WWU MÜNSTER

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

Recent advances for code generation in the *HPC²SE* project

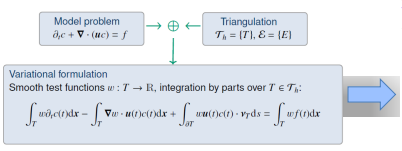
Hardware- and Performance-aware Codegeneration for Computational Science and Engineering

Engwer, Fahlke, Koch, Bastian, Kempf, Heß, Gorlatch, Fey, Rüde, Köstler, Hönig

18. October 2019

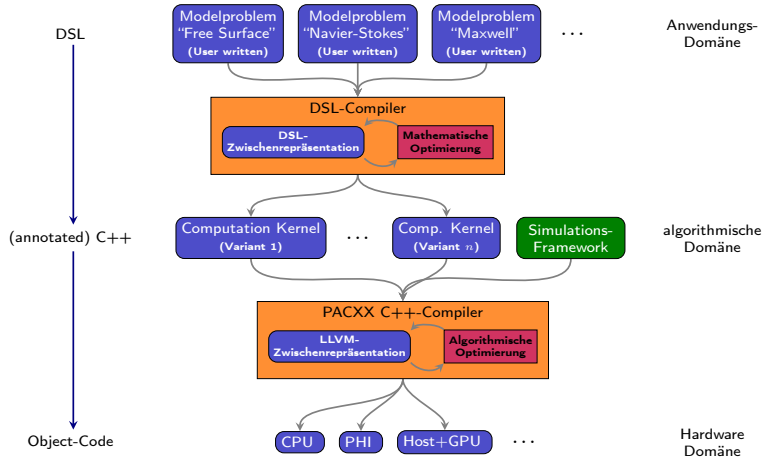
Goals for Code Generation

- Fast problem transformations (productivity)
  - ➡ Math to Code
- Fast implementations (performance)
  - ➡ Efficient use of hardware resources
- Fast code transformations (portability)
  - ➡ For algorithms and different platforms




```
#include "MultiGrid/MultiGrid.h"
void Smoother_4() {
 exchsolutionData_4();
#pragma omp parallel for schedule(static) num_threads(8)
 for (int fragmentIdx = 0; fragmentIdx < 8; ++fragmentIdx) {
  if (isValidForSubdomain[fragmentIdx]) {
   for (int y = iterationOffsetBegin[fragmentIdx][1];
                                  y < (iterationOffsetEnd[fragmentIdx][1]-1); y += 1)
    for (int x = iterationOffsetBegin[fragmentIdx][0];
                                  x < (iterationOffsetEnd[fragmentIdx][0]-1); x += 1)
     slottedFieldData_Solution[1][fragmentIdx][1][(((y*12)+12)+(x+1))] =
              (slottedFieldData_Solution[0][fragmentIdx][1][(((y*12)+12)+(x+1))]
              +((((1.0e-01)/fieldData_LapICoeff[fragmentIdx][1][(((y*1)+12))*5.0e-01)
              *(fieldData_RHS[fragmentIdx][1][(((y*1)+12)+x)]
              -((((((fieldData_LapICoeff[fragmentIdx][1][((y*1)+12)]
              *slottedFieldData_Solution[0][fragmentIdx][1][(((y*12)+11)+12)+(x+1))]))
              +(fieldData_LapICoeff[fragmentIdx][1][(((y*1)+12)+12)]
              *slottedFieldData_Solution[0][fragmentIdx][1][(((y*12)+13)+(x+1))]))))
              +(fieldData_LapICoeff[fragmentIdx][1][(((y*1)+12)+12)]
              *slottedFieldData_Solution[0][fragmentIdx][1][(((y*12)+11)+12)+(x+1))]))
              +(fieldData_LapICoeff[fragmentIdx][1][(((y*1)+12)+12)]
              *slottedFieldData_Solution[0][fragmentIdx][1][(((y*12)+13)+(x+1))]))
              +(fieldData_LapICoeff[fragmentIdx][1][(((y*1)+12)+12)]
              *slottedFieldData_Solution[0][fragmentIdx][1][(((y*12)+12)+(x+1))])))));
...
```

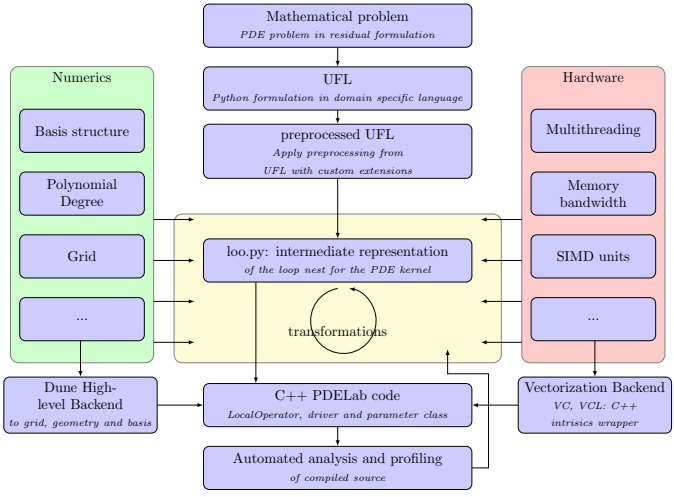
HPC²SE code generation appraoch



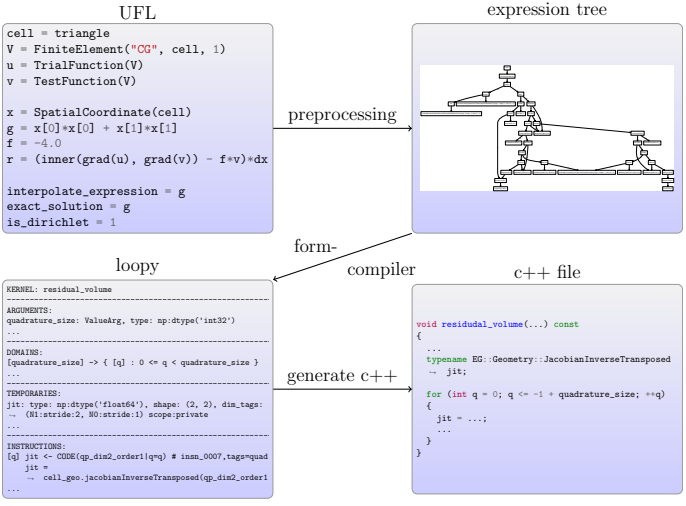
Outline

1 Our Code Generation Approach

Workflow


Mathematical problem
PDE problem in residual formulation
UFL
Python formulation in domain specific language
preprocessed UFL
Apply preprocessing from
UFL with custom extensions
Numerics
Basis structure
Polynomial Degree
Grid
...
loo.py: intermediate representation
of the loop nest for the PDE kernel
transformations
Hardware
Multithreading
Memory bandwidth
SIMD units
...
Dune High-level Backend
to grid, geometry and basis
C++ PDELab code
LocalOperator, driver and parameter class
Vectorization Backend
VC, VCL: C++ intrisics wrapper
Automated analysis and profiling
of compiled source

Workflow


UFL
cell = triangle
V = FiniteElement("CG", cell, 1)
u = TrialFunction(V)
v = TestFunction(V)
x = SpatialCoordinate(cell)
g = x[0]*x[0] + x[1]*x[1]
f = -4.0
r = (inner(grad(u), grad(v)) - f*v)*dx
interpolate_expression = g
exact_solution = g
is_dirichlet = 1
preprocessing
expression tree
form-compiler
loopy
KERNEL: residual_volume
ARGUMENTS:
quadrature_size: ValueArg, type: np:dtype('int32')
DOMAINS:
[quadrature_size] -> { [q] : 0 <= q < quadrature_size }
TEMPORARIES:
jit: type: np:dtype('float64'), shape: (2, 2), dim_tags:
(N1:stride:2, N0:stride:1) scope:private
INSTRUCTIONS:
[q] jit <- CODE(qp_dim2_order1|q*q) # insn_0007,tags=quad
jit =
cell_geo.jacobianInverseTransposed(qp_dim2_order1
generate c++
c++ file
void residudal_volume(...) const
{
typename EG::Geometry::JacobianInverseTransposed
jit;
for (int q = 0; q <= -1 + quadrature_size; ++q)
{
jit = ...;
}
}

Current State

Low-level optimization

- Performance optimizations on intermediate representation (eg. loop reordering in tensor contractions)
- Built in autotuning approach using micro benchmarks for finding good performance transformations
- Vectorization strategies for good SIMD parallelism

Code generation options

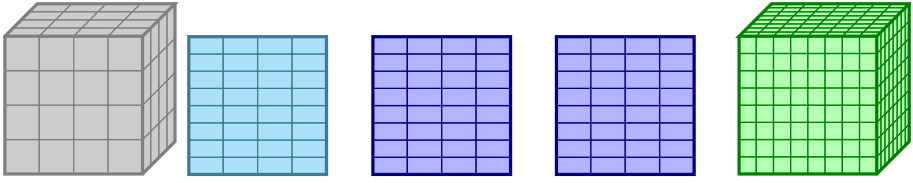
- Sumfactorization for High order DG
  → reduces algorithmic complexity
- Block-structured meshes for low order CG
  → Increase of arithmetic intensity by handling multiple elements in local operator
- Stencil kernels on structured meshes
  → On-the-fly smoothers for MG (→ EXA-Stencil)

2 Higher-order DG

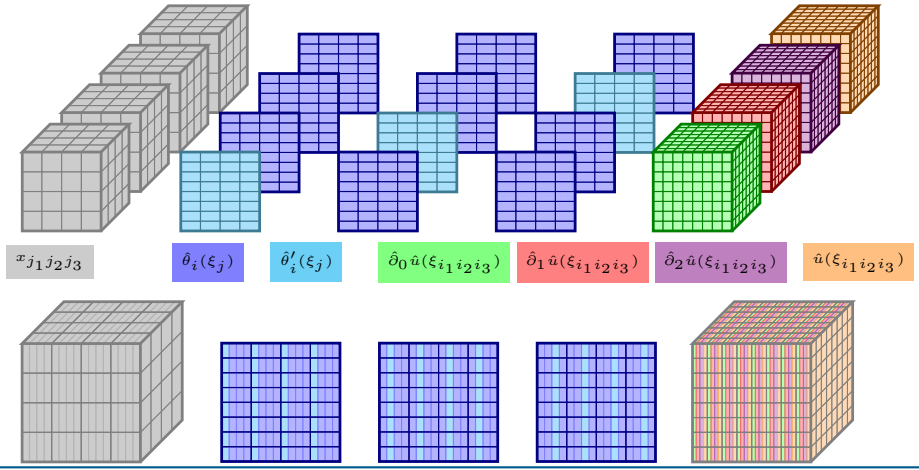
Evaluation of $\hat{\partial}_0 \hat{u}$ in 3D using sum factorization

$$\hat{\partial}_0 \hat{u}(\xi_{i_1 i_2 i_3}) = \underbrace{\sum_{j_3 \in J^{(3)}} A^{(3)}_{i_3,j_3} \underbrace{\sum_{j_2 \in J^{(2)}} A^{(2)}_{i_2,j_2} \underbrace{\sum_{j_1 \in J^{(1)}} A^{(1)}_{i_1,j_1} x_{j_1 j_2 j_3}}_{y_{j_2 j_3 i_1}}}_{y_{j_3 i_1 i_2}}}_{y_{i_1 i_2 i_3}}$$

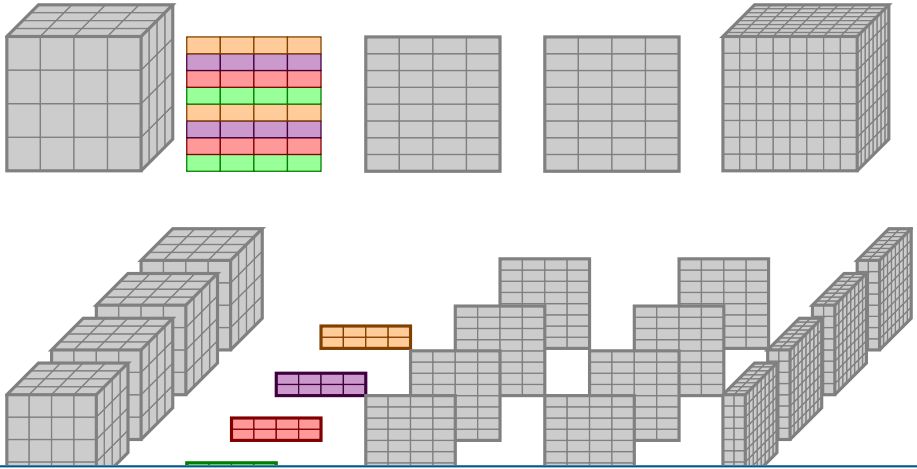
$A^{(1)}_{i,j} = \hat{\theta}'_i(\xi^{(1)}_j)$: Derivatives of the 1D Basis at the 1D quadrature points. $A^{(k)}_{i,j} = \hat{\theta}_i(\xi^{(k)}_j)$, $k \rangle 1$: Evaluations of the 1D Basis at the 1D quadrature points.



Vectorization by Loop Fusion

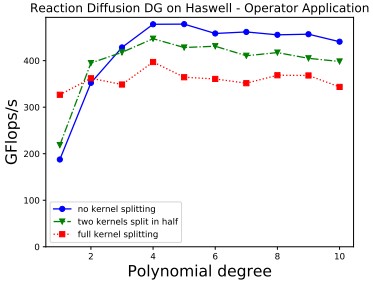


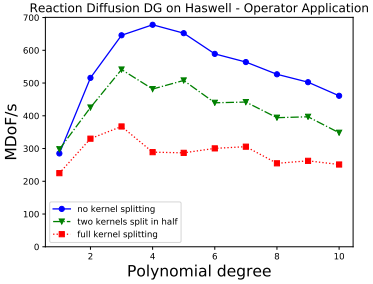
Parallelism through Loop Splitting



Performance on Intel Haswell

- Intel Xeon processor E5-2698 v3
- 2×16 cores, 2.3 GHz
- Theoretical node peak performance: 1.17 TFlops/s
- Operator application of DG-Method for a diffusion reaction problem with full diffusion tensor.





Peak performance?

Haswell

- Around 50% of peak performance for polynomial degree $k > 2$
- Volume integrals around 70% of peak performance
- Skeletons around 40% of peak performance (lower flop per byte ratio)

Skylake

- Around 30% of peak performance for polynomial degree $k > 2$
- Around 70% of peak for volume integrals but room for improvement on skeletons
- Up to 850 GFlops/s compared to 500 GFlops/s for Haswell

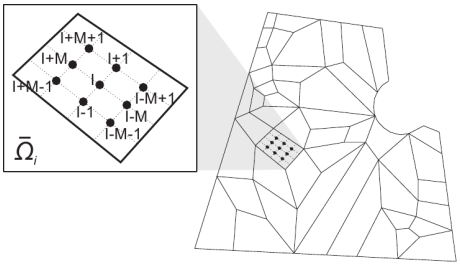
3 Block-structured meshes

Locally block-structured FEM

Goal: Generate efficient lower order FEM code
Idea:

- Coarse grid of macro elements
- Refine macro-Elements in k Micro-Elements
- Substructure managed within the local kernel

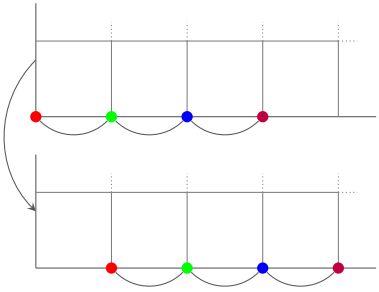
→ increased arithmetic intensity



Vectorization

- Vectorize over neighboring micro-elements
- Automatically generated as transformation within the Loo.py IR

- Example vector width 4:
  → local kernel speedup of $\sim 3.5x$



Performance - Matrix free operator application

Linear elasticity (LE)	Poisson (P)
Unstructured 3D mesh	Axiparallel strukturierted 2D mesh
Local kernel with high arithmetic intensity	Local kernel with low arithmetic intensity
40MB DoFs per kernel	95MB DoFs per kernel

Problem	Variant	MDof/s	Speedup vs $k = 1$
LE	$k = 8$	2.1	2.2x
	$k = 8$ + SIMD4	7.1	7.8x
P	$k = 16$	18.4	5.6x
	$k = 16$ + SIMD4	41.3	12.6x

Performance - Matrix free operator application

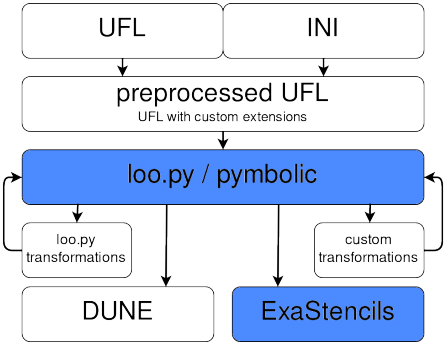
Linear elasticity (LE)	Poisson (P)
Unstructured 3D mesh	Axiparallel strukturierted 2D mesh
Local kernel with high arithmetic intensity	Local kernel with low arithmetic intensity
40MB DoFs per kernel	95MB DoFs per kernel
Low data transfer overhead → low speedup due tu block-strukturing	High data transfer overhead → hoher speedup due tu block-strukturing
High arithmetic intensity → high speedup due to vektorization	Low arithmetic intensity → low speedup due to vektorization

4 Stencil code generation

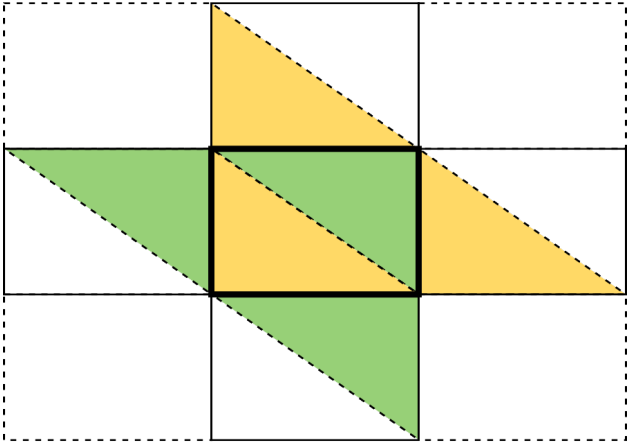
Stencil generation for HPC platforms

ExaStencils

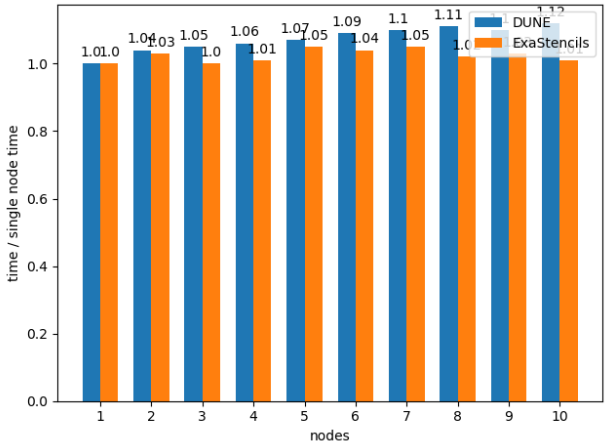
- Whole program generation framework
- Focus on Multigrid methods and stencil codes
- Own DSL: ExaSlang
- Target Code: C++/CUDA
- Automatically introduces SIMD, OpenMP, MPI
- Heterogeneus computation



DG Stencil for ExaStencils



Scaling on SuperMUC-NG



5 Platform portability?!

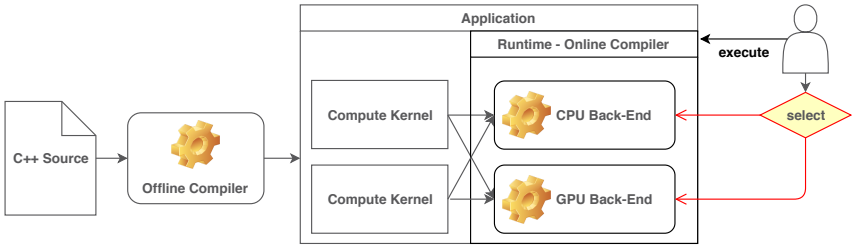
PACXX: Platform portable C++ compiler

- PACXX is used as a complete compiler
- Drop-in replacement for Clang
- Compilation result contains »Runtime« + all support routines

Objectives

- Unified programming for parallel architectures in C++.
- Platform independence by modular design.
- “Black-box” parallelisation.

Unified programming for parallel architectures in C++



- Inspired by CUDA and OpenCL.
- »Kernel« / »Devices« paradigm.
- One codebase for »Host« and »Device«.
- Based on C++14 Lambdas.

Example: MatMul

```
auto& exec = Executor::get(0);

auto& dev_a = exec.allocate<double>(matrix_size);
auto& dev_b = exec.allocate<double>(matrix_size);
auto& dev_c = exec.allocate<double>(matrix_size);

dev_a.upload(a, matrix_size);
dev_b.upload(b, matrix_size);
dev_c.upload(c, matrix_size);

auto pa = dev_a.get();
auto pb = dev_b.get();
auto pc = dev_c.get();

auto matMultKernel = [=](auto &config)
{
   auto column = config.get_global(0);
   auto row    = config.get_global(1);
   double val  = 0;
   for (unsigned i = 0; i < width; ++i)
      val += pa[row * width + i] * pb[i * width + column];
   pc[row * width + column] = val;
};

exec.launch(matMultKernel, {{width/threads , width} , {threads , 1}});

dev_c.download(c, matrix_size);
```

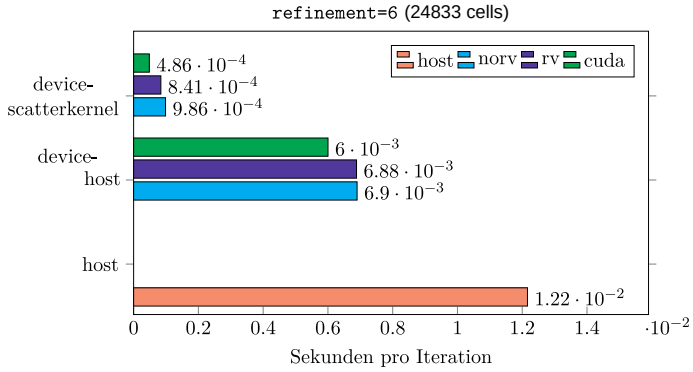
Example: MatMul

```
auto& exec = Executor::get(0);

auto& dev_a = exec.allocate<double>(matrix_size);
auto& dev_b = exec.allocate<double>(matrix_size);
auto& dev_c = exec.allocate<double>(matrix_size);

dev_a.upload(a, matrix_size);
dev_b.upload(b, matrix_size);
dev_c.upload(c, matrix_size);

auto pa = dev_a.get();
auto pb = dev_b.get();
auto pc = dev_c.get();

auto matMultKernel = [=](auto &config)
{
   auto column = config.get_global(0);
   auto row    = config.get_global(1);
   double val  = 0;
   for (unsigned i = 0; i < width; ++i)
      val += pa[row * width + i] * pb[i * width + column];
   pc[row * width + column] = val;
};

exec.launch(matMultKernel, {{width/threads , width} , {threads , 1}});

dev_c.download(c, matrix_size);
```

Using PACXX GPU backend for grid-based methods

- Models: »Linear Elasticity« and »Poisson«
- Reimplement certain DUNE components “GPU friendly”
- Reformulate operator/assembler simular to `Map/Reduce`
- Split into two separate kernels

Intel Xeon CPU E5-1620 v2 vs. Nvidia Tesla K20c



WWU
MÜNSTER

Advanced Solvers for modern Architectures 2019

7th Applied Mathematics Symposium Münster
November 11–13, 2019 | Münster, Germany

uni-muenster.de/AMM/num/solvers2019/

Speakers

Andrew T. Barker (LLNL)
Erin Carson (Charles University)
Andreas Frommer (University of Wuppertal)
Laura Grigori (INRIA)
Eike Müller (University of Bath)
Nicole Spillane (École Polytechnique)
Wim Vanroose (University of Antwerp)
Stefano Zampini (KAUST)

Discussion

- Flexible code generation pipeline
- Mathematically increased arithmetic intensity via local structure
  - Higher-order DG
  - Block-structured refinement
- Generating preconditioners
- Platform portability
  - GPU friendly mesh data structures
  - Map-reduce style assembly
  - PACXX concepts not generic enough for our optimizations
- Open Questions:
  - Transfer to real application
  - Finalize work on DD preconditioners
  - Fully integrate with PACXX

Discussion

- Flexible code generation pipeline
- Mathematically increased arithmetic intensity via local structure
- Generating preconditioners
  - Stencil generation
  - *Non-verlapping DD methods (WIP)*
- Platform portability
  - GPU friendly mesh data structures
  - Map-reduce style assembly
  - PACXX concepts not generic enough for our optimizations
- Open Questions:
  - Transfer to real application
  - Finalize work on DD preconditioners
  - Fully integrate with PACXX

Questions?