Hardware Oriented Numerics for PDEs

Dominik Göddeke (and the FEAST group)

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

PDESoft 2012
Münster, May 18

technische universität dortmund

fakultät für mathematik

What's this all about?

Hardware isn't our friend any more

- 'Power wall + memory wall + ILP wall = brick wall'
- Frequency scaling and Pax MPI is over
- Paradigm shift towards parallelism and heterogeneity
- Data movement cost gets prohibitively expensive
- In single chips, workstations, nodes and large-scale machine

Challenges in numerical HPC

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

Alternative Approach: Hardware-Oriented Numerics

Conflicting situations

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

Challenge: new algorithmic way of thinking

- Balance these conflicting goals
- Much more than just ‘good implementation’
- Rather: Scalable, arbitrarily parallelisable, locality maximising numerical schemes

Important

- Consider short-term hardware details in actual implementations, but long-term hardware trends in the design of numerical schemes!

The Memory Wall Problem

Worst-case example: Vector addition

- Compute $\mathbf{c} = \mathbf{a} + \mathbf{b}$ for large N in double precision
- Arithmetic intensity: N flops for $3N$ memory operations
- My machine: 12 GFLOP/s and 10 GB/s peak

Back-of-an-envelope calculation

- To run at 12 GFLOP/s, we need $12 \cdot 3$ Gdoubles, i.e., 288 GB/s
- Bad: maximum performance is 3.5% of what we could do

Performance of SpMV

- Similar upper bound: no reuse in matrix data, indirection (bad caching) in coefficient vector
- Obviously, GFLOP/s are not a clever metric for this

The Memory Wall Problem

Moving data is becoming prohibitively expensive

- Affects all levels of the memory hierarchy
- Between cluster nodes, from main memory to CPU, from CPU to GPU, within chips

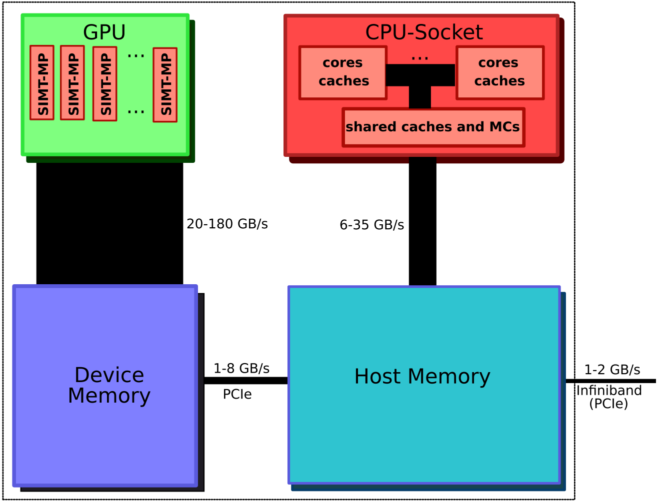
Multicores make this worse

- Number of memory controllers does not scale with number of cores
- It can sometimes make sense to leave cores idle
- NUMA and shared last-level caches

Data locality is the only solution

- Maximise data reuse (manually or via choice of data structures)
- Maximise coherent access patterns for block-transfers and avoid jumping through memory

GPUs and the Memory Wall Problem



GPUs: Myth, Marketing and Reality

Raw marketing numbers

- > 3 TFLOP/s peak single precision floating point performance
- Lots of papers claim $> 100\times$ speedup

Looking more closely

- Single or double precision floating point (same precision on both devices)?
- Sequential CPU code vs. parallel GPU implementation?
- 'Standard operations' or many low-precision graphics constructs?

Reality

- GPUs are undoubtedly fast, but so are CPUs
- Quite often: CPU codes significantly less carefully tuned
- Anything between 5–30x speedup is realistic (and worth the effort)

Example #1:

Mixed Precision
Iterative Refinement

Combatting the memory wall problem

Motivation

Switching from double to single precision (DP→SP)

- 2x effective memory bandwidth, 2x effective cache size
- At least 2x compute speed (often 4–12x)

Problem: Condition number

- Theory for linear system $\mathbf{Ax} = \mathbf{b}$

$$\mathrm{cond}_2(\mathbf{A}) \sim 10^s; \frac{\|\mathbf{A} + \delta\mathbf{A}\|}{\|\mathbf{A}\|}, \frac{\|\mathbf{b} + \delta\mathbf{b}\|}{\|\mathbf{b}\|} \sim 10^{-k}(k > s) \quad \Rightarrow \quad \frac{\|\mathbf{x} + \delta\mathbf{x}\|}{\|\mathbf{x}\|} \sim 10^{s-k}$$

In our setting

- Truncation error in 7–8th digit increased by s digits

Numerical Example

Poisson problem on unit square

- Simple yet fundamental model problem
- $\text{cond}_2(\mathbf{A}) \approx 10^5$ for $L = 10$ (1M bilinear FE, regular grid)
- Condition number usually much higher: anisotropies in grid and operator

	Data+Comp. in DP		Data in SP, Compute in DP		Data+Comp. in SP	
Level	L_2 Error	Red.	L_2 Error	Red.	L_2 Error	Red.
5	1.1102363E-3	4.00	1.1102371E-3	4.00	1.1111655E-3	4.00
6	2.7752805E-4	4.00	2.7756739E-4	4.00	2.8704684E-4	3.87
7	6.9380072E-5	4.00	6.9419428E-5	4.00	1.2881795E-4	2.23
8	1.7344901E-5	4.00	1.7384278E-5	3.99	4.2133101E-4	0.31
9	4.3362353E-6	4.00	4.3757082E-6	3.97	2.1034461E-3	0.20
10	1.0841285E-6	4.00	1.1239630E-6	3.89	8.8208778E-3	0.24

$\Rightarrow$ **Single precision insufficient for moderate problem sizes already**

Mixed Precision Iterative Refinement

Iterative refinement

- Established algorithm to provably guarantee accuracy of computed results (within given precision)
  - High precision: $\mathbf{d} = \mathbf{b} - \mathbf{A}\mathbf{x}$ (cheap)
  - Low precision: $\mathbf{c} = \mathbf{A}^{-1}\mathbf{d}$ (expensive)
  - High precision: $\mathbf{x} = \mathbf{x} + \mathbf{c}$ (cheap) and iterate (expensive?)
- Convergence to high precision accuracy if $\mathbf{A}$ *'not too ill-conditioned'*
- Theory: Number of iterations $\approx f(\log(\text{cond}_2(\mathbf{A})), \log(\varepsilon_{\text{high}}/\varepsilon_{\text{low}}))$

New idea (Hardware-oriented numerics)

- Use this algorithm to improve time to solution and thus efficiency of linear system solves
- Goal: Result accuracy of high precision with speed of low precision floating point format

Iterative Refinement for Large Sparse Systems

Refinement procedure not immediately applicable

- 'Exact' solution using 'sparse LU' techniques too expensive
- Convergence of iterative methods not guaranteed in single precision

Solution

- Interpretation as a preconditioned mixed precision defect correction iteration

$$\mathbf{x}_{\mathsf{DP}}^{(k+1)} = \mathbf{x}_{\mathsf{DP}}^{(k)} + \mathbf{C}_{\mathsf{SP}}^{-1}(\mathbf{b}_{\mathsf{DP}} - \mathbf{A}_{\mathsf{DP}}\mathbf{x}_{\mathsf{DP}}^{(k)})$$

- Preconditioner $\mathbf{C}_{\mathsf{SP}}$ in single precision:
  'Gain digit(s)' or 1-3 MG cycles instead of exact solution

Results (MG and Krylov for Poisson problem)

- Speedup at least 1.7x (often more) without loss in accuracy
- Asymptotic optimal speedup is 2x (bandwidth limited)

Example #2:

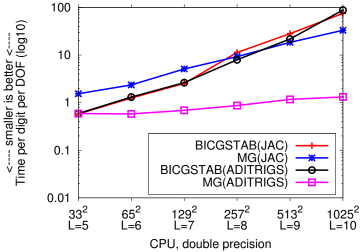
Parallelising Inherently Sequential Operations

Multigrid with strong smoothers
(Re-) discover parallelism

Motivation: Why Strong Smoothers?

Test case: anisotropic diffusion in generalised Poisson problem

- $-\text{div}\ (\mathbf{G}\ \ \text{grad}\ \ \mathbf{u}) = \mathbf{f}$, same grid as before
- $\mathbf{G} = \mathbf{I}$: standard Poisson problem, $\mathbf{G} \neq \mathbf{I}$: arbitrarily challenging
- Example: $\mathbf{G}$ introduces anisotropic diffusion along some vector field



Only multigrid with a strong smoother is competitive

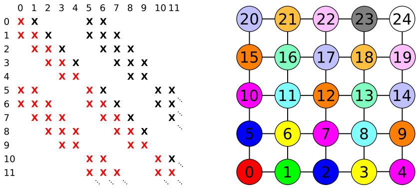
Gauß-Seidel Smoother

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

Sequential algorithm

- Forward elimination, sequential dependencies between matrix rows
- Illustrative: coupling to the left and bottom (numbering yields banded matrix)

1st idea: classical wavefront-parallelisation (exact)

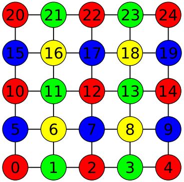


- Pro: always works to resolve *explicit* dependencies
- Con: irregular parallelism and access patterns, implementable?

Gauß-Seidel Smoother

2nd idea: decouple dependencies via multicolouring (inexact)

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



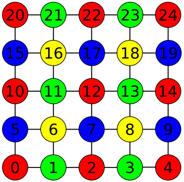
Analysis

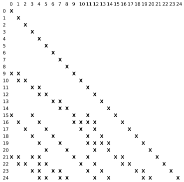
- Parallel efficiency: 4 sweeps with $\approx N/4$ parallel work each
- Checkerboard access pattern challenging for SIMD/GPU due to strided access (solution: merge colours into one kernel)
- Numerical efficiency: sequential coupling only in last sweep

Gauß-Seidel Smoother

3rd idea: multicolouring = renumbering

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



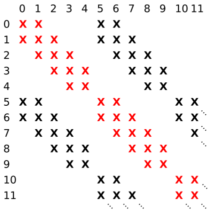


- Recoupling: Jacobi (red) – coupling to left and right (green) – top and bottom (blue) – all 8 neighbours (yellow)
- More computations that standard decoupling
- Experiments: convergence rates of sequential variant recovered (in absence of preferred direction)

Tridiagonal Smoother (Line Relaxation)

Starting point

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



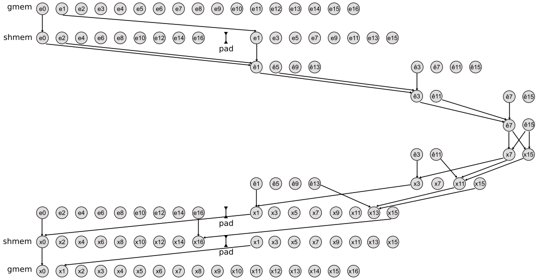
Observations

- One independent tridiagonal system per mesh row
  $\Rightarrow$ top-level parallelisation across mesh rows
- Implicit coupling: wavefront and colouring techniques not applicable

Tridiagonal Smoother (Line Relaxation)

Cyclic reduction for tridiagonal systems

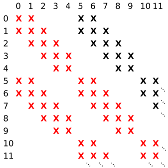
- Exact, stable (w/o pivoting) and cost-efficient
- Problem: classical formulation parallelises computation but not memory accesses on GPUs (bank conflicts in shared memory)
- Developed a better formulation, 2-4x faster
- Index nightmare, general idea: recursive padding between odd and even indices on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



Combined GS and TRIDI

Starting point

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



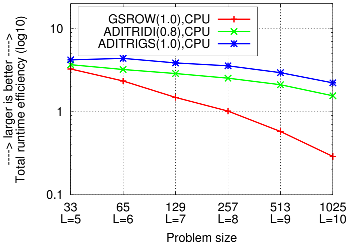
Observations and implementation

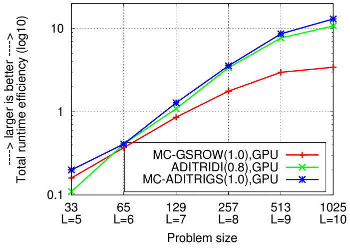
- Difference to tridiagonal solvers: mesh rows depend sequentially on each other
- Use colouring ($\#c \geq 2$) to decouple the dependencies between rows (more colours $=$ more similar to sequential variant)

Evaluation: Total Efficiency on CPU and GPU

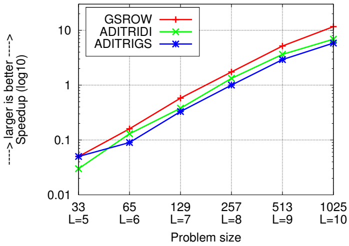
Test problem: generalised Poisson with anisotropic diffusion

- Total efficiency: (time per unknown per digit $(\mu s))^{-1}$
- Mixed precision iterative refinement multigrid solver





Speedup GPU vs. CPU



Summary: structured grid smoother parallelisation

- Factor 8–30 (dep. on HW, precision, smoother selection) speedup over already highly tuned CPU implementation
- Same functionality on CPU and GPU
- Balancing of numerical and parallel efficiency, best speedup for worst method

Example #3:

Grid- and Matrix Structures

Flexibility $\leftrightarrow$ Performance
Robust parallel smoothers

Grid- and Matrix Structures

General sparse matrices (unstructured grids)

- CSR (and ELLR-T for GPUs): matrix format for arbitrary grids
- Maximum flexibility, but during SpMV
  - Indirect, irregular memory accesses
  - Index overhead reduces already low arithm. intensity further
- Performance depends on nonzero pattern (DOF numbering)

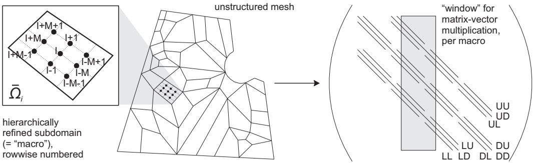
Structured matrices (structured grids)

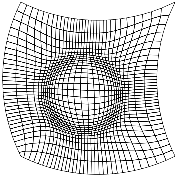
- As above: structured grids, suitable numbering $\Rightarrow$ band matrices
- Important: no stencils, fully variable coefficients
- direct regular memory accesses (fast), mesh-independent performance
- Structure exploitation in the design of MG components (ex. 2)

Approach in FEAST

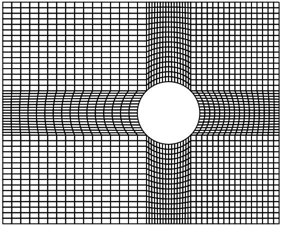
Combination of respective advantages

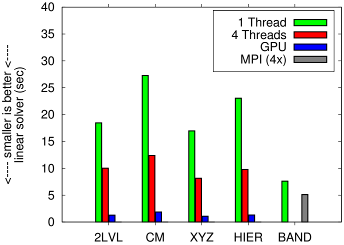
- Global macro-mesh: unstructured, flexible
- Local micro-meshes: structured (logical TP-structure), fast
- Important: structured $\neq$ cartesian meshes (r-adaptivity)
- Reduce numerical linear algebra to sequences of operations on structured data (maximise locality)
- Developed for large clusters (later), but generally useful





Example: Poisson on Unstructured Grid



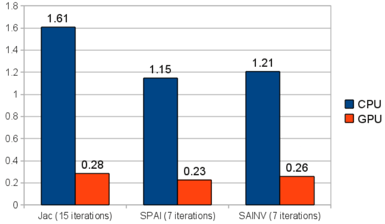


Intel Nehalem vs. NVIDIA Tesla (GTX280)

- $\approx$ 2M bilinear FE, MG-JAC (no influence of numbering on numerics)
- Unstructured formats highly numbering-dependent
- Multicore 2–3x over singlecore, GPU 8–12x over multicore
- Banded format (here: 8 'blocks') 2–3x faster than best unstructured layout and predictably on par with multicore
- Multilevel r-adaptivity across patch boundaries better than h-adaptivity?

Example: Poisson on Unstructured Grid

GPU/multicore parallelisation also possible for strong smoothers



Intel Westmere vs. NVIDIA Fermi (C2070)

- Same problem and discretisation as before, XYZ numbering
- SPAI (asymptotically GS) and SAINV (close to ILU(0)) smoothers
- Reasonable speedups of GPU over multicore
- More on ‘unstructured GPU’ for FEM assembly: talk by Matthias Möller, Tuesday morning

Example #4:

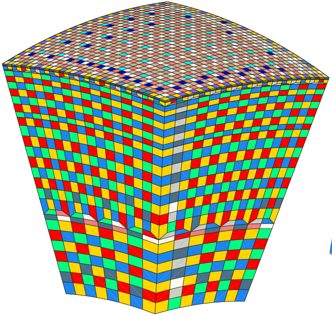
Integrating GPUs into Large-scale Software

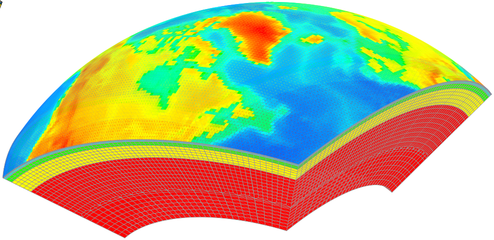
Re-implementation vs. acceleration

SPECFEM3D-GLOBE: Seismic Wave Propagation

Problem description

- Elastic waves in strongly heterogeneous media
- Earthquake modeling at the scale of the Earth
- Gordon-Bell 2003, finalist 2008
- *Very* well-tuned MPI-only CPU reference implementation

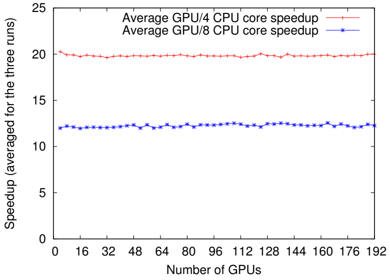




SPECFEM3D-GLOBE: Seismic Wave Propagation

GPU parallelisation

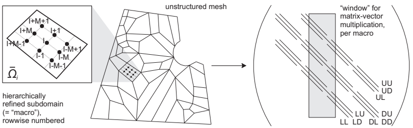
- Algorithm: explicit in time, SEM+GLL discretisation $\Rightarrow$ 90% of time to solution into SEM assembly
- One 'PhD-year' in 2008 for single-GPU re-implementation of simple Earth models ($\neq$ full production code)
- Two 'professor-weeks' in 2009 to get overlapping of MPI and PCIe-GPU completely hidden



ScaRC: Coarse-Grained Parallel Geometric Multigrid

ScaRC for scalar systems

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

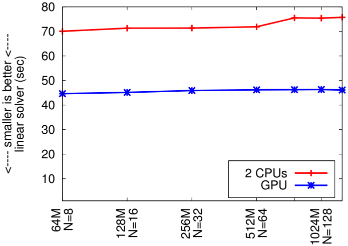


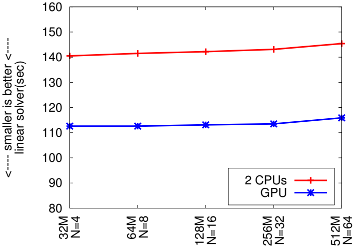
global Krylov
preconditioned by
 global multilevel (V 1+1)
 additively smoothed by
  for all Ω_i: **local multigrid**
 coarse grid solver: UMFPACK

Weak Scalability

Simultaneous doubling of problem size and resources

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

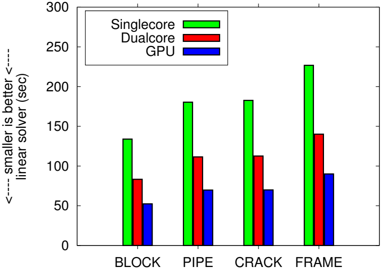


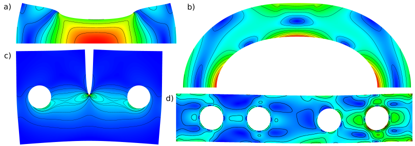


Results

- No loss of weak scalability despite local acceleration
- 1.3 billion DOF (no stencil!) on 160 ancient GPUs in less than 50 s

Speedup Linearised Elasticity





- USC cluster in Los Alamos, 16 dualcore nodes (Opteron Santa Rosa, Quadro FX5600)
- Problem size 128 M DOF
- Dualcore 1.6x faster than singlecore (memory wall)
- GPU 2.6x faster than singlecore, 1.6x than dualcore

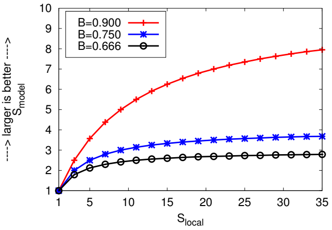
Speedup Analysis

Theoretical model of expected speedup

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

This example

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



Summary and Conclusions

Summary

High-level take-away messages of this talk

- Things numerical software people might want to know about hardware
- Thinking explicitly of data movement and in parallel is mandatory
- Unfortunately, there are many levels of parallism, each with its own communication characteristics
- Parallelism is (often) natural, we 'just' have to rediscover it

Selected examples: Multilevel solvers and GPUs

- Mixed precision iterative refinement techniques
- Extracting fine-grained parallelism from inherently sequential ops
- FEM-multigrid (geometric) for structured and unstructured grids
- Integrating GPUs in numerical software

Outlook and Current Work

Minimising Amdahl's impact

- Properly doable only with C++
- FEM-Assembly (almost done)
- Smoothers for convection-dominated problems: tricky because numerica requires different numbering than parallelisation

Road towards exascale

- Promising results on cluster of 256 Tegra-2 smartphone SoC: '2 GFLOP/s at 0.5 Watts'
- 10x slower execution more than compensated by using 10x more processors for less 'energy to solution'
- Implication: GPU-style scalability required at the level currently implied by MPI

Acknowledgements

Collaborative work with

- FEAST group (TU Dortmund): Ch. Becker, S.H.M. Buijssen, C. Christof, M. Geveler, D. Göddeke, J. Greif, M. Köster, D. Ribbrock, Th. Rohkämper, S. Turek, P. Zajac
- Robert Strzodka (Max Planck Institut Informatik, currently NVIDIA Research)
- Jamaludin Mohd-Yusof, Patrick McCormick (Los Alamos National Laboratory)
- Dimitri Komatitsch (CNRS Marseille), Gordon Erlebacher (UFL), David Michéa (BGRM)

Supported by

- DFG: TU 102/22-1, TU 102/22-2, TU 102/27-1, TU102/11-3
- BMBF: *HPC Software für skalierbare Parallelrechner*: SKALB project 01IH08003D

http://www.mathematik.tu-dortmund.de/~goeddeke