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

Markus Geveler, Hannes Ruelmann, Stefan Turek

@ Sci. Comp. + ML, Nottingham 2019
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

www.mathematik.tu-dortmund.de/lsiil/
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
How we solve the incompressible NSEs

\[
\begin{align*}
u_t - \nu \Delta u + u \cdot \nabla u + \nabla p &= f, \\
-\nabla \cdot u &= 0
\end{align*}
\]

**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

**weak form**

\[
\int_{\Omega} \partial_t u \: v \: dx + \int_{\Omega} (u \cdot \nabla u) \: v \: dx - \nu \int_{\Omega} \Delta u \: v \: dx + \int_{\Omega} \nabla p \: v \: dx = \int_{\Omega} f \: v \: dx \\
\int_{\Omega} (\nabla \cdot u) \: q \: dx = 0
\]

**FEM**

\[
(\partial_t u_h, v_h) + (u_h \cdot \nabla u_h, v_h) + \nu (\nabla u_h, \nabla v_h) - (p_h, \nabla \cdot v_h) = (f, v_h) \\
-(q_h, \nabla \cdot u_h) = 0
\]

**matrix form**

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

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

**theta scheme**

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

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

**smoother** determines overall efficiency

**decouple + magic**

\[
S(\tilde{u}) \cdot \tilde{u} = f_{FP} \\
P \cdot q = \frac{1}{\Delta t} B^T \cdot \tilde{u}
\]
Influences on incore performance of technical flow simulation:

- 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

\[ x^{k+1} = x^k + \omega M(b - Ax^k) \]

\[ \| I - MA \|_F^2 = \sum_{k=1}^{n} \| e^T_k - m_k^T A \|_2^2 = \sum_{k=1}^{n} \| A^T m_k - e_k \|_2^2 \]

\[ \min_{m_k} \| A^T m_k - e_k \|_2, \quad k = 1, \ldots, n. \]
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 bandwidth 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?
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**

---

Many Householder transforms-based QR decompositions, Batched on GPU

Partial randomization via Markov Chains

\[ m_y = \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} \left( \sum_{q=0}^{N} \frac{W_q}{A_k} \delta_{q,d} \right) \]

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

<table>
<thead>
<tr>
<th>cuBLAS, cuSPARSE and MKL</th>
<th>Partial randomization via Markov Chains</th>
<th>Treat matrices as discrete function in a function regression, train a Deep Learning Net with pairs ((A, A^{-1})), make inference for (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~4x GPU v CPU</td>
<td>[ m_y = \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} \left( \sum_{q=0}^{N} \frac{W_q}{A_k} \delta_{q,d} \right) ]</td>
<td>first results with fixed sparsity, implementation with TensorFlow CPU and GPU</td>
</tr>
</tbody>
</table>

---

CUDA and MKL

~5x GPU v CPU
The problem:
No smoother for (some) real world meshes

Real world applications generate complex geometry and mesh anisotropies.
The "how" part

How we use Machine Learning in the basic building 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

\[
(A_h)_{ij} = \sum_{m=1}^{N} \int_{K_m} \nabla \phi_j \cdot \nabla \phi_i \, dx = \sum_{m=1}^{N} A^{(m)}_{ij}
\]

\[
(b_h)_i = \int_{\Omega} f \phi_i \, dx
\]

\[A_h x = b_h\]

\[M \approx A_h^{-1}\]

\[x^{(k+1)} = x^{(k)} + \omega M (b_h - A_h x^{(k)})\]
Ingredients
- Multi-Layer Perceptron
- Supervised Learning
- Back Propagation
- Random initialisation
Test and Training: Data

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

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

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

\[(b_h)_i = \int_{\Omega} f \phi_i \, dx \quad A_h x = b_h\]

**Algorithm 1** test and training phase

**input:** \(n_{h1}, n_i, n_{\text{epoch}}, n_{\text{batch}}, n_{\text{train}}, l, n_{\text{test}}, n_{\text{testbatch}}\)

- define the neural network \((n_{h1}, n_i)\)
- initialize weights
- initialize bias neurons \(b\)
- define error function and optimizer

start training:

**for** \(i \in n_{\text{epoch}}\) **do**

\(A = \text{load\_training\_matrices}\left(n_{\text{train}}\right)\)

\(A_{\text{inv}} = \text{load\_training\_inverses}\left(n_{\text{train}}\right)\)

**for** \(j \in n_{\text{batch}}\) **do**

\(x = A_{\text{batch}}\_j\)

\(y = A_{\text{inv\_batch}}\_j\)

apply optimizer: \((W, b) = \text{opt}(x, y, l)\)

**test phase:**

**for** \(i \in n_{\text{testbatch}}\) **do**

\(A = \text{load\_test\_data}\left(n_{\text{test}}\right)\)

evaluate the neural network

apply error function or test scenario

**output:** \(W, b\)
First numerical experiments

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

<table>
<thead>
<tr>
<th>lvl</th>
<th>n</th>
<th>(J_{\omega=0.7})</th>
<th>GS</th>
<th>NN</th>
<th>it down</th>
<th>(\kappa) before</th>
<th>(\kappa) after</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>9</td>
<td>49</td>
<td>17</td>
<td>8</td>
<td>2.13</td>
<td>5.9</td>
<td>1.6</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
<td>273</td>
<td>95</td>
<td>21</td>
<td>4.52</td>
<td>29.8</td>
<td>2.9</td>
</tr>
<tr>
<td>4</td>
<td>225</td>
<td>1323</td>
<td>463</td>
<td>66</td>
<td>7.02</td>
<td>127.3</td>
<td>7.8</td>
</tr>
<tr>
<td>5</td>
<td>961</td>
<td>5879</td>
<td>2057</td>
<td>39</td>
<td>52.74</td>
<td>516.0</td>
<td>23.4</td>
</tr>
</tbody>
</table>

NN: 50 and 225 neurons per layer resp.
Numerical experiments: Aniso

<table>
<thead>
<tr>
<th>dim</th>
<th>Jac (0.7)</th>
<th>GS</th>
<th>NN</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>422</td>
<td>147</td>
<td>26</td>
</tr>
<tr>
<td>121</td>
<td>1955</td>
<td>683</td>
<td>39</td>
</tr>
<tr>
<td>529</td>
<td>8622</td>
<td>3017</td>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>dim</th>
<th>Jac (0.7)</th>
<th>GS</th>
<th>NN</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1101</td>
<td>385</td>
<td>22</td>
</tr>
<tr>
<td>121</td>
<td>5036</td>
<td>1762</td>
<td>32</td>
</tr>
<tr>
<td>529</td>
<td>div</td>
<td>7939</td>
<td>37</td>
</tr>
</tbody>
</table>

(w. different nets)
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!
Backup: Numerical experiments: Sparse

\[ O(n^2 \cdot M) \quad O(\bar{n} \cdot M) \]

<table>
<thead>
<tr>
<th>(n)</th>
<th>49</th>
<th>225</th>
<th>961</th>
<th>3,969</th>
<th>16,129</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>2,401</td>
<td>50,625</td>
<td>923,521</td>
<td>15,752,961</td>
<td>260,144,641</td>
</tr>
<tr>
<td>(\bar{n})</td>
<td>289</td>
<td>1,457</td>
<td>6,481</td>
<td>27,281</td>
<td>111,889</td>
</tr>
<tr>
<td>diag</td>
<td>180</td>
<td>868</td>
<td>3,780</td>
<td>15,748</td>
<td>64,260</td>
</tr>
<tr>
<td>%</td>
<td>7.497</td>
<td>1.715</td>
<td>0.409</td>
<td>0.100</td>
<td>0.025</td>
</tr>
</tbody>
</table>

system matrix: dense to sparse

<table>
<thead>
<tr>
<th></th>
<th>(NN_{sparse} (1,000))</th>
<th>(NN_{full} (1,500))</th>
<th>(NN_{full} (2,000))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\omega)</td>
<td>0.6 0.7 0.8</td>
<td>0.6 0.7 0.8</td>
<td>0.6 0.7 0.8</td>
</tr>
<tr>
<td>1</td>
<td>33  27  23</td>
<td>118 110  88</td>
<td>29  24  37</td>
</tr>
<tr>
<td>2</td>
<td>37  31  26</td>
<td>110  94  82</td>
<td>31  25  20</td>
</tr>
<tr>
<td>3</td>
<td>35  29  25</td>
<td>109  93  81</td>
<td>29  24  22</td>
</tr>
</tbody>
</table>

different weighting and different size of training data for 3 matrices
filter matrix entries $< \epsilon$

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

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!