Hardware-oriented Numerics for PDEs

Dominik Göddeke (and the FEAST group)

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

Software Problems for Challenging Computational Problems
ACMAC, Heraklion, Greece, January 15 2013

Hardware (r)evolution

Parallelism, specialisation and heterogeneity

- Frequency scaling is over, we now scale 'cores' and SIMD
- Even CPU-only systems are heterogeneous
- CPUs and accelerators in the same system
- Visible on all architectural levels
  - Fine-grained: SSE/AVX, SIMT within chips
  - Medium-grained: GPUs, CPUs, NUMA within a node
  - Coarse-grained: MPI between heterogeneous nodes

This paradigm shift is here to stay

- Power is the root cause of this
- Driven by move towards peta- and exascale, but: affects all 'levels' of scientific computing

Consequences ...

A hardware problem suddenly becomes a software problem

- Existing codes no longer run faster automatically
- Swapping an underlying library doesn't help
- Neither does relying on auto-parallelisation and vectorisation by general-purpose high-level compilers
  - We discuss topics that DOLPHIN (recall Garth's talk) outsources

A hardware problem suddenly becomes a methodological problem

- Existing textbook numerical schemes are often designed for sequential or coarse-grained parallel execution
- Unit cost for flops and data movement

Conflicting goals

Good numerical performance

- Same convergence and properties as sequential code
- Methodological scalability (weak and strong)

Good hardware performance

- Extract a decent fraction of peak (whatever the peak is)
- Parallel scalability (weak and strong)

Portability and man-power

- Re-usable software components
- Hide hardware and implementation details from application people
- Performance portability of implementations

Our approach: hardware-oriented numerics

Challenge: new algorithmic way of thinking

- Balance these conflicting goals
- Goal: scalable, arbitrarily parallelisable, locality maximising, robust, h-independent, $O(N)$, adaptive, high-order ... schemes

Important: more than 'good implementation'

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

Two options

- Production codes: tune once for many runs, re-implementation may be justified
- Research code bases: (re-) design to facilitate a good compromise between the above goals

Talk outline

Hardware evolution

- Brief, high-level overview of challenges
- GPUs vs. CPUs

Case study 1

- Seismic wave propagation modeling on large GPU clusters
- Spectral element method

Case study 2

- Co-design of grid, matrix and solver structures
- ScaRC: FEM-multilevel solvers for large, sparse systems
- Partial acceleration by GPUs

Outlook: energy efficiency

- Case study on an ARM ('smartphone-powered') cluster

Hardware evolution

An impenetrable brick wall in electrical engineering

Power wall

- Small scale: power is proportional to voltage cubed
- Large scale: nuclear power plant next door?

Memory wall

- Gap between data movement (latency and bandwidth) and peak compute rates widens exponentially
- Affects all levels of the memory hierarchy

ILP wall

- Compilers and complex OoO execution circuits cannot extract enough independent instructions to keep deep pipelines fully busy

The power wall problem

Short digression into electrical engineering

- Power is proportional to voltage$^2\times$ frequency
- Frequency is proportional to voltage (at same process size)
- Similarly bad: power also proportional to temperature

A simple back-of-the-envelope calculation

	cores	V	freq	perf $\sim f$	power $\sim V^3$	power eff. perf/power
singlecore	1	1	1	1	1	1
faster singlecore	1	1.5	1.5	1.5	3.3	0.45
dualcore	2	0.75	0.75	1.5	0.8	1.88

A-ha!

- 50% more performance with 20% less power

The power wall is a money wall

Power requirement of HPC installations

- Current 1–20 PFLOP/s installations require O(10) MW per year
- Rule of thumb: 1 MW/a = 1 MEUR/a
- 20 MW considered an upper bound for future power envelopes

Our machine in Dortmund

- 440 dual socket nodes, 50 TFLOP/s
- Deployed in 2009 (#249 in the TOP500) for roughly 1 MEUR
- Annual electricity bill incl. A/C: 280 KEUR
- This year, the accumulated running cost exceeds its initial deployment cost
- Administration won't pay for a follow-up machine

The memory wall problem

Illustrative example: vector addition

- Compute $\mathbf{c} = \mathbf{a} + \mathbf{b}$ for large N in double precision
- Arithmetic intensity: N flops for $3N$ values to/from memory

GFLOP/s are meaningless for this

- Dual Intel Westmere server (2011): 200 GFLOP/s and 36 GB/s peak
- Vector addition at 200 GFLOP/s needs 600 Gdoubles, i.e., 4.8 TB/s
- Worse: maximum performance is less than one percent of what the chip could do

More generally

- Data movement becomes prohibitively expensive
- Affects all levels of the memory hierarchy, moving data over networks takes O(1e4–1e6) longer than computing
- Moving data also burns more Joules

The ILP wall (and more on-chip bottlenecks)

Hardware and compiler ILP run out of steam

- Lots of complex circuits to make a single instruction stream run fast
  - Branch prediction, prefetchers, caches, OoO execution, ...
  - Lots of legacy features for backward ABI compatibility in x86
- Conventional CPUs are not very power-efficient
- Only 1 % of the die area actually does compute!

Enter GPUs

- Designed to maximise the throughput of many data elements
- OpenCL is not performance-portable
- NVIDIA CUDA: co-design of hardware and programming model
- The 'beauty' of CUDA
  - Allows much less 'implementational corner-cutting'
  - Forces us to think about performance already during algorithm design

From CPUs to GPUs

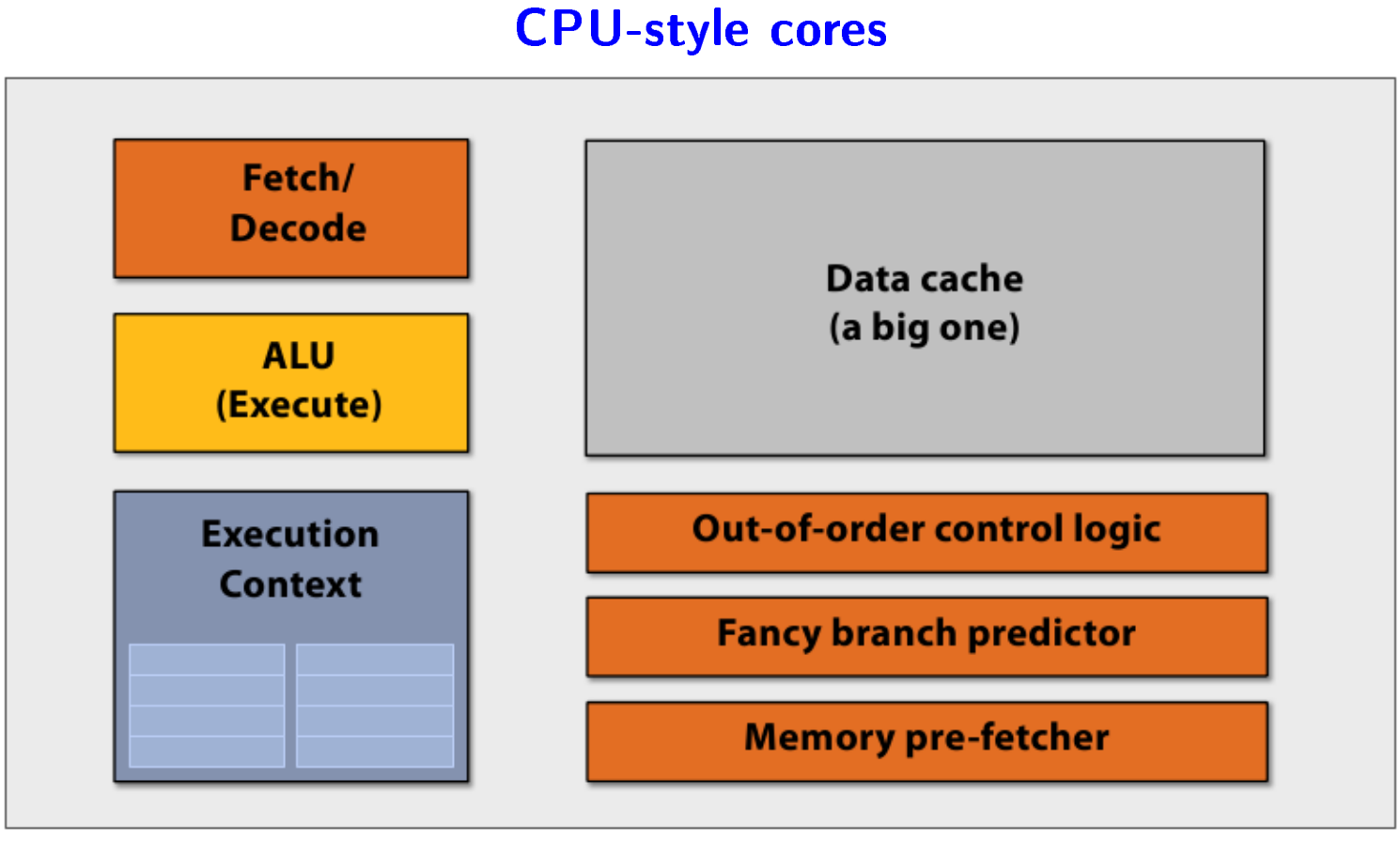


Image courtesy Kayvonf Fatahalian

From CPUs to GPUs

Skimming down

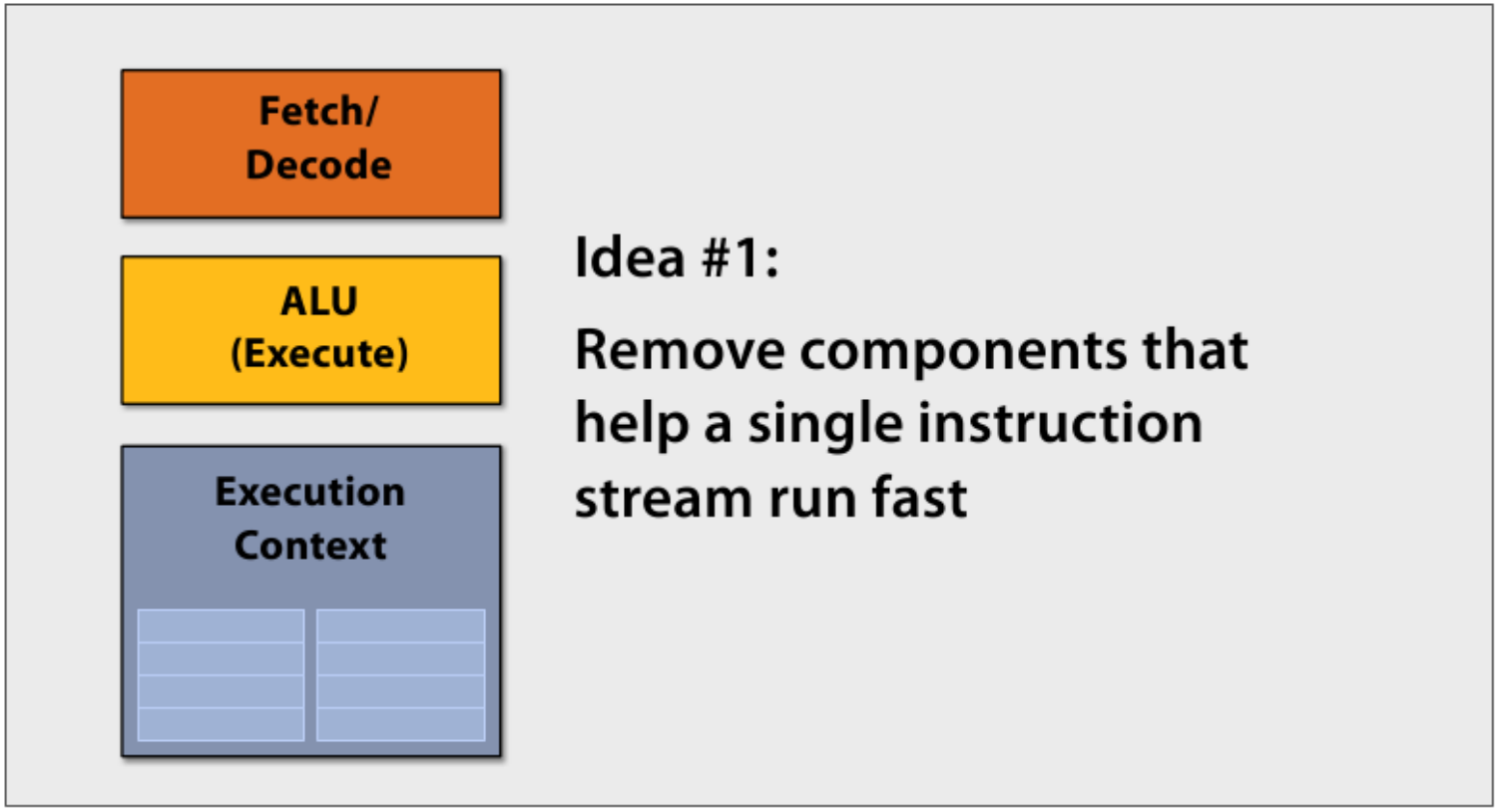


Image courtesy Kayvonf Fatahalian

From CPUs to GPUs

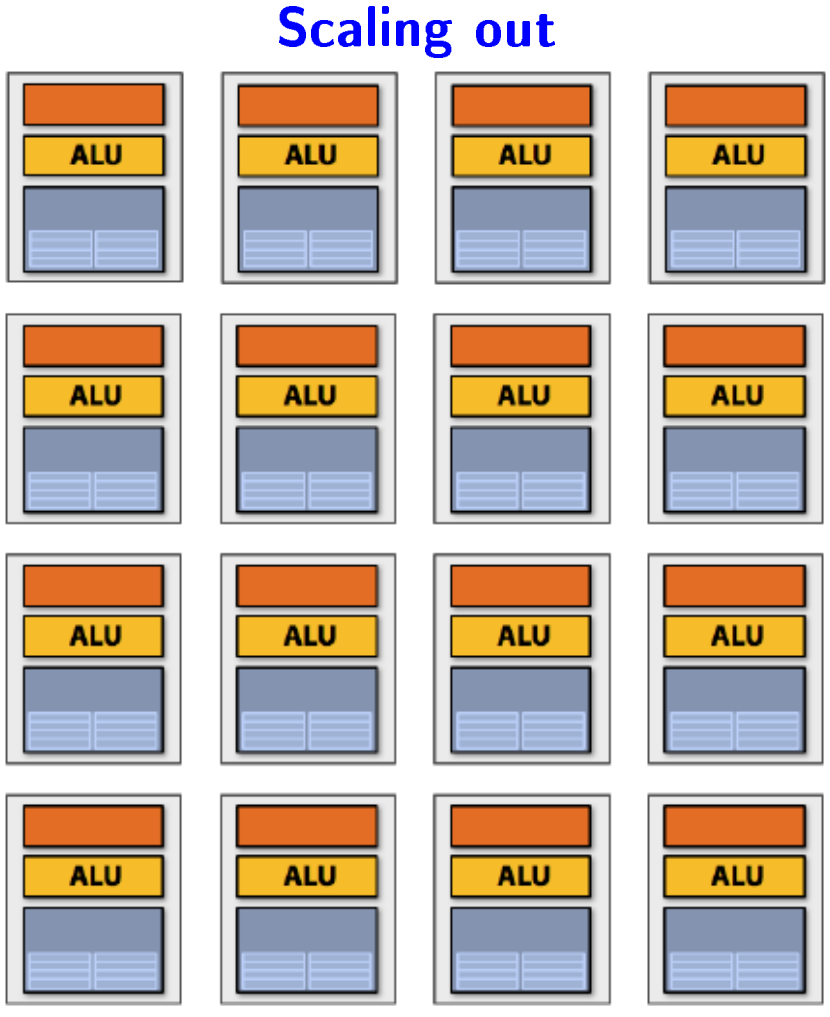


Image courtesy Kayvonf Fatahalian

From CPUs to GPUs

Amortise instruction cost: SIMD processing

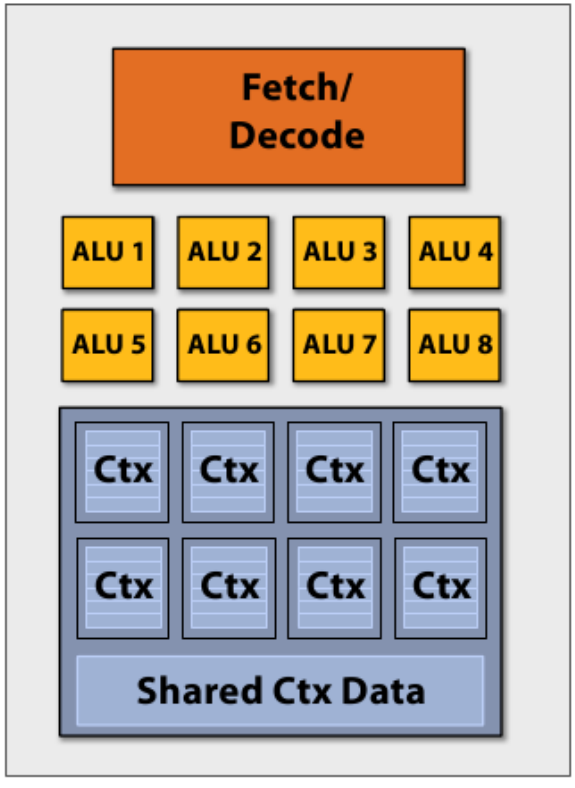
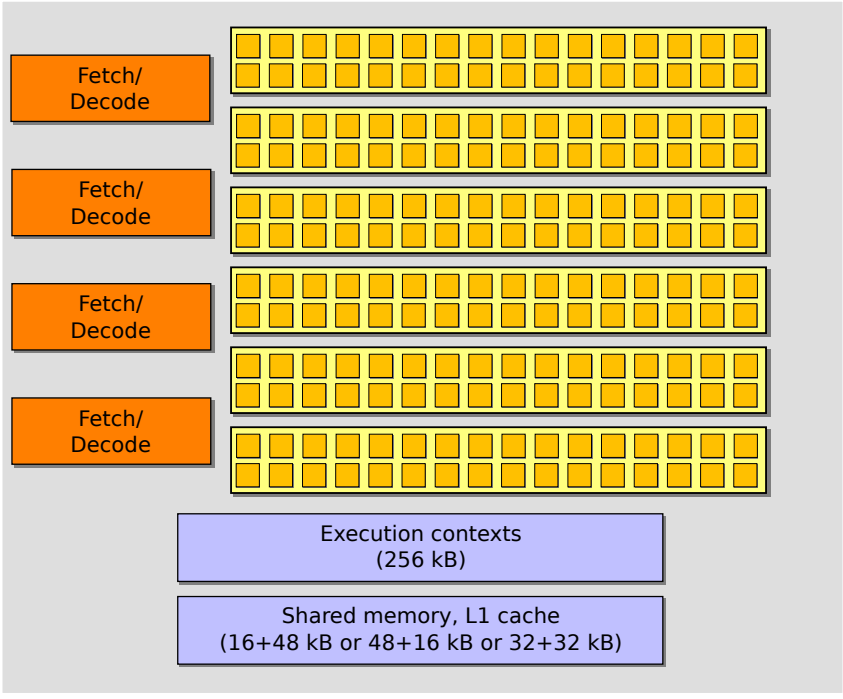


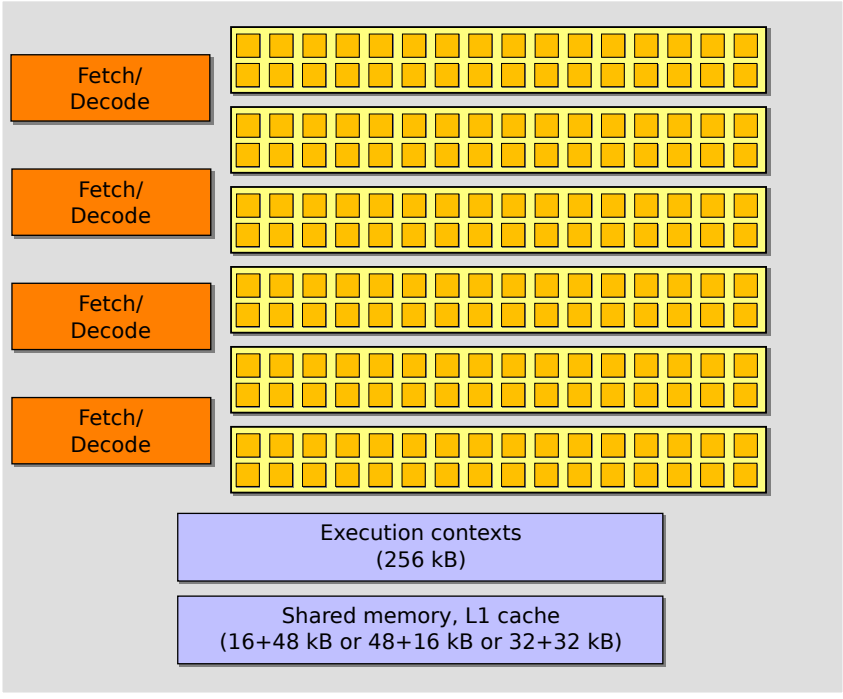
Image courtesy Kayvonf Fatahalian

Real GPU design: NVIDIA Kepler



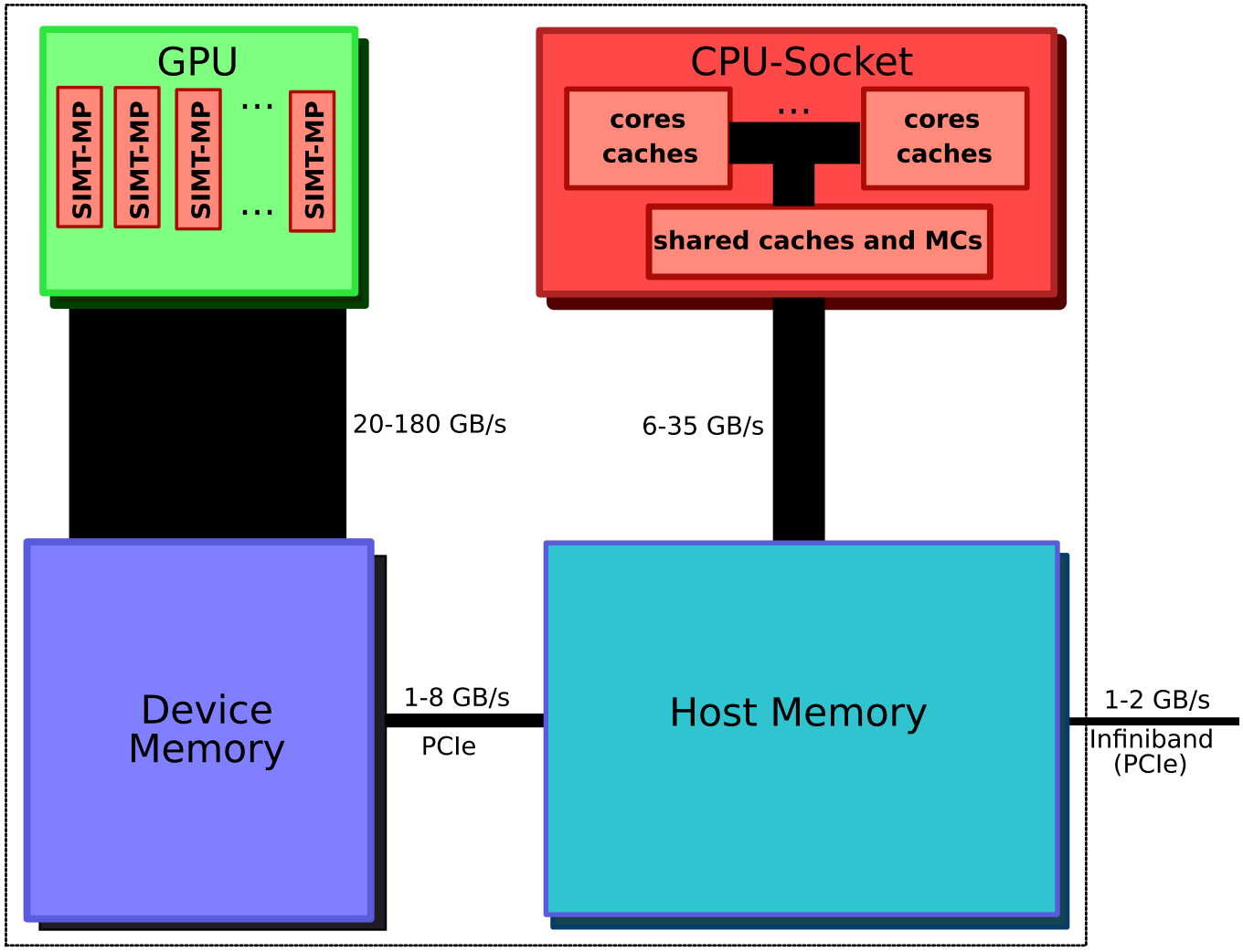
- Figure shows a single core of this design
- Not shown: 32 SFUs exclusively for transcendentals, DP units
- 32 threads (one warp) share an instruction stream
  - Each of the four schedulers can issue two instructions per cycle

Real GPU design: NVIDIA Kepler



- Programming abstraction: block = virtualised core
- Up to 32 warps can synchronise and share data in one block
- Threads from different blocks only synchronise at kernel scope
- Up to 16 blocks with together up to 64 warps simultaneously, if one stalls, another one is swapped in overhead-free

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)?
- Sequential CPU code vs. parallel GPU implementation?
- 'Standard operations' or many low-precision graphics constructs?
- Different levels of performance tuning?

Reality

- GPUs are undoubtedly fast, but so are CPUs (I am a GPU person!)
- GPUs tolerate much less 'implementational corner cutting', force us to think about performance while we code
- Tuning for GPUs almost always yields ideas to improve CPU code

Case study 1

Seismic wave propagation modeling

Introduction and challenges

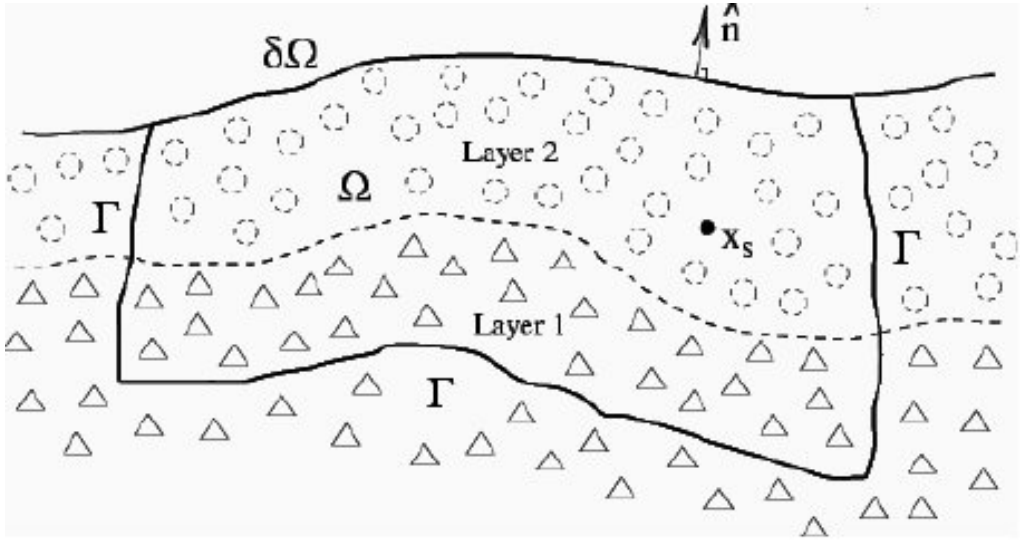
Topography and sedimentary basins

- Densely populated areas are often located in sedimentary basins
- Surrounding mountains reflect seismic energy back and amplify it (think rigid Dirichlet boundary conditions)
- Seismic shaking thus much more pronounced in basins

Introduction and challenges

Spatially varying resolution

- Local site effects and topography
- Discontinuities between heterogeneous sedimentary layers and faults in the Earth



High seismic frequencies need to be captured

- High-order methods and finely-resolved discretisation in space and time required

Starting point: waves in elastic anisotropic solids

Unknown displacement field

- $u(x,t), x = (x_1, x_2, x_3)$ in some domain Ω, $(x,t) \mapsto \mathbb{R}^3$
- Recall: displacement u, velocity $\partial_t u$, acceleration $\partial_t^2 u$

Constitutive law for elastic solids: $\sigma = E : \varepsilon$

- $\mathbf{A} : \mathbf{B}$ tensor contraction
- Stress tensor σ
- Elasticity tensor $E = E(x)$ ('the Earth') with 21 independent components in 3D, comprises anisotropies, i.e., different material properties in different directions (think slate)
- Strain tensor $\varepsilon = \frac{1}{2}\left(\nabla u + (\nabla u)^{\mathsf{T}}\right)$: Hooke's law, linear relation between strain and gradient of displacement – or a suitable generalisation of Hooke's law for inelastic media

Starting point: waves in elastic anisotropic solids

Strong form of the wave equation as IVP

$$
\begin{aligned}
\rho \partial_t^2 u &= \nabla \cdot \sigma + f && \text{in } \Omega \times T \\
\sigma \cdot n &= \bar{t} && \text{on } \Gamma_N \\
u &= g && \text{on } \Gamma_D \\
u|_{t=0} &= u_0 && \forall x \in \Omega \\
\partial_t u|_{t=0} &= u_1 && \forall x \in \Omega
\end{aligned}
$$

- Density ρ, acceleration $\partial_t^2 u$, source terms f
- Body forces (inner forces) $\mathrm{div}(\sigma)$ caused by the displacement
- On Neumann boundaries, external stresses $\bar{t}$ in normal direction
- Homogeneous Neumann boundaries: free surface condition
- Fixed load on Dirichlet boundaries
- Suitable initial conditions

More complex models

Layer model of the Earth

- Crust, mantle and inner core can be modeled as such solids, magma in the mantle is sufficiently 'solid'
- Elasticity tensor models discontinuities only within a solid layer
- Outer core is liquid, oceans on top of the crust cannot be neglected

Interfaces

- Fluids: outer core is compressible, oceans are incompressible
- Suitable interface conditions, reflections, refractions, free surface

Can be made arbitrarily complex

- Coriolis force, self-gravitation, PML conditions if less than the full Earth is modeled, sources, attenuation, ...

Discretisation with the spectral element method

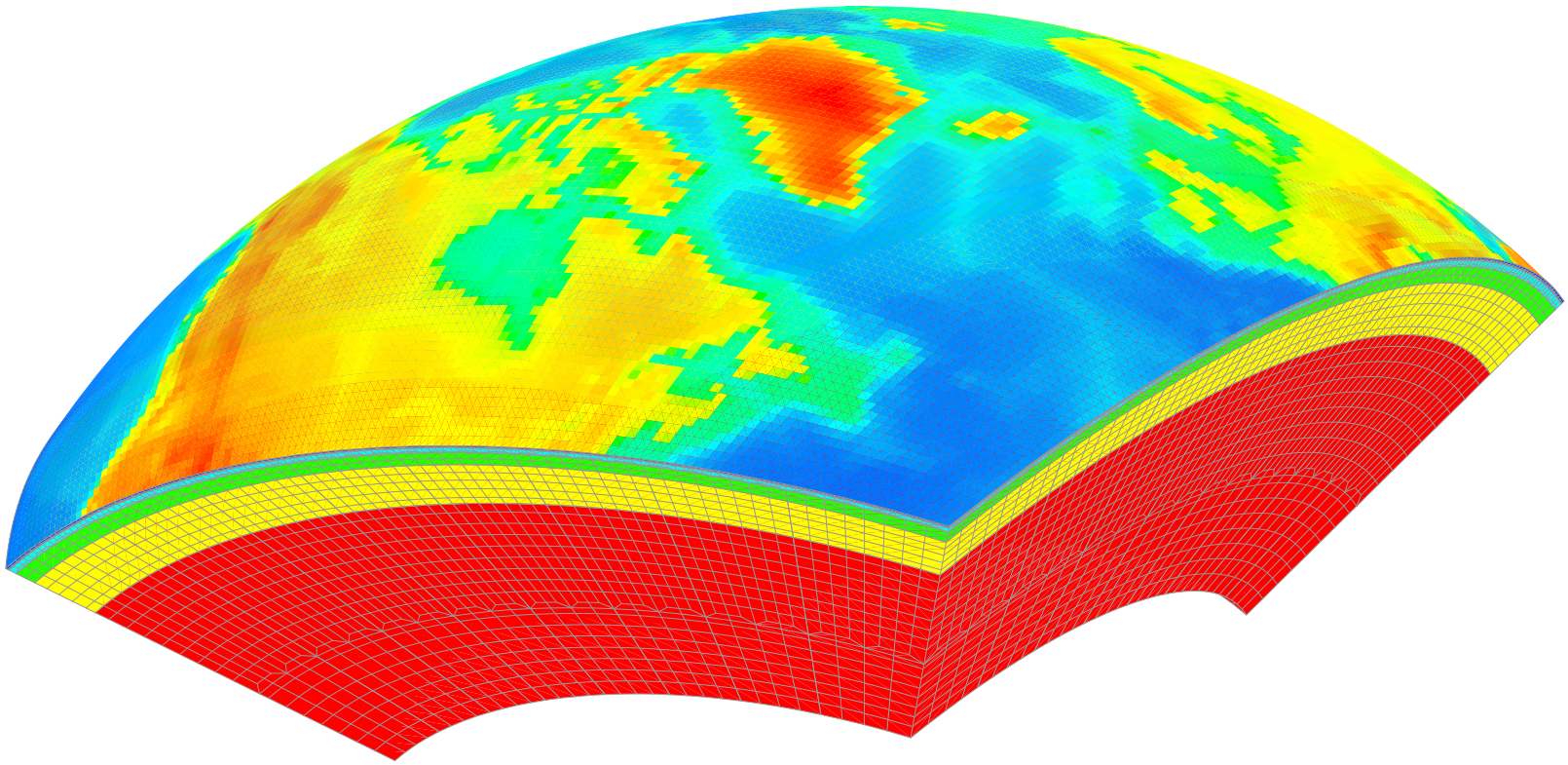
Discretisation trade-offs

- (Pseudo-) spectral methods
  - Pro: accuracy and exponential approximation order
  - Con: parallelisation, boundary conditions
- Finite element methods
  - Geometric flexibility (topographic details, faults and fractures, internal discontinuities and material transitions, ...), boundary conditions (parallelisation)
  - Possibly ill-conditioned systems, high-order expensive (wide supports)

SEM as a hybrid approach

- Essentially based on high-order interpolation to represent fields per element of a classical unstructured curvilinear FEM mesh
- Good compromise between conflicting goals
- Can be interpreted as a continuous Galerkin method

Cubed sphere unstructured mesh



SEM for the seismic wave equation

Discretise the weak form

- Same as in finite element methods

Base everything on Lagrange polynomials

- Use triple tensor products of degree-two Lagrange polynomials as shape functions for each curvilinear element
- Represent physical fields by interpolation based on triple tensor products of Lagrange basis polynomials
  - Degree 4–10, 4 is a good compromise between accuracy and speed
  - Use Gauß-Lobatto-Legendre control points (rather than just Gauß or Lagrange points), GLL points are the roots of $(1 - x^2)l'_k(x)$, so closely related to degree-k Lagrange polynomials $l_k(x)$
  - Degree+1 GLL points per spatial dimension per element (so 125 for degree 4 in 3D)

SEM for the seismic wave equation

Clever trick

- Use GLL points also as cubature points for quadrature
- GLL Lagrange interpolation combined with GLL quadrature yields strictly diagonal mass matrix

Important consequence: algorithm significantly simplified

- Explicit time stepping schemes become feasible
- In this talk: second order centred finite difference Newmark time integration
- Solving of linear systems becomes trivial
- Drawback: timestep depends on mesh spacing

Solution algorithm

Problem to be solved in algebraic notation

- $\mathbf{M}\ddot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}$ for waves in simple solids
- Mass matrix $\mathbf{M}$, stiffness matrix $\mathbf{K}$
- Displacement vector $\mathbf{u}$, sources $\mathbf{f}$, velocity $\mathbf{v} = \dot{\mathbf{u}}$, acceleration $\mathbf{a} = \ddot{\mathbf{u}}$

Three steps per timestep

- Update second half-step of velocity $\dot{\mathbf{u}} + \frac{\Delta t}{2}\ddot{\mathbf{u}}$ and global displacement $\mathbf{u} + \Delta t\dot{\mathbf{u}} + \frac{\Delta t}{2}\ddot{\mathbf{u}}$
- Compute new $\mathbf{K}\mathbf{u}$ and $\mathbf{M}$ (the tricky bit, called 'SEM assembly') to obtain intermediate acceleration
- Compute new acceleration $\mathbf{M}^{-1}\ddot{\mathbf{u}}$, and velocity for next half-timestep $\dot{\mathbf{u}} + \frac{\Delta t}{2}\ddot{\mathbf{u}}$ (cannot be merged into steps 2 and 1 because of data dependencies)

Implementation

All open source

- SPECFEM3D, `http://www.geodynamics.org`

Steps 1 and 3 are trivial

- AXPY-like updates on globally numbered data

Step 2 is most demanding step in the algorithm

- Measurements indicate around 90% of the total runtime
- Employ 'assembly by elements' technique on CPUs and GPUs
- First stage: all-local computations
  - DOF mapping, Jacobians, quadrature, ... $\Rightarrow$ per-element acc.
- Second stage: scatter local acceleration vector into global matrix
  - Some care required to avoid race conditions for shared data

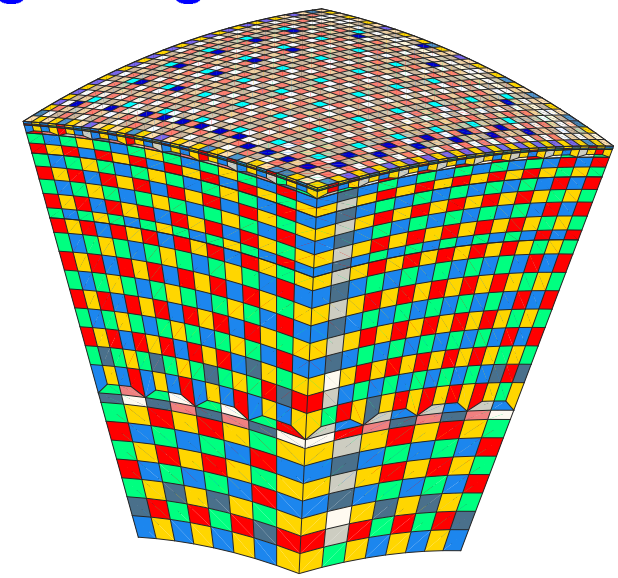
GPU implementation

One CUDA kernel for entire step 2

- One CUDA block per element, 125/128 threads to useful work
- A 'bit' painful to alleviate register and shared memory pressure
- Deville-like unrolled matrix matrix multiplication for the 5x5 cutplanes of the 5x5x5 tensor product GLL/DOF set
- Uncoalesced memory accesses for global data (unstructured mesh)

Atomic memory updates or colouring during accumulation

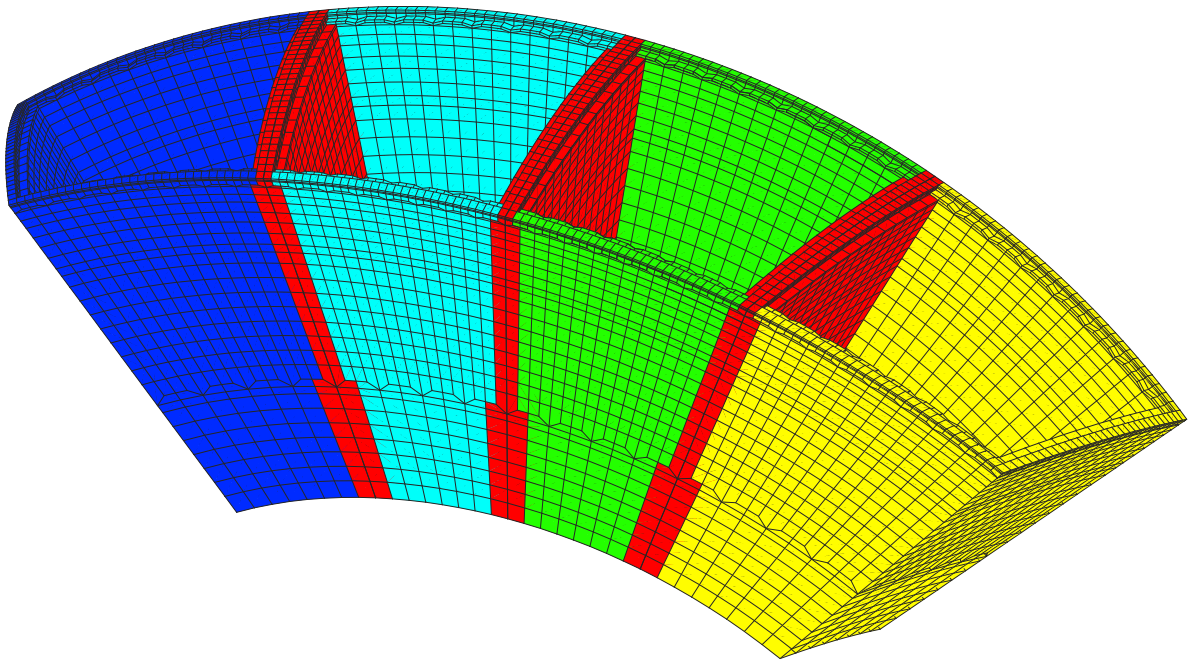
- Atomics are trivial to implement, but still slow on K20
- Use greedy colouring instead



Mapping to GPU clusters with MPI

Overlap computation with communication (non-blocking MPI)

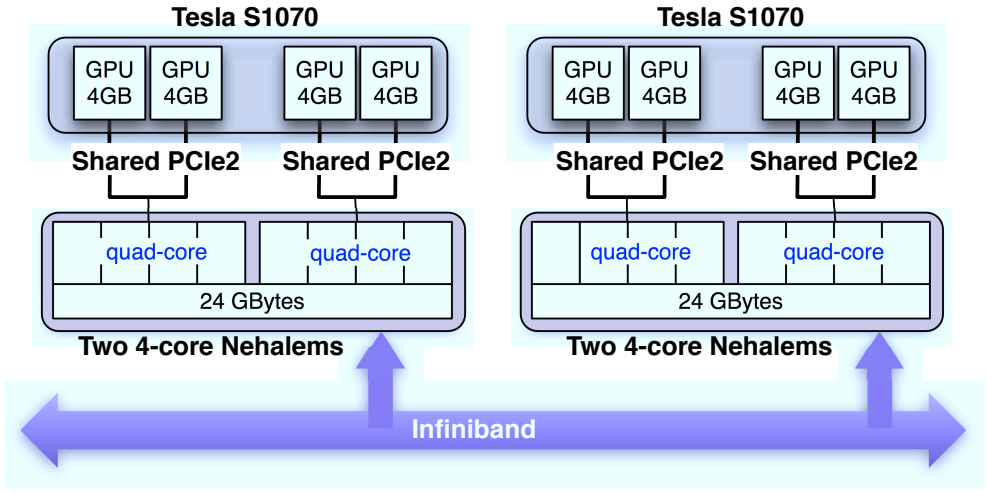
- PCIe to/from GPU adds additional latency bottleneck
- Higher surface-to-volume ratio required
- Pack send buffers on GPU, unpack on host prior to `MPI_Isend()` and vice versa on the receiving side



Testbed

Titane: Bull Novascale R422 E1 GPU cluster

- Installed at CCRT/CEA/GENCI, Bruyères-le-Châtel, France
- 48 dual-Nehalem quad-Tesla nodes



CPU reference code is heavily optimised

- Cooperation with Barcelona Supercomputing Center
- Extensive cache optimisation using ParaVer

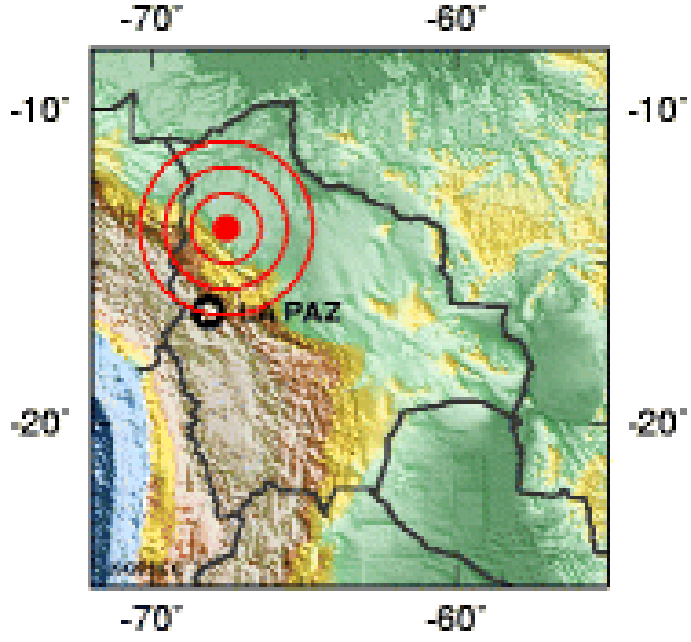
Numerical validation

Single vs. double precision

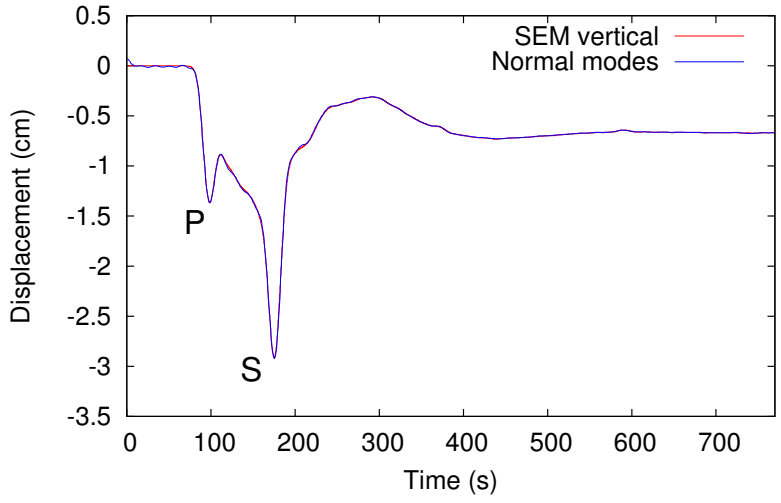
- Single precision is sufficient for this problem class
- So use single precision on CPU and GPU for a fair comparison
- Same results between single and double except minimal (but unavoidable) floating point noise

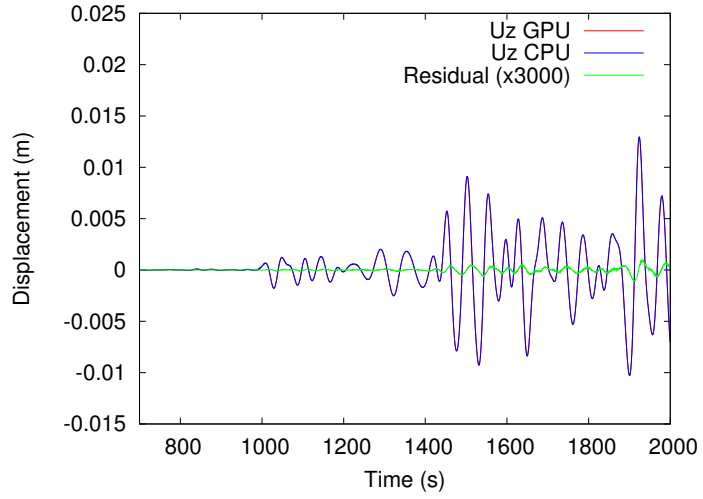
Application to a real earthquake

- Bolivia 1994, $M_w = 8.2$
- Lead to a static offset (permanent displacement) several 100 km wide
- Reference data from BANJO sensor array and quasi-analytical solution computed via summation of normal modes from sensor data



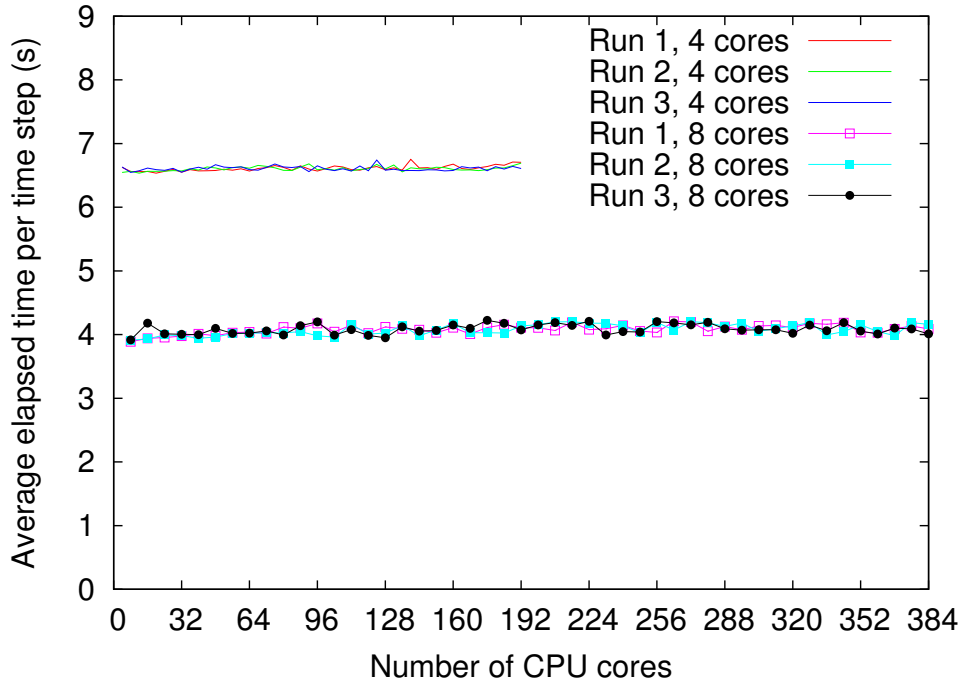
Numerical validation





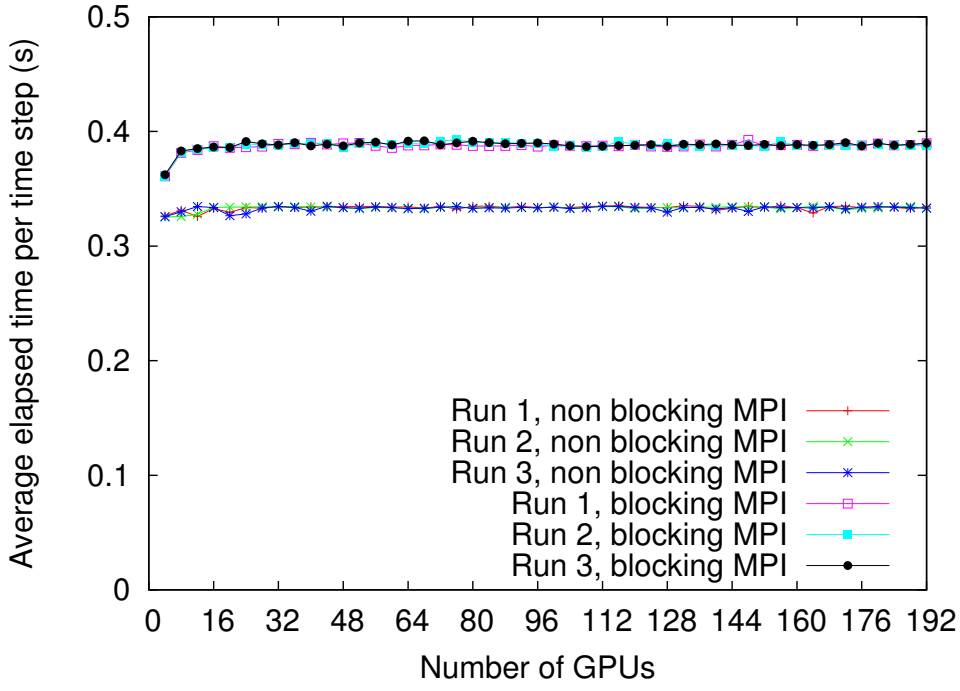
- Pressure and shear waves are accurately computed
- Static offsets are reproduced
- No difference between CPU and GPU solution
- Amplification shows that only differences are floating point noise

CPU weak and strong scaling



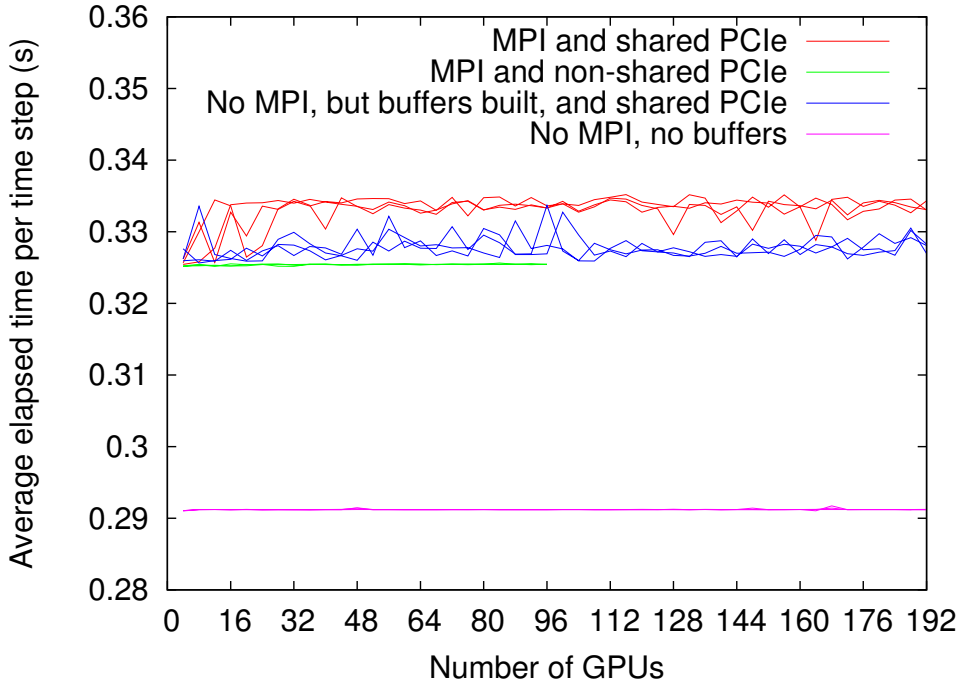
- Constant problem size per node (4x3.6 or 8x1.8 GB)
- Weak scaling excellent up to full machine (17 billion unknowns)
- 4-core version actually uses 2+2 cores per node (process pinning)
- Strong scaling only 60% due to memory bus and network contention

GPU weak scaling



- Constant problem size per node (4x3.6 GB)
- Blocking MPI: compute-PCIe-network, non-blocking as on CPU but with hidden PCIe transfers
- Weak scaling excellent up to full machine (17 billion unknowns)
- Blocking MPI results in 20% slowdown

GPU performance breakdown



- Effect of overlapping (no MPI = replace send-receive by `memset()`)
- Red vs. green: 3% penalty for using shared PCIe
- Red vs. blue: difference $\leq$ 2.8%, excellent overlap
- Green vs. magenta: total overhead of running this problem on a cluster is $\leq$ 12% for building, processing and transmitting buffers

Summary

Speedup by GPUs

- 13–21 times faster than highly tuned MPI-only code
- Depending on how many cores a single GPU is compared to

Full costly re-implementation

- Four weeks to get the single-element code decently tuned
  - By two GPU experts – tuning means getting indices right
- Four weeks to get MPI-GPU overlap in place and to do the runs

Techniques can be used in any FEM code

- Coloured or atomic assembly, and overlapping
- See JCP paper by Dimitri Komatitsch, Gordon Erlebacher, DG, David Michéa, 2011

FEAST and ScaRC

FEM-Multigrid for large systems

Introduction

Hardware-oriented numerics

- FEM-multigrid for sparse systems
- Multigrid is the only algorithmically scalable sparse iterative solver
- Lots and lots of trade-offs between hardware, parallel, numerical efficiency, scalability, portability, ...

Examples in this section

- Smoother/preconditioner parallelisation
- Grid- and matrix structures, performance vs. flexibility
- ScaRC solvers
- Minimally invasive GPU integration
- Some (oldish) results

Smoother parallelisation

Key numerical ingredient

- Good smoothers and preconditioners are essential for good convergence rates
- Focus for the next minutes: elementary (single-grid) smoothers

Different scopes

- Block preconditioning on the MPI level (later in this talk)
- Good general smoothers in a shared memory setting
  - Typically based on strong recursive coupling
  - Challenging to parallelise for the GPU
- Important building blocks in FEAST: strong smoothers on generalised tensor product grids
  - Lexicographical numbering $\Rightarrow$ banded matrix (variable coefficients)

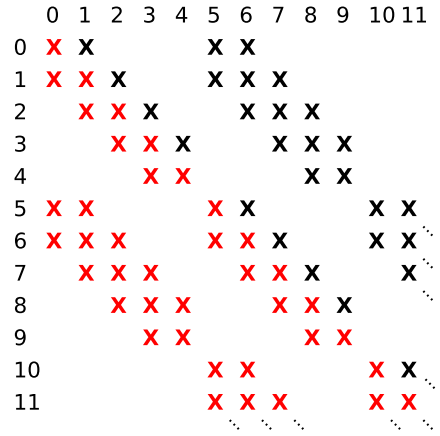
Gauß-Seidel smoother

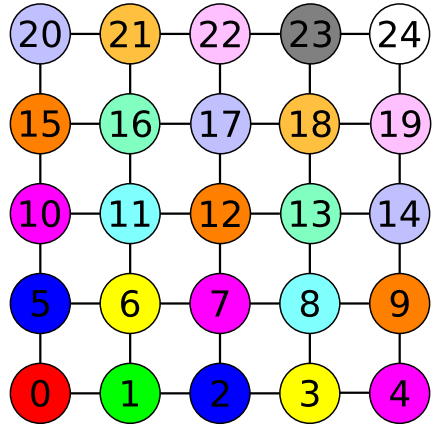
Disclaimer: Not necessarily a good smoother, but a good didactical example.

Sequential algorithm

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

1st idea: classical wavefront-parallelisation (exact)



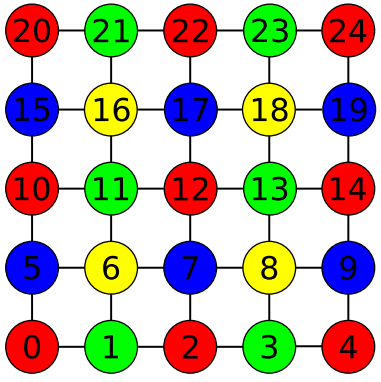


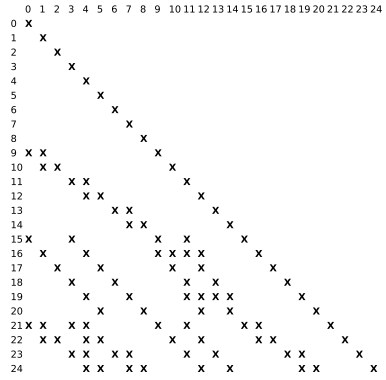
- Pro: always works to resolve *explicit* dependencies
- Con: irregular parallelism and access patterns, implementable?

Gauß-Seidel smoother

Better idea: multicolouring (inexact)

- Always use all already available results
- Jacobi (red) – coupling to left and right (green) – top and bottom (blue) – all 8 neighbours (yellow)



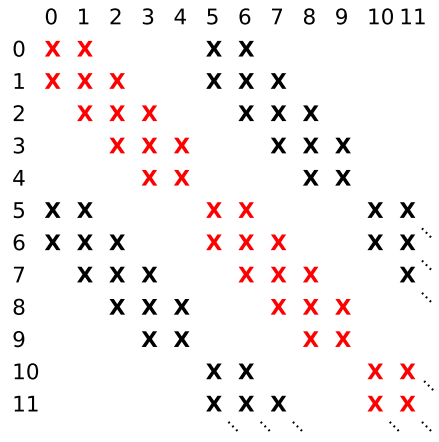


- Parallel efficiency: 4 sweeps with $\approx N/4$ parallel work each
- Regular data access, but checkerboard pattern challenging for SIMD/GPUs due to strided access
- Numerical efficiency: different coupling than sequentially

Tridiagonal smoother (line relaxation)

Starting point

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



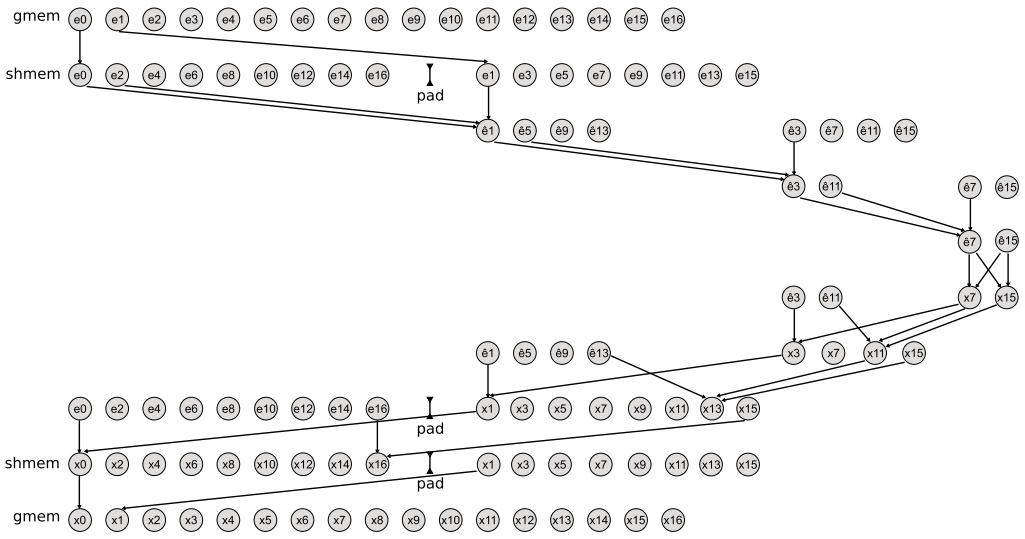
Observations

- One independent tridiagonal system per mesh row
- Top-level parallelisation across mesh rows trivial
- Implicit coupling: wavefront and colouring techniques not applicable

Tridiagonal smoother (line relaxation)

Cyclic reduction for tridiagonal systems

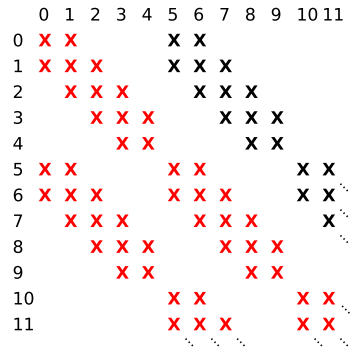
- Combine odd equations linearly from even ones
- Exact, stable (w/o pivoting) and cost-efficient
- Problem: classical formulation parallelises computation but not memory accesses on GPUs (bank conflicts in shared memory)
- Developed a better formulation, 2-4x faster
- Index nightmare, general idea: 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 know) for this matrix structure



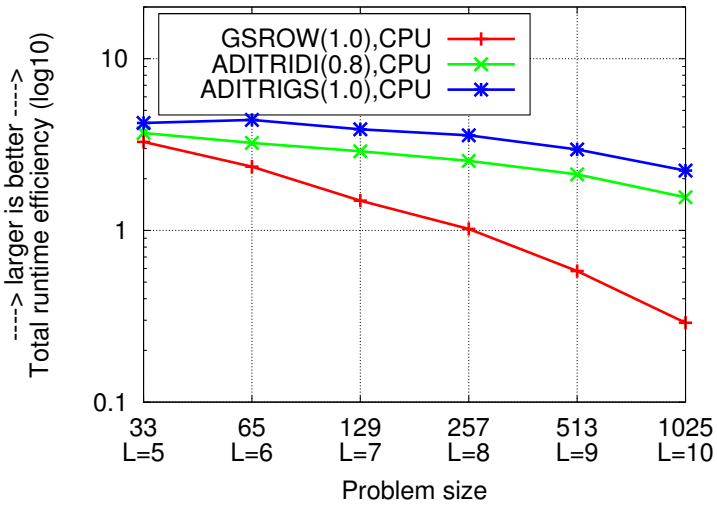
Observations and implementation

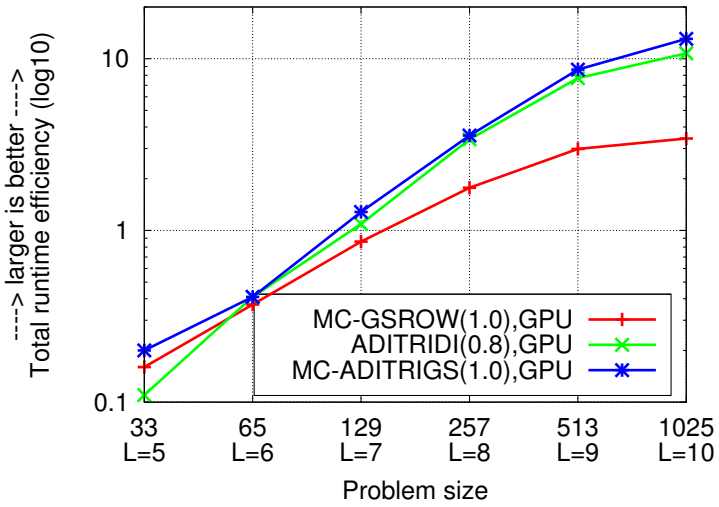
- Difference to tridiagonal solvers: mesh rows depend sequentially on each other
- Use colouring ($\#c \geq 2$) to decouple the dependencies between rows (more colours = more similar to sequential variant)

Evaluation: total efficiency on CPU and GPU

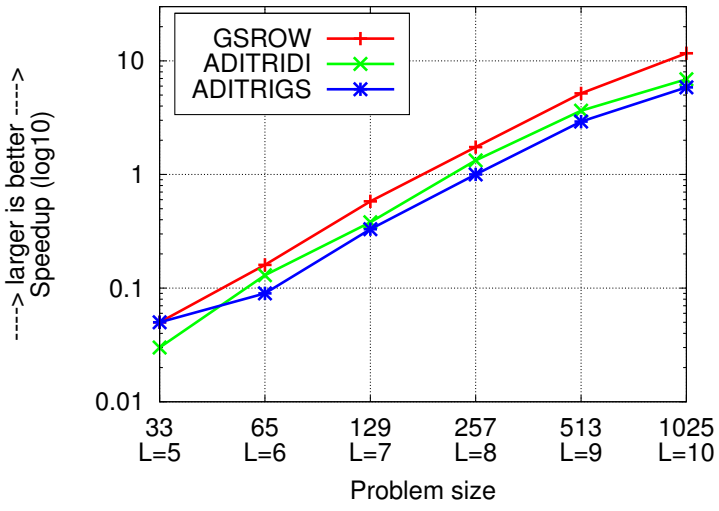
Test problem: generalised Poisson with anisotropic diffusion

- Total efficiency: (time per unknown per digit (μs))$^{-1}$
- Mixed precision iterative refinement multigrid solver
- Core2Duo vs. GTX280 (older results)





Speedup GPU vs. CPU



Summary: smoother parallelisation

- Factor 10-30 (dep. on precision and smoother selection) speedup over already highly tuned CPU implementation
- Same functionality on CPU and GPU
- Balancing of various efficiency targets

Grid- and matrix structures

General sparse matrices (on unstructured grids)

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

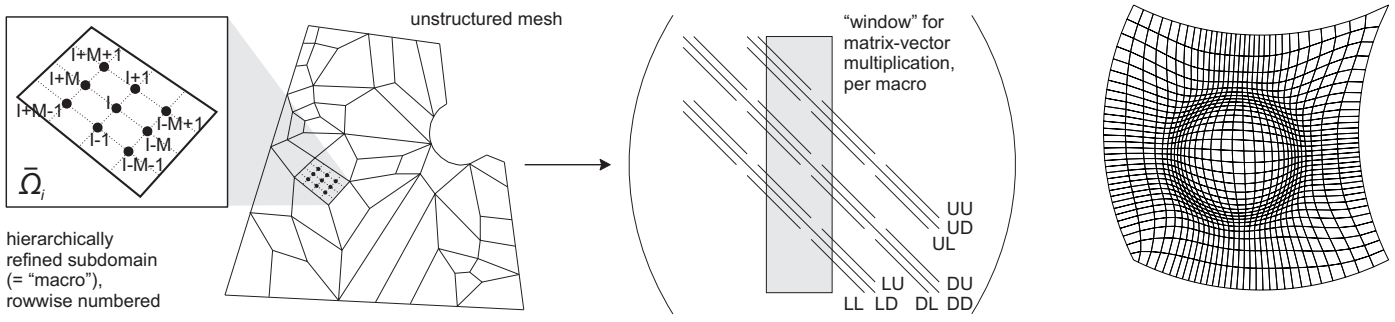
Structured matrices

- Example: 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

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 larger clusters (later), but generally useful: cache-friendly, locality-maximising



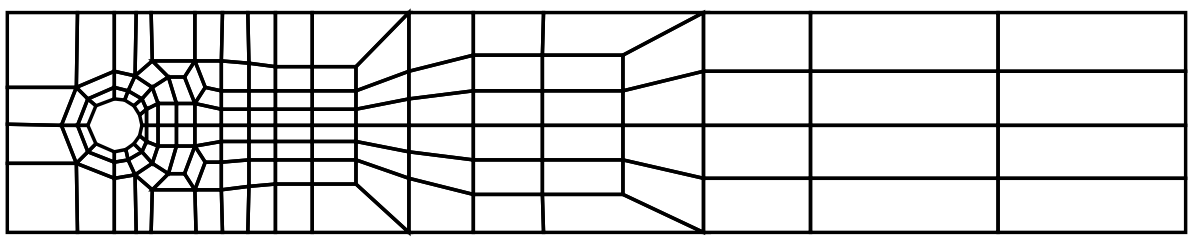
Examples

Test problem

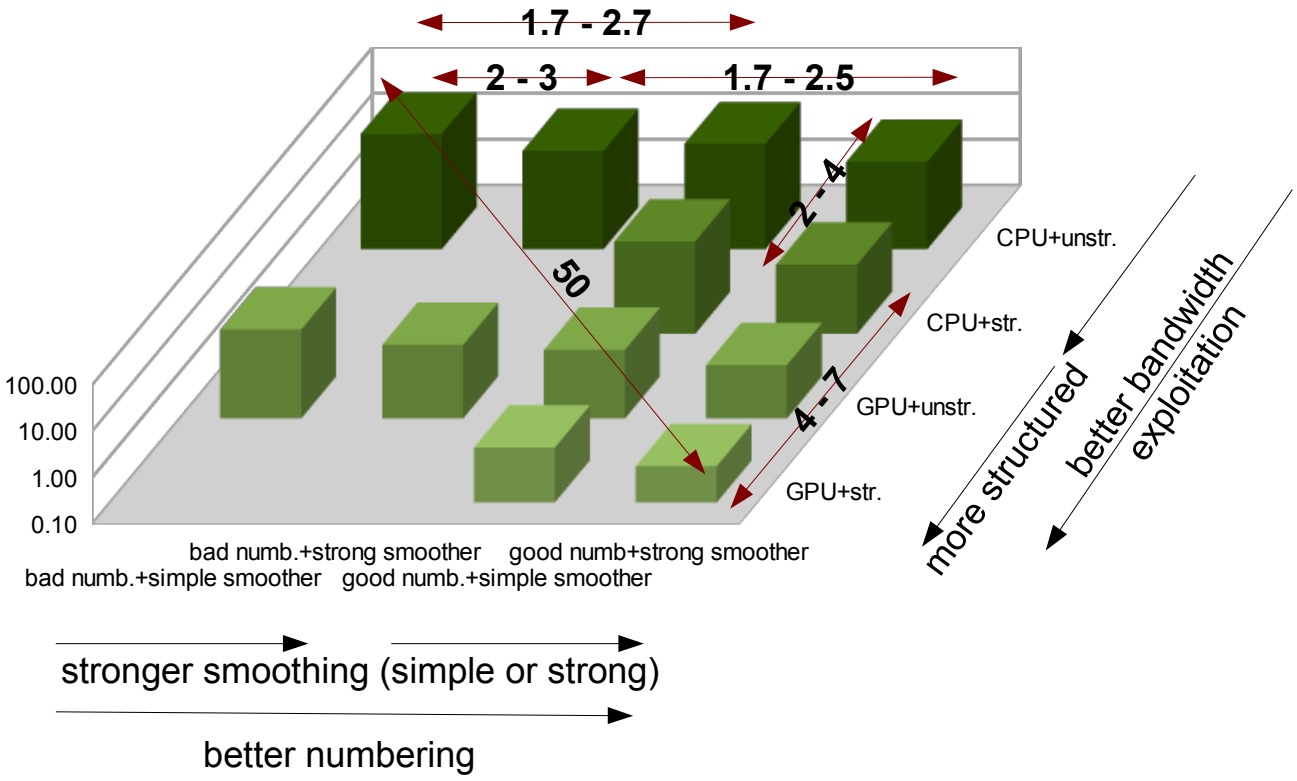
- $-\Delta u = 1$, Q1 FEM, cyclinderflow grid
- Inhomogeneous Dirichlet boundary conditions
- Multigrid solver

Big fat parameter space

- ELLR-T vs. FEAST matrix format
- ELLR-T with two different sortings (random and lexicographic)
- Jacobi vs. strong smoother (unstructured SPAI or ADI-TRIGS)
- Westmere Corei7-X980 vs. Fermi C2070



Results on cylinder grid



- Largest to smallest contribution: GPU, structured, smoother, numbering

Coarse-grained parallel multigrid

Goals

- Parallel efficiency: strong and weak scalability
- Numerical scalability, i.e. convergence rates independent of N
- Robust for different partitionings, anisotropies, etc.

Most important challenge

- Minimising communication between cluster nodes
- Concepts for strong smoothers so far not applicable (shared memory) due to high communication cost and synchronisation overhead
- Insufficient parallel work on coarse levels

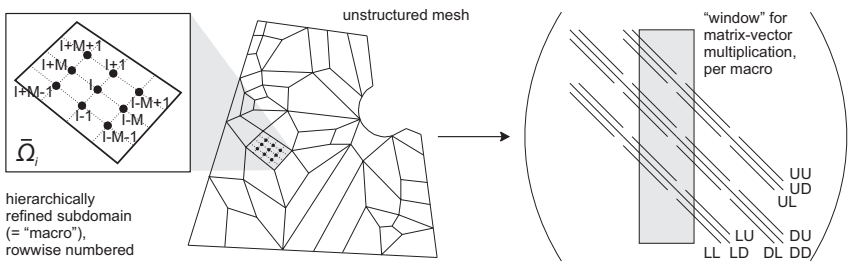
Our approach: Scalable Recursive Clustering (ScaRC)

- Under development in our group

ScaRC: concepts

ScaRC for scalar systems

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



global BiCGStab

preconditioned by

global multilevel (V 1+1)

additively smoothed by

for all Ω_i: **local multigrid**

coarse grid solver: UMFPACK

ScaRC for multivariate problems

Block-structured systems

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

Examples (2D case)

- Linearised elasticity with compressible material (2x2 blocks)
- Saddle point problems: Stokes (3x3 with zero blocks), elasticity with (nearly) incompressible material, Navier-Stokes with stabilisation (3x3 blocks)

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} = \mathbf{f}, \quad \begin{pmatrix} \mathbf{A}_{11} & \mathbf{0} & \mathbf{B}_1 \\ \mathbf{0} & \mathbf{A}_{22} & \mathbf{B}_2 \\ \mathbf{B}_1^T & \mathbf{B}_2^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{p} \end{pmatrix} = \mathbf{f}, \quad \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{B}_1 \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{B}_2 \\ \mathbf{B}_1^T & \mathbf{B}_2^T & \mathbf{C}_C \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{p} \end{pmatrix} = \mathbf{f}$$

$\mathbf{A}_{11}$ and $\mathbf{A}_{22}$ correspond to scalar (elliptic) operators

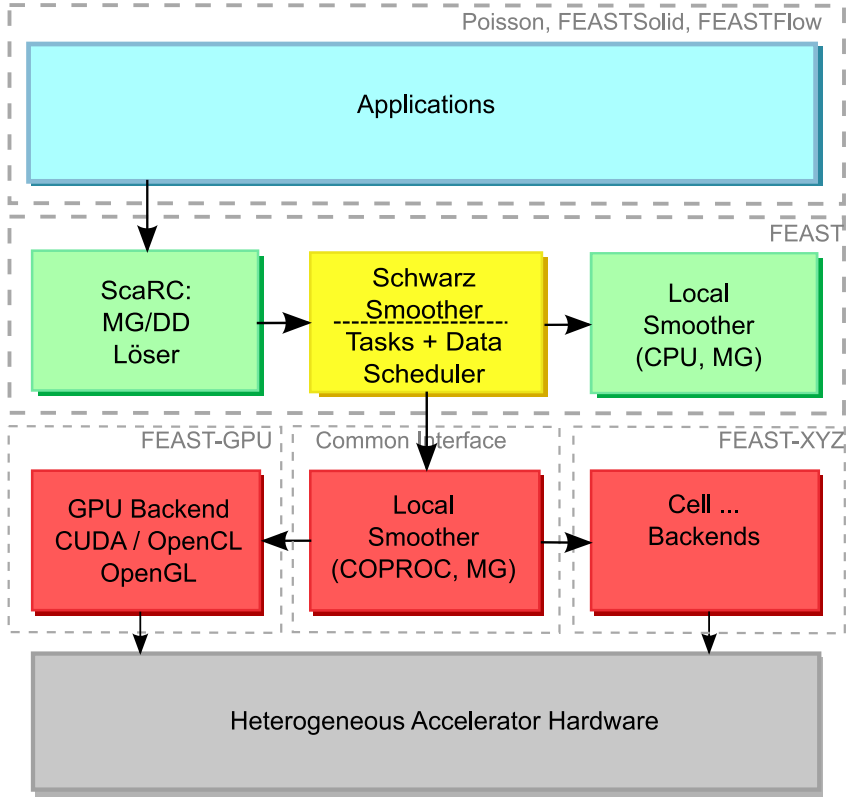
$\Rightarrow$ Tuned linear algebra **and** tuned solvers

Minimally invasive GPU integration

Naive straight-forward approach

- GPUs as accelerators of the most time-consuming component
- CPUs: outer MLDD solver, no changes to applications
- Wouldn't do it this way any more today, yet still instructive

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

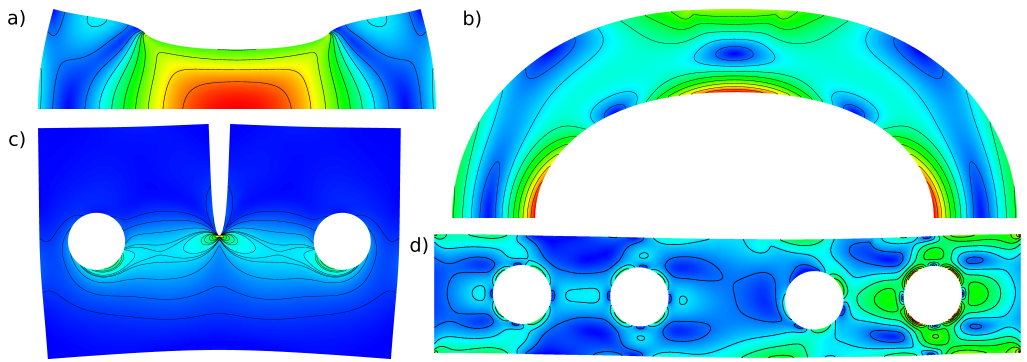


Example: linearised elasticity

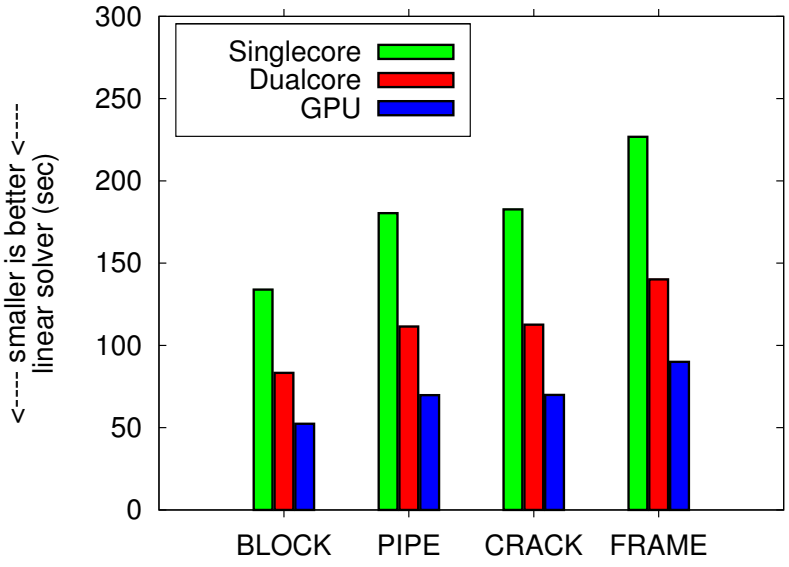
$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix} = \mathbf{f}$$

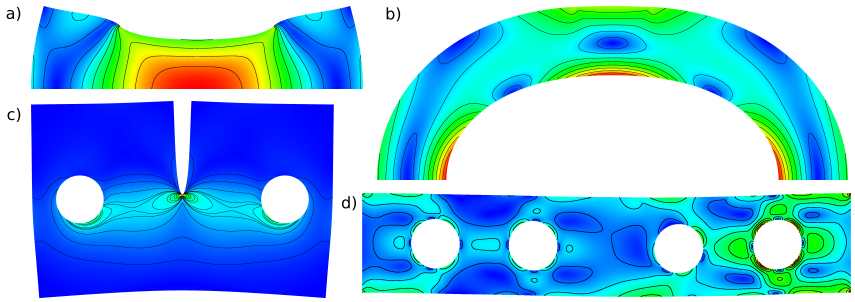
$$\begin{pmatrix} (2\mu+\lambda)\partial_{xx} + \mu\partial_{yy} & (\mu+\lambda)\partial_{xy} \\ (\mu+\lambda)\partial_{yx} & \mu\partial_{xx} + (2\mu+\lambda)\partial_{yy} \end{pmatrix}$$

global multivariate BiCGStab
block-preconditioned by
Global multivariate multilevel (V 1+1)
additively smoothed (block GS) by
for all Ω_i: solve $\mathbf{A}_{11}\mathbf{c}_1 = \mathbf{d}_1$ by
local scalar multigrid
update RHS: $\mathbf{d}_2 = \mathbf{d}_2 - \mathbf{A}_{21}\mathbf{c}_1$
for all Ω_i: solve $\mathbf{A}_{22}\mathbf{c}_2 = \mathbf{d}_2$ by
local scalar multigrid
coarse grid solver: UMFPACK



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

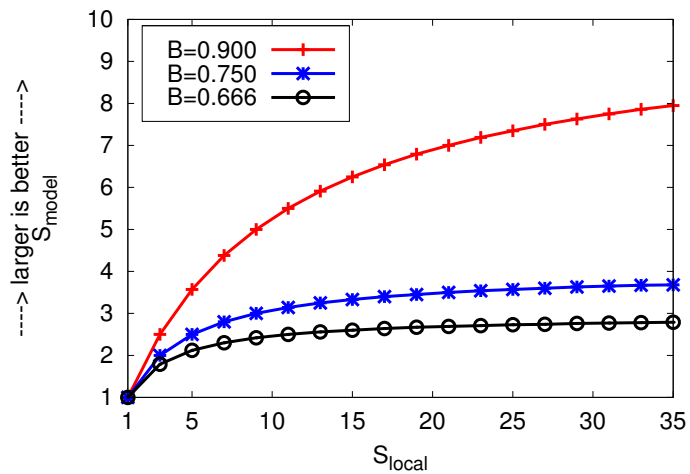
Speedup analysis

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

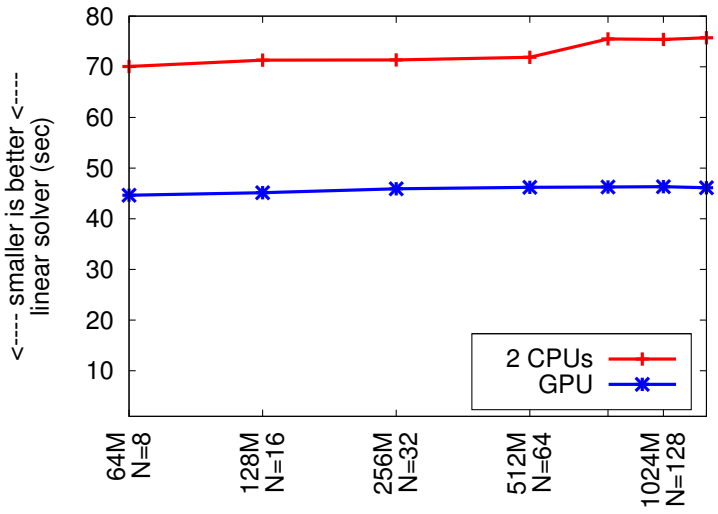
Accelerable fraction R_{acc}	66%
Local speedup S_{local}	9x
Modeled speedup S_{model}	2.5x
Measured speedup S_{total}	2.6x
Upper bound S_{max}	3x

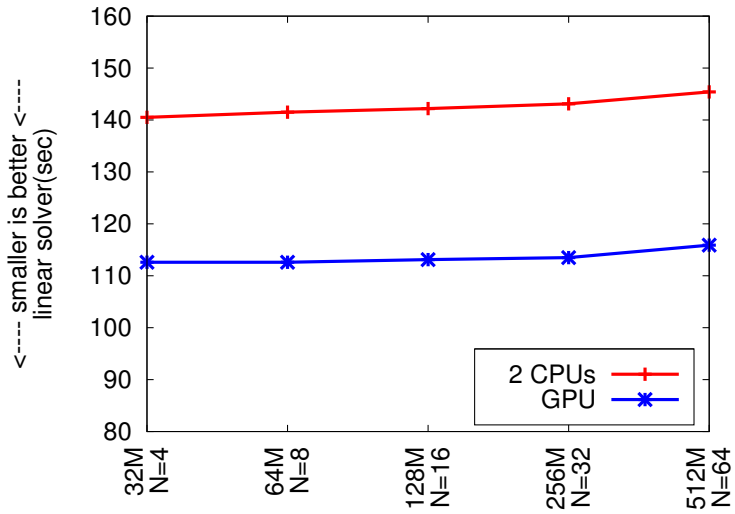


Weak scalability

Simultaneous doubling of problem size and resources

- Warning: ancient pre-CUDA results (dual 1c-Xeon / FX1400 nodes)
- Left: Poisson, 160 nodes, max. 1.3 B DOF
- Right: linearised elasticity, 64 nodes, max. 0.5 B DOF
- No loss of weak scalability despite local acceleration





Energy efficiency

Case study on an ARM cluster

Motivation

Importance of energy efficiency

- At exascale, *everything* is measured by its energy efficiency
  - 30× improvement in flops/Watt required over current BG/Q, 50× over commodity clusters just to stay in the reasonable power envelope of 20 MW
- Even at reasonable scales, running cost of a machine matters most

Promising 'new' processor architecture

- Low-power processor designs stemming from embedded systems now capable enough for HPC: dedicated FPUs, multicores, vector registers, reasonably-sized pipelines ...
- Tremendous market volume, probably couple hundred GFLOP/s in the smartphones and tablets in this room
- Almost all these chips contain IP by ARM Ltd., e.g. Apple A4/A5, ARM Cortex-A series, Samsung, ...

What is 'energy efficiency'?

In fact, a tricky question for application people

- Flops/Watt not meaningful at application level (Linpack anyone)
- Slower units more energy-efficient even if it takes longer?
- Many more units to reach same speed still more energy-efficient?
- Time-to-solution and energy-to-solution!

Weak and strong scaling equally critical

- Weak: need many more units because of much lower memory/node
- Strong: need many more units to compensate for slower per-node execution
- Application-specific sweet spots, how much slower can we afford to be if it cuts our electricity bill in half?

Prototype cluster: tibidabo @ BSC

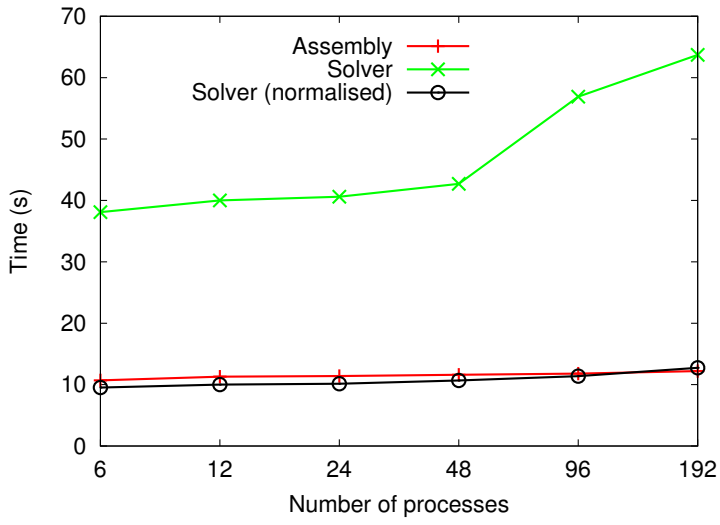
Machine details

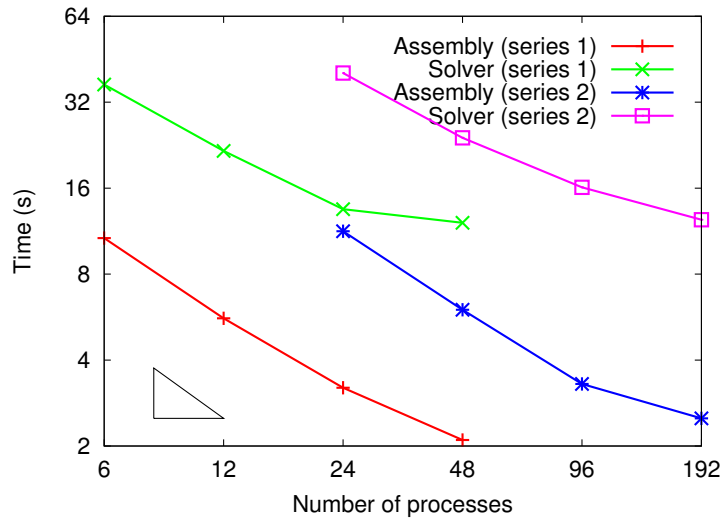
- 96 dual-core NVIDIA Tegra-2 SoCs based on ARM Cortex-A9
- Hosted in SECO Q7 carrier boards (nodes, essentially developer kits)
- 1 GB memory per dualcore, diskless (network FS), 1 GbE MPI
- Measured 5.7–7.5 W per node depending on load
- 2 W for the ARM alone, plus other SoC components, USB, GbE, blinking LEDs

Porting effort

- Standard Linux OS with GNU compiler and debugger toolchain
- Main limitation: instruction issue rate $\Rightarrow$ sophisticated cache-blocking strategies counterproductive, bookkeeping and index overhead for deeper nested loops
- Serial codes only reach half of the memory bandwidth per node!

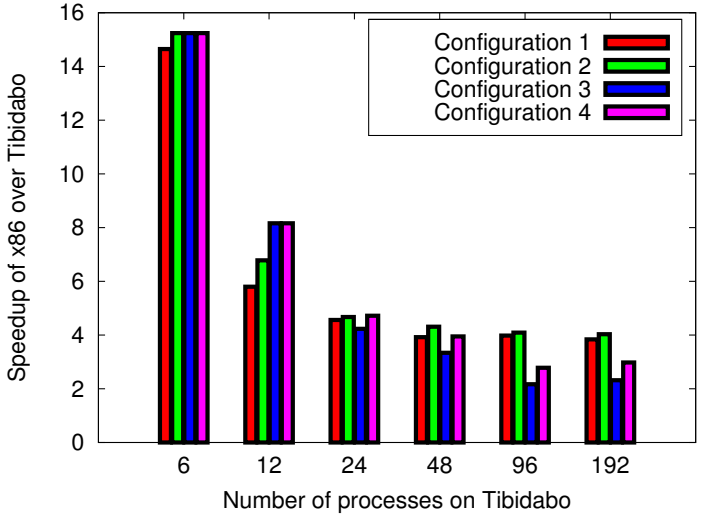
Scalability: FEAST

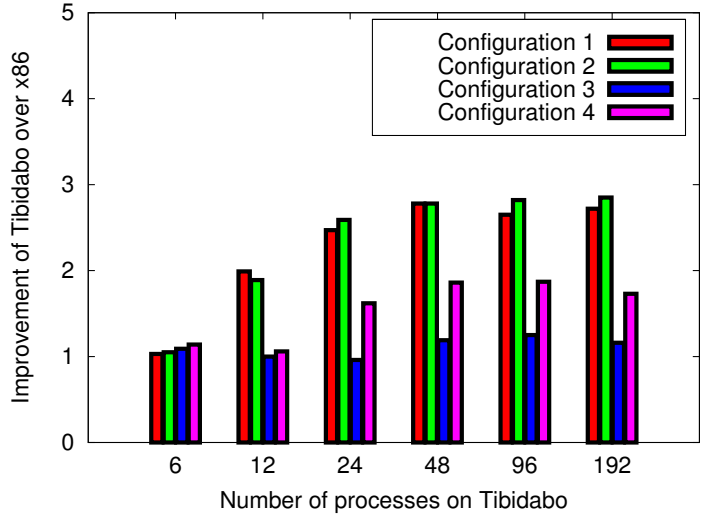




- Weak (left) and strong (right) scaling
- Weak scaling as expected (bump is a granularity effect from 4 to 5 solver iterations)
- Strong scaling quite ok (gigE and a multilevel method)

Power-performance analysis: FEAST

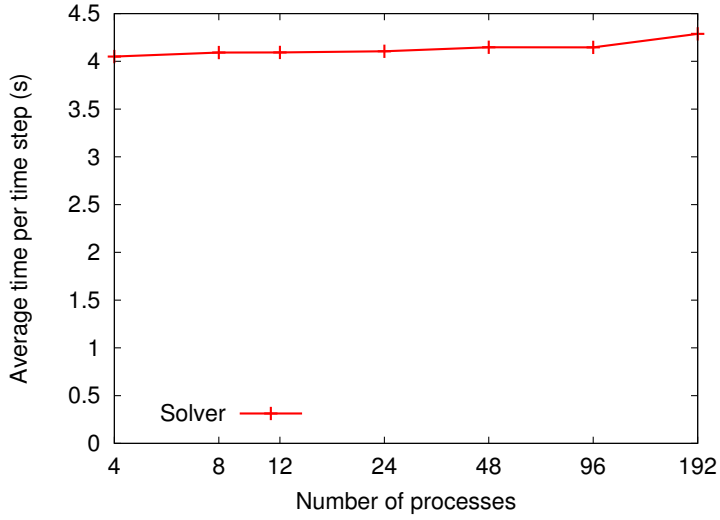


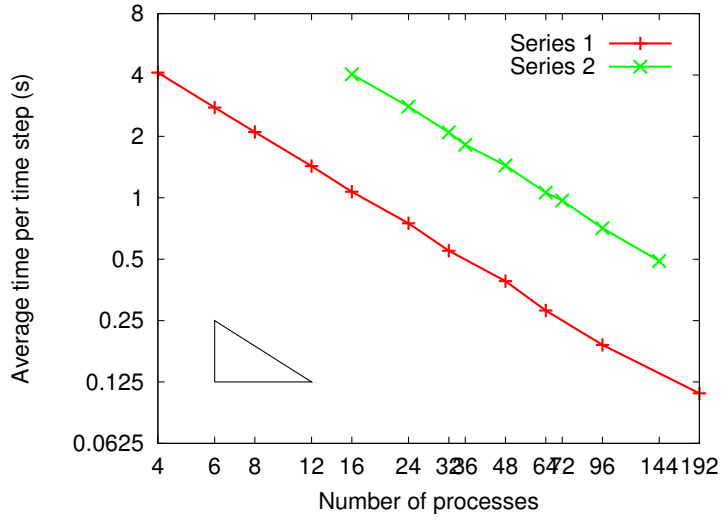


- Speedup x86 over ARM (left), improvement energy-to-solution ARM over x86 (right)
- Always more beneficial to use the ARM cluster
- As long as ≥ 2 x86 nodes are necessary, slowdown only 2–4

reference system: 32 2-way 4-core Nehalem system, 24 GB per node
(1) same partitioning (6 x86 cores/node), (2) re-partition for 8 cores/node,
(3) as few x86 cores/node as possible, (4) twice of that

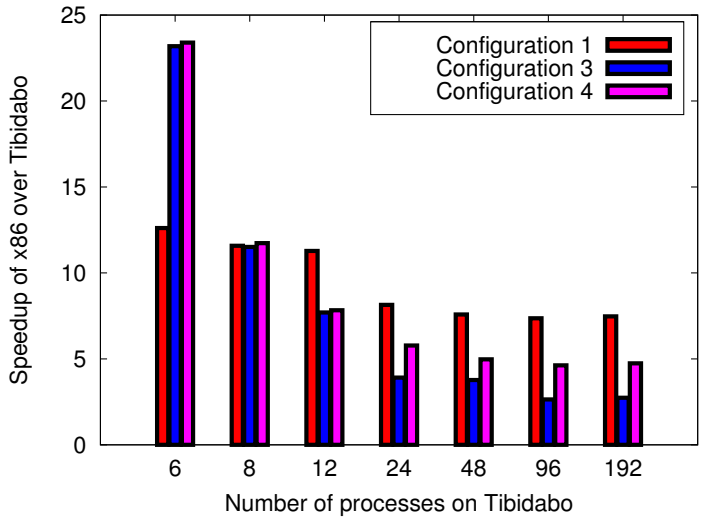
Scalability: SPECFEM3D_GLOBE

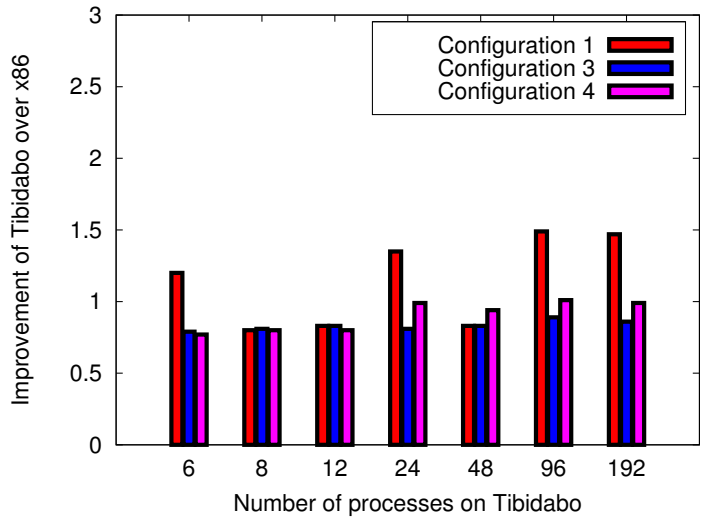




- Weak (left) and strong (right) scaling
- Both perfect (explicit in time, quite dense)

Power-performance analysis: SPECFEM3D_GLOBE





- Speedup x86 over ARM (left), improvement energy-to-solution ARM over x86 (right)
- Not always more beneficial to use the ARM cluster due to many small dense matrix matrix multiplications

Summary

Summary

Hardware-oriented numerics

- Parallelism, heterogeneity and memory wall on all scales of scientific computing, from chips to big machines
- Without incorporating knowledge on the way hardware evolves into the design of numerical methods, only fractions of peak performance can be achieved and codes might become slower over time

Energy efficiency

- Will become the ultimate quality criterion
- Using many low-power resources necessitates perfect weak and strong scaling even for moderate-size problems

Acknowledgements

Collaborative work with

- C. Becker, S. Buijssen, S. Turek (TU Dortmund)
- R. Strzodka (Max Planck Institut Informatik, currently NVIDIA Research)
- J. Mohd-Yusof, P. McCormick (Los Alamos National Laboratory)
- D. Komatitsch (CNRS Marseille), A. Ramirez, N. Rajovic, N. Puzovic (Barcelona Supercomputing Center)

Papers, links, tutorials and other stuff...

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

Funding

- DFG Priority Program 1648: 'Software for Exascale Computing'
- NVIDIA Teaching Center Program