Total efficiency of core components in Finite Element frameworks

Markus Geveler

Inst. for Applied Mathematics
TU Dortmund University of Technology, Germany
markus.geveler@math.tu-dortmund.de

MAFELAP13: Large scale computing with applications
London, June 13th 2013

technische universität dortmund
fakultät für mathematik

motivation

today's HPC facilities

- comprise heterogeneous compute nodes
  - multicore CPU(s) + some accelerator very common (GPU, Cell)
  - next generation accelerators upcoming (Intel XEON Phi)
  - even heterogeneity on-a-chip (SoCs)
- cost efficiency dominated by *energy-efficiency*

today's large-scale FEM codes

- have to adapt to target hardware
  - heterogeneity and frameworking
  - parallelisation of applications (DD mostly)
  - parallelisation of core components (e.g. 'linear solver on GPU')
  - optimisation with respect to many details (data-flow and SIMD mostly)
- can we have the same results with less energy? $\rightarrow$ TTS vs TES

total efficiency

(some) aspects of efficiency

- numerical efficiency
  - dominates asymptotic behaviour and wall clock time
- hardware-efficiency
  - exploit all levels of parallelism provided by hardware (SIMD, multi-threading on a chip/device/socket, multi-processing in a cluster, hybrids)
  - *then* try to reach good scalability (communication optimisations, block comm/comp)
- energy-efficiency
  - by hardware mostly but codes may have to be adjusted ($\rightarrow$ portability)

Hardware-oriented Numerics: Enhance both hardware- and numerical efficiency simultaneously, use (most) energy-efficient hardware where available! Attention: codependencies!
Today's major example: (local) unstructured grid geometric Multigrid with Approximate Inverse smoothers on GPUs

FE-gMG

Today's (first) example: ingredients

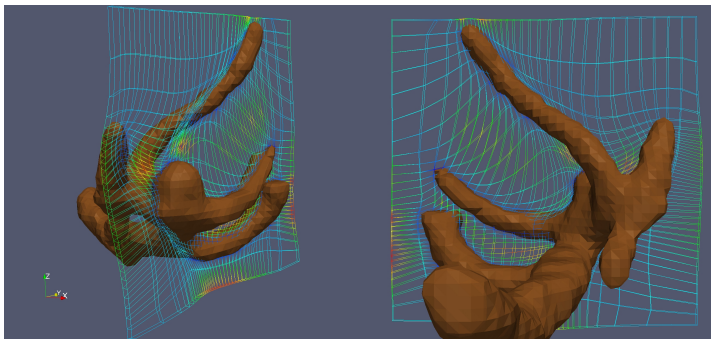
- (local) geometric multigrid
- for unstructured grids
- with Approximate Inverse smoothers
- with FE transfer operators
- with clever DOF sorting
- on GPUs (and multicore CPUs)
- all based on one kernel: SpMV

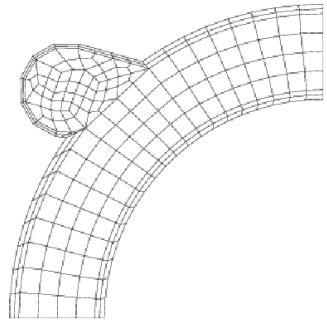
why local?

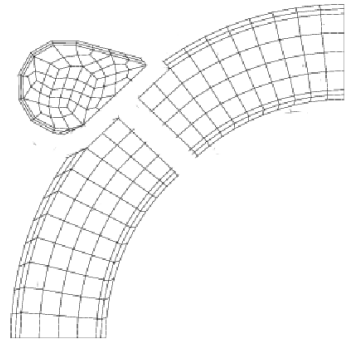
- because large scale HPC starts 'in the little' $\rightarrow$ on one heterogeneous compute node
- consider a very slow compute node $\rightarrow$ perfect scaling, but good?
- consider a very bad single-node implementation $\rightarrow$ perfect scaling, but good?

FE-gMG as a core-component

Coarse grained parallelism by domain decomposition (Schwarz)



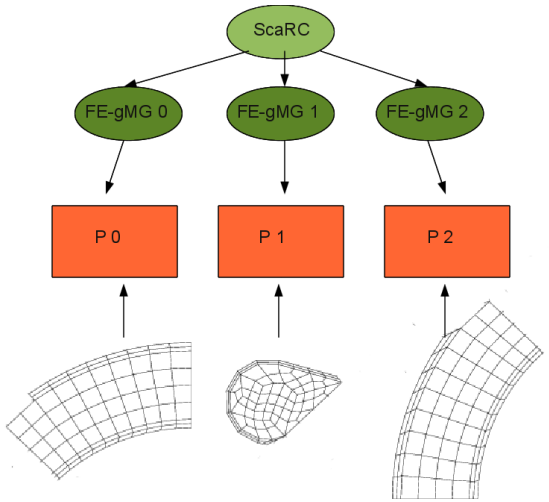




1. use conformal coarse mesh as starting point
2. cluster patches for local problem assembly (structured and unstructured! - many HPC/GPGPU examples are structured)
3. load-balance patches

linear solver

ScaRC pattern



- define data-parallel solver pattern globally (multigrid)
- use special smoother as local solver: *recursion* or *blockwise local solver* $\rightarrow$ *FE-gMG*
- patch type determines solver components
- apply the smoother: global defect $\rightarrow$ local solvers (recursion or FE-gMG) $\rightarrow$ global correction

ScaRC-preconditioner:

1. $d \leftarrow b - Ax$
2. $y \leftarrow \widetilde{\sum}_i R_i^T \mathsf{MG}(d)$, where $\mathsf{MG}(d)$:
   1. $d_i \leftarrow R_i d$
   2. $y_i \leftarrow \mathsf{FE\text{-}gMG}(B_i, d_i)$
   3. $y \leftarrow \widetilde{\sum}_i R_i^T y_i$
3. $x \leftarrow x + y$

FE-gMG

concentrate all tuning in one kernel: sparse matrix vector multiply (SpMV)

- in coarse-grid solver: preconditioned Krylov subspace methods
- smoother:
  - preconditioned Richardson iteration or
  - Krylov subspace method
  - local preconditioners by *approximate inverses*
- defect

the remainings

- a little BLAS-1 (dot-product, norm, scale, ...)
- important: *grid transfer operators* $\rightarrow$ can also be realised as SpMV

advantages

- flexibility (only matrices are switched) $\rightarrow$ blackbox
- oblivious of FE-space, dimension, ...
- performance-tuning concentrated

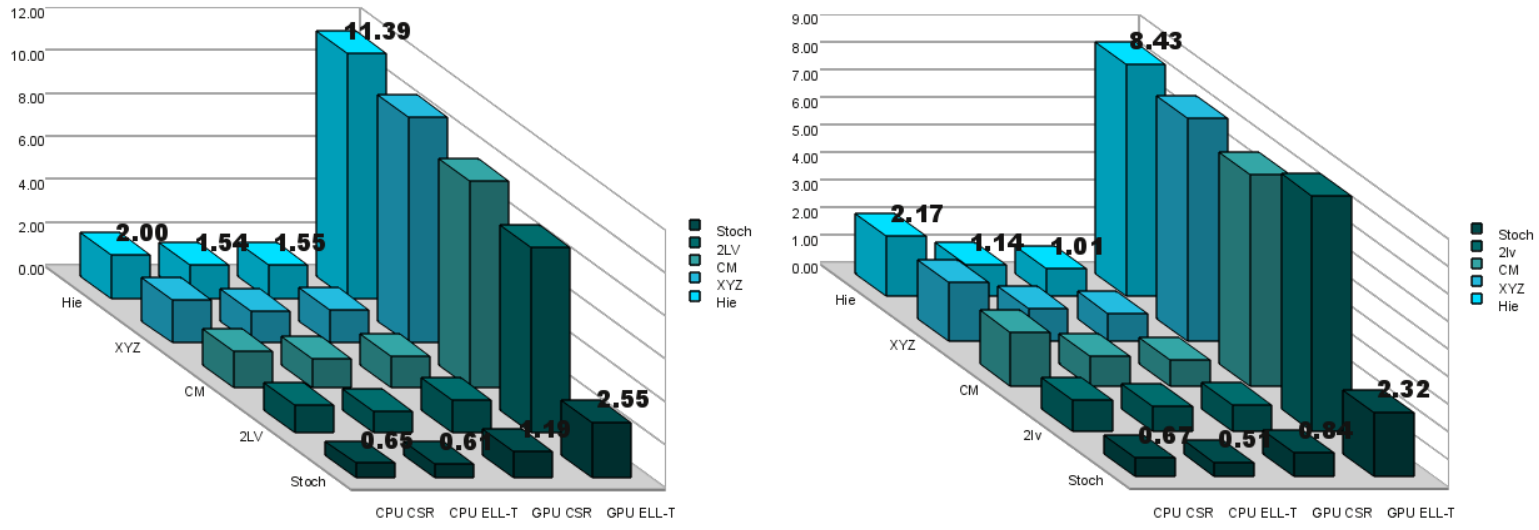
disadvantages

- we somewhat move the problem from solver to assembly of matrices

SpMV: CSR vs. ELLPACK-R

SpMV kernel - performance

- example: stiffness-matrices from a 3D-Poisson problem, left: Q_1, right Q_2 (more nonzeros)
- DOF numbering scheme: bright to dark $\rightarrow$ larger matrix-bandwidth



- $\rightarrow$ porting CSR-SpMV to GPUs catastrophic (access pattern)
- $\rightarrow$ numbering of DOFs / number of nonzeros performance-critical

SpMV on GPUs

ELLPACK-R

- store sparse matrix S in two arrays `A` (non-zeros in column-major order) and `j` (column-Index for each entry in `A`)
- `A` has (#rows in S) $\times$ (maximum #non-zeros in rows of S)
- shorter rows are filled
- additional array `rl` to store effective non-zeros count per row (get stop on row right)

$$S = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \Rightarrow \texttt{A} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \texttt{j} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix} \texttt{rl} = \begin{bmatrix} 2 \\ 2 \\ 3 \\ 2 \end{bmatrix}$$

advantages

- complete regular access pattern to $\mathbf{y}$ and A
- GPU implementation:
  - one thread for each element y_i
  - $\rightarrow$ access to all ELLPACK-R arrays and $\mathbf{y}$ completely coalesced (column-major)
  - $\rightarrow$ access on $\mathbf{x}$: use texture-cache (FERMI: L2-cache)
  - $\rightarrow$ no synchronisation between threads needed
  - $\rightarrow$ no branch-divergence
  - in addition: multiple threads can access one row (ELLPACK-T)
- *access to* $\mathbf{x}$ *depends on non-zero pattern of* A $\rightarrow$ bandwidth given by DOF-numbering

SpMV in gMG (1)

smoother, coarse grid solver

- preconditioned Richardson iteration:

$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \omega M(\mathbf{b} - A\mathbf{x}^k)$$

- CG or BiCGStab: preconditioner, defect, ...

smoother construction

- Jacobi only does not suffice
- good preconditioners often are inherently sequential
- preconditioner of the smoother reduced to SpMV
- $\rightarrow$ *Sparse Approximate Inverse*

smoother construction

SPAI

$$\| I - MA \|_F^2 \quad = \quad \sum_{k=1}^{n} \| e_k^{\mathrm{T}} - m_k^{\mathrm{T}} A \|_2^2 \quad = \quad \sum_{k=1}^{n} \| A^{\mathrm{T}} m_k - e_k \|_2^2$$

where e_k is the k-th unit vektor and m_k the k-th row of M. $\rightarrow$ for n columns of $M \rightarrow n$ *least squares* opt-problems:

$$min_{m_k} \| A^{\mathrm{T}} m_k - e_k \|_2, \quad k = 1, \ldots n.$$

- use non-zero pattern of the stiffness matrix for M

SAINV

- *Stabilised Approximate Inverse*
- calculate factorisation $A^{-1} = ZD^{-1}Z^T$ where Z and D are calculated explicitly: A-biconjugation applied to unit base
- Z is assembled incompletely: use drop-tolerance
- no structure constraints possible (as opposed to SPAI)
- $\rightarrow$ sometimes: better approximation of A^{-1}
- SAINV approximately as good as ILU(0)
- problem: inherently sequential

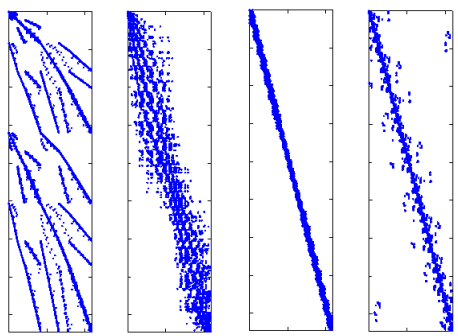
SpMV in gMG (2)

SpMV in grid transfer:

- two conformal FE-spaces V_{2h} and V_h
- with Lagrange-Basis: interpolation (grid-transfer) can be expressed as SpMV

prolongation matrix

$$(P_{2h}^h)_{ij} = \varphi_{2h}^{(j)}(\xi_h^{(i)})$$



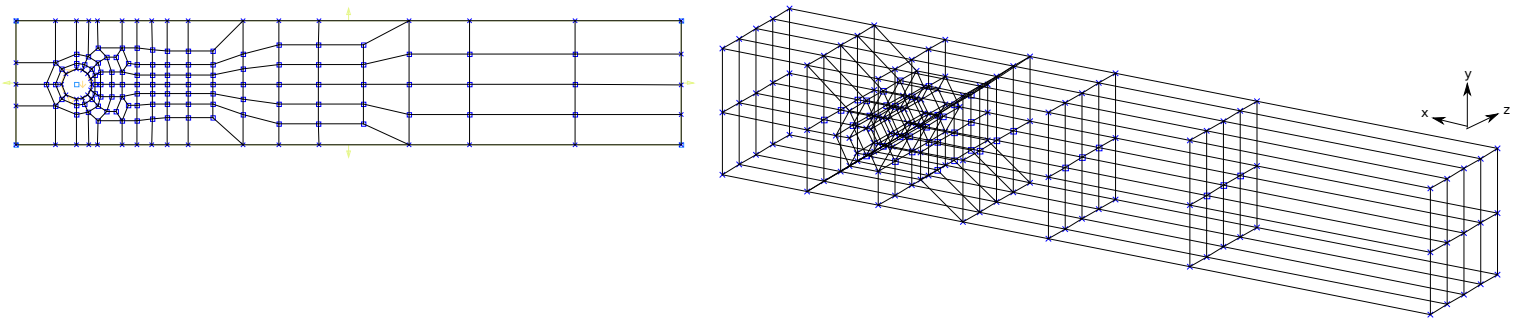
restriction matrix

$$R_h^{2h} = (P_{2h}^h)^T$$

- DOF numbering technique $\rightarrow$ performance

results

benchmark



- popular grid, unstructured, Poisson problem
- 2D and 3D, Q_1 and Q_2 FE, CPU (Core i7 980X, 6 threads) und GPU (Tesla C2070)
- Approximate Inverse strong smoothing (SPAI, SAINV)

$$\begin{cases} -\Delta u = 1, & \mathbf{x} \in \Omega \\ u = 0, & \mathbf{x} \in \Gamma_1 \\ u = 1, & \mathbf{x} \in \Gamma_2 \end{cases}$$

- different FE-spaces
- different DOF numbering techniques

results

FE-gMG: (2D)

Q1	CPU								GPU							
	Jacobi		SPAI			SAINV			Jacobi			SPAI			SAINV			
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	4.04	13	2.54	5	1.59	3.59	6	1.12	1.06	13	3.82	0.56	5	1.88	4.53	1.19	6	0.89	3.01
CM	3.65	13	2.19	5	1.66	3.29	6	1.11	1.03	13	3.55	0.72	5	1.43	3.05	0.82	6	1.26	4.03
XYZ	3.48	13	2.06	5	1.69	4.44	9	0.78	0.98	13	3.53	0.51	5	1.93	4.04	1.03	9	0.96	4.32
Stoch	4.04	13	2.57	5	1.57	3.19	5	1.27	1.74	13	2.33	1.04	5	1.66	2.46	1.29	5	1.35	2.47
Hie	3.49	13	2.07	5	1.69	3.07	6	1.14	0.97	13	3.59	0.50	5	1.94	4.14	0.77	6	1.26	3.98

Q2	CPU								GPU							
	Jacobi		SPAI			SAINV			Jacobi			SPAI			SAINV			
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	13.19	22	4.87	5	2.71	10.24	9	1.29	2.27	22	5.80	0.93	5	2.44	5.22	2.04	9	1.12	5.02
CM	11.40	22	4.40	5	2.59	10.58	12	1.08	2.50	22	4.56	1.02	5	2.44	4.30	2.20	12	1.13	4.80
XYZ	11.29	22	4.21	5	2.68	9.58	12	1.18	2.41	22	4.69	0.99	5	2.44	4.26	2.70	12	0.89	3.55
Stoch	12.92	22	5.14	5	2.51	4.64	9	2.79	4.78	22	2.70	2.04	5	2.35	2.52	1.57	9	3.05	2.96
Hie	11.25	22	4.24	5	2.66	8.68	9	1.30	2.44	22	4.60	1.00	5	2.43	4.22	1.88	9	1.30	4.61

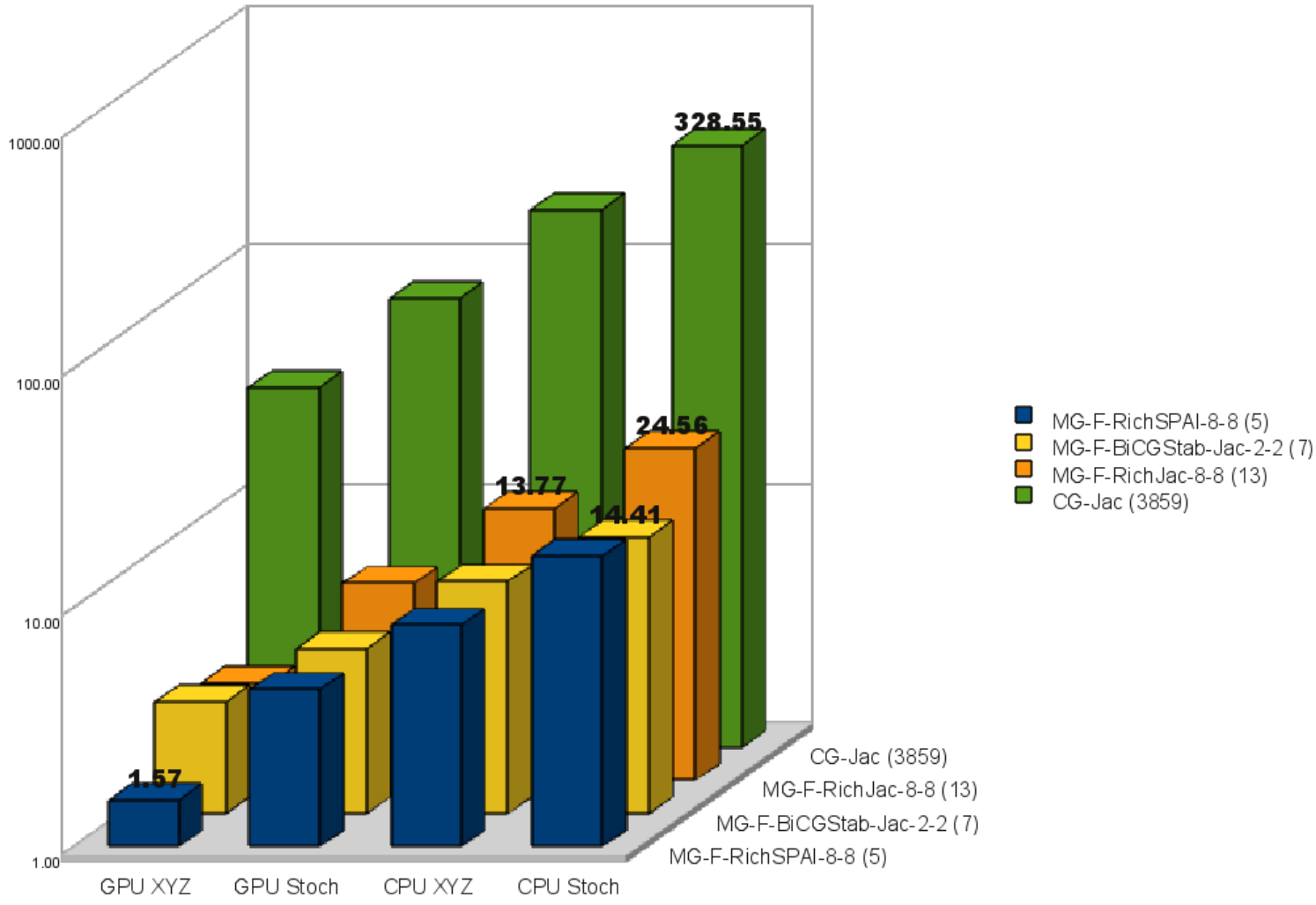
FE-gMG: (3D)

Q1	CPU								GPU							
	Jacobi		SPAI			SAINV			Jacobi			SPAI			SAINV			
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	2.43	26	1.08	7	2.25	1.03	9	2.37	0.66	26	3.71	0.27	7	2.39	3.94	0.28	9	2.32	3.63
CM	2.34	26	1.02	7	2.30	0.98	9	2.37	0.66	26	3.53	0.28	7	2.39	3.67	0.29	9	2.26	3.36
Stoch	2.63	26	1.18	7	2.23	1.28	10	2.06	0.75	26	3.48	0.33	7	2.32	3.61	0.38	10	1.98	3.35

Q2	CPU								GPU							
	Jacobi		SPAI			SAINV			Jacobi			SPAI			SAINV			
sort	time	#iter	time	#iter	speedup jac	time	#iter	speedup jac	time	#iter	speedup cpu	time	#iter	speedup jac	speedup cpu	time	#iter	speedup jac	speedup cpu
2lv	9.86	42	3.09	8	3.19	2.44	10	4.04	2.01	42	4.90	0.58	8	3.44	5.29	0.56	10	3.60	4.37
CM	7.46	42	2.50	8	2.99	2.41	12	3.10	2.31	42	3.23	0.70	8	3.32	3.59	0.73	12	3.18	3.32
Stoch	8.89	42	3.14	8	2.83	2.90	12	3.07	2.92	42	3.04	0.92	8	3.18	3.41	0.92	12	3.19	3.16

results

FE-gMG: combined effects: gMG+ AI smoother + DOF numbering + GPU



results

complete geometric multigrid on GPUs

- completely unstructured grids possible
- high extensibility potentials: smoother
- DOF-numbering still critical
- careful combination of hardware- and numerical efficiency offers up to 3 orders of magnitude speedup!
- matrix assembly not considered (stiffness/mass, transfer, preconditioner) $\rightarrow$ random access matrices needed!

heterogeneity on a node also means incorporating all resources!

---

example solver: SWE with multiple extensions

- hardware-oriented numerics: use well parallelisable algorithms, where possible (accuracy!)
- here: sophisticated free-surface flow solver based on SWE solved with LBM (bed friction, wind, pollutant transport, FSI): MPI + PThreads + SSE + CUDA

$$\frac{\partial h}{\partial t}+\frac{\partial (hu_j)}{\partial x_j}=0 \quad \text{and} \quad \frac{\partial hu_i}{\partial t}+\frac{\partial (hu_iu_j)}{\partial x_j}+g\frac{\partial}{\partial x_i}(\frac{h^2}{2})=S_i^{\text{bed}}+S_i^{\text{wind}}$$

$$\frac{\partial hc}{\partial t}+\frac{\partial (hu_jc)}{\partial x_j}=\frac{\partial}{\partial x_j}\left(Dh\frac{\partial c}{\partial x_j}\right)+S^{\text{poll}}$$

$$S_i^{\text{bed}}=-g\left(h\frac{\partial b}{\partial x_i}+n_b^2h^{-\frac{1}{3}}u_i\sqrt{u_ju_j}\right)$$

$$S_i^{\text{wind}}=(\rho_\alpha 10^{-3}\times(0.75+0.0067\sqrt{w_1^2+w_2^2}))(w_1\sqrt{w_1^2+w_2^2})$$

$$S^{\text{poll}}=-Khc+S_0h$$

heterogeneity on a node also means incorporating all resources!

LBM for SWE

$$f_\alpha(\mathbf{x}+\mathbf{e}_\alpha\Delta t, t+\Delta t) = f_\alpha(\mathbf{x},t) + Q(f_\alpha, f_\beta), \qquad \beta = 1,\ldots,k.$$

$$f_\alpha^{\text{temp}}(\mathbf{x},t) = f_\alpha(\mathbf{x},t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq})$$

$$f_\alpha^{\text{eq}} = \begin{cases} h(1 - \frac{5gh}{6e^2} - \frac{2}{3e^2}u_i u_i) & \alpha = 0 \\ h(\frac{gh}{6e^2} + \frac{e_{\alpha i}u_i}{3e^2} + \frac{e_{\alpha j}u_i u_j}{2e^4} - \frac{u_i u_i}{6e^2}) & \alpha = 1,3,5,7 \\ h(\frac{gh}{24e^2} + \frac{e_{\alpha i}u_i}{12e^2} + \frac{e_{\alpha j}u_i u_j}{8e^4} - \frac{u_i u_i}{24e^2}) & \alpha = 2,4,6,8 \end{cases}$$

$$f_\alpha(\mathbf{x}+\mathbf{e}_\alpha\Delta t, t+\Delta t) = f_\alpha(\mathbf{x},t) - \frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + \frac{\Delta t}{6e^2}e_{\alpha i}S_i, \ \alpha = 0,\ldots,8.$$

$$h(\mathbf{x},t) = \sum_\alpha f_\alpha(\mathbf{x},t) \quad \text{and} \quad u_i(\mathbf{x},t) = \frac{1}{h(\mathbf{x},t)}\sum_\alpha e_{\alpha i} f_\alpha,$$

$$g_\alpha^{\text{temp}}(\mathbf{x},t) = g_\alpha(\mathbf{x},t) - \frac{1}{\tau_{\text{poll}}^{\text{var}}}(g_\alpha - g_\alpha^{eq})$$

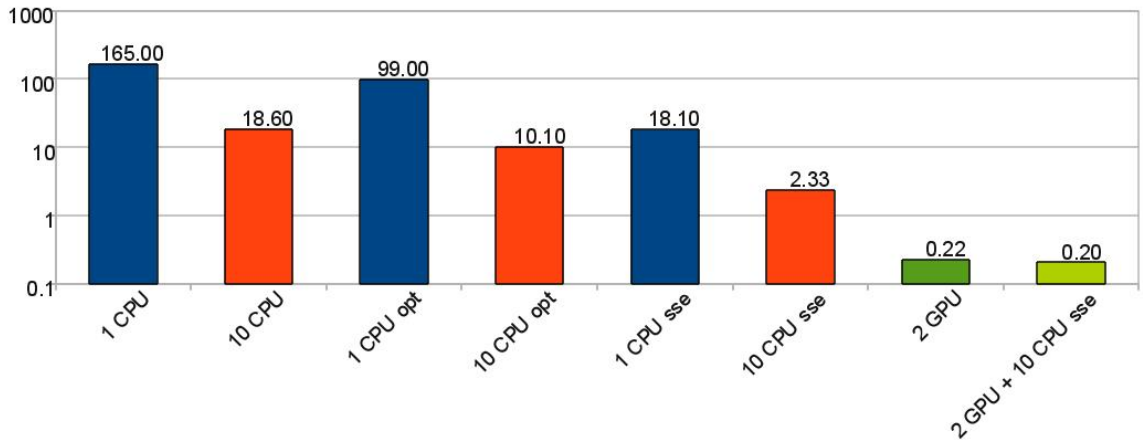
$$\tau_{\text{poll}}^{\text{var}} = 1/2 + h(\mathbf{x},t) \times (\tau_{\text{poll}} - 1/2)$$

$$g_\alpha^{\text{eq}} = \begin{cases} c(h - 5/9) & \alpha = 0 \\ c(1/9 + \frac{h}{3e^2}e_{\alpha i}u_i) & \alpha = 1,3,5,7 \\ c(1/36 + \frac{h}{12e^2}e_{\alpha i}u_i) & \alpha = 2,4,6,8 \end{cases}$$

heterogeneity on a node also means incorporating all resources!

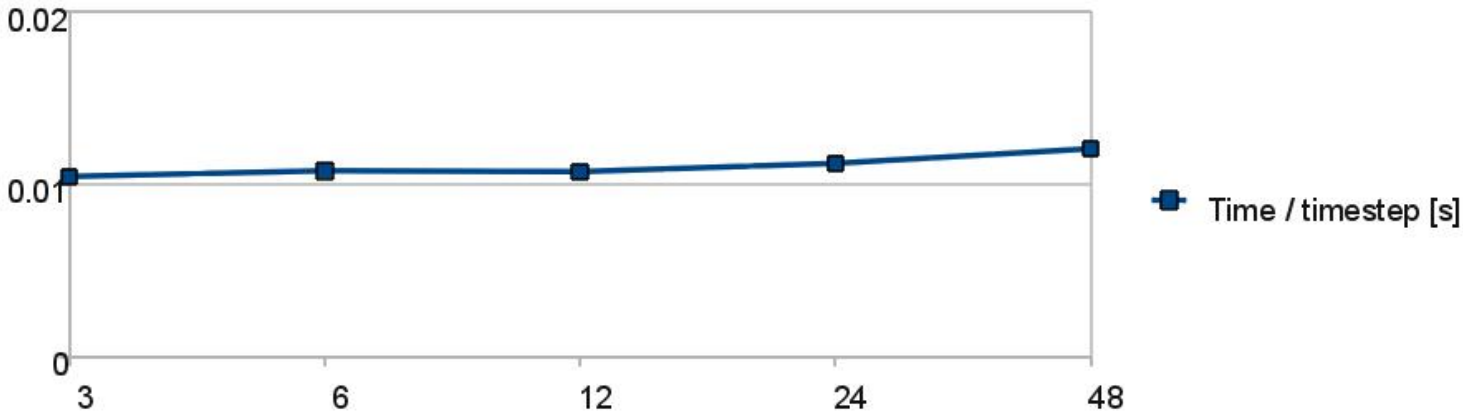
example, single node performance

- CINECA IBM-PLX GPU cluster:
  - 2 6-core Westmeres and 2 NVIDIA Tesla GPUs per node
  - Infiniband
  - full features (flow + pollutant)
  - $2^l * (2000 \times 2000)$ lattice sites and $3 * (2^l)$ nodes on refinement level l



heterogeneous compute nodes

example, scaling and finals



- → optimisation concerning vectorisation is crucial for CPU performance
- → in some cases: compiler unable to vectorise kernel loops at all (bed force term)
- → good serial performance only granted by organising loops / register usage by hand
- **→ hybrid pays off (10 percent is quite good!), if CPU kernels are reasonably optimised, load reasonably balanced**

total time to solution vs total energy to solution

so far: combining hardware- and numerical efficiency

- now: what about energy?
- example: GPGPU: specialist accelerator
- in general: exploiting hardware that is considered to be more energy-efficient because it stems from the embedded fields (lower transistor count due to lower instr. set compatibility, mainly)
- often acceptable: decrease in total energy consumption bought with increase of total time to solution

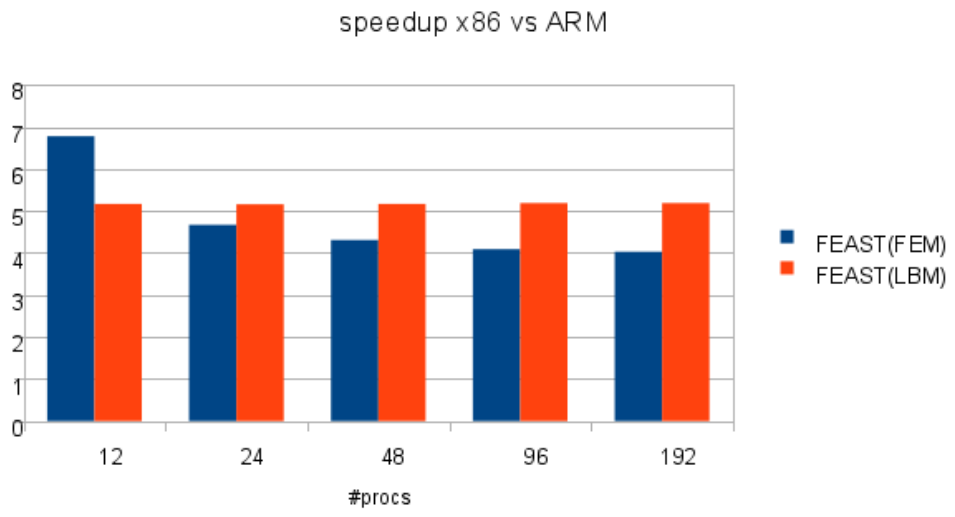
example: TIBIDABO prototype cluster (BSC)

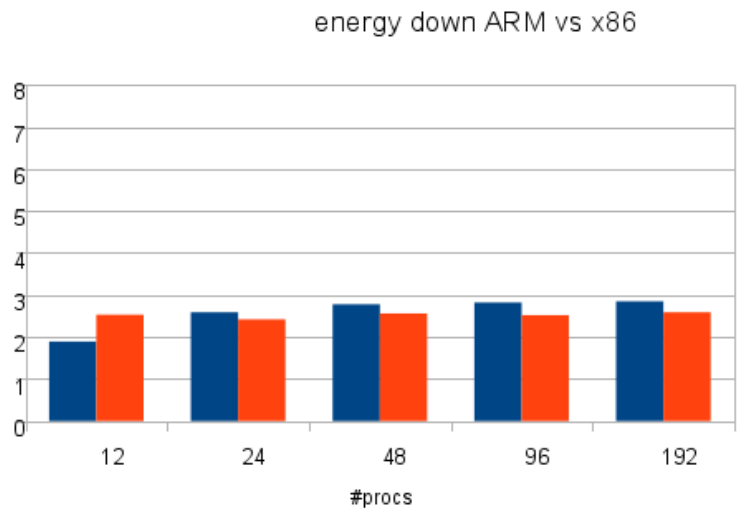
- 1 NVIDIA Tegra 2 SoC (dual core ARM Cortex-A9) per core
- LPDDR2 memory at low timings
- no SIMD

ARM vs x86

TIBIDABO vs LiDO

- TIBIDABO ARM cluster, up to 96 nodes
- Dortmund x86 cluster LiDO, up to 32 nodes (2x Nehalem dual socket quad core, SSE, DDR3)





conclusions

total efficiency

- hardware efficiency and numerical efficiency have to be augmented carefully and simultaneously $\rightarrow$ codependencies
- less total time to solution can (quite) easily be traded for less energy consumption
- energy efficiency of ARM architecture is (and is expected to be) increasing rapidly (SIMD, better caches, faster memory)

TODOs

- matrix assembly!!
- 'block-Jacobi' character of the parallel scheme
- overlapping comm/comp

Acknowledgements

Thanks to BSC for hardware access, especially to Alex Ramirez, Nicola Rajovic and Nicola Puzovic.
Thanks to the Dortmund LiDO team at DOWIR.
Thanks to all contributors to FEAST, especially Dirk Ribbrock, Peter Zajac and Dominik Göddeke.

This work was granted access to the HPC resources of CINECA made available within the Distributed European Computing Initiative by the PRACE-2IP, receiving funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement number RI-283493.

Supported by BMBF, *SKALB project 01IH08003D*.
Supported by German Research Foundation (DFG), projects SFB 708/TB 1 and SPP 1423 (TU 102/32-2)

Thanks to Raphael Münster for image material.