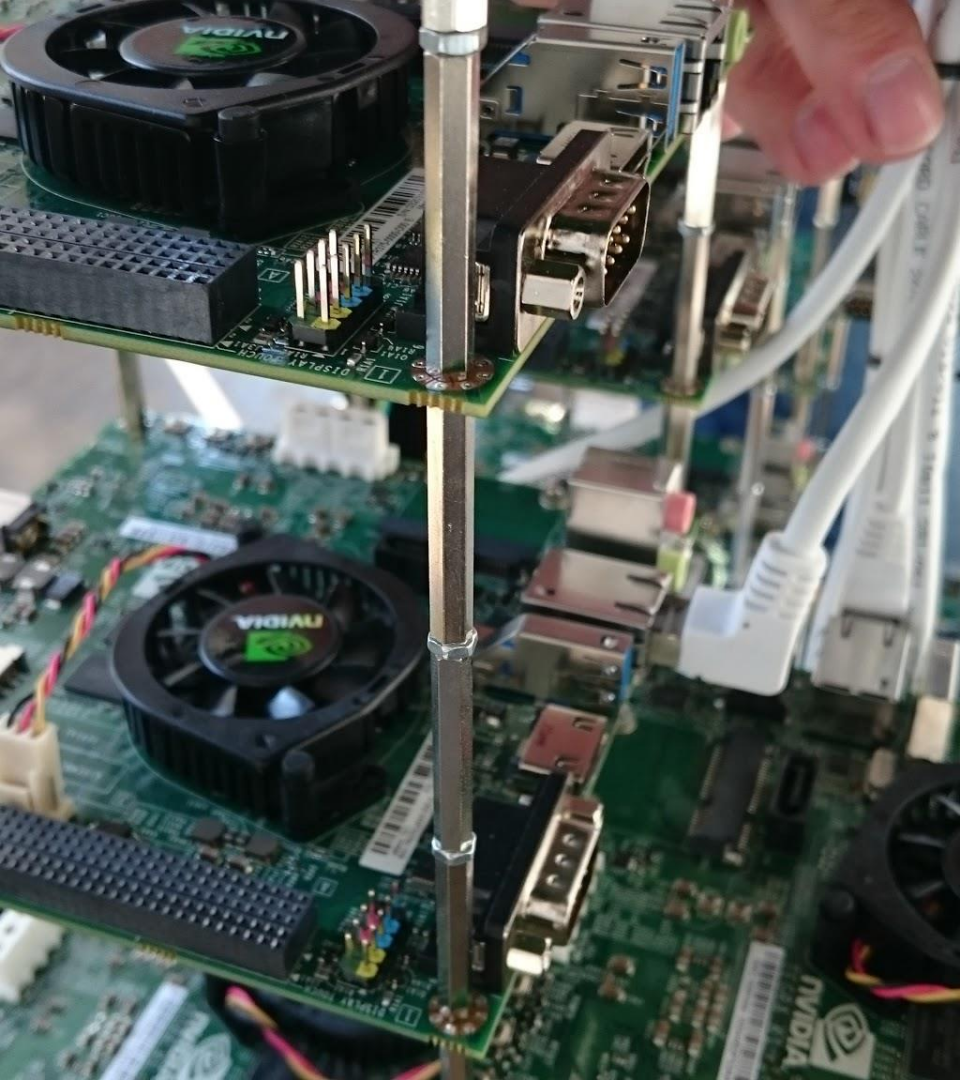
Machine Learning Approaches for the Acceleration of the Linear Solver in PDE Simulations

Markus Geveler,
Hannes Ruelmann,
Stefan Turek

@ Sci. Comp. + ML,
Nottingham 2019

technische universität dortmund
fakultät für mathematik



technische universität dortmund

fakultät für mathematik

This is us @ TU Dortmund University

markus.geveler@math.tu-dortmund.de

hannes.ruelmann@math.tu-dortmund.de

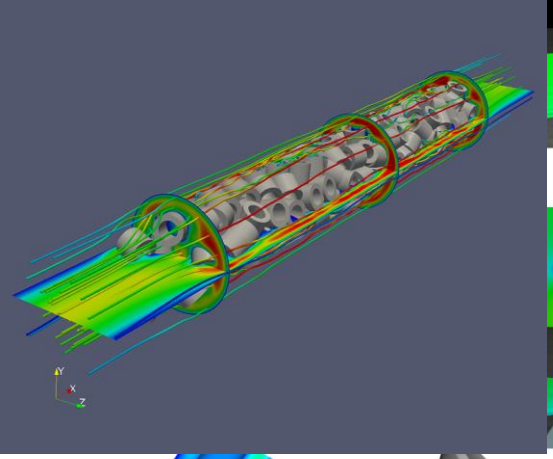
stefan.turek@math.tu-dortmund.de

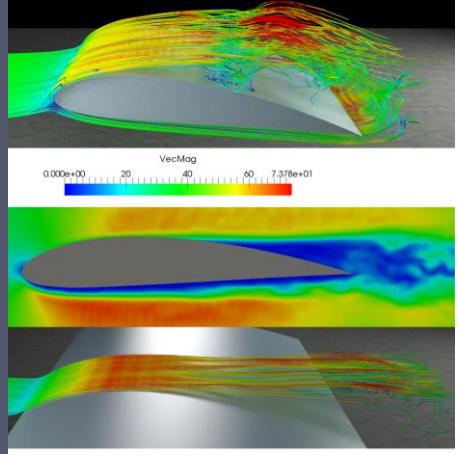
Advanced PDE techniques for incompressible flows,
Hardware-oriented Numerics,
Multilevel domain decomposition solvers,
High performance and high quality CFD simulations

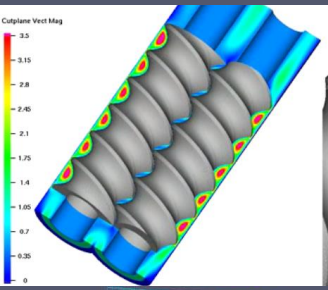
Unconventional High Performance Computing

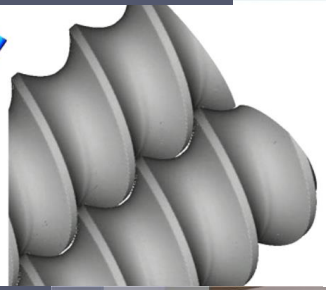
FEAT

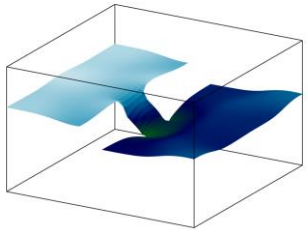
www.mathematik.tu-dortmund.de/lsiii/





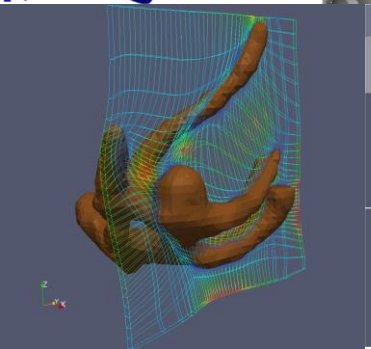


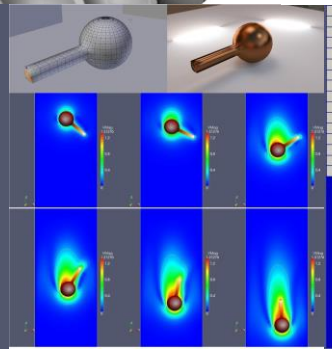


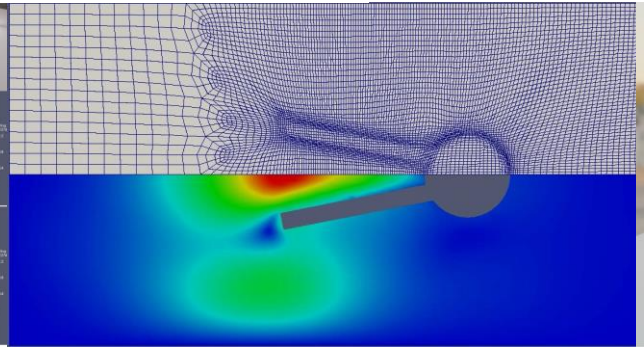


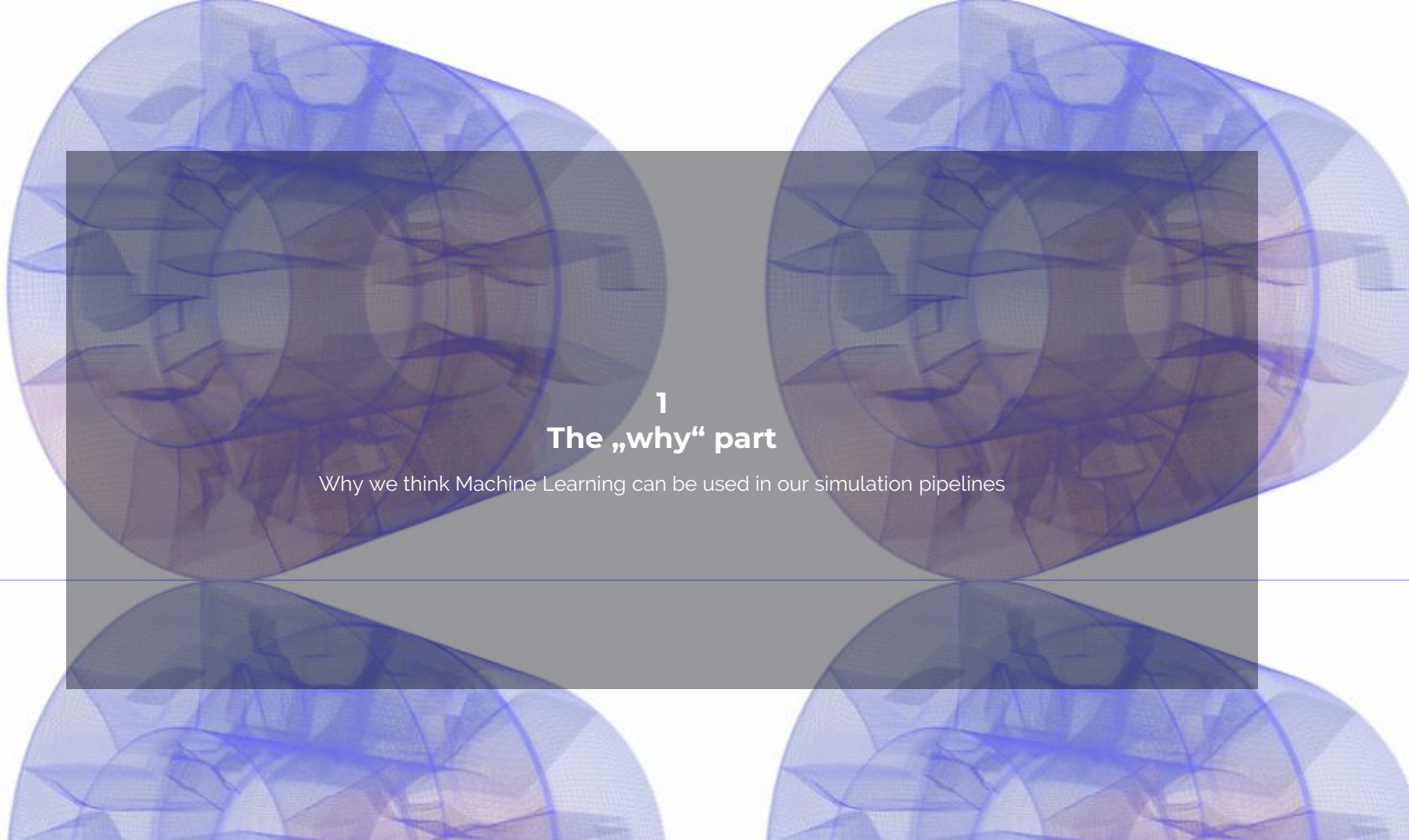
Where everything is leading: simulation of technical flow

- **Task**: determine physical quantities for a huge number of points in space and repeat very often
- **Multiphysics:** increasing complexity of problems and thus methods
- **Models are versatile**: applications in many fields
- **Ressource-hungry:** memory, time, energy
- **Methods:** DD Newton-Krylov-***Multigrid schemes +FEM***









1
The „why“ part

Why we think Machine Learning can be used in our simulation pipelines

Parallel Multilevel **Pressure** Schur Complement solver

- **Pressure Poisson Problem** consumes most of the time
- Recursive domain decomposition Newton-Krylov-***multigrid schemes***
- Local **geometric multigrid** highly hardware-optimised and accelerated with GPUs or similar
- **smoother** determines overall efficiency

How we solve the incompressible NSEs

NSEs

$$\begin{aligned} u_t - \nu\Delta u + u\cdot\nabla u + \nabla p &= f, \\ -\nabla\cdot u &= 0 \end{aligned}$$

weak form

$$\begin{aligned} \int_\Omega \partial_t u\ v\ dx + \int_\Omega (u\cdot\nabla u)\ v\ dx - \nu\int_\Omega \Delta u\ v\ dx + \int_\Omega \nabla p\ v\ dx &= \int_\Omega f\ v\ dx \\ \int_\Omega (\nabla\cdot u)\ q\ dx &= 0 \end{aligned}$$

FEM

$$\begin{aligned} (\partial_t u_h, v_h) + (u_h\cdot\nabla u_h, v_h) + \nu(\nabla u_h, \nabla v_h) - (p_h, \nabla\cdot v_h) &= (f, v_h) \\ -(q_h, \nabla\cdot u_h) &= 0 \end{aligned}$$

matrix form

$$\begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \partial_t u \\ 0 \end{bmatrix} + \begin{bmatrix} K(u)+\nu L & B \\ -B^T & 0 \end{bmatrix}\begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} F \\ 0 \end{bmatrix}$$

theta scheme

$$S(u) = \alpha M + \theta\Delta t(K(u)+\nu L)$$

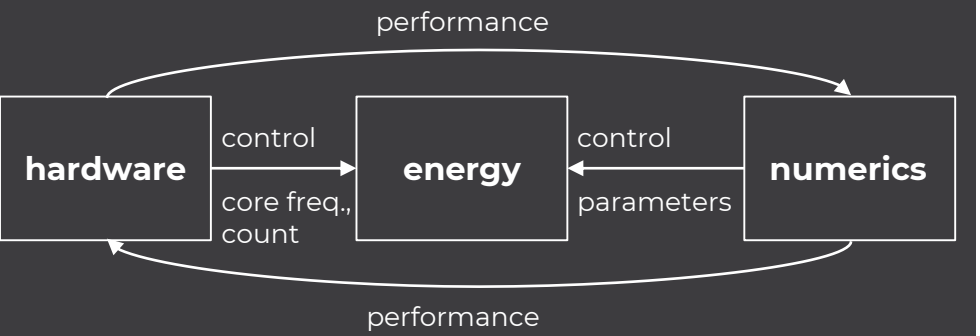
$$g = [M - (1-\theta)\Delta t(K(u)+\nu L)]\cdot u^n + \theta\Delta t F^{n+1} + (1-\theta)\Delta t F^n.$$

$$\begin{bmatrix} S(u) & B \\ -B^T & 0 \end{bmatrix}\begin{bmatrix} u^{n+1} \\ p^{n+1} \end{bmatrix} = \begin{bmatrix} g \\ 0 \end{bmatrix}$$

decouple + magic

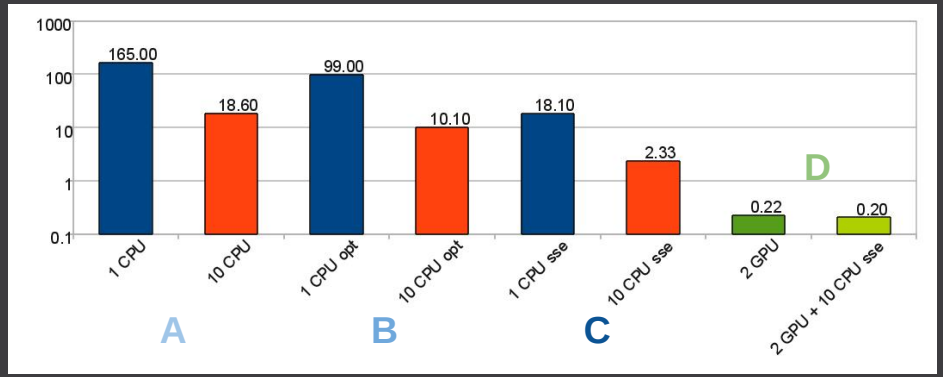
$$S(\tilde{u})\cdot\tilde{u} = f_{FP}$$

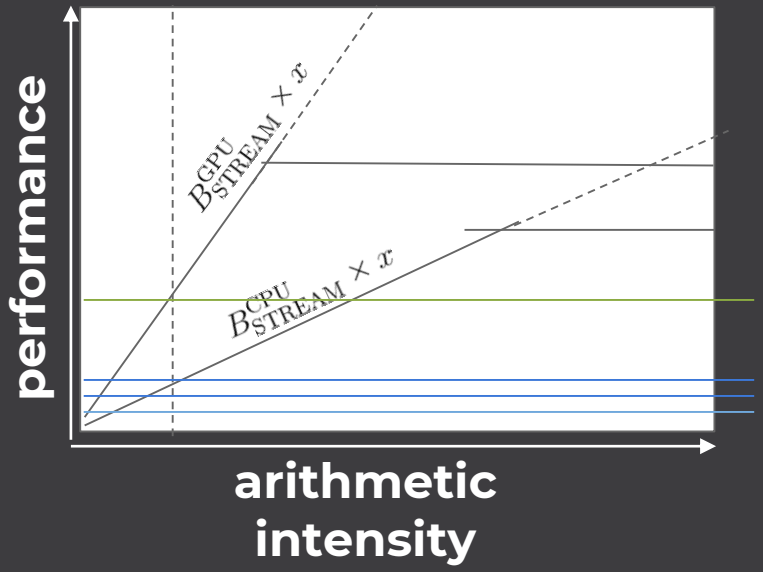
$$P\cdot q = \frac{1}{\Delta t}B^T\cdot\tilde{u}$$



Performance engineering for technical flow simulation

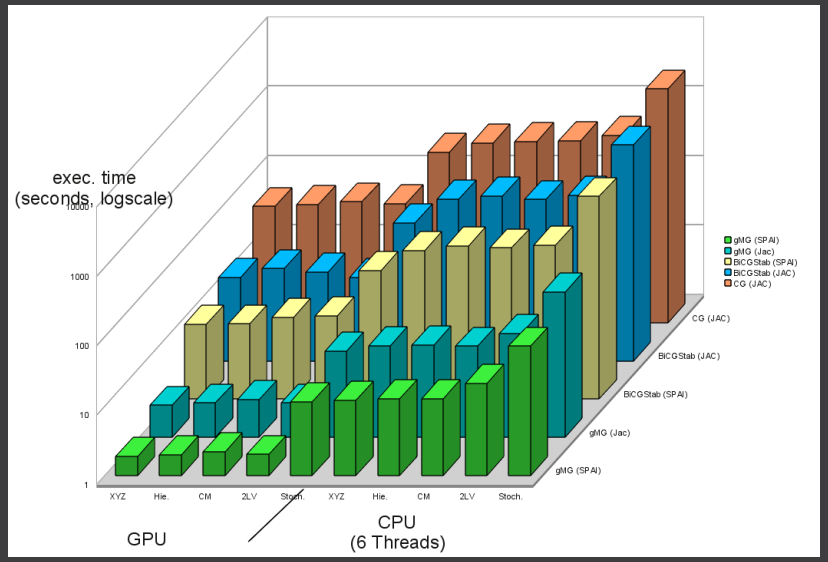
- Influences on **incore performance** of technical flow simulation **(hardware** efficiency**):**
  - FEM space(s)
  - mesh adjacencies (fully unstructured)
  - DOF numbering
  - matrix storage (SELL)
  - accuracy (mixed, low)
  - assembly of matrices (SPAI methods)
- Influences on **incore performance** of technical flow simulation **(numerical** efficiency**):**
  - solver scheme
  - preconditioners / smoothers
  - **mesh anisotropies**

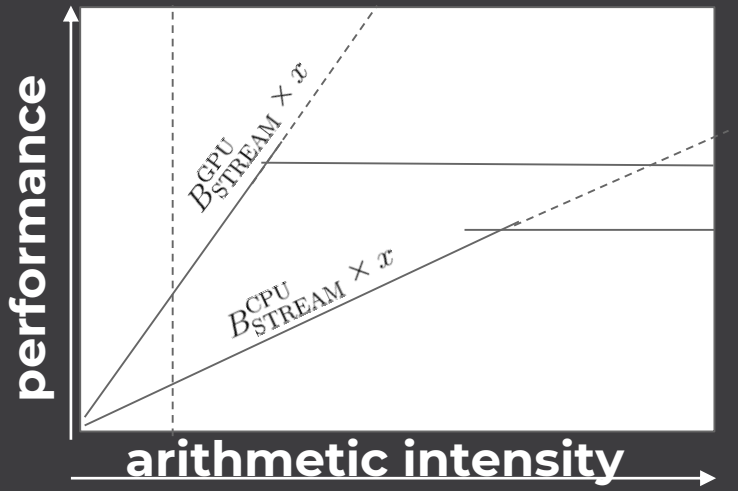




Hardware efficiency and performance engineering for technical flow simulation

- most of our codes are **memory bandwidth-bound**
- proper exploitation of **SIMD** is key to single core performance. Often: optimised **SpMV.**
- the memory interface is **saturated with a (small) amount of cores**
- **GPGPU** usually gives us a speedup of 5 - 10 through larger on-chip memory bandwidth. GPUs can also **saturate that bandwidth**.
- **mixed precision** provides another x1.5 max sustainable (double-single). More possible (double-half)
- low precision: with some methods, perhaps....
- **baseline power** of all devices **has to be amortized via hybrid computation with careful load balancing**





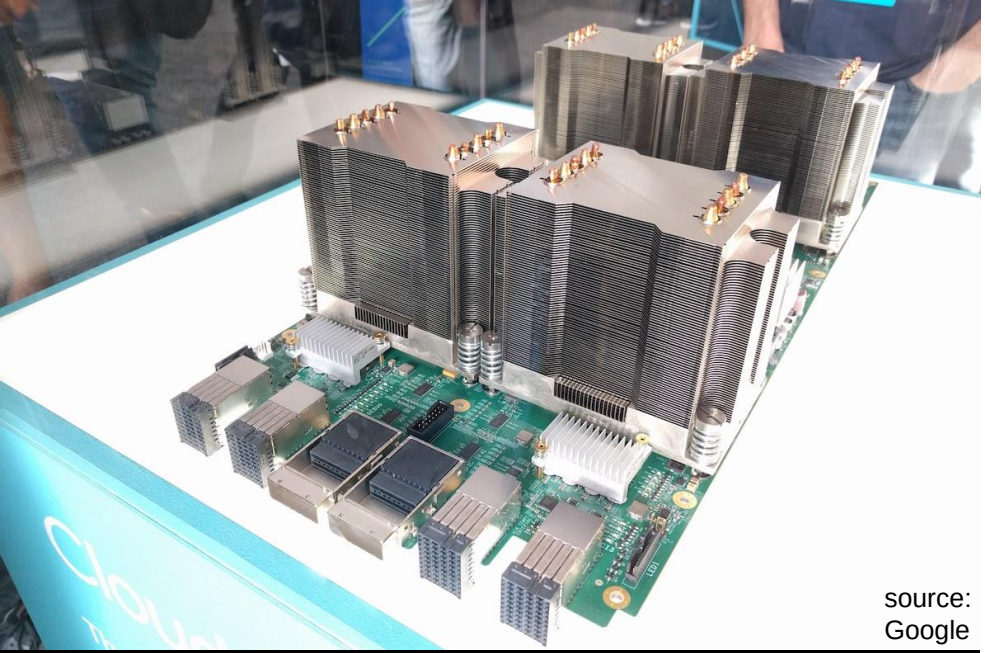
Numerical efficiency and performance engineering for technical flow simulation

- clever smoother construction example: **SPAI-types**
- in theory: **SPAI with same structure as A gives convergence rates like GS (SPAI-1)**
- **works very well as MG smoother**
- **construction phase:** different ways to do this
- **application phase: SpMV**

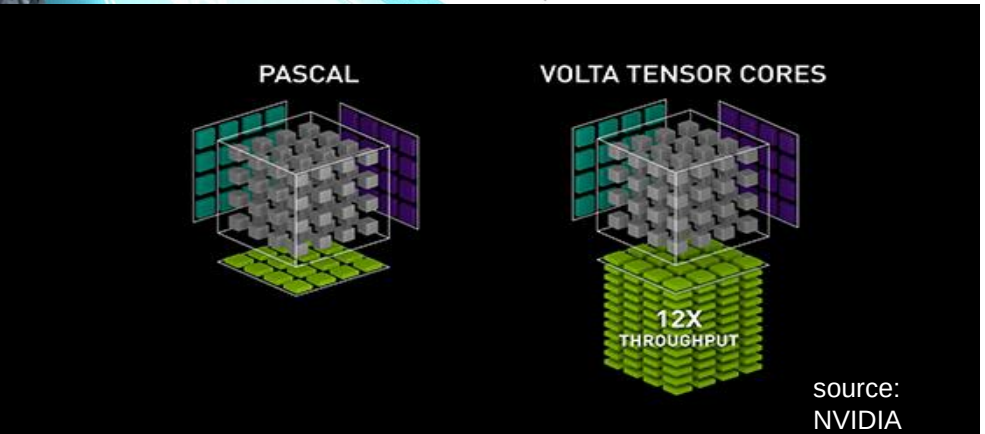
$$\mathbf{x}^{k+1} \leftarrow \mathbf{x}^k + \omega M(\mathbf{b} - A\mathbf{x}^k)$$

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

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



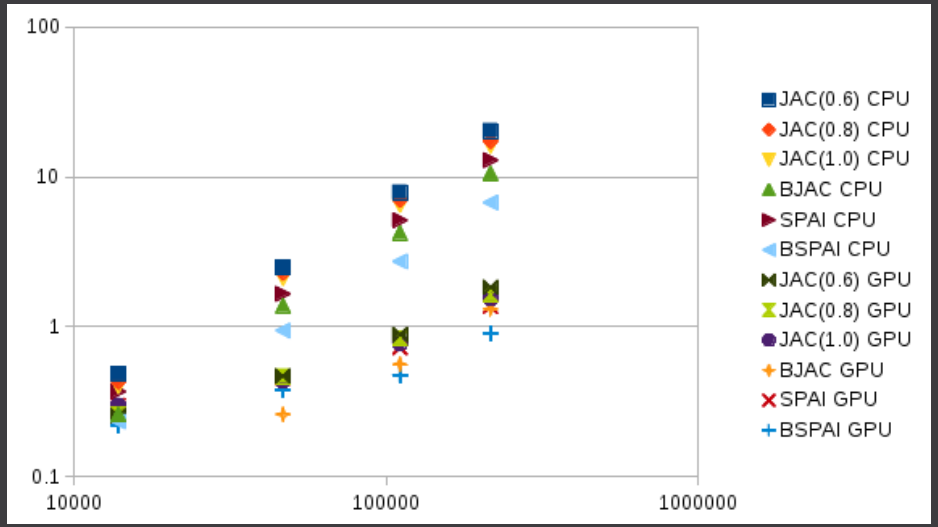
source: Google



source: NVIDIA

The Successor of GPUs?

- Vendors adjust their chips to ML
- Scientists are never in the position to determine hardware micro architecture
- With GPGPU, we have been able to exploit a rough x10 in bandwith in the last decade
- ML Hardware:
  - (Cloud-)TPUs: Inferencing is fast and low precision
  - Volta and Turing GPUs: Tensor Processing Cores as in TPUs, low precision instruction sets and libraries
  - How can that help?



Many Householder transforms -based QR decompositions, Batched on GPU

cuBLAS, cuSPARSE and MKL

~4x GPU v CPU

Partial randomization via Markov Chains

$$m_{ij} = \lim_{N\to\infty} \frac{1}{N} \sum_{s=1}^{N} \left(\sum_{q=0}^{\infty} \frac{W_q}{A_{ii}} \delta_{s_q j} \right)$$

CUDA and MKL

~5x GPU v CPU

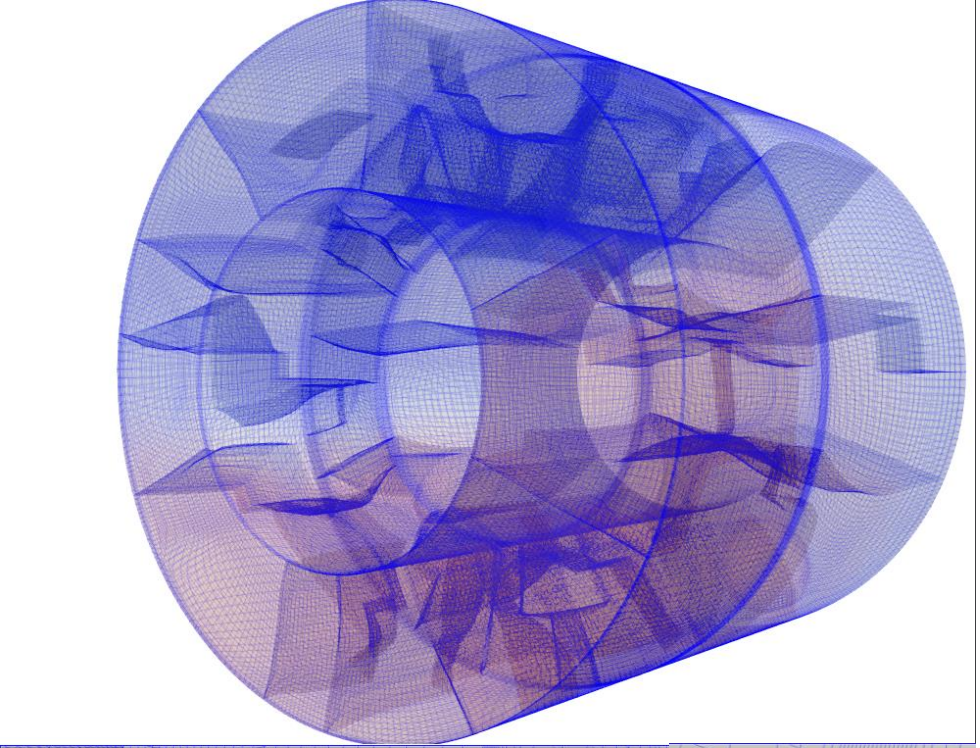
Treat matrices as discrete function in a function regression, train a Deep Learning Net with pairs (A, A^{-1}) make inference for M

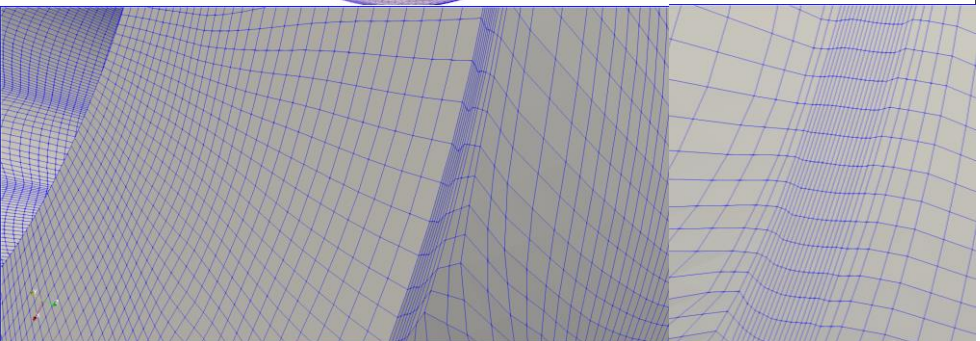
first results with fixed sparsity, implementation with TensorFlow CPU and GPU

Numerical efficiency: recent SPAI preconditioner results from the EXA-DUNE project

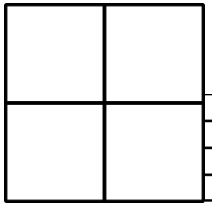
- **SPAI is exceptionally adaptable**
- allows for good balancing of **effort/energy to effectivity** of preconditioner/smoother
- high **reuse potential** of once created approximate inverse
- many screws to adapt to hardware (assembly stage)
  - predefined sparsity pattern (SPAI-1)
  - refinement of sparsity pattern
  - refinement of coefficients
  - **rough inverses often good enough**

SPPEXA



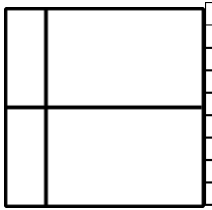


The problem: No smoother for (some) real world meshes

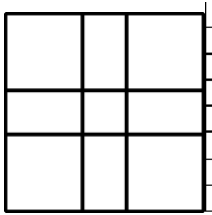


MG(8xRich,8xRich), Tesla P100

lvl	dofs	Jacobi (0,8)	GS (1.0)	SPAI-1 (1,0)	ILU-0 (0,5)
10	4.198.401	4	4	3	4
9	1.050.625	4	4	3	4
8	263.169	4	4	3	4

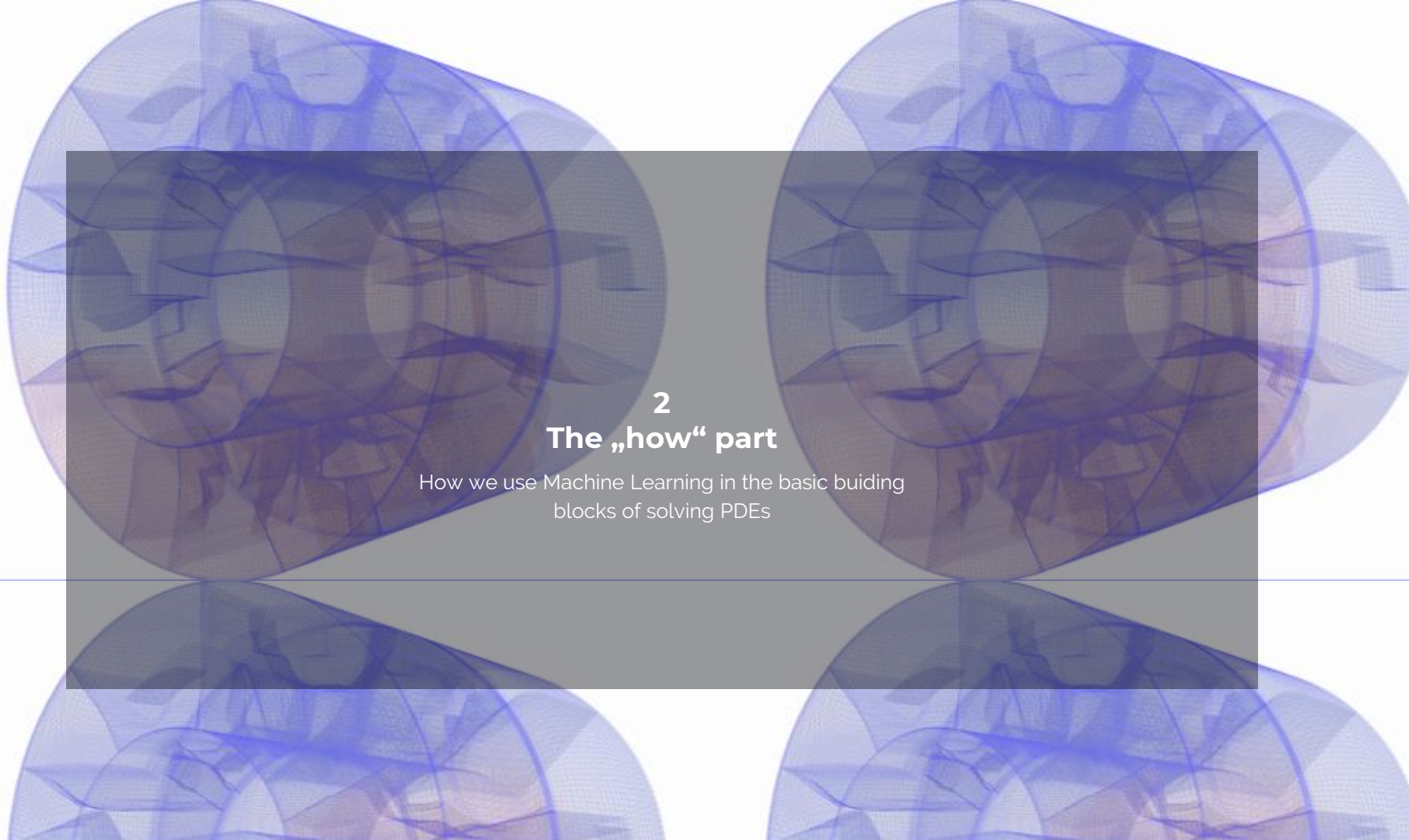


lvl	dofs	Jacobi (0,5)	GS (1.0)	SPAI-1 (1,0)	ILU-0 (0,5)	ILU-RCMK
10	2.100.225	109	33	24	26	7
9	525.825	106	32	23	25	7
8	131.841	103	30	22	24	7
7	32.960	98	28	21	23	7
6	8.240	90	25	19	21	6
5	2.060	78	22	17	18	6
4	515	55	16	13	12	6
3	129	35	9	10	8	6



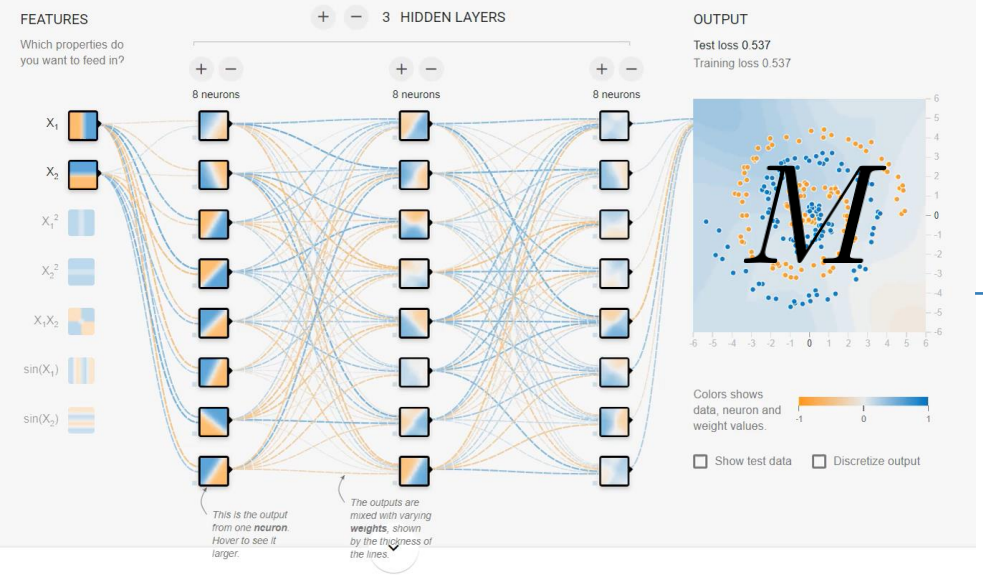
lvl	dofs	Jacobi (0,5)	GS (0,7)	ILU (0,7)	SPAI-1 (1,0)
9	2.362.369	654	370	102	140
8	591.361	619	350	95	130
7	148.225	562	319	85	118
6	37.249	486	289	75	103
5	9.409	377	218	57	80
4	2.401	258	166	34	51
3	625	175	96	20	31

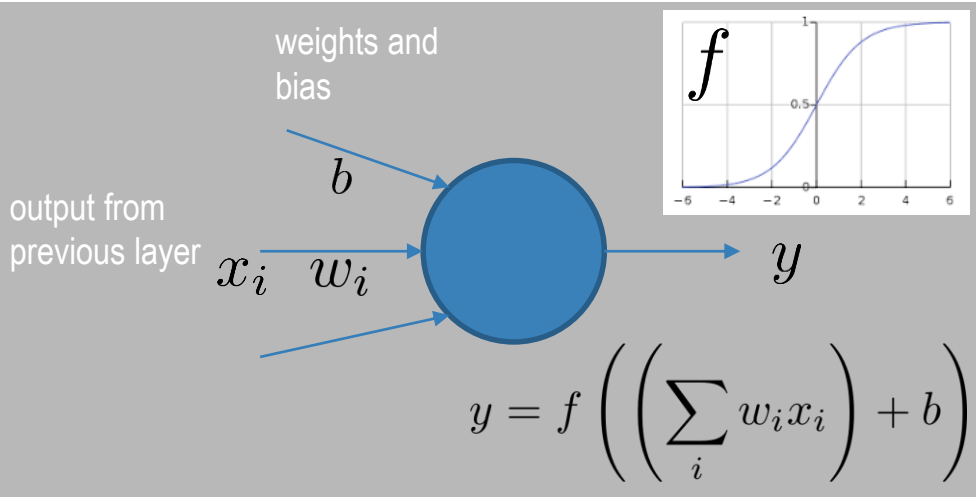
Real world applications generate **complex geometry and mesh anisotropies.**



2
The „how“ part

How we use Machine Learning in the basic buiding blocks of solving PDEs





Our idea: train an artificial neural network to guess an approximate inverse...

- ...and: do it faster than SPAI can be assembled
- ...and: do it better than SPAI in the case of mesh anisotropies

$$(A_h)_{ij} = \sum_{m=1}^{N} \int_{K_m} \nabla\phi_j \cdot \nabla\phi_i dx = \sum_{m=1}^{N} A_{ij}^{(m)}$$

$$(\mathrm{b}_h)_i = \int_\Omega f\phi_i dx$$

$$A_h x = b_h$$

$$M \approx A_h^{-1}$$

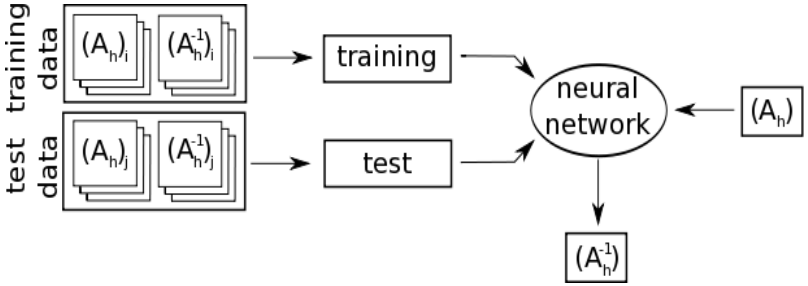
smoother

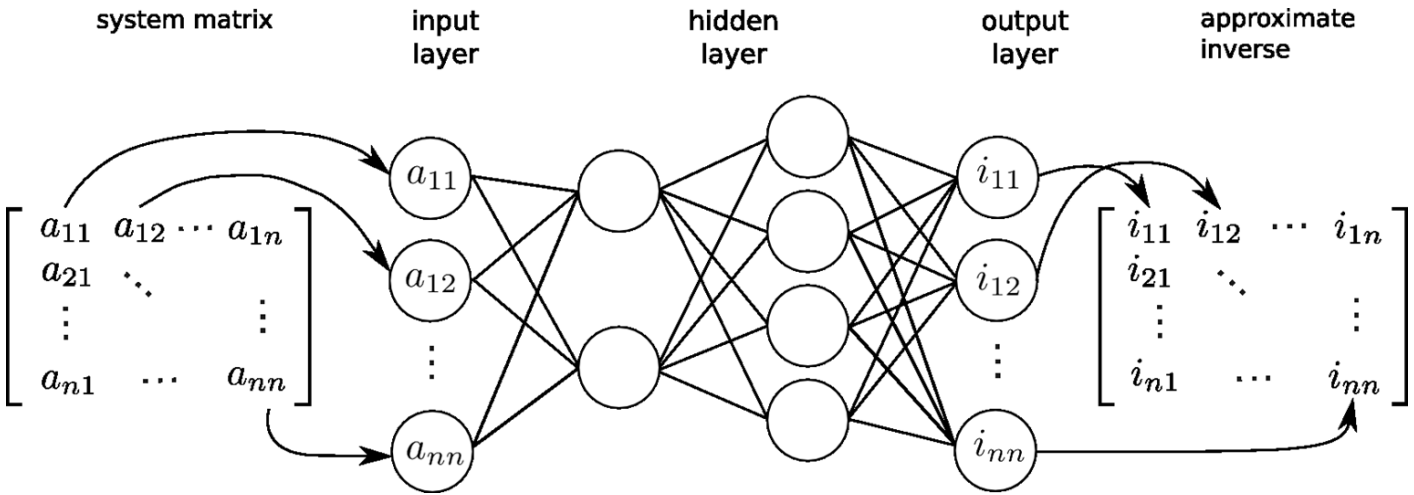
$$x^{(k+1)} = x^{(k)} + \omega M(b_h - A_h x^{(k)})$$

Test and Training

Ingredients

- Multi-Layer Perceptron
- Supervised Learning
- Back Propagation
- Random initialisation





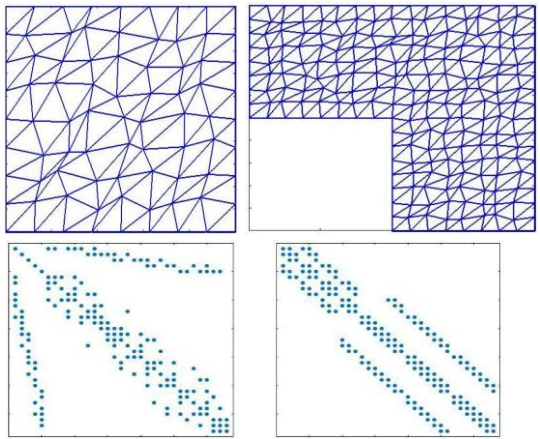
Test and Training: Data

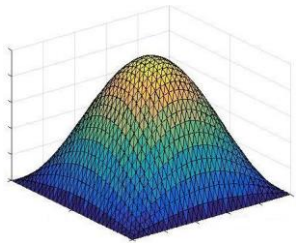
$$-\Delta u = f \quad \text{in } \Omega, \quad u = 0 \quad \text{on } \partial\Omega$$

$$a_h(u_h, v_h) = b_h(v_h) \quad \forall v_h \in V_h$$

$$(A_h)_{ij} = \sum_{m=1}^{N} \int_{K_m} \nabla\phi_j \cdot \nabla\phi_i dx = \sum_{m=1}^{N} A_{ij}^{(m)}$$

$$(\mathrm{b}_h)_i = \int_\Omega f\phi_i dx \qquad A_h x = b_h$$





Algorithm 1 test and training phase

input: n_{hl}, n_{i}, n_{epoch}, n_{batch}, n_{train}, l, n_{test}, $n_{\text{testbatch}}$

- define the neural network (n_{hl}, n_{i})
- initialize weights
- initialize bias neurons b
- define error function and optimizer

start training:

for i in n_{epoch} **do**

A = load_training_matrices (n_{train})

A_inv = load_training_inverses (n_{train})

for j in n_{batch} **do**

x = A_batch$_j$

y = A_inv_batch$_j$

apply optimizer: (W, b) = opt(x, y, l)

test phase:

for i in $n_{\text{testbatch}}$ **do**

A = load_test_data (n_{test})

evaluate the neural network

apply error function or test scenario

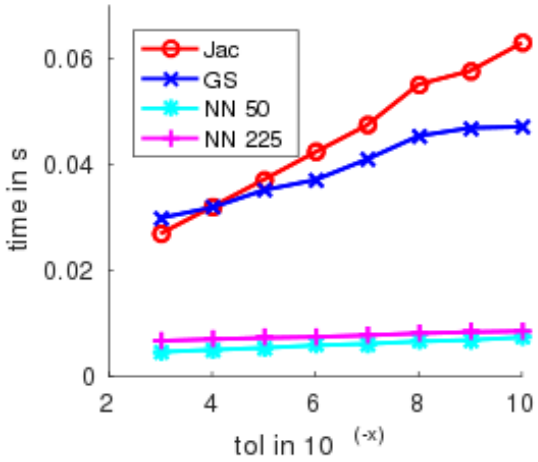
output: W, b

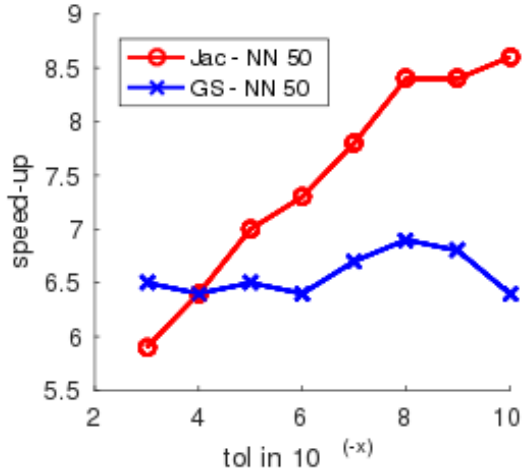
First numerical experiments

$$x^{(k+1)} = x^{(k)} + \omega M(b_h - A_h x^{(k)})$$

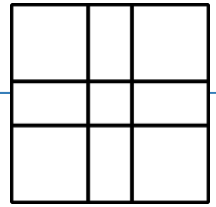
		# it			it down	κ	
lvl	n	$J_{\omega=0.7}$	GS	NN		before	after
2	9	49	17	8	2.13	5.9	1.6
3	49	273	95	21	4.52	29.8	2.9
4	225	1 323	463	66	7.02	127.3	7.8
5	961	5 879	2 057	39	52.74	516.0	23.4

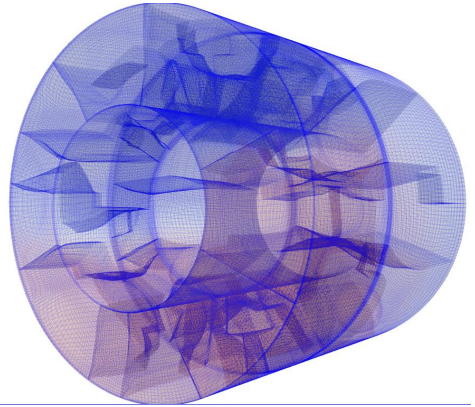
NN:
50 and
225
neurons
per layer
resp.





Numerical experiments: Aniso





1:3

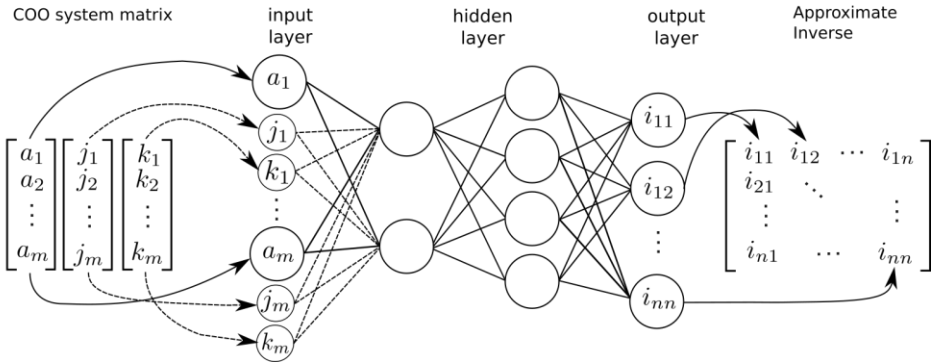
dim	Jac (0,7)	GS	NN
25	422	147	26
121	1955	683	39
529	8622	3017	64

1:10

dim	Jac (0,7)	GS	NN
25	1101	385	22
121	5036	1762	32
529	div	7939	37

(w. different nets)

source: Google



Current work

- How fast is it in **actual MG** compared to standard SPAI/ILU?
- Even when it is slower: What is the total efficiency for high anisotropies?
- What about actual TP hardware?
- From prototype to production: Problem size, sparsification.
- Going deeper into ML: Networking anybody?
- Other ideas: optimal control of parameters in linear solvers like relaxation in SSOR, damping in Jacobi.
- Going beyond the linear solver…

Thanks!

This work has been supported in part by the German Research Foundation (DFG) through the Priority Program 1648 ‘Software for Exascale Computing’.

Meet us at ENUMATH 2019: We have a Minisymposium there!
30th sep - 4th okt 2019, Egmond aan Zee, The Netherlands.

Backup: Numerical experiments: Sparse

$O(n^2 \cdot M)$ $O(\bar{n} \cdot M)$

n	49	225	961	3 969	16 129
full	2 401	50 625	923.521	15 752 961	260 144 641
$\bar{n}$	289	1 457	6 481	27 281	111 889
diag	180	868	3 780	15 748	64 260
%	7.497	1.715	0.409	0.100	0.025

system matrix: dense to sparse

	NN_{sparse} (1 000)			NN_{full}(1 500)			NN_{full} (2 000)		
ω	0.6	0.7	0.8	0.6	0.7	0.8	0.6	0.7	0.8
1	33	27	23	118	110	88	29	24	37
2	37	31	26	110	94	82	31	25	20
3	35	29	25	109	93	81	29	24	22

different weighting and different size of training data for 3 matrices

Backup: Numerical experiments: Sparse

filter matrix entries < ϵ

ϵ	# it (NN)	$\bar{n}$(NN)	%	# it (exact)	$\bar{n}$ (exact)
0.00	23	50 625	100.0	1	50 625
0.01	24	35 469	70.1	9	35 319
0.02	29	26 755	52.8	12	26 663
0.03	48	21 029	41.5	21	21 095
0.04	58	17 133	33.8	38	17 173
0.05	279	14 171	28.0	620	14 233

inverse: dense (semi) sparse

Thanks!

This work has been supported in part by the German Research Foundation (DFG) through the Priority Program 1648 ‘Software for Exascale Computing’.

Meet us at ENUMATH 2019: We have a Minisymposium there!
30th sep - 4th okt 2019, Egmond aan Zee, The Netherlands.