Accelerating Double Precision FEM Simulations with GPUs

Dominik Göddeke[1 3] Robert Strzodka[2] Stefan Turek[1]

dominik.goeddeke@math.uni-dortmund.de

[1]Mathematics III: Applied Mathematics and Numerics, University of Dortmund

[2]Computer Science, Stanford University, California

[3]Computer Science VII: Computer Graphics, University of Dortmund

ASIM 2005 – 18th Symposium on Simulation Technique
Workshop Parallel Computing and Graphics Processors
Erlangen, Germany, September 12th-15th 2005

Overview

Overview

Overview

Overview

Motivation and related work

As described in previous talk: GPUs exhibit the attractive potential to accelerate FEM simulations by at least one order of magnitude.

Available *half* (s10e5) and *single* (s23e8 NVIDIA, s16e7 ATI, IEEE 754 on CPUs) floating point formats do not provide sufficient accuracy for numerical simulations in technical applications, e.g. quantitatively correct drag and lift values in virtual wind tunnels compared to qualitatively correct flow field calculations.

Native double precision support in hardware is unlikely to appear in the medium-term: Not required for graphics purposes.

Possible approach: Emulate double through single precision (essentially doubling the mantissa, so-called **single-double**). Feasibility in a GPU context currently being analyzed in the GAIA project (Lawrence Livermore Labs), drawback: Factor 10 increase in number of operations.

Motivation and related work

As described in previous talk: GPUs exhibit the attractive potential to accelerate FEM simulations by at least one order of magnitude.

Available *half* (s10e5) and *single* (s23e8 NVIDIA, s16e7 ATI, IEEE 754 on CPUs) floating point formats do not provide sufficient accuracy for numerical simulations in technical applications, e.g. quantitatively correct drag and lift values in virtual wind tunnels compared to qualitatively correct flow field calculations.

Native double precision support in hardware is unlikely to appear in the medium-term: Not required for graphics purposes.

Possible approach: Emulate double through single precision (essentially doubling the mantissa, so-called **single-double**). Feasibility in a GPU context currently being analyzed in the GAIA project (Lawrence Livermore Labs), drawback: Factor 10 increase in number of operations.

Our approach

Context: Use the GPU as a co-processor for computationally intensive components of FEM solvers, e.g. iterative solvers for linear systems.

Revitalize **Mixed Precision Defect Correction** techniques, combining the high performance of the GPU and the high accuracy of the CPU.

Core algorithm: Outer defect correction loop running in double precision on the CPU, GPU-based iterative solver in single precision works as a **preconditioner** for the outer solver to ensure reducing the defect by few digits per iteration.

Time won by running the preconditioner on the GPU easily outweights the (potentially expensive) data transfers between CPU and GPU and the additional iterations implied by the Richardson approach.

Our approach

Context: Use the GPU as a co-processor for computationally intensive components of FEM solvers, e.g. iterative solvers for linear systems.

Revitalize **Mixed Precision Defect Correction** techniques, combining the high performance of the GPU and the high accuracy of the CPU.

Core algorithm: Outer defect correction loop running in double precision on the CPU, GPU-based iterative solver in single precision works as a **preconditioner** for the outer solver to ensure reducing the defect by few digits per iteration.

Time won by running the preconditioner on the GPU easily outweights the (potentially expensive) data transfers between CPU and GPU and the additional iterations implied by the Richardson approach.

Notation and input parameters

Note:
Reformulated algorithm compared to proceedings (relative instead of absolute termination criteria). Presented results are based on new version and an improved implementation.

Notation and parameters:

- ϵ_{inner} and ϵ_{outer}: stopping criteria for the inner and outer solver.
- $A\mathbf{x} = \mathbf{b}$: given linear system of equations to be solved in double precision.
- α: scaling factor.
- d^0: norm of initial defect for convergence control.
- Subscript $_{32}$: single precision vectors stored in GPU memory as textures.
- Subscript $_{64}$: double precision vectors stored in CPU memory.

The algorithm

1. Set initial values and calculate initial defect: $\alpha_{64} = 1.0$, $\mathbf{x}_{64} =$ initial guess, $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$, $d_{64}^0 = ||\mathbf{d}_{64}||$.
2. Set initial values for inner solver: $\mathbf{d}_{32} = \mathbf{d}_{64}$, $\mathbf{c}_{32} =$ initial guess, $d_{32}^0 = ||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}||$
3. Iterate inner solver until $||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}|| < \epsilon_{inner} \cdot d_{32}^0$.
4. Update outer solution: $\mathbf{x}_{64} = \mathbf{x}_{64} + \alpha_{64}\mathbf{c}_{32}$.
5. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
6. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
7. Check for convergence ($\alpha_{64} < \epsilon_{outer} \cdot d_{64}^0$); otherwise scale defect: $\mathbf{d}_{64} = \frac{1}{\alpha_{64}}\mathbf{d}_{64}$ and continue with step 2.

The algorithm

1. Set initial values and calculate initial defect: $\alpha_{64} = 1.0$, $\mathbf{x}_{64} =$ initial guess, $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$, $d_{64}^0 = ||\mathbf{d}_{64}||$.
2. Set initial values for inner solver: $\mathbf{d}_{32} = \mathbf{d}_{64}$, $\mathbf{c}_{32} =$ initial guess, $d_{32}^0 = ||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}||$
3. Iterate inner solver until $||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}|| < \epsilon_{inner} \cdot d_{32}^0$.
4. Update outer solution: $\mathbf{x}_{64} = \mathbf{x}_{64} + \alpha_{64}\mathbf{c}_{32}$.
5. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
6. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
7. Check for convergence ($\alpha_{64} < \epsilon_{outer} \cdot d_{64}^0$); otherwise scale defect: $\mathbf{d}_{64} = \frac{1}{\alpha_{64}}\mathbf{d}_{64}$ and continue with step 2.

The algorithm

1. Set initial values and calculate initial defect: $\alpha_{64} = 1.0$, $\mathbf{x}_{64} =$ initial guess, $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$, $d_{64}^0 = ||\mathbf{d}_{64}||$.
2. Set initial values for inner solver: $\mathbf{d}_{32} = \mathbf{d}_{64}$, $\mathbf{c}_{32} =$ initial guess, $d_{32}^0 = ||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}||$
3. Iterate inner solver until $||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}|| < \epsilon_{inner} \cdot d_{32}^0$.
4. Update outer solution: $\mathbf{x}_{64} = \mathbf{x}_{64} + \alpha_{64}\mathbf{c}_{32}$.
5. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
6. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
7. Check for convergence ($\alpha_{64} < \epsilon_{outer} \cdot d_{64}^0$); otherwise scale defect: $\mathbf{d}_{64} = \frac{1}{\alpha_{64}}\mathbf{d}_{64}$ and continue with step 2.

The algorithm

1. Set initial values and calculate initial defect: $\alpha_{64} = 1.0$, $\mathbf{x}_{64} =$ initial guess, $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$, $d^0_{64} = ||\mathbf{d}_{64}||$.
2. Set initial values for inner solver: $\mathbf{d}_{32} = \mathbf{d}_{64}$, $\mathbf{c}_{32} =$ initial guess, $d^0_{32} = ||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}||$
3. Iterate inner solver until $||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}|| < \epsilon_{inner} \cdot d^0_{32}$.
4. Update outer solution: $\mathbf{x}_{64} = \mathbf{x}_{64} + \alpha_{64}\mathbf{c}_{32}$.
5. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
6. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
7. Check for convergence ($\alpha_{64} < \epsilon_{outer} \cdot d^0_{64}$); otherwise scale defect: $\mathbf{d}_{64} = \frac{1}{\alpha_{64}}\mathbf{d}_{64}$ and continue with step 2.

The algorithm

1. Set initial values and calculate initial defect: $\alpha_{64} = 1.0$, $\mathbf{x}_{64} =$ initial guess, $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$, $d^0_{64} = ||\mathbf{d}_{64}||$.
2. Set initial values for inner solver: $\mathbf{d}_{32} = \mathbf{d}_{64}$, $\mathbf{c}_{32} =$ initial guess, $d^0_{32} = ||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}||$
3. Iterate inner solver until $||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}|| < \epsilon_{inner} \cdot d^0_{32}$.
4. Update outer solution: $\mathbf{x}_{64} = \mathbf{x}_{64} + \alpha_{64}\mathbf{c}_{32}$.
5. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
6. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
7. Check for convergence ($\alpha_{64} < \epsilon_{outer} \cdot d^0_{64}$); otherwise scale defect: $\mathbf{d}_{64} = \frac{1}{\alpha_{64}}\mathbf{d}_{64}$ and continue with step 2.

The algorithm

1. Set initial values and calculate initial defect: $\alpha_{64} = 1.0$, $\mathbf{x}_{64} =$ initial guess, $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$, $d_{64}^0 = ||\mathbf{d}_{64}||$.
2. Set initial values for inner solver: $\mathbf{d}_{32} = \mathbf{d}_{64}$, $\mathbf{c}_{32} =$ initial guess, $d_{32}^0 = ||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}||$
3. Iterate inner solver until $||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}|| < \epsilon_{inner} \cdot d_{32}^0$.
4. Update outer solution: $\mathbf{x}_{64} = \mathbf{x}_{64} + \alpha_{64}\mathbf{c}_{32}$.
5. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
6. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
7. Check for convergence ($\alpha_{64} < \epsilon_{outer} \cdot d_{64}^0$); otherwise scale defect: $\mathbf{d}_{64} = \frac{1}{\alpha_{64}}\mathbf{d}_{64}$ and continue with step 2.

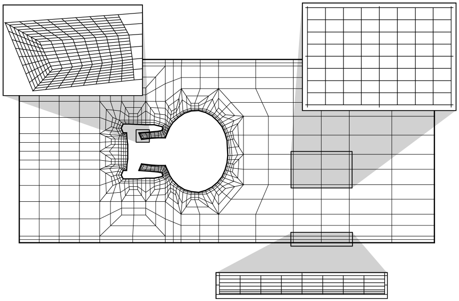
The algorithm

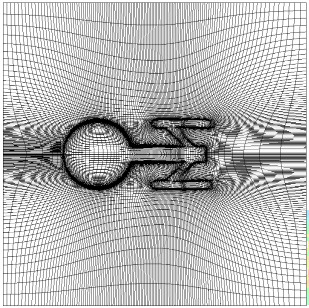
1. Set initial values and calculate initial defect: $\alpha_{64} = 1.0$, $\mathbf{x}_{64} =$ initial guess, $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$, $d^0_{64} = ||\mathbf{d}_{64}||$.
2. Set initial values for inner solver: $\mathbf{d}_{32} = \mathbf{d}_{64}$, $\mathbf{c}_{32} =$ initial guess, $d^0_{32} = ||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}||$
3. Iterate inner solver until $||\mathbf{d}_{32} - A_{32}\mathbf{c}_{32}|| < \epsilon_{inner} \cdot d^0_{32}$.
4. Update outer solution: $\mathbf{x}_{64} = \mathbf{x}_{64} + \alpha_{64}\mathbf{c}_{32}$.
5. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
6. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
7. Check for convergence ($\alpha_{64} < \epsilon_{outer} \cdot d^0_{64}$); otherwise scale defect: $\mathbf{d}_{64} = \frac{1}{\alpha_{64}}\mathbf{d}_{64}$ and continue with step 2.

Remarks

Test setup is strictly prototypical:

- Inner solver: Unpreconditioned conjugate gradients.
- Mesh: Regularly refined cartesian grid due to potential instabilities of CG solver on deformed meshes.
- Goal: Multigrid as inner preconditioner will greatly enhance performance and allow arbitrarily deformed and adapted **generalized tensorproduct meshes**.





Test setup

- Solve Poisson equation $-\Delta \mathbf{u} = \mathbf{f}$: Dirichlet boundary conditions on $\Omega = [0,1]^2$, conforming bilinear Finite Elements (Q_1) for spatial discretization, levels of refinement from $N = 3^2$ to $N = 1025^2$ unknowns.
- Perform error analysis with test function $\mathbf{u}_0(x,y) := x(1-x)y(1-y)$.
- Measure error nodewise with the scaled l_2 norm (approximate L_2) of analytic solution and computed results.
- Reference CPU data through cache-aware highly optimized FEAST simulation package on Opteron 250 node ($\sim$ 4 GFLOP/s LINPACK).
- GPU timings on NVIDIA GeForce 6800 graphics card (AGP).

Influence of input data precision (I)

Goal: Analyze influence of input data precision on the overall error.

- Compute RHS as **discrete** Laplacian $\mathbf{f} := -\Delta_h \mathbf{u}_0$ to avoid discretization errors.
- Clamp from double ($\mathbf{f}_{64}$) to single ($\mathbf{f}_{32}$) and half ($\mathbf{f}_{16}$) floating point precision.
- Perform computation in the CPU reference solver and the GPU-based defect correction solver with full accuracy until norm of residuals drops below $d^0_{64} \cdot \epsilon_{outer} = 10^{-10}$.

Influence of input data precision (II)

Results for this test case, $N = 257^2$ and $N = 513^2$:

$\mathbf{f}_{16}$ (CPU)	$\mathbf{f}_{16}$ (GPU-CPU)
$2.333 \cdot 10^{-6}$	$2.333 \cdot 10^{-6}$
$1.008 \cdot 10^{-6}$	$1.008 \cdot 10^{-6}$

$\mathbf{f}_{32}$ (CPU)	$\mathbf{f}_{32}$ (GPU-CPU)
$7.718 \cdot 10^{-10}$	$7.717 \cdot 10^{-10}$
$7.726 \cdot 10^{-10}$	$7.725 \cdot 10^{-10}$

$\mathbf{f}_{64}$ (CPU)	$\mathbf{f}_{64}$ (GPU-CPU)
$2.750 \cdot 10^{-13}$	$2.806 \cdot 10^{-13}$
$1.051 \cdot 10^{-12}$	$1.049 \cdot 10^{-12}$

- Representing all data in double precision is essential.
- Improving internal computational precision without introducing a higher precision format for input and output is useless.
- GPU-CPU defect correction yields identical precision.

Influence of solver accuracies (I)

Goal: Analyze overall error of the continuous problem and compare possible accuracy with varying solver precision.

- Compute RHS $\mathbf{f} := -\Delta \mathbf{u}_0$ with **continuous** Laplacian to obtain the continuous problem $-\Delta \mathbf{u} = \mathbf{f}$ in Ω.
- Analytically known solution: $\mathbf{u} = \mathbf{u}_0$.
- Run pure CPU and the GPU-CPU solver at half, single and double precision, with input data in the corresponding precision, and solve until the initial defect is reduced by 10^{-10}.

Influence of solver accuracies (II)

N	CPU16	GPU16	CPU32	GPU32	CPU64	GPU-CPU64
3^2	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$
5^2	$1.444 \cdot 10^{-3}$	$1.509 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$
9^2	$2.675 \cdot 10^{-4}$	$7.556 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$
17^2	$4.047 \cdot 10^{-4}$	$2.332 \cdot 10^{-3}$	$1.015 \cdot 10^{-4}$	$1.016 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$
33^2	$2.120 \cdot 10^{-3}$	$2.253 \cdot 10^{-3}$	$2.611 \cdot 10^{-5}$	$2.648 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$
65^2	noise	noise	$6.464 \cdot 10^{-6}$	$8.324 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$
129^2	noise	noise	$1.656 \cdot 10^{-6}$	$8.554 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$
257^2	noise	noise	$5.927 \cdot 10^{-7}$	$2.781 \cdot 10^{-5}$	$4.181 \cdot 10^{-7}$	$4.181 \cdot 10^{-7}$
513^2	noise	noise	$2.803 \cdot 10^{-5}$	$1.119 \cdot 10^{-4}$	$1.047 \cdot 10^{-7}$	$1.047 \cdot 10^{-7}$
1025^2	noise	noise	$7.708 \cdot 10^{-5}$	$4.463 \cdot 10^{-4}$	$2.620 \cdot 10^{-8}$	$2.620 \cdot 10^{-8}$

- No discernible difference for $N \geq 17^2$ in a given visualization: 16 bit floating point arithmetic is sufficient for *visual accuracy*.
- *Half* precision: Inappropriate beyond visual accuracy.
- *Single* precision: Discretization error dominates for the first few levels (factor 4 error reduction per refinement as expected), but due to insufficient precision, further refinement increases the error again.
- Note the difference in single precision accuracy between CPU and GPU.
- GPU-CPU approach yields identical accuracy compared to pure CPU implementation.

Influence of solver accuracies (II)

N	CPU16	GPU16	CPU32	GPU32	CPU64	GPU-CPU64
3^2	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$
5^2	$1.444 \cdot 10^{-3}$	$1.509 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$
9^2	$2.675 \cdot 10^{-4}$	$7.556 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$
17^2	$4.047 \cdot 10^{-4}$	$2.332 \cdot 10^{-3}$	$1.015 \cdot 10^{-4}$	$1.016 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$
33^2	$2.120 \cdot 10^{-3}$	$2.253 \cdot 10^{-3}$	$2.611 \cdot 10^{-5}$	$2.648 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$
65^2	noise	noise	$6.464 \cdot 10^{-6}$	$8.324 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$
129^2	noise	noise	$1.656 \cdot 10^{-6}$	$8.554 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$
257^2	noise	noise	$5.927 \cdot 10^{-7}$	$2.781 \cdot 10^{-5}$	$4.181 \cdot 10^{-7}$	$4.181 \cdot 10^{-7}$
513^2	noise	noise	$2.803 \cdot 10^{-5}$	$1.119 \cdot 10^{-4}$	$1.047 \cdot 10^{-7}$	$1.047 \cdot 10^{-7}$
1025^2	noise	noise	$7.708 \cdot 10^{-5}$	$4.463 \cdot 10^{-4}$	$2.620 \cdot 10^{-8}$	$2.620 \cdot 10^{-8}$

- No discernible difference for $N \geq 17^2$ in a given visualization: 16 bit floating point arithmetic is sufficient for *visual accuracy*.
- *Half* precision: Inappropriate beyond visual accuracy.
- *Single* precision: Discretization error dominates for the first few levels (factor 4 error reduction per refinement as expected), but due to insufficient precision, further refinement increases the error again.
- Note the difference in single precision accuracy between CPU and GPU.
- GPU-CPU approach yields identical accuracy compared to pure CPU implementation.

Influence of solver accuracies (II)

N	CPU16	GPU16	CPU32	GPU32	CPU64	GPU-CPU64
3^2	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$
5^2	$1.444 \cdot 10^{-3}$	$1.509 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$
9^2	$2.675 \cdot 10^{-4}$	$7.556 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$
17^2	$4.047 \cdot 10^{-4}$	$2.332 \cdot 10^{-3}$	$1.015 \cdot 10^{-4}$	$1.016 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$
33^2	$2.120 \cdot 10^{-3}$	$2.253 \cdot 10^{-3}$	$2.611 \cdot 10^{-5}$	$2.648 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$
65^2	noise	noise	$6.464 \cdot 10^{-6}$	$8.324 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$
129^2	noise	noise	$1.656 \cdot 10^{-6}$	$8.554 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$
257^2	noise	noise	$5.927 \cdot 10^{-7}$	$2.781 \cdot 10^{-5}$	$4.181 \cdot 10^{-7}$	$4.181 \cdot 10^{-7}$
513^2	noise	noise	$2.803 \cdot 10^{-5}$	$1.119 \cdot 10^{-4}$	$1.047 \cdot 10^{-7}$	$1.047 \cdot 10^{-7}$
1025^2	noise	noise	$7.708 \cdot 10^{-5}$	$4.463 \cdot 10^{-4}$	$2.620 \cdot 10^{-8}$	$2.620 \cdot 10^{-8}$

- No discernible difference for $N \geq 17^2$ in a given visualization: 16 bit floating point arithmetic is sufficient for *visual accuracy*.
- *Half* precision: Inappropriate beyond visual accuracy.
- *Single* precision: Discretization error dominates for the first few levels (factor 4 error reduction per refinement as expected), but due to insufficient precision, further refinement increases the error again.
- Note the difference in single precision accuracy between CPU and GPU.
- GPU-CPU approach yields identical accuracy compared to pure CPU implementation.

Influence of solver accuracies (II)

N	CPU16	GPU16	CPU32	GPU32	CPU64	GPU-CPU64
3^2	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$
5^2	$1.444 \cdot 10^{-3}$	$1.509 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$
9^2	$2.675 \cdot 10^{-4}$	$7.556 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$
17^2	$4.047 \cdot 10^{-4}$	$2.332 \cdot 10^{-3}$	$1.015 \cdot 10^{-4}$	$1.016 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$
33^2	$2.120 \cdot 10^{-3}$	$2.253 \cdot 10^{-3}$	$2.611 \cdot 10^{-5}$	$2.648 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$
65^2	noise	noise	$6.464 \cdot 10^{-6}$	$8.324 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$
129^2	noise	noise	$1.656 \cdot 10^{-6}$	$8.554 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$
257^2	noise	noise	$5.927 \cdot 10^{-7}$	$2.781 \cdot 10^{-5}$	$4.181 \cdot 10^{-7}$	$4.181 \cdot 10^{-7}$
513^2	noise	noise	$2.803 \cdot 10^{-5}$	$1.119 \cdot 10^{-4}$	$1.047 \cdot 10^{-7}$	$1.047 \cdot 10^{-7}$
1025^2	noise	noise	$7.708 \cdot 10^{-5}$	$4.463 \cdot 10^{-4}$	$2.620 \cdot 10^{-8}$	$2.620 \cdot 10^{-8}$

- No discernible difference for $N \geq 17^2$ in a given visualization: 16 bit floating point arithmetic is sufficient for *visual accuracy*.
- *Half* precision: Inappropriate beyond visual accuracy.
- *Single* precision: Discretization error dominates for the first few levels (factor 4 error reduction per refinement as expected), but due to insufficient precision, further refinement increases the error again.
- Note the difference in single precision accuracy between CPU and GPU.
- GPU-CPU approach yields identical accuracy compared to pure CPU implementation.

Influence of solver accuracies (II)

N	CPU16	GPU16	CPU32	GPU32	CPU64	GPU-CPU64
3^2	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$
5^2	$1.444 \cdot 10^{-3}$	$1.509 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$
9^2	$2.675 \cdot 10^{-4}$	$7.556 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$
17^2	$4.047 \cdot 10^{-4}$	$2.332 \cdot 10^{-3}$	$1.015 \cdot 10^{-4}$	$1.016 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$
33^2	$2.120 \cdot 10^{-3}$	$2.253 \cdot 10^{-3}$	$2.611 \cdot 10^{-5}$	$2.648 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$
65^2	noise	noise	$6.464 \cdot 10^{-6}$	$8.324 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$
129^2	noise	noise	$1.656 \cdot 10^{-6}$	$8.554 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$
257^2	noise	noise	$5.927 \cdot 10^{-7}$	$2.781 \cdot 10^{-5}$	$4.181 \cdot 10^{-7}$	$4.181 \cdot 10^{-7}$
513^2	noise	noise	$2.803 \cdot 10^{-5}$	$1.119 \cdot 10^{-4}$	$1.047 \cdot 10^{-7}$	$1.047 \cdot 10^{-7}$
1025^2	noise	noise	$7.708 \cdot 10^{-5}$	$4.463 \cdot 10^{-4}$	$2.620 \cdot 10^{-8}$	$2.620 \cdot 10^{-8}$

- No discernible difference for $N \geq 17^2$ in a given visualization: 16 bit floating point arithmetic is sufficient for *visual accuracy*.
- *Half* precision: Inappropriate beyond visual accuracy.
- *Single* precision: Discretization error dominates for the first few levels (factor 4 error reduction per refinement as expected), but due to insufficient precision, further refinement increases the error again.
- Note the difference in single precision accuracy between CPU and GPU.
- GPU-CPU approach yields identical accuracy compared to pure CPU implementation.

Performance evaluation

Iteration counts and timings for $\epsilon_{outer} = 10^{-10}$ (relative), FEAST gaining 10 decimals directly vs. combined GPU-CPU solver gaining 2-3 decimals per (inner) iteration, (timings include all necessary data transfers):

N	Iters FEAST	Iters GPU-CPU	Time FEAST	Time GPU-CPU
127^2	188	437 (4)	0.27s	0.62s
257^2	368	877 (4)	2.30s	1.89s
513^2	707	1762 (4)	18.38s	8.81s
1025^2	1392	3506 (4)	191.39s	53.13s

- CPU outperforms GPU for smaller problems.
- FEAST convergence behaviour as expected $O(h^{-1})$. GPU-CPU solver shows surprisingly high iteration counts.
- GPU-CPU solver outperforms FEAST by a factor of 3.6 despite unoptimized "proof of concept" implementation and an increase in iterations by 2.5.

Performance evaluation

Iteration counts and timings for $\epsilon_{outer} = 10^{-10}$ (relative), FEAST gaining 10 decimals directly vs. combined GPU-CPU solver gaining 2-3 decimals per (inner) iteration, (timings include all necessary data transfers):

N	Iters FEAST	Iters GPU-CPU	Time FEAST	Time GPU-CPU
127^2	188	437 (4)	$0.27s$	$0.62s$
257^2	368	877 (4)	$2.30s$	$1.89s$
513^2	707	1762 (4)	$18.38s$	$8.81s$
1025^2	1392	3506 (4)	$191.39s$	$53.13s$

- CPU outperforms GPU for smaller problems.
- FEAST convergence behaviour as expected $O(h^{-1})$. GPU-CPU solver shows surprisingly high iteration counts.
- GPU-CPU solver outperforms FEAST by a factor of 3.6 despite unoptimized "proof of concept" implementation and an increase in iterations by 2.5.

Analysis of GPU-CPU convergence behaviour

Exemplary setup: $N = 257^2$ (left) and $N = 513^2$ (right), $\epsilon_{outer} = 10^{-10}$, iteration counts for different values of ϵ_{inner}:

ϵ_{inner}	CPU 64-64	CPU 64-32	GPU 64-32	CPU 64-64	CPU 64-32	GPU 64-32
10^{-1}	970 (10)	998 (10)	956 (10)	1626 (10)	1931 (10)	1634 (10)
10^{-2}	654 (5)	776 (5)	831 (5)	1139 (5)	1818 (5)	1508 (5)
10^{-3}	830 (4)	941 (4)	877 (4)	1649 (4)	1984 (4)	1762 (4)
10^{-5}	531 (2)	1111 (3)	927 (3)	1034 (2)	2241 (4)	2822 (4)
10^{-10}	368 (1)	2222 (3)	1780 (3)	707 (1)	5182 (3)	4848 (5)

- Defect correction scheme itself is main reason for iteration increase: Inner CG solver 'unsuitable' (bad restart values? loss of orthogonality?).
- 2 decimals in the inner solver do not translate to 2 decimals in the outer defect, this scales with the problem size!
- Will be further examined as soon as more powerful (MG) inner preconditioners are available.

Potential improvements, conclusions

Do not check for convergence of the inner solver in every iteration, but heuristically based on expected convergence behaviour.

Incorporate *asynchroneous readbacks* for convergence checks, do norm evaluation during potentially superfluous next inner iteration.

Evaluate better scaling schemes, e.g. use $(d_i^{-1})_i$ as preconditioner.

Remark: The same algorithm can also be implemented completely in the GPU to improve a *half* precision result to *single* precision, taking advantage of the 50% bandwidth reduction.

Conclusion: We have presented an approach that allows double precision calculations while using the GPU as a fast co-processor. Convergence behaviour needs to be further examined, but for practical purposes, the algorithm works very well and delivers good speedups compared to a highly tuned CPU-only solution.