrow \mathfrak{r}$ does not require any communication and is a local operation.

Calculating the global value of a scalar product $\sigma \leftarrow \mathfrak{S}(\mathfrak{r}, \mathsf{s})$ requires s simple MPI all-gather operation and accumulation of a single value. Scalar products are expensive because they enforce a synchronization point in the parallel code path.

(4) Sequential Algebraic Multigrid Algorithm

- **Main ingredients of the algebraic multigrid setup**
  - Coarse and fine node selection process: $I = C \cup F$
  - Construction of prolongation P and restriction R operators
  - Triple matrix product: $A_c = RAP$

Coarsening Algorithm

Simplified Ruge-Stüben based coarsening algorithm using the strong connection concept.

```
$C \leftarrow \emptyset, F \leftarrow \emptyset, T \leftarrow I$
while $T \neq \emptyset$ do
    Find next node $i \in T$
    $C \leftarrow C \cup \{i\}$
    $F \leftarrow F \cup \{j \in I | j \notin C \cup F \wedge i \neq j \wedge |A_{ij}| > \epsilon |A_{ii}|\}$
    $T \leftarrow T \backslash (C \cup F)$
end while
```

Prolongation Operator

$$P = \begin{pmatrix} \mathbf{1}_{CC} \\ P_{FC} \end{pmatrix} \tag{1}$$

Define the number of strongly coupled coarse grid nodes with respect to the fine grid node $i \in F$:

$$n_i := \#\{j \in C || A_{ij}| > \epsilon |A_{ii}|\} \tag{2}$$

The matrix P_{FC} is then defined as:

$$(P_{FC})_{ij} := \begin{cases} 1/n_i, & |A_{ij}| > \epsilon |A_{ii}| \\ 0, & \text{else} \end{cases} \tag{3}$$

Matrix Graph Representations

$$A = \begin{pmatrix} 2 & -1 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & -1 & 2 \end{pmatrix} \tag{4}$$

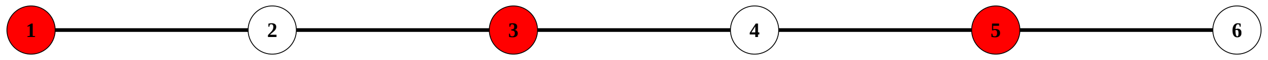


Figure 4: The undirected matrix graph of the 1D Laplace operator.

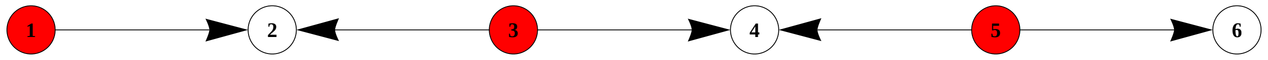


Figure 5: The directed graph of the prolongation operator.

Sequential Classic Multigrid Algorithm

```
Require: $f_l, u_l, r_l, s_l, A_l, R_l, P_l, S_l, T_l, \quad 0 \leq l < L$
  $f_0 \leftarrow f$
  if $l < L$ then
    $u_l \leftarrow 0$ {Initial guess}
    $u_l \leftarrow S_l(u_l, f_l)$ {Presmoothing}
    $r_l \leftarrow f_l - A_l u_l$ {Calculation of residual}
    $f_{l+1} \leftarrow R_l r_l$ {Restriction of residual}
    multigrid(l+1) {Multigrid recursion}
    $s_l \leftarrow P_l u_{l+1}$ {Prolongation of correction}
    $u_l \leftarrow u_l + s_l$ {Addition of correction}
    $u_l \leftarrow T_l(u_l, f_l)$ {Postsmoothing}
  else
    $u_L \leftarrow A_L^{-1} f_L$ {Coarse grid solver}
  end if
  $u \leftarrow u_0$
```

ω-Jacobi and Forward/Backward Gauss-Seidel Smoother

ω-Jacobi Smoother

```
Require: $A_l, D \leftarrow \mathrm{diag}(A_l), \omega \in (0,1]$
  return $u_l \leftarrow u_l + \omega D^{-1}(f_l - A_l u_l)$
```

Forward/Backward Gauss-Seidel Smoother

```
Require: $A_l, D \leftarrow \mathrm{diag}(A_l)$
  return $u_l \overset{\text{forward}}{\longleftarrow} u_l + D^{-1}(f_l - A_l u_l)$
```

```
Require: $A_l, D \leftarrow \mathrm{diag}(A_l)$
  return $u_l \overset{\text{backward}}{\longleftarrow} u_l + D^{-1}(f_l - A_l u_l)$
```

Preconditioned Conjugate Gradient Algorithm

```
Require: $m \in \mathbb{N}, \alpha, \beta, \sigma, \sigma_f, \sigma_l, \tau \in \mathbb{R}, r, s, v, w$
  $r \leftarrow f - Au$ {Calculation of residual}
  $s \leftarrow Pr$ {Apply preconditioner}
  $\sigma \leftarrow S(s, r)$ {Scalar product}
  $\sigma_f \leftarrow \sigma_l \leftarrow \sigma, \quad m \leftarrow 0$
  while $m < M \wedge \sigma/\sigma_f > \epsilon^2$ do
    $m \leftarrow m + 1, \quad v \leftarrow As$
    $\tau \leftarrow S(s, v)$ {Scalar product}
    $\alpha \leftarrow \sigma/\tau$
    $u \leftarrow u + \alpha s$ {Update solution}
    $r \leftarrow r - \alpha v$ {Update residual}
    $w \leftarrow Pr$ {Apply preconditioner}
    $\sigma \leftarrow S(w, r)$ {Scalar product}
    $\beta \leftarrow \sigma/\sigma_l, \quad \sigma_l \leftarrow \sigma$
    $s \leftarrow \beta s + w$ {Update search direction}
  end while
```

(5) Parallel Algebraic Multigrid Algorithm

- **Challenges of the parallel algebraic multigrid setup**
  - Parallel coarse and fine node selection must be globally consistent
  - Skeleton based coarsening equivalent to sequential algorithm with reordering
  - Local prolongation and restriction operators without communication requirements

Parallel Coarsening Algorithm

```
Construct the boundary node set $B$ and the submatrix $A_{BB}$
(a)
$C_B^p \leftarrow \emptyset, F_B^p \leftarrow \emptyset, T_B^p \leftarrow B$
while $T_B^p \neq \emptyset$ do
    Find next node $i \in T_B^p$
    $C_B^p \leftarrow C_B^p \cup \{i\}$
    $F_B^p \leftarrow F_B^p \cup \{j \in B | j \notin C_B^p \cup F_B^p \wedge i \neq j \wedge |A_{BB;ij}| > \epsilon |A_{BB;ii}|\}$
    $T_B^p \leftarrow T_B^p \setminus (C_B^p \cup F_B^p)$
end while
(b)
$C^p \leftarrow I^p \cap C_B^p, F^p \leftarrow I^p \cap F_B^p, T^p \leftarrow I^p \cap C_B^p$
while $T^p \neq \emptyset$ do
    Find next node $i \in T^p$
    $F^p \leftarrow F^p \cup \{j \in I^p | j \notin C^p \cup F^p \wedge i \neq j \wedge |A_{ij}^p| > \epsilon |A_{ii}^p|\}$
    $T^p \leftarrow T^p \setminus \{i\}$
end while
(c)
$T^p \leftarrow I^p \setminus (C^p \cup F^p)$
```

```
**while** $T^p \neq \emptyset$ **do**
    Find next node $i \in T^p$
    $C^p \leftarrow C^p \cup \{i\}$
    $F^p \leftarrow F^p \cup \{j \in I^p | j \notin C^p \cup F^p \wedge i \neq j \wedge |A^p_{ij}| > \epsilon |A^p_{ii}|\}$
    $T^p \leftarrow T^p \setminus (C^p \cup F^p)$
**end while**
```

Parallel Coarsening: Step (a)

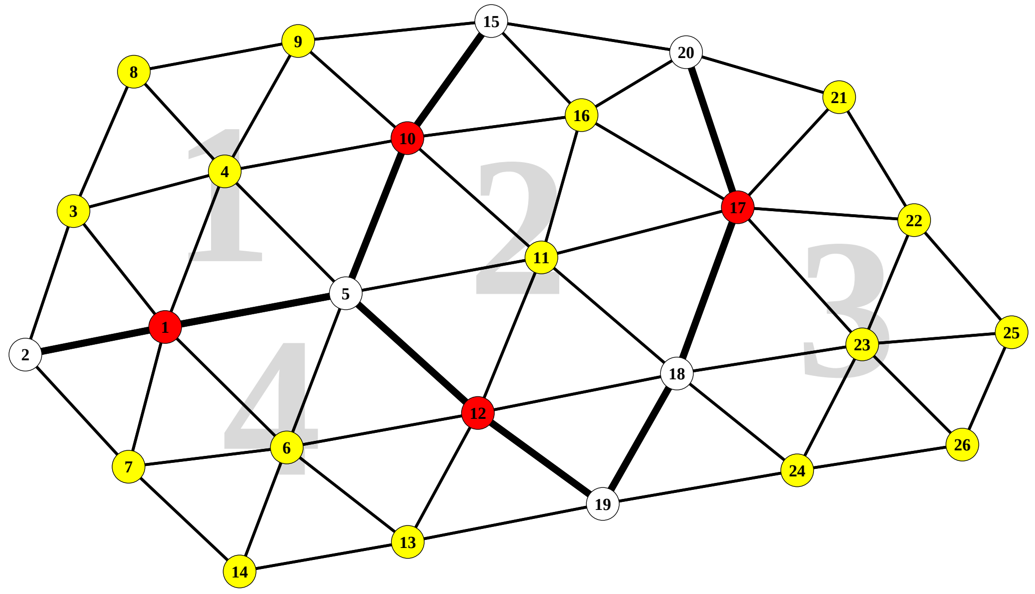


Figure 6: Parallel Coarsening Step (a): The boundary nodes are assigned either red coarse grid nodes or white fine grid nodes. The unassigned nodes in the inner of the processor domains are depicted in yellow color.

Parallel Coarsening: Step (b)

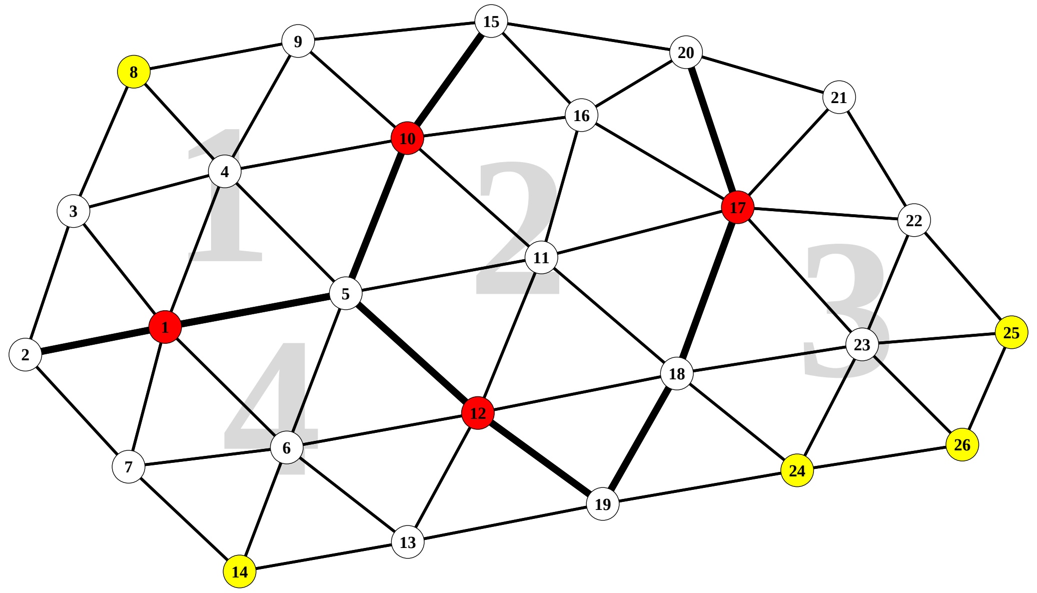


Figure 7: Parallel Coarsening Step (b): Fine grid nodes in the inner of the processor domains depending on the coarse boundary nodes are assigned.

Parallel Coarsening: Step (c)

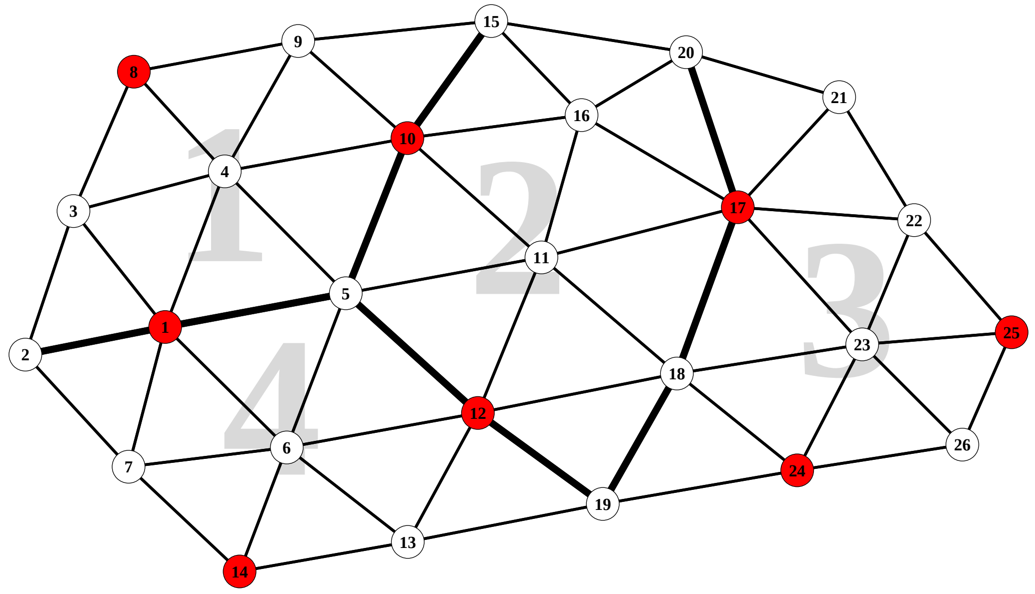


Figure 8: Parallel Coarsening Step (c): The coarsening is completed on the remaining nodes in the inner of the processor domains.

Parallel Prolongation Operator

The local prolongation operator on processor p requires *foreign* coarse grid nodes C^{*p} to interpolate the boundary fine grid nodes F^{*p}.

```
$F^{*p} \leftarrow F^p \cap B^p$
Construct the submatrix $A_{F^{*p}C}$
$C^{*p} \leftarrow \{j \in C | \exists i \in F^{*p} : |A_{F^{*p}C;ij}| > \epsilon |A_{F^{*p}C;ii}|\} \backslash C^p$
$J^p \leftarrow C^p \cup C^{*p}$
Construct $P^p := P_{I^p J^p}$ using $A_{F^{*p}C}$ and $A^p$
```

Interpolation on the Finite Element Mesh

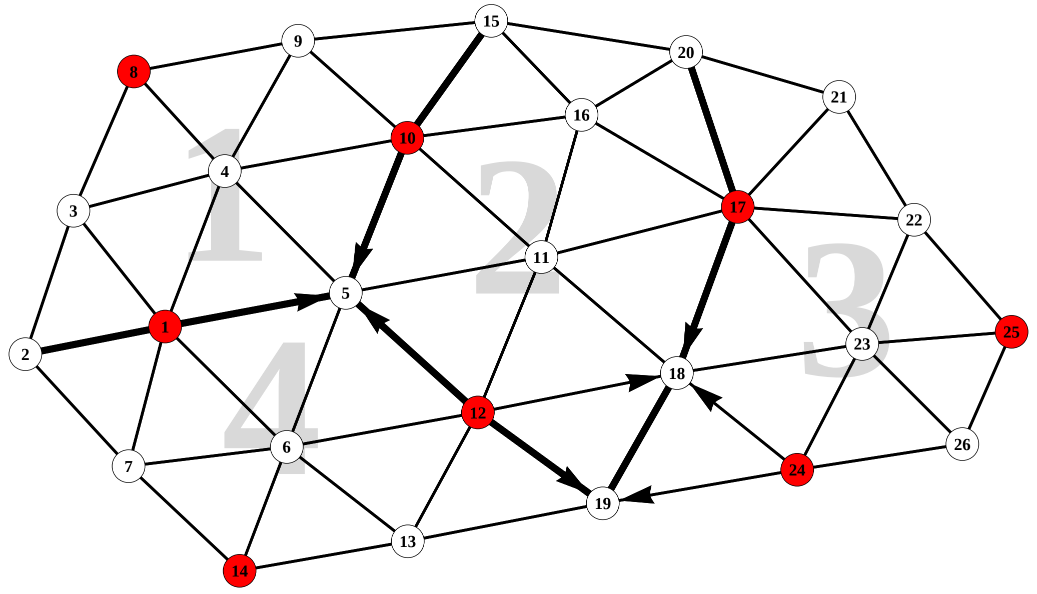


Figure 9: White fine grid nodes are interpolated by red coarse grid nodes. The black arrows show the boundary crossing interpolation for the three fine grid nodes 5, 18, and 19.

Global Prolongation Operator

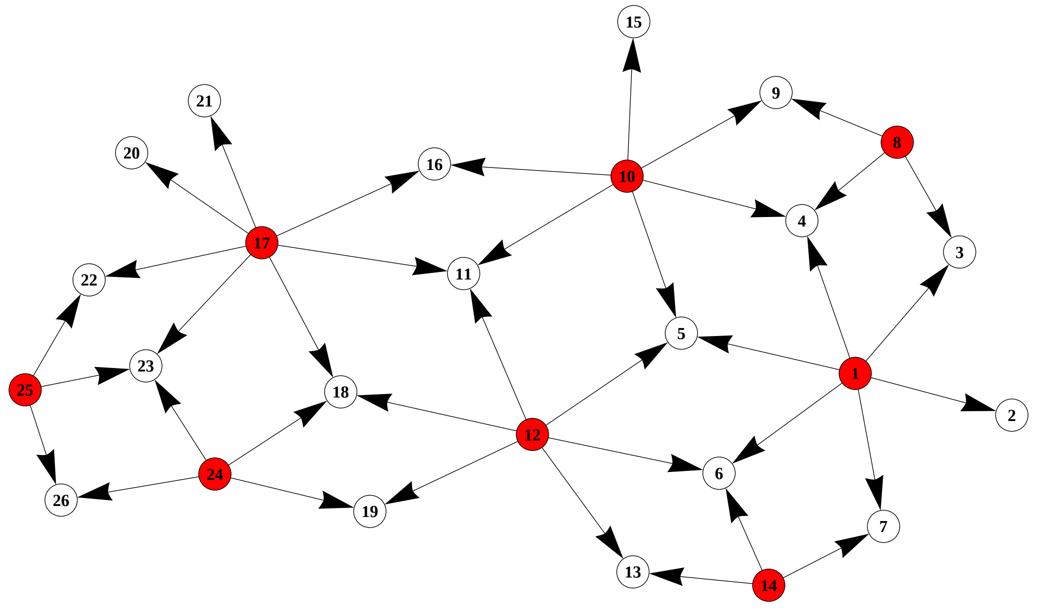


Figure 10: The global prolongation operator on the whole simulation domain represented as a directed graph.

Interpolation on the Color-Coded Finite Element Mesh

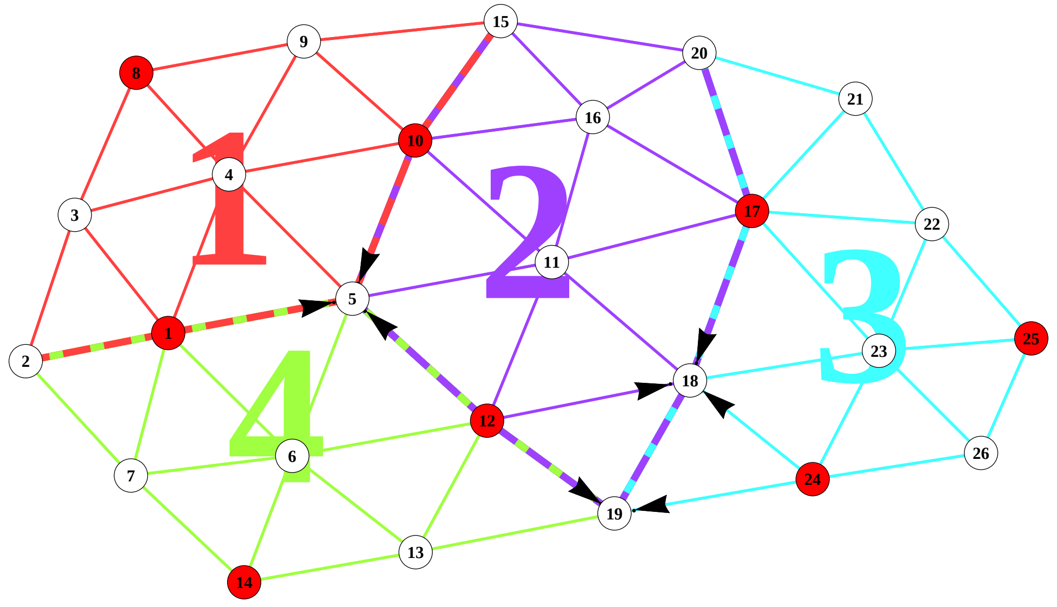


Figure 11: The processor domains embedded in the finite element mesh are shown in color. The cross-boundary interpolation is depicted for the fine grid nodes 5, 18, and 19.

Color-Coded Global Prolongation Operator

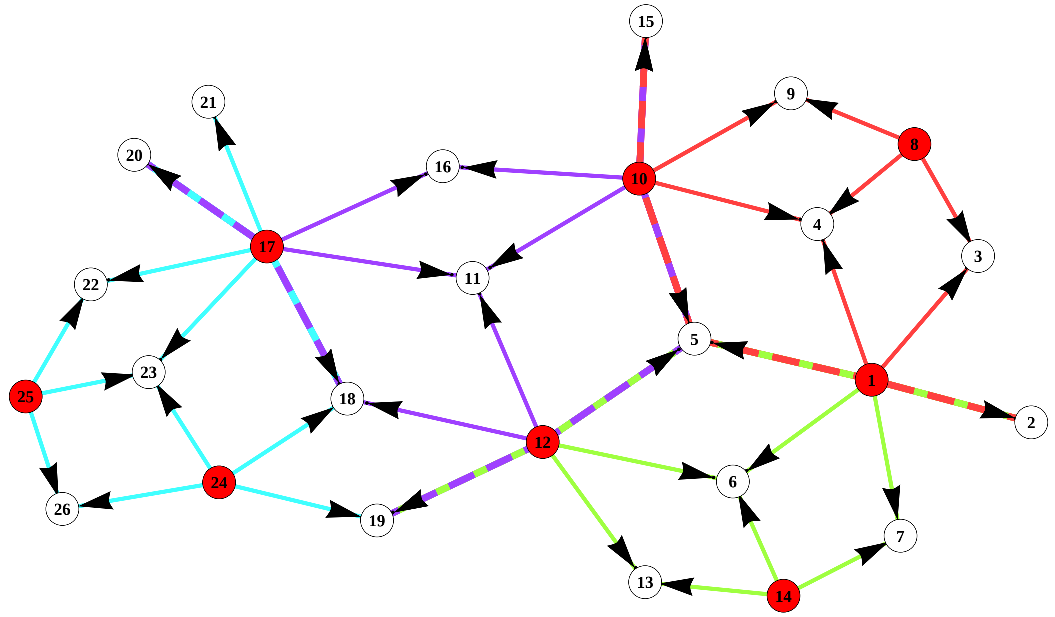


Figure 12: The global prolongation operator on the whole simulation domain represented as a color-coded directed graph.

Local Prolongation Operator on Processor 1

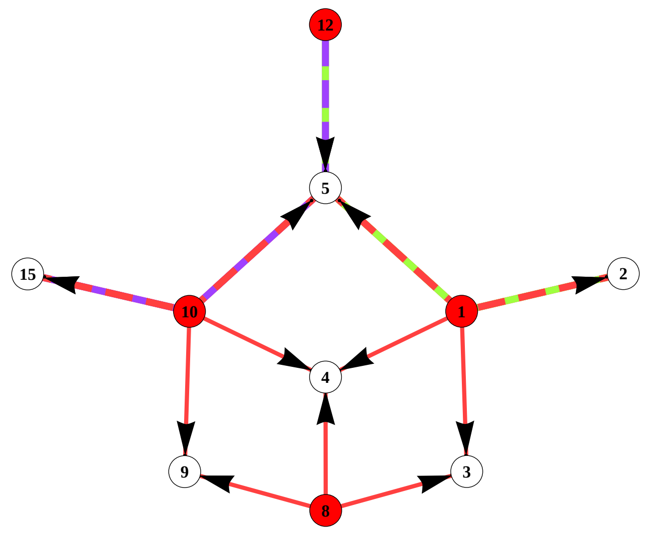


Figure 13: The local prolongation operator on processor one represented as colorcoded directed graph. Note the foreign coarse grid node 12 and the crossboundary interpolation of node 5.

Local Prolongation Operator on Processor 2

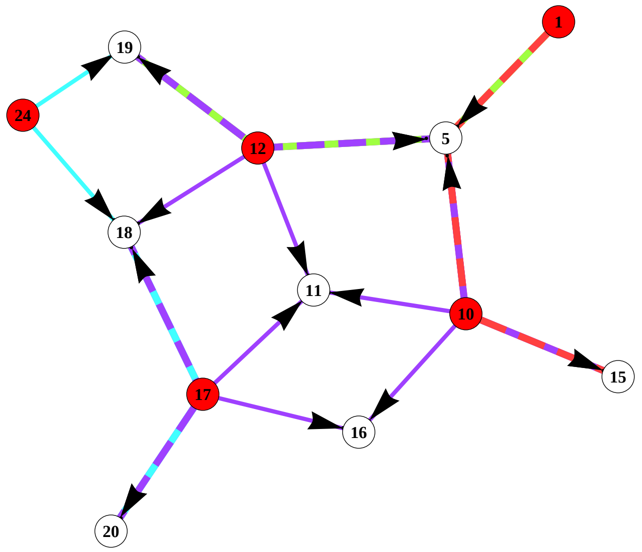


Figure 14: The local prolongation operator on processor two represented as colorcoded directed graph. Note the foreign coarse grid nodes 1 and 24 and the cross-boundary interpolation of the nodes 5, 18, and 19.

Local Prolongation Operator on Processor 3

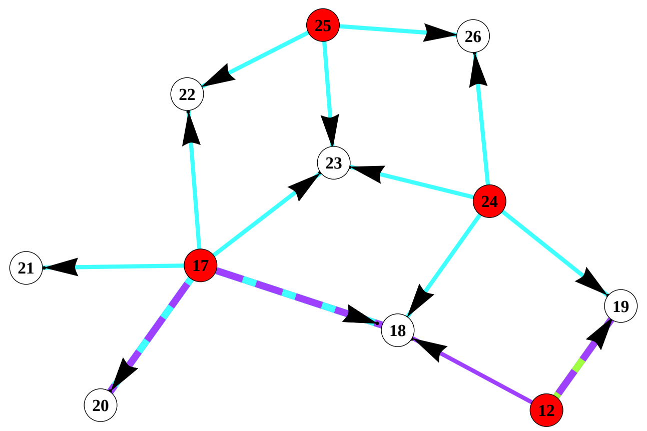


Figure 15: The local prolongation operator on processor three represented as colorcoded directed graph. Note the foreign coarse grid node 12 and the crossboundary interpolation of the nodes 18 and 19.

Local Prolongation Operator on Processor 4

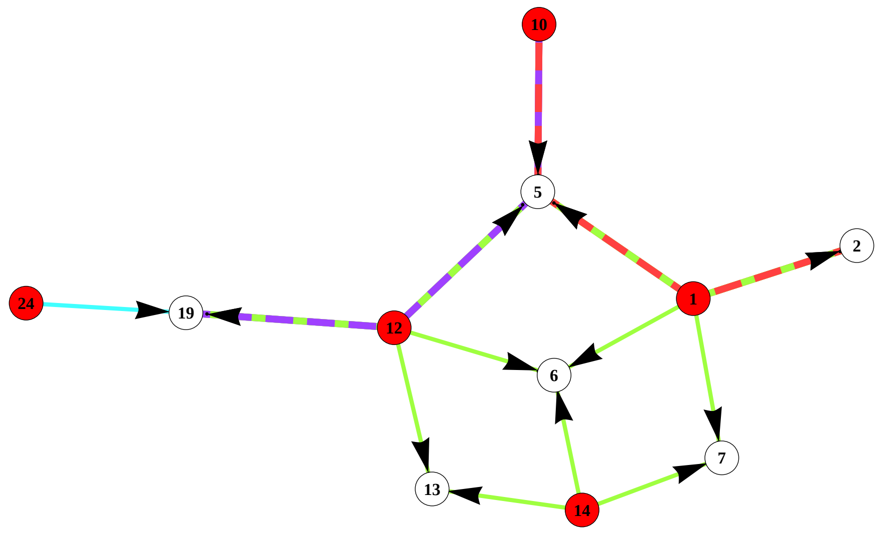


Figure 16: The local prolongation operator on processor four represented as colorcoded directed graph. Note the foreign coarse grid nodes 10 and 24 and the cross-boundary interpolation of the nodes 5 and 19.

Parallel Classic Multigrid Algorithm

```
Require: $\mathsf{f}_l, \mathfrak{u}_l, \mathsf{r}_l, \mathfrak{s}_l, R_l, P_l, S_l, T_l, \quad 0 \le l < L$
  $\mathsf{f}_0 \leftarrow \mathsf{f}$
  if $l < L$ then
    $\mathfrak{u}_l \leftarrow 0$ {Initial guess}
    $\mathfrak{u}_l \leftarrow S_l(\mathfrak{u}_l, \mathfrak{f}_l)$ {Presmoothing}
    $\mathsf{r}_l \leftarrow \mathsf{f}_l - \mathsf{A}_l\mathfrak{u}_l$ {Calculation of residual}
    $\mathsf{f}_{l+1} \leftarrow R_l\mathsf{r}_l$ {Restriction of residual}
    multigrid(l+1) {Multigrid recursion}
    $\mathfrak{s}_l \leftarrow P_l\mathfrak{u}_{l+1}$ {Prolongation of correction}
    $\mathfrak{u}_l \leftarrow \mathfrak{u}_l + \mathfrak{s}_l$ {Addition of correction}
    $\mathfrak{u}_l \leftarrow T_l(\mathfrak{u}_l, \mathsf{f}_l)$ {Postsmoothing}
  else
    $\mathfrak{u}_L \Leftarrow (\mathfrak{A}_L)^{-1}\mathsf{f}_L$ {Coarse grid solver}
  end if
  $\mathfrak{u} \leftarrow \mathfrak{u}_0$
```

Parallel ω-Jacobi and Forward/Backward Gauss-Seidel Smoother

Parallel ω-Jacobi Smoother

```
**Require:** $A_l^p, \mathfrak{D} \Leftarrow \operatorname{diag}(A_l^p), \omega \in (0,1]$
  $\mathfrak{v}_l^p \Leftarrow (\mathsf{f}_l^p - A_l^p \mathfrak{u}_l^p)$
  **return** $\mathfrak{u}_l^p \leftarrow \mathfrak{u}_l^p + \omega \mathfrak{D}^{-1} \mathfrak{v}_l^p$
```

Parallel Forward/Backward Gauss-Seidel Smoother

```
**Require:** $A_l^p, \mathfrak{D} \Leftarrow \operatorname{diag}(A_l^p), \omega \in (0,1]$
  $\mathfrak{v}_l^p \Leftarrow (\mathsf{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On boundary nodes}
  $\mathfrak{u}_l^p \leftarrow \mathfrak{u}_l^p + \omega \mathfrak{D}^{-1} \mathfrak{v}_l^p$ {On boundary nodes}
  $\mathfrak{u}_l^p \xleftarrow{\text{forward}} \mathfrak{u}_l^p + \mathfrak{D}^{-1}(\mathsf{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On inner nodes}
  **return** $\mathfrak{u}_l^p$
```

```
**Require:** $A_l^p, \mathfrak{D} \Leftarrow \operatorname{diag}(A_l^p), \omega \in (0,1]$
  $\mathfrak{u}_l^p \xleftarrow{\text{backward}} \mathfrak{u}_l^p + \mathfrak{D}^{-1}(\mathsf{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On inner nodes}
  $\mathfrak{v}_l^p \Leftarrow (\mathsf{f}_l^p - A_l^p \mathfrak{u}_l^p)$ {On boundary nodes}
  $\mathfrak{u}_l^p \leftarrow \mathfrak{u}_l^p + \omega \mathfrak{D}^{-1} \mathfrak{v}_l^p$ {On boundary nodes}
  **return** $\mathfrak{u}_l^p$
```

Parallel Conjugate Gradient Algorithm

```
**Require:** $m \in \mathbb{N}, \alpha, \beta, \sigma, \sigma_f, \sigma_l, \tau \in \mathbb{R}, \mathsf{v}, \mathsf{r}, \mathfrak{s}, \mathfrak{w}$
  $\mathsf{r} \leftarrow \mathsf{f} - \mathsf{A}\mathfrak{u}$ {Calculation of residual}
  $\mathfrak{s} \leftarrow \mathfrak{P}\mathsf{r}$ {Apply preconditioner}
  $\sigma \leftarrow \mathfrak{S}(\mathfrak{s}, \mathsf{r})$ {Scalar product}
  $\sigma_f \leftarrow \sigma_l \leftarrow \sigma, \quad m \leftarrow 0$
  **while** $m < M \wedge \sigma/\sigma_f > \epsilon^2$ **do**
      $m \leftarrow m + 1, \quad \mathsf{v} \leftarrow \mathsf{A}\mathfrak{s}$
      $\tau \leftarrow \mathfrak{S}(\mathfrak{s}, \mathsf{v})$ {Scalar product}
      $\alpha \leftarrow \sigma/\tau$
      $\mathfrak{u} \leftarrow \mathfrak{u} + \alpha\mathfrak{s}$ {Update solution}
      $\mathsf{r} \leftarrow \mathsf{r} - \alpha\mathsf{v}$ {Update residual}
      $\mathfrak{w} \leftarrow \mathfrak{P}\mathsf{r}$ {Apply preconditioner}
      $\sigma \leftarrow \mathfrak{S}(\mathfrak{w}, \mathsf{r})$ {Scalar product}
      $\beta \leftarrow \sigma/\sigma_l, \quad \sigma_l \leftarrow \sigma$
      $\mathfrak{s} \leftarrow \beta\mathfrak{s} + \mathfrak{w}$ {Update search direction}
  **end while**
```

(6) Parallelization on GPU-Accelerated Hybrid Architectures

- **Parallelization Concepts on CPU and GPU clusters for PCG-AMG**
  - Coarse grained MPI parallelization for CPU and GPU nodes
  - Additional fine grained parallelization on GPUs
  - Many-core implementation of sparse matrix-vector multiplication
  - Interleaved CRS data format for *coalesced memory access* on GPU
  - Nvidia CUDA Technology for C/C++ GPU programming

Sparse Matrix-Vector Multiplication Kernel

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 Multiplication 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!

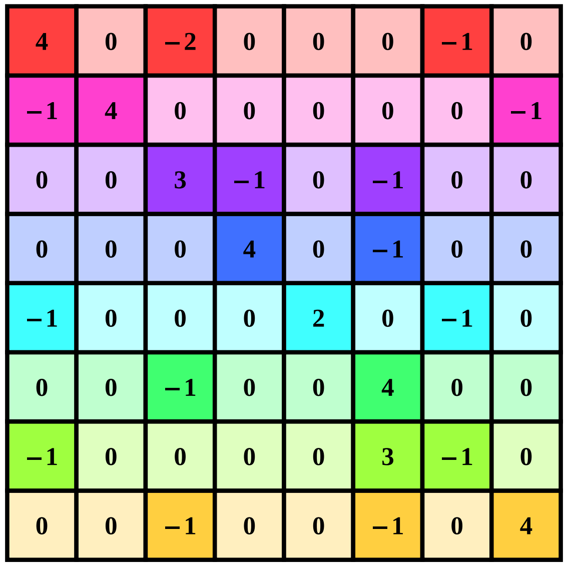
Problems and Solutions

- **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 the texture unit of the GPU for random access to b vector
  - Texturing is optimized for spacial locality: Small read-only cache

GPU-Accelerated Sparse Matrix-Vector Multiplication

An efficient sparse matrix-vector multiplication is key to the PCG-AMG solver performance.

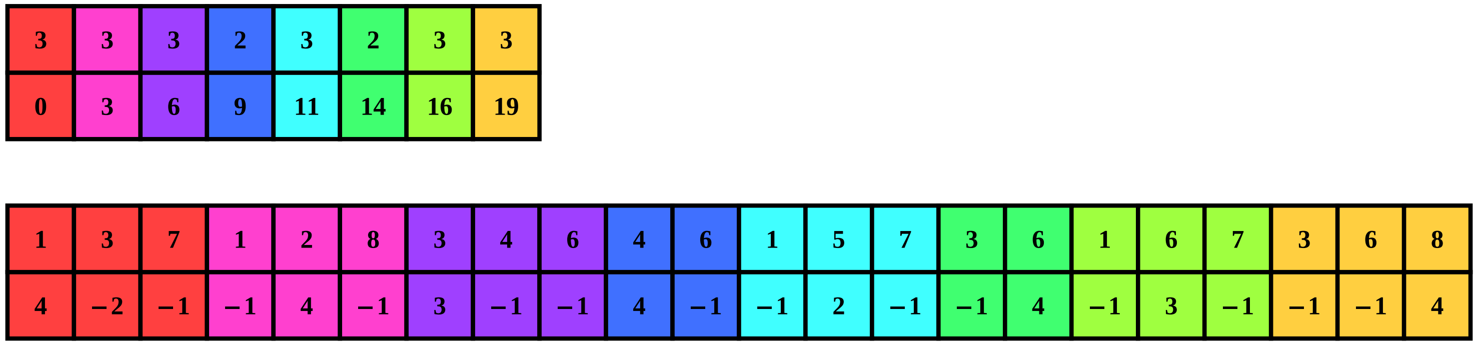


4	0	−2	0	0	0	−1	0
−1	4	0	0	0	0	0	−1
0	0	3	−1	0	−1	0	0
0	0	0	4	0	−1	0	0
−1	0	0	0	2	0	−1	0
0	0	−1	0	0	4	0	0
−1	0	0	0	0	3	−1	0
0	0	−1	0	0	−1	0	4

Figure 17: A sample matrix with the rows colored in different hues.

Compressed Row Storage Data Format (CRS)

A flexible data format for sparse matrices.



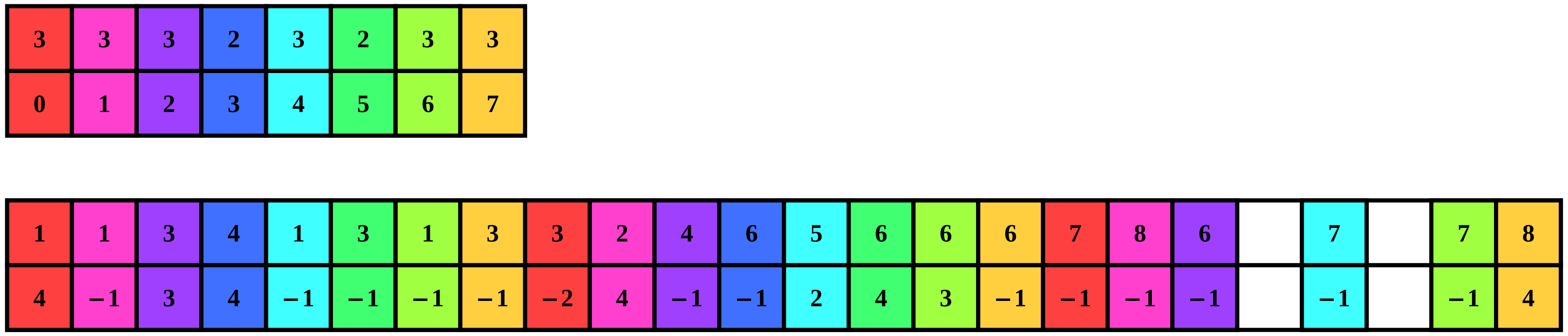
3	3	3	2	3	2	3	3
0	3	6	9	11	14	16	19

1	3	7	1	2	8	3	4	6	4	6	1	5	7	3	6	1	6	7	3	6	8
4	−2	−1	−1	4	−1	3	−1	−1	4	−1	−1	2	−1	−1	4	−1	3	−1	−1	−1	4

Figure 18: CRS data structure for the sample matrix with the count and displacement vector on top and the column indices and matrix entries below.

Interleaved Compressed Row Storage Data Format (ICRS)

Coalesced memory access patterns on GPUs are required to achieve high performance.



3	3	3	2	3	2	3	3
0	1	2	3	4	5	6	7

1	1	3	4	1	3	1	3	3	2	4	6	5	6	6	6	7	8	6		7		7	8
4	−1	3	4	−1	−1	−1	−1	−2	4	−1	−1	2	4	3	−1	−1	−1	−1		−1		−1	4

Figure 19: ICRS data structure for the sample matrix with the count and displacement vector on top and the interleaved column indices and matrix entries below. The eight interleaved matrix rows create holes in the data structure represented by the white boxes.

CUDA Code Sample for Sparse Matrix-Vector Multiplication

```
#define L 256
struct linear_operator_params {
    const int *cnt;       //ICRS count vector
    const int *dsp;       //ICRS displacement vector
    const int *col;       //ICRS column indices
    const double *ele;    //ICRS matrix entries
    const double *u;      //Input vector
    double *v;            //Output vector
    int n;                //Matrix dimension
};
texture<int2> tex_u;

void _device_linear_operator(int *cnt, int *dsp, int *col, double *ele,
    int m, int n, double *u, double *v)
{
    cudaBindTexture(0, tex_u, (int2*)u, sizeof(double) * m); //Bind the texture

    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); //GPU kernel launch

    cudaUnbindTexture(tex_u);                                //Unbind the texture
}
```

```
__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 + L * parms.cnt[j];
        double s = 0.0;
        for(unsigned int i = blkStart; i < blkStop; i += L)
        {
            unsigned int q = parms.col[i];          //Load column index
            double a = parms.ele[i];                //Load matrix entry
            int2 c = tex1Dfetch(tex_u, q);          //Load vector entry using texture mapping
            double b = __hiloint2double(c.y, c.x);  //Convert texture entries to double number
            s += a * b;                             //Calculate the sparse scalar product
        }
        parms.v[j] = s;                             //Store the sparse scalar product
    }
}
```

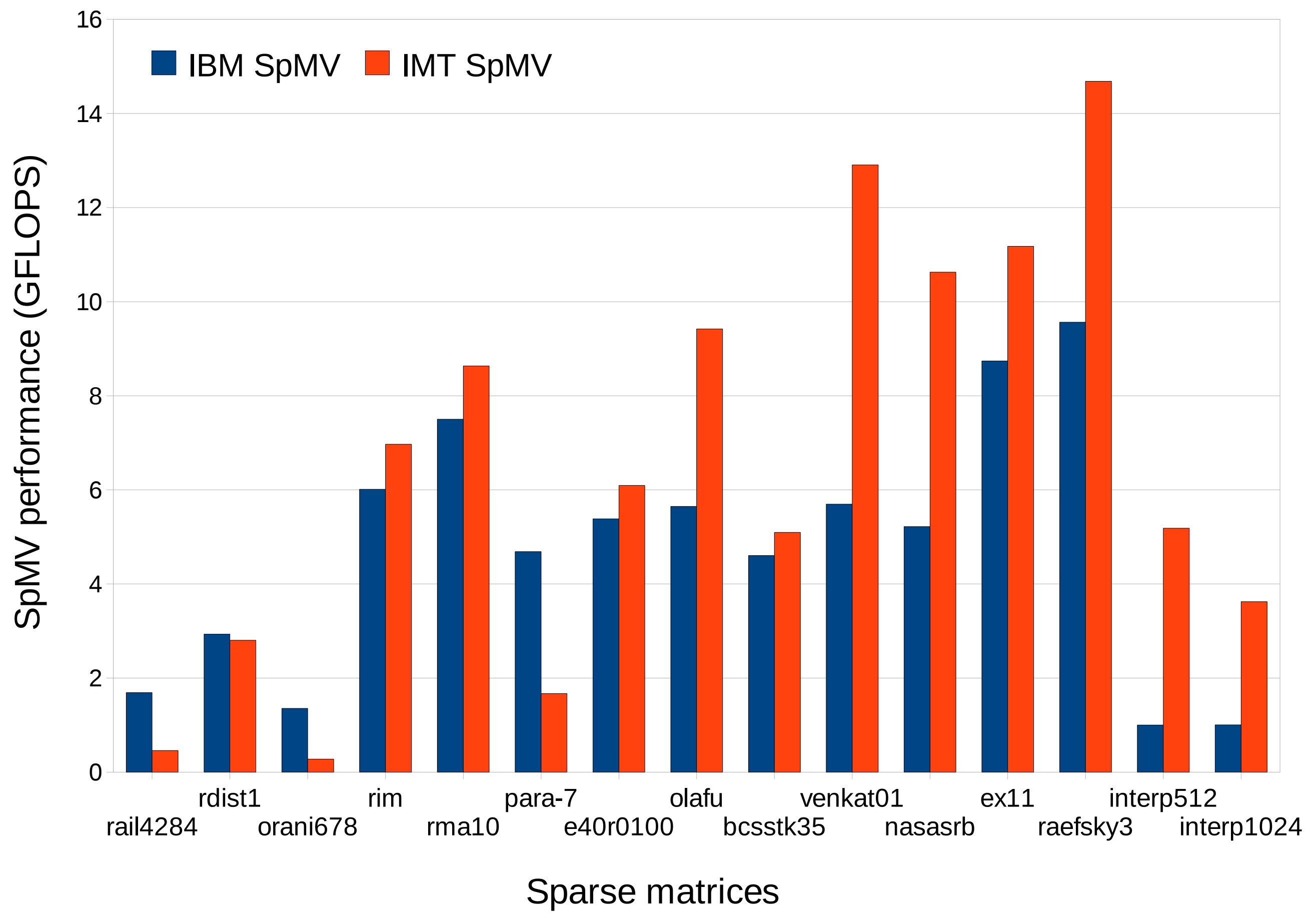


Figure 20: Performance comparison of IBM SpMV and IMT SpMV based on ICRS data format on GeForce 8800 GTX

Conclusions

- **GPU-Accelerated Hybrid Architectures**
  - GPUs have entered main stream high performance computing
  - TOP 500: 1. Tianhe-1A (GPU), 2. Jaguar (CPU), 3. Nebulae (GPU)
  - Many applications can profit from GPU acceleration now
  - Algorithms and data structures have to be adapted
  - *Flops are free, memory access is expensive!*

Part II

Large Scale Simulations of the Euler Equations on GPU Clusters

Overview

- The Vijayasundaram Method for Multi-Physics Euler Equations
- ARMO CPU/GPU Algorithms
- ARMO CPU/GPU Benchmarks

(1) The Vijayasundaram Method for Multi-Physics Euler Equations

The Euler equations are given by a system of differential equations. We consider two gas species with densities ρ_1 and ρ_2 for the simulations and ideal gas state equations. More complicated and realistic state equation can also be handled by the ARMO simulation code.

Let ρ_1, ρ_2 be the densities of the gas species and $\rho = \rho_1 + \rho_2$ the density of the gas, p the pressure, and p_1, p_2, p_3 the components of the gas momentum density, and E the total energy density. Let $\mathbf{x} = \{x_1, x_2, x_3\} \in \Omega \subset \mathbb{R}^3$ and $t \in (0, T) \subset \mathbb{R}$ be the space time coordinates. Then the conserved quantity $\mathbf{w}(\mathbf{x}, t)$ is given by

$$\mathbf{w} = \begin{pmatrix} \rho_1 \\ \rho_2 \\ p_1 \\ p_2 \\ p_3 \\ E \end{pmatrix} \tag{5}$$

and the flux vectors are defined as

$$\mathbf{f}_k(\mathbf{w}) = \begin{pmatrix} \rho_1 p_k/\rho \\ \rho_2 p_k/\rho \\ p_1 p_k/\rho + \delta_{1k} p \\ p_2 p_k/\rho + \delta_{2k} p \\ p_3 p_k/\rho + \delta_{3k} p \\ (E+p) p_k/\rho \end{pmatrix}, \quad k \in \{1,2,3\} \tag{6}$$

The Euler equations on the domain $\Omega \times (0,T)$ can then be expressed as

$$\frac{\partial}{\partial t}\mathbf{w}(\mathbf{x},t) + \frac{\partial}{\partial x_1}\mathbf{f}_1(\mathbf{w}(\mathbf{x},t)) + \frac{\partial}{\partial x_2}\mathbf{f}_2(\mathbf{w}(\mathbf{x},t)) + \frac{\partial}{\partial x_3}\mathbf{f}_3(\mathbf{w}(\mathbf{x},t)) = 0 \tag{7}$$

and together with suitable boundary conditions the system can be solved with the finite volume approach.

The finite volume method can be formulated by applying Green's theorem

$$\frac{d}{dt}\int_{\Omega} \mathbf{w}(\mathbf{x},t)d\mathbf{x} = -\int_{\partial\Omega} \mathbf{f}_1 n_1 + \mathbf{f}_2 n_2 + \mathbf{f}_3 n_3 dS \tag{8}$$

where $\mathbf{n} = (n_1, n_2, n_3)$ denotes the outer normal to the boundary $\partial\Omega$. The discrete version is then derived by integration over a time intervall $[t_n, t_n + \Delta t]$ and averaging over the cells K_i.

$$\mathbf{w}^{(n+1)}|_{K_i} = \mathbf{w}^{(n)}|_{K_i} - \Delta t \sum_{j\in S(i)} \frac{|\Gamma_{ij}|}{|K_i|} \sum_{k=1}^{3} \mathbf{F}_{k,\Gamma_{ij}}(\mathbf{w}^{(n)}|_{K_i}, \mathbf{w}^{(n)}|_{K_j}) n_k \tag{9}$$

With a tetrahedral approximation to $\Omega \approx \{K_i\}_{i\in I}$ and Γ_{ij} are the interfaces between the cells K_i, K_j and the set $S(i)$ stores the indices of the neighboring cells of K_i

The **Vijayasundaram method** defines the fluxes as

$$\mathbf{F}_{k,\Gamma_{ij}}(\mathbf{u},\mathbf{v}) = \mathbb{A}_k^+\left(\frac{\mathbf{u}+\mathbf{v}}{2}\right)\mathbf{u} + \mathbb{A}_k^-\left(\frac{\mathbf{u}+\mathbf{v}}{2}\right)\mathbf{v}, \quad k = 1,2,3 \tag{10}$$

The essence of the Vijayasundaram method is the calculation of an eigenspace decomposition of $\mathbb{A}_k = d\mathbf{f}_k/d\mathbf{w}, k = 1,2,3$ into positive and negative subspaces. Thus the matrices $\mathbb{A}_k^+$, $\mathbb{A}_k^-$ are constructed from the positive and negative eigenvalues of $\mathbb{A}_k = \mathbb{R}_k\Lambda_k\mathbb{L}_k$ with $\Lambda_k = \mathrm{diag}(\lambda_{k,1},\ldots,\lambda_{k,6})$ and $k = 1,2,3$.

$$\mathbb{A}_k^\pm = \mathbb{R}_k\Lambda_k^\pm\mathbb{L}_k, \tag{11}$$

$$\Lambda_k^\pm = \mathrm{diag}(\lambda_{k,1}^\pm,\ldots,\lambda_{k,m}^\pm), \tag{12}$$

$$\lambda_{k,i}^+ = \max(\lambda_{k,i},0), \quad \lambda_{k,i}^- = \min(\lambda_{k,i},0), \quad i = 1,\ldots,6 \tag{13}$$

(2) ARMO CPU/GPU Algorithms

High level parallel CPU algorithm:

```
Require: $f, g, com, nei, geo, pio$
Require: $t_{max}, i_{max}, C, \sigma, m, n$
  $t \leftarrow 0, i \leftarrow 0$
  while $t < t_{max}$ and $i < i_{max}$ do
    exchange$(m, n, f, g, com)$
    mpi_alltoall$(m, n, g, f)$
    vijaya$(n, nei, geo, pio, f, g, \sigma)$
    mpi_allreduce_max$(\sigma)$
    update$(n, f, g, \sigma, C)$
    $i \leftarrow i + 1$
    $t \leftarrow t + C/\sigma$
  end while
```

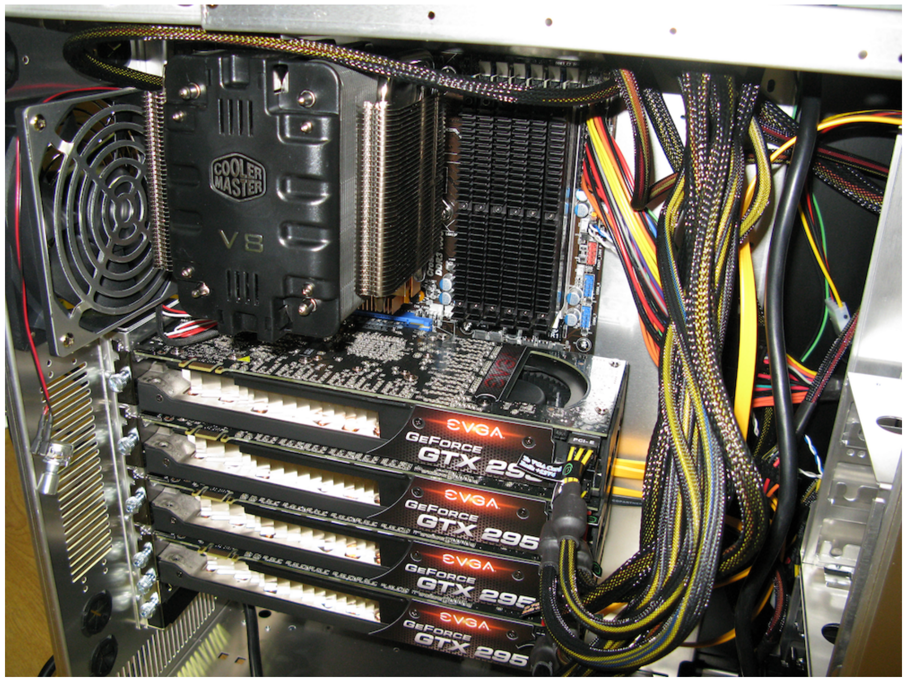
High level parallel GPU algorithm:

```
**Require:** $f_D, g_D, com_D, nei_D, geo_D, pio_D, \sigma_D$
**Require:** $t_{max}, i_{max}, C, \sigma, m, n, snd, rcv$
  $t \leftarrow 0, i \leftarrow 0$
  **while** $t < t_{max}$ **and** $i < i_{max}$ **do**
    $\texttt{exchange}_D(m, n, f_D, g_D, com_D)$
    $\texttt{device\_to\_host}(n, g_D, snd)$
    $\texttt{mpi\_alltoall}(snd, rcv)$
    $\texttt{host\_to\_device}(n, f_D, rcv)$
    $\texttt{vijaya}_D(n, nei_D, geo_D, pio_D, f_D, g_D, \sigma_D)$
    $\texttt{device\_to\_host}(\sigma_D, \sigma)$
    $\texttt{mpi\_allreduce\_max}(\sigma)$
    $\texttt{host\_to\_device}(\sigma_D, \sigma)$
    $\texttt{update}_D(n, f_D, g_D, \sigma_D, C)$
    $i \leftarrow i + 1$
    $t \leftarrow t + C/\sigma$
  **end while**
```

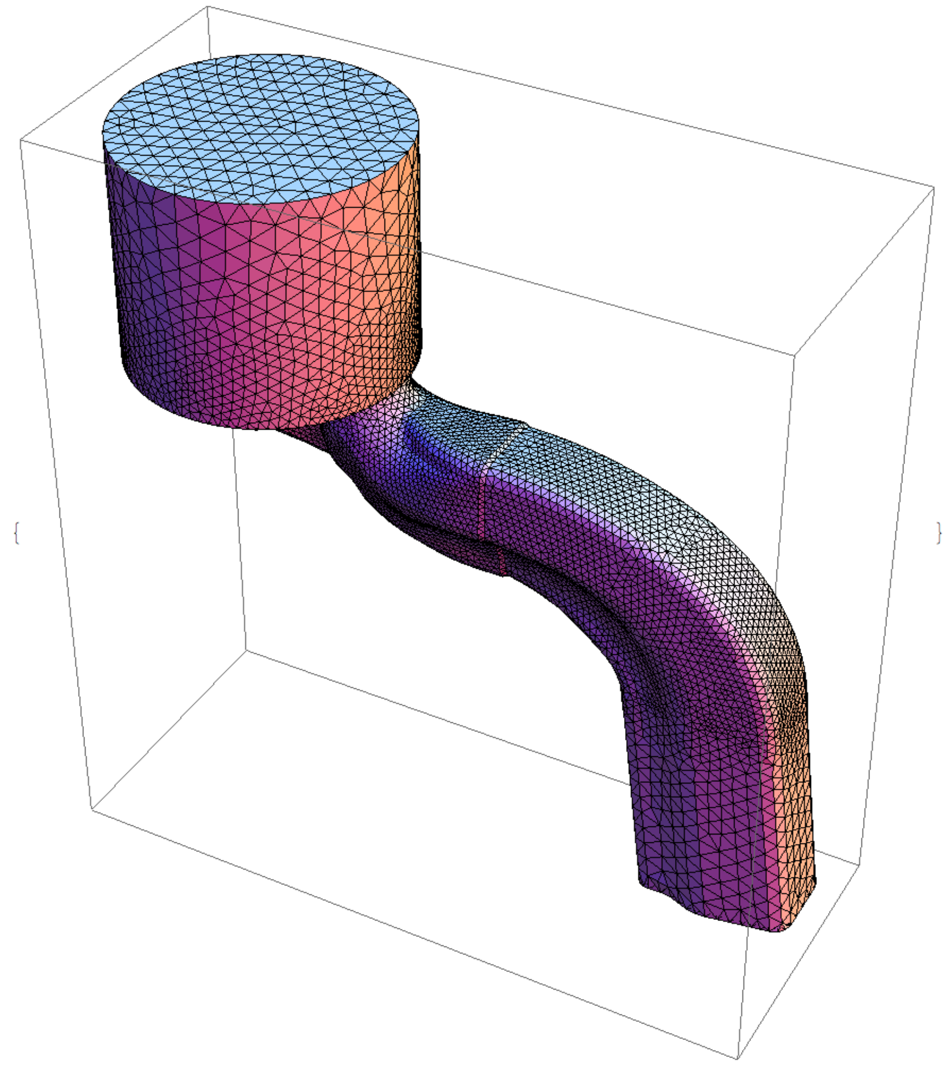
(3) ARMO CPU/GPU Benchmarks

- **CPU / GPU Hardware for the Benchmarks**
  - *memo*: 8x Intel Xeon X7560 @ 2.27 GHz with 1024 GB RAM
  - *quad2*: 4x AMD Opteron 8347 @ 1.9 GHz with 32 GB RAM
  - *gtx*: AMD Phenom 9950 @ 2.6 GHz with 8 GB RAM and 4x Nvidia GTX 280
  - *ro2009*: Intel Core i7 920 @ 2.66 GHz with 6 GB RAM and Nvidia GTX 280
  - *iscsergpu*: 8x Intel Core i7 965 @ 3.2 GHz with 12 GB RAM and 32x Nvidia GTX 295
  - *penge*: 12x Dual Intel Xeon E5450 @ 3.0 GHz with 16 GB RAM
  - *fermi*: Intel Core i7 920 @ 2.66 GHz with 12 GB RAM and 2x Nvidia GTX 480
  - *tesla*: 2x Intel Xeon E5620 @ 2.4 GHz with 48 GB RAM and 2x Tesla C2050
  - *sandy*: Intel Core i7 2600K @ 3.4 GHz with 16 GB RAM and 1x Nvidia GTX 580

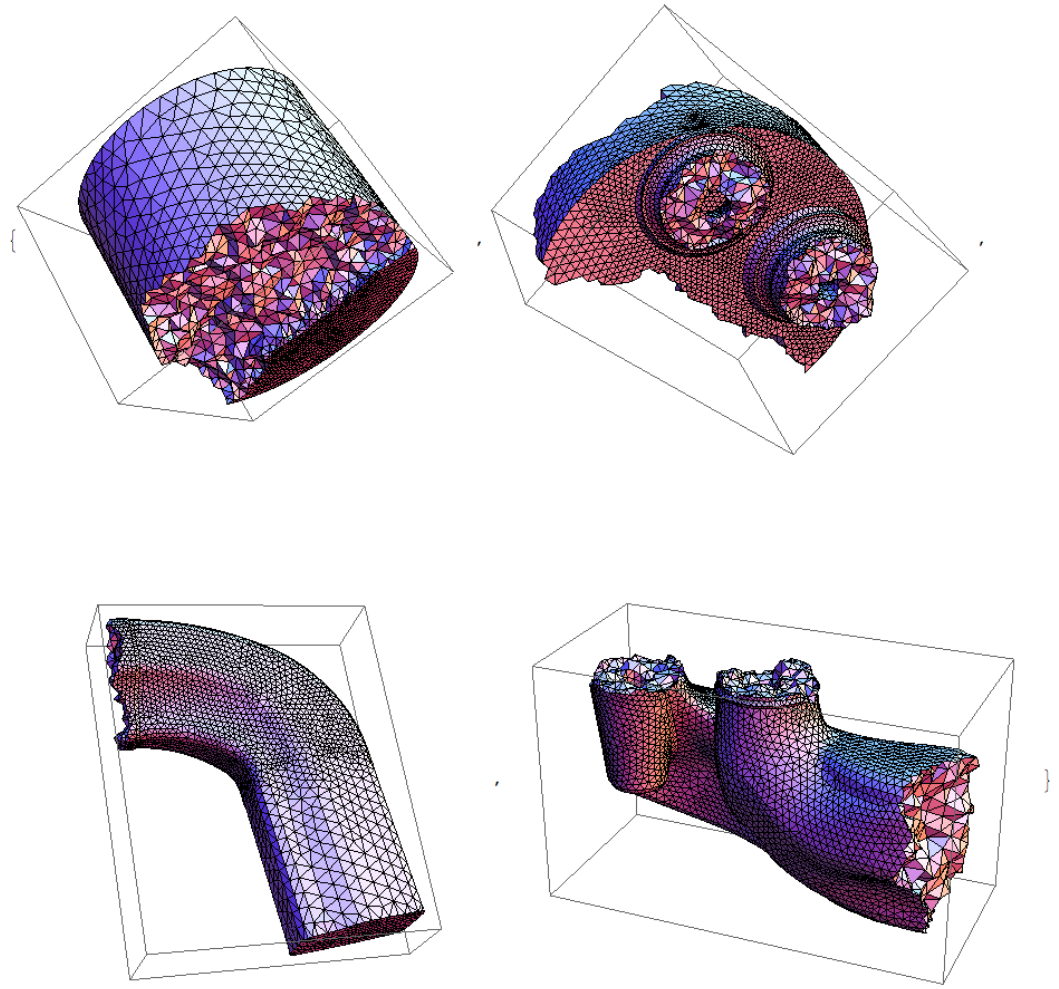
- **GPU Computing Hardware**
  - *gtx*: 4x Nvidia Geforce GTX 280 (960 cores / 4 GB on-board RAM)
  - *ro2009*: Nvidia Geforce GTX 280 (240 cores / 1 GB on-board RAM)
  - *iscsergpu*: 32x Nvidia Geforce GTX 295 (15,360 cores / 56 GB on-board RAM)
  - *fermi*: 2x Nvidia Geforce GTX 480 (960 cores / 3 GB on-board RAM)
  - *tesla*: 2x Nvidia Tesla C2050 (896 cores / 6 GB on-board ECC RAM)
  - *sandy*: 1x Nvidia Geforce GTX 580 (512 cores / 1.5 GB on-board RAM)



Benchmark example: Intake port of a diesel engine with 155,325 elements.



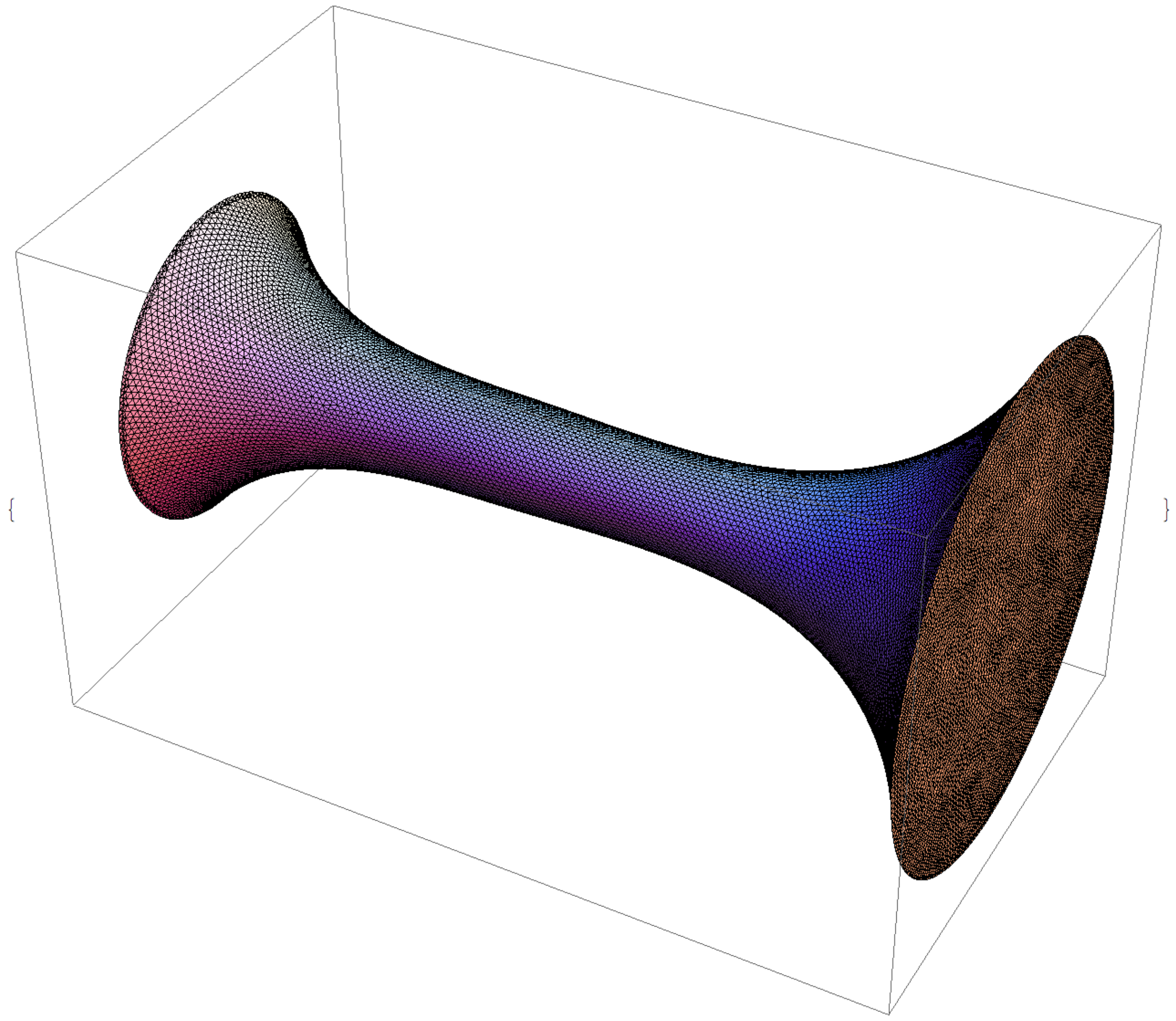
Four pieces of the intake port for parallel processing using domain decomposition.



CPU cores	memo	quad2	gtx	ro2009	iscsergpu	penge	fermi	tesla	sandy
1	12.348	33.58	19.37	10.84	9.32	11.74	10.37		6.56
2	5.942	16.07	9.26	5.28	4.55	5.08	5.02		3.27
4	2.960	7.59	4.47	2.66	2.29	2.47	2.54		1.76
8	1.442	3.13		2.14	1.81 [1]	1.268 [1]	2.11		1.54
16	0.682	1.38			1.09 [2]	0.641 [2]			
32	0.350				0.65 [4]	0.330 [4]			
64	0.181					**0.174** [8]			
Speedup	**68.22**	24.21	4.33	5.07	14.34	67,47	4.91		4.26
Efficiency	1.07	**1.51**	1.08	0.63	0.45	1.05	0.61		0.53
GPU cores	memo	quad2	gtx	ro2009	iscsergpu	penge	fermi	tesla	sandy
1			0.284	0.254	0.380		0.156	0.205	0.125
2			0.141		0.175		0.090	0.116	
4			0.086		0.098				
8					**0.069** [1]				
Speedup			3.30	1.00	**5.51**		1.73	1.76	1.00
Efficiency			0.82	1.00	0.69		0.86	**0.88**	1.00

Table 2: Parallel scalability benchmark for an intake-port with 155,325 elements.

Benchmark example: Nozzle with 642,700, 2,570,800, and 10,283,200 elements.



CPU cores	quad2	gtx	ro2009	iscsergpu	fermi	tesla	sandy
1	135.80	79.65	49.54	40.62	47.41		31.77
2	65.85	38.55	24.54	20.13	23.50		16.35
4	32.73	19.06	12.45	10.23	11.89		8.78
8	15.67		9.76	7.86 [1]	9.41		7.17
16	7.61			4.22 [2]			
32				**2.42** [4]			
Speedup	**19.06**	4.13	4.87	17.27	5.04		4.43
Efficiency	**1.19**	1.03	0.61	0.54	0.63		0.55
GPU cores	**quad2**	**gtx**	**ro2009**	**iscsergpu**	**fermi**	**tesla**	**sandy**
1		1.186	1.048	1.561	0.617	0.829	0.487
2		0.540		0.702	0.312	0.415	
4		0.275		0.337			
8				**0.185** [1]			
Speedup		5.00	1.00	**11.60**	1.98	2.00	1.00
Efficiency		1.25	1.00	**1.45**	0.99	1.00	1.00

Table 3: Parallel scalability benchmark for a nozzle with 642,700 elements.

CPU cores	quad2	gtx	ro2009	iscsergpu	fermi	tesla	sandy
1	415.00	259.89	176.69	142.83	174.55		120.68
2	203.15	128.70	95.04	72.03	85.06		62.67
4	105.69	65.90	45.51	37.27	43.64		34.03
8	55.34		36.52	29.47 [1]	35.17		27.75
16	29.16			14.77 [2]			
32				7.40 [4]			
64				**3.75** [8]			
Speedup	14.23	3.94	4.84	**38.09**	4.96		4.35
Efficiency	0.89	**0.99**	0.60	0.60	0.62		0.54
GPU cores	quad2	gtx	ro2009	iscsergpu	fermi	tesla	sandy
1		3.955	4.180	4.683	2.160	2.705	1.718
2		1.694		2.052	1.082	1.371	
4		0.841		1.002			
8				0.514 [1]			
16				**0.320** [2]			
Speedup		4.70	1.00	**14.63**	2.00	1.97	1.00
Efficiency		**1.18**	1.00	0.91	1.00	0.99	1.00

Table 4: Parallel scalability benchmark for a nozzle with 2,570,800 elements.

CPU cores	quad2	gtx	ro2009	iscsergpu	fermi	tesla	sandy
1	1384.5	916.89	*	508.74	603.83		409.63
2	693.25	462.34	*	257.83	305.15		218.61
4	361.81	238.70	*	132.20	156.57		118.57
8	200.29		*	110.17 [1]	128.98		102.84
16	108.48			55.93 [2]			
32				28.20 [4]			
64				**14.11** [8]			
Speedup	12.76	3.84		**36.05**	4.68		3.98
Efficiency	0.80	**0.96**		0.56	0.59		0.50
GPU cores	quad2	gtx	ro2009	iscsergpu	fermi	tesla	sandy
1		*	*	*	7.896	10.356	6.538
2		6.602		7.619	3.964	5.264	
4		3.088		3.529			
8				1.725 [1]			
16				0.935 [2]			
32				0.701 [4]			
64				**0.495** [8]			
Speedup		2.14		**15.39**	1.99	1.97	1.00
Efficiency		**1.07**		0.48	1.00	0.98	1.00

Table 5: Parallel scalability benchmark for a nozzle with 10,283,200 elements.

Effective GFLOPS for ARMO Simulator

CPU / GPU Hardware	Intake-port 155,325	Nozzle 642,700	Nozzle 2,570,800	Nozzle 10,283,200
Opteron 8347 CPU core	0.636	0.651	0.852	**1.022**
GTX 580 GPU board	170.982	181.592	205.903	**216.422**
GPU cluster	309.75 [8]	478.03 [8]	1105.44 [16]	**2858.52** [64]
Speedup GPU / CPU	268.64	**278.85**	241.56	211.76
Speedup GPU cluster / CPU	486.67	734.05	1296.87	**2796.97**

Table 6: Effective GFLOPS for ARMO simulator.

Conclusions

- **GPUs deliver excellent performance for CFD problems!**
  - $2800\times$ speedup on GPU cluster with 64 GPUs compared to one AMD Opteron core
  - New GPU hardware: Fermi architecture brings further performance improvements
  - 10,000,000 finite volume cells per typical GPU possible
  - Close to 1,000,000,000 cells on GPU cluster with 64 GPUs possible
  - Software: PGI compiler introduces unified programming model for CPUs and GPUs
  - Websites: **www.nvidia.com** and **www.pgroup.com**

Thank you!