Towards Generating Solvers for the Simulation of non-Newtonian Fluids

Harald Köstler, Sebastian Kuckuk
FAU Erlangen-Nürnberg
22.12.2015
Outline
Outline

- Scope and Motivation
- Project ExaStencils
- The Application at Hand
- Work on the Code Generator
- Preliminary results
- Next Steps
Scope and Motivation
Research Vision

1. Open your IPython notebook
2. Specify your continuous model in latex-like language
3. Apply symbolic algebra transformations and derive discrete model
4. Describe numerical algorithms in Matlab-like language
5. Generate efficient code automatically
6. Compute results in the cloud and visualize them within notebook
PDEs

- Goal: Solve a partial differential equation approximately by solving a discretized form

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

\[ A u_h = f_h \]
Multigrid

Smoothing of High Frequency Errors

Coarsened Representation of Low Frequency Errors
Multigrid V-Cycle

- Smoothing
- Restriction
- Prolongation & Correction
- Coarse Grid Solving
Computational Grids

- Regular grids
- Block-Structured grids -> deferred for now
Main Focus: HPC

- Highly optimized and highly scalable geometric multigrid solvers
- ‘Traditional’ supercomputers like JUQUEEN and SuperMUC
- Alternative architectures like Piz Daint and Mont Blanc
Problems to Deal With

- Composition of optimal solvers requires choosing
  - MG components: Cycle Type? Which smoother(s)? Which coarse grid solver? Which inter-Grid operators?
  - MG parameters: Relaxation? Number of smoothing steps? Other component dependent parameters?
  - …
Problems to Deal With

- Composition of optimal solvers requires choosing
  - MG components: …
  - MG parameters: …
  - Applied Optimizations: …
- But the choice relies heavily on various influences
  - Hardware: CPU, GPU or both? Number of nodes, sockets and cores? Cache characteristics? Network characteristics?
  - Software: MPI, OpenMP or both? CUDA or OpenCL? Which versions?
  - Problem description: Which PDE? Which boundary conditions?
  - Discretization: Finite Differences, Finite Elements or Finite Volumes?
  - Domain: Uniform or block-structured? How to partition?
  - …
Possible Solutions

● What do we want?
  ● Efficient and robust multi-grid solvers
  ● Performance portability
  ● Easy to adapt to new hardware
  ● Easy to extend
  ● …

● Solutions?
  ● Optimizing by hand?
  ● Extensive Libraries?
  ● Auto-Tuning?
  ● Code generation?
Programming Approaches in CSE

● One problem – one code
  ● Everything is implemented from scratch or one uses common libraries
  ● Can be easily specialized and therefore optimized
  ● Problem has to be worth it

● One library – several problems
  ● Higher maintenance and user support effort
  ● Hard to fit all users needs and achieve optimal performance
  ● A whole community can benefit from it

● One generator / compiler – several problems
  ● Designing generator requires very high effort
  ● Problem-specific optimizations possible
Domain-driven Projects

Users from different application fields

Mathematician

Software specialist

Hardware specialist

Description of application in domain specific language

Feature Model

Automatic selection of algorithmic components

Code generation for specific application

Automatic Tuning on specific hardware

PDE

\[
\text{Operators::Laplacian(Data::solution)} = \text{Data::rhs}
\]
Manual vs. Automatic

- Performance (necessary)
  - Autotuning success is limited
  - Manual Performance Engineering with detailed Performance Measurement (tools!) and solid hardware-specific Performance Models is state-of-the-art
  - Performance Prediction based on analytic models, parameter fitting, machine learning current research

- Portability (nice to have)
  - Manual effort usually necessary for new architectures
  - Easier for common frameworks (automatic tests!)

- Productivity (is it your time?)
  - Manual implementations in one language hard to maintain
  - Code generation is first higher programming effort but then pays off for similar problems
  - Requires new skills from developers
Different Performance Models

Analytic Performance models: Roofline, ECM

Heuristic Performance model

\[ f(p) = \sum_{k=1}^{n} c_k p^{i_k} log^{j_k}(p) \]

Performance-influence model

\[ \pi_1 + 27 \cdot \text{pre} + 323 \cdot \text{GaussSeidel} \cdot \text{pre} - 1.8 \cdot \text{ip\_amg} \]
Analytic Performance Model for V-cycle I

\[ t_V = \sum_{l=1}^{L} t_l = \sum_{l=1}^{L} \sum_{k=1}^{n_l} t_l^k = \sum_{l=1}^{L} \sum_{k=1}^{n_l} t \left( d_l^k \right) \]

\[ = \sum_{l=1}^{L} \sum_{k=1}^{n_l} \max \left( \frac{d_l^k}{b(d_l^k)} ; t_k \right) \]

Input: kernels t, data transfers d, bandwidth b, call overhead t
Analytic Performance Model for V-cycle II

\[ t_{1}^{\text{block}} = t(2s_1 + s_2) + (\nu - 1) \cdot t(2s_1) \]

\[ t_{l}^{\text{block}} = t(1.5s_l) + t(2s_l + s_{l+1}) + t(1.5s_l + s_{l-1}) + (\nu - 3) \cdot t(2s_l) \quad \text{for } 1 < l < (L - 4) \]

\[ \sum_{l=L-4}^{L} t_{l}^{\text{block}} = t_{31 \times 31} + t(1.5s_{L-4} + s_{L-5}) \cdot \text{correction of level (L-5)} \]
Project ExaStencils
Project ExaStencils

- Sebastian Kuckuk
- Harald Köstler
- Ulrich Rüde
- Alexander Grebhahn
- Sven Apel
- Christian Schmitt
- Frank Hannig
- Jürgen Teich
- Hannah Rittich
- Matthias Bolten
- Stefan Kronawitter
- Armin Größlinger
- Christian Lengauer
ExaStencils – Layers

Continuous Domain & Continuous Model
  discretization

Discrete Domain & Discrete Model
  parametrization

Algorithmic Components & Parameters
  specification

Complete Program Specification
  optimization

Internal Representation
  generation

Exascale C++ Code
Jacobi Smoother (Layer 4)

```plaintext
function Smoother@((coarsest + 1) to finest) ( ) : Unit {
    communicate Solution[0]@(current)
    loop over inner on Solution@(current) {
        Solution[1]@current =
            Solution[0]@current
            + ( ( ( 1.0 / diag ( Lapl@(current) ) ) * 0.8 )
                * ( RHS@current - Lapl@(current) * Solution[0]@current ) )
    }
    ...
```
#include "MultiGrid/MultiGrid.h"

void Smoother_4() {
    exchsolutionData_4(0);
    #pragma omp parallel for schedule(static) num_threads(8)
    for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
        if (isValidForSubdomain[fragmentIdx][0]) {
            for (int y = iterationOffsetBegin[fragmentIdx][0][1];
                 y < (iterationOffsetEnd[fragmentIdx][0][1]+17); y += 1)
                for (int x = iterationOffsetBegin[fragmentIdx][0][0];
                     x < (iterationOffsetEnd[fragmentIdx][0][0]+17); x += 1)
                    slottedFieldData_Solution[1][fragmentIdx][4][(((y*19)+19)+(x+1))] =
                        (slottedFieldData_Solution[0][fragmentIdx][4][(((y*19)+19)+(x+1))]
                         +(((1.0e+00/fieldData_LaplCoeff[fragmentIdx][4][((y*17)+x)])*8.0e-01)
                            *(fieldData_RHS[fragmentIdx][4][((y*17)+x)]
                             -((((fieldData_LaplCoeff[fragmentIdx][4][((y*17)+x)])
                                    *slottedFieldData_Solution[0][fragmentIdx][4][(((y*19)+19)+(x+1))])
                                    +(fieldData_LaplCoeff[fragmentIdx][4][(((y*17)+289)+x)])
                                    *slottedFieldData_Solution[0][fragmentIdx][4][(((y*19)+19)+(x+2))])))
                                    +(fieldData_LaplCoeff[fragmentIdx][4][(((y*17)+578)+x)])
                                    *slottedFieldData_Solution[0][fragmentIdx][4][(((y*19)+19)+x)])
                                    +(fieldData_LaplCoeff[fragmentIdx][4][(((y*17)+867)+x)])
                                    *slottedFieldData_Solution[0][fragmentIdx][4][(((y*19)+38)+(x+1))])))
                                    +(fieldData_LaplCoeff[fragmentIdx][4][(((y*17)+1156)+x)])
                                    *slottedFieldData_Solution[0][fragmentIdx][4][(((y*19)+(x+1)))]));
    }
}
#include "MultiGrid/MultiGrid.h"

void Smoother_4() {
  exchsolutionData_4(0);
  #pragma omp parallel for schedule(static) num_threads(8)
  for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
    if (isValidForSubdomain[fragmentIdx][0]) {
      for (int c0 = iterationOffsetBegin[fragmentIdx][0][1];
           c0 <= (iterationOffsetEnd[fragmentIdx][0][1] + 16)); c0 = (c0 + 1))
        double* slottedFieldData_Solution_1_fragmentIdx_4_p1 =
          &(slottedFieldData_Solution[1][fragmentIdx][4][(19*c0)]);
      double* fieldData_RHS_fragmentIdx_4_p1 = &fieldData_RHS[fragmentIdx][4][(17*c0)];
      double* slottedFieldData_Solution_0_fragmentIdx_4_p1 = ...
      double* fieldData_LaplCoeff_fragmentIdx_4_p1 = ...
      for (int c1 = iterationOffsetBegin[fragmentIdx][0][0];
           c1 <= (iterationOffsetEnd[fragmentIdx][0][0] + 16)); c1 = (c1 + 1)) {
        slottedFieldData_Solution_1_fragmentIdx_4_p1[(c1+20)] =
          (slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+20)]
          + (((1.0e+00/fieldData_LaplCoeff_fragmentIdx_4_p1[c1]) * 8.0e01) * fieldData_RHS_fragmentIdx_4_p1[c1] -
            (((fieldData_LaplCoeff_fragmentIdx_4_p1[c1] * slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+20)] +
              (fieldData_LaplCoeff_fragmentIdx_4_p1[c1+289] * slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+21)]) +
              (fieldData_LaplCoeff_fragmentIdx_4_p1[c1+578] * slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+19)]) +
              (fieldData_LaplCoeff_fragmentIdx_4_p1[c1+39]) +
              (fieldData_LaplCoeff_fragmentIdx_4_p1[c1+1156] * slottedFieldData_Solution_0_fragmentIdx_4_p1[(c1+1)])))));
    } // end for (int c1 = iterationOffsetBegin[fragmentIdx][0][0];
    } // end for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
  } // end for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx)
## Runtime Results for Different Problems

Sizes 4096x4096 resp. 256x256x256

<table>
<thead>
<tr>
<th></th>
<th>cpu</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>8</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Double</td>
<td>Float</td>
<td>Double</td>
<td>Float</td>
<td>Double</td>
<td>Float</td>
</tr>
<tr>
<td>Laplace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D Jacobi</td>
<td>546</td>
<td>1.17</td>
<td>3.50</td>
<td>3.50</td>
<td>6.42</td>
<td>9.10</td>
</tr>
<tr>
<td>2D GaussSeidel</td>
<td>453</td>
<td>1.16</td>
<td>2.42</td>
<td>3.24</td>
<td>3.41</td>
<td>4.72</td>
</tr>
<tr>
<td>3D Jacobi</td>
<td>608</td>
<td>1.08</td>
<td>2.60</td>
<td>3.00</td>
<td>6.83</td>
<td>10.31</td>
</tr>
<tr>
<td>3D GaussSeidel</td>
<td>608</td>
<td>1.08</td>
<td>2.60</td>
<td>3.01</td>
<td>4.22</td>
<td>5.96</td>
</tr>
<tr>
<td>Complex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diffusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2D Jacobi</td>
<td>9235</td>
<td>1.02</td>
<td>2.81</td>
<td>2.87</td>
<td>23.99</td>
<td>33.58</td>
</tr>
<tr>
<td>2D GaussSeidel</td>
<td>7504</td>
<td>1.11</td>
<td>2.18</td>
<td>2.29</td>
<td>11.62</td>
<td>16.42</td>
</tr>
<tr>
<td>3D Jacobi</td>
<td>8799</td>
<td>1.15</td>
<td>2.77</td>
<td>2.91</td>
<td>15.94</td>
<td>39.28</td>
</tr>
<tr>
<td>3D GaussSeidel</td>
<td>9048</td>
<td>1.80</td>
<td>2.60</td>
<td>2.87</td>
<td>10.28</td>
<td>24.19</td>
</tr>
</tbody>
</table>

ms speedup speedup speedup speedup speedup speedup

Experimental Setup

- Windows 7, Visual Studio 2012, CUDA 5.5
- Quad-core Intel Xeon E5-1620 v2, 3.7 GHz
  - 59.7 GB/s memory bandwidth
  - 118.4 GFLOP/s peak performance
- NVIDIA GeForce GTX 680
  - 192.2 GB/s memory bandwidth
  - 3100 GFLOP/s peak performance (single precision)
  - 128.8 GFLOP/s peak performance (double precision)
- Unknows 16.8 million on finest grid
  - 2D: factor 1.33 for coarser grids -> 22.3 million points
  - 3D: factor 1.14 for coarser grids -> 19.1 million points
Transfer and Compute Times

- Quad-core Intel Xeon E5-1620 v2, 3.7 GHz
  - 0.75 ms (1 L + 1 S, 2D)
  - 0.64 ms (1 L + 1 S, 3D)
  - 0.18 ms (1 FLOP per point)

- NVIDIA GeForce GTX 680
  - 0.23 ms (1 L + 1 S, 2D)
  - 0.20 ms (1 L + 1 S, 3D)
  - 0.007 ms (1 FLOP per point, single)
  - 0.17 ms (1 FLOP per point, double)
Jacobi 2D Implementation

- Quad-core Intel Xeon E5-1620 v2, 3.7 GHz
  - Runtime 546 ms
  - Means 3033 FLOP per point
  - Means 1456 transfers (L + S)
- NVIDIA GeForce GTX 680
  - Runtime 85 ms (double), 60 ms (single)
  - Means 500 FLOP (double), 8571 FLOP (single)
  - Means 739 transfers (double), 521 (single)
Algorithmic model Jacobi 2D

- 5 V(2,2)-cycles
  - Means (4+1)*5 = 25 transfers without intergrid ops
  - Means (4+1)*5 = 25 FLOP without intergrid ops
- Best case: factor 20 worse!
- Hand-tuned implementation on GPU is about factor 7.5 faster than generated code: factor 2.8 worse!
The Application at Hand
Overall Goal

Replication/ fully generated code of “Parallel finite volume method simulation of three-dimensional fluid flow and convective heat transfer for viscoplastic non-Newtonian fluids”, D. A. Vasco, N. O. Moraga and G. Haase

- FORTRAN90 Code
- Finite volume discr.
- Staggered grid
- SIMPLE Algorithm
- TDMA solvers
Modelling

● Incompressible non-isothermal/non-Newtonian fluid
  ● Temperature-dependent density
  ● Viscosity according to different models such as power law, Herschely-Bulkley, Bingham and Casson

● Mixed boundary conditions
  ● No-slip BC for velocities (Dirichlet)
  ● Neumann boundary conditions for pressure, viscosity, etc.
  ● Temperature BC split into
    ● uniform hot and cold temperature on the vertical left and right walls, respectively (Dirichlet)
    ● otherwise adiabatic conditions (Neumann)

● Additional driving gravitational force is given by \( g \cdot \rho \cdot V_{CV} \)
Discretization

- Domain: Rectangular box with < 50 mm edge length
- Discretization using finite volumes and
  - power-law differencing scheme (similar to hybrid, i.e. a combination of central and upwind differencing schemes)
  - Weighted harmonic mean interpolation for components with high jumps
    \[
    \frac{\omega_0 + \omega_1}{\omega_0 x_0 + \omega_1 x_1}
    \]
- Implicit time integration for each component – explicit coupling using SIMPLE
The SIMPLE Algorithm

- **Semi-Implicit Method for Pressure Linked Equations**
- **Concept:**
  
  - Update properties
  - Solve for velocity components
  - Solve for pressure correction
  - Apply pressure correction
The SIMPLE Algorithm

- Temperature can be added as a separate step

- Update properties
- Solve for velocity components
- Solve for pressure correction
- Apply pressure correction
- Solve for temperature
Staggered grid

- Finite volume discretization on a staggered grid

- Values associated with the cell centers, e.g. $p$ and $\theta$
- Values associated with the $x$-staggered grid, e.g. $U$
- Values associated with the $y$-staggered grid, e.g. $V$
- Control volumes associated with cell-centered values
- Control volumes associated with $x$-staggered values
- Control volumes associated with $y$-staggered values
Non-uniform staggered grid

- Grid complicates
  - Interpolation of values
  - Integration across (staggered) control volume interfaces
  - Incorporation of boundary conditions

- Open questions:
  - How to specify grid setup/ layout
  - How to parallelize -> generate separate kernels for uniform portions?
Work on the Code Generator
Extensions

Necessary preparations:

- Interface from/to FORTRAN
  - Data and functions
  - Allows replacing single portions of the code, one step at a time

- Data layouts for staggered grids

- Requires, apart from the language extension itself,
  - adapted field layouts
  - new communication routines
  - specialized boundary conditions
  - (refined coarsening and interpolation stencils)
Porting code

- Straight-forward for simple kernels

```fortran
subroutine advance_fields ()
  !$omp parallel do &
  !$omp private(i,j,k) &
  !$omp firstprivate(l1,m1,n1) &
  !$omp shared(rho,rho0) &
  !$omp schedule(static) default(none)
  do k=1,n1
    do j=1,m1
      do i=1,l1
        rho0(i,j,k)=rho(i,j,k)
      end do
    end do
  end do
!$omp end parallel do
```

```plaintext
Function AdvanceFields@finest ()
  : Unit {
    loop over rho@current {
      rho[next]@current = rho[active]@current
    }
    advance rho@current
  }
```
Porting code

- But what about more complicated code? How do we get from this ...

```plaintext
! if not at the boundary
fl = xcv(i) * v(i,jp,k) * (fy(jp)*rho(i,jp,k) + fym(jp)*rho(i,j,k))
flm = xcvip(im) * v(im,jp,k) * (fy(jp)*rho(im,jp,k) + fym(jp)*rho(im,j,k))
flownu = zcv(k) * (fl+flm)
gm = xcv(i) * vis(i,j,k) * vis(i,jp,k) / (ycv(j)*vis(i,jp,k) + ycv(jp)*vis(i,j,k) + 1.e-30)
gmm = xcvip(im) * vis(im,j,k) * vis(im,jp,k) / (ycv(j)*vis(im,jp,k) + ycv(jp)*vis(im,j,k) + 1.e-30)
diff = 2. * zcv(k) * (gm+gmm)
call diflow(flownu,diff,acof)
adc = acof + max(0.,flownu)
anu(i,j,k) = adc - flownu
```
Porting code

… to this?

```plaintext
flownu@current = integrateOverXStaggeredNorthFace ( v[active]@current * rho[active]@current )

Val diffnu : Real = integrateOverXStaggeredNorthFace ( evalAtXStaggeredNorthFace ( vis@current, "harmonicMean" ) ) / vf_stagCVWidth_y@current@[0, 1, 0]

AuStencil@current:[0, 1, 0] = -1.0 * ( calc_diflow ( flownu@current, diffnu ) + max ( 0.0, flownu@current ) - flownu@current )
```
Extended boundary conditions

```plaintext
Function ApplyBC_u@finest ( ) : Unit {
  loop over u@current only duplicate [ 1, 0] on boundary {
    u@current = 0.0
  }
  loop over u@current only duplicate [ -1, 0] on boundary {
    u@current = 0.0
  }
  loop over u@current only duplicate [ 0, 1] on boundary {
    u@current = wall_velocity
  }
  loop over u@current only duplicate [ 0, -1] on boundary {
    u@current = -1.0 * wall_velocity
  }
}
Field Solution < global, DefNodeLayout, ApplyBC_u>@finest
```
Virtual fields

- Rework and extension of the way geometric information is used
- Many virtual fields (positions of (staggered) nodes/ faces/ cells, interpolation and integration parameters, etc.)
- Tradeoff between on-the-fly calculation and explicit storage

```cpp
1 // previous access through functions doesn’t allow offset access
2 rhs@current = sin ( nodePosition_x@current ( ) )
3
4 // newly introduced virtual fields allow virtually identical behavior ...
5 rhs@current = sin ( vf_nodePosition_x@current )
6
7 // ... and in addition allow offset accesses
8 Val dif : Real = ( 
9     vf_nodePosition_x@current@[ 1, 0, 0] 
10       - vf_nodePosition_x@current@[ 0, 0, 0] )
```
Specialized functions

- Specialized evaluation and integration functions

```plaintext
1  // evaluate with respect to cells of the grid
2  evalAtSouthFace ( rho[active]@current )
3
4  // integrate expressions across faces of grid cells
5  integrateOverEastFace ( u[active]@current * rho[active]@current )
6
7  // integrate expressions across faces of cells of the staggered grid
8  integrateOverXStaggeredEastFace ( u[active]@current * rho[active]@current )
9
10 // integrate expressions using specialized interpolation schemes
11 integrateOverZStaggeredEastFace ( evalAtZStaggeredEastFace ( vis@current, "harmonicMean" ) )
```
Preliminary Results
Preliminary results

- Initial version without non-Newtonian model
- Comparison is difficult due to the different exit criteria, e.g.
- Solving the single components
  - Reference: fixed to 1 solve step (TDMA)
  - Our implementation
    - At least one step (RBGS or GMG)
      \[ \| r \| \leq \alpha (1 + \beta \| b \|) \]
      \[ \| r_{i+1} \| - \| r_i \| < \bar{\alpha} \]
- SIMPLE
  - Reference: a certain percentage of all cells (including boundaries) doesn’t change beyond a fixed threshold
  - Our implementation: convergence for all components is achieved after updating the matrix but before starting the solve function
Preliminary results

- One socket of our cluster (Intel Xeon E7-4830)
- 16 OMP threads
- 100 timesteps
Next Steps
Next Steps

- Long-term direction of ExaStencils
  - Work on ExaSlang Layer 1 – 3
  - Extension of the framework, esp. multi-GPU
- Short term – which is most promising?
  - More advanced applications such as solidification
  - Distributed memory parallelization -> next step
  - Extension of solver components
    - Galerkin coarsening
    - Solve for all three velocity components in one step (-> staggered)
    - Solve for the whole linearized, coupled system
    - Non-linear multigrid
  - More work on optimization (low-level as well as algorithmic)
Thank you for your Attention!

Questions?