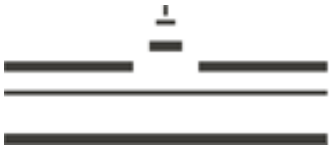
HPC²SE: Hardware- and Performance-aware Codegeneration for Computational Science and Engineering

7. HPC-Status-Konferenz der Gauß-Allianz
Stuttgart, December 5th 2017

Harald Köstler, Martin Bauer, Jö Fahlke, Michael Haidl, Marcel Koch, Dominic Kempf, et al
Chair for System Simulation
Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany



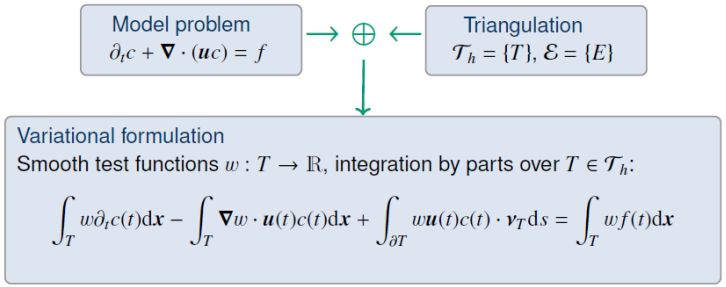
Westfälische Wilhelms-Universität Münster

Universität Heidelberg
Zukunft seit 1386

Friedrich-Alexander Universität Erlangen-Nürnberg
Technische Fakultät

Requirements 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(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))])))));
...
```

Code generation approaches

- Generate full code from scratch **vs.** generate kernels or modules for existing framework
- Develop new language **vs.** embed in existing language
- Output general purpose code (e.g. C++) **vs.** machine-specific code (PTX)
- Prescribe parameters and code transformations **vs.** find them automatically

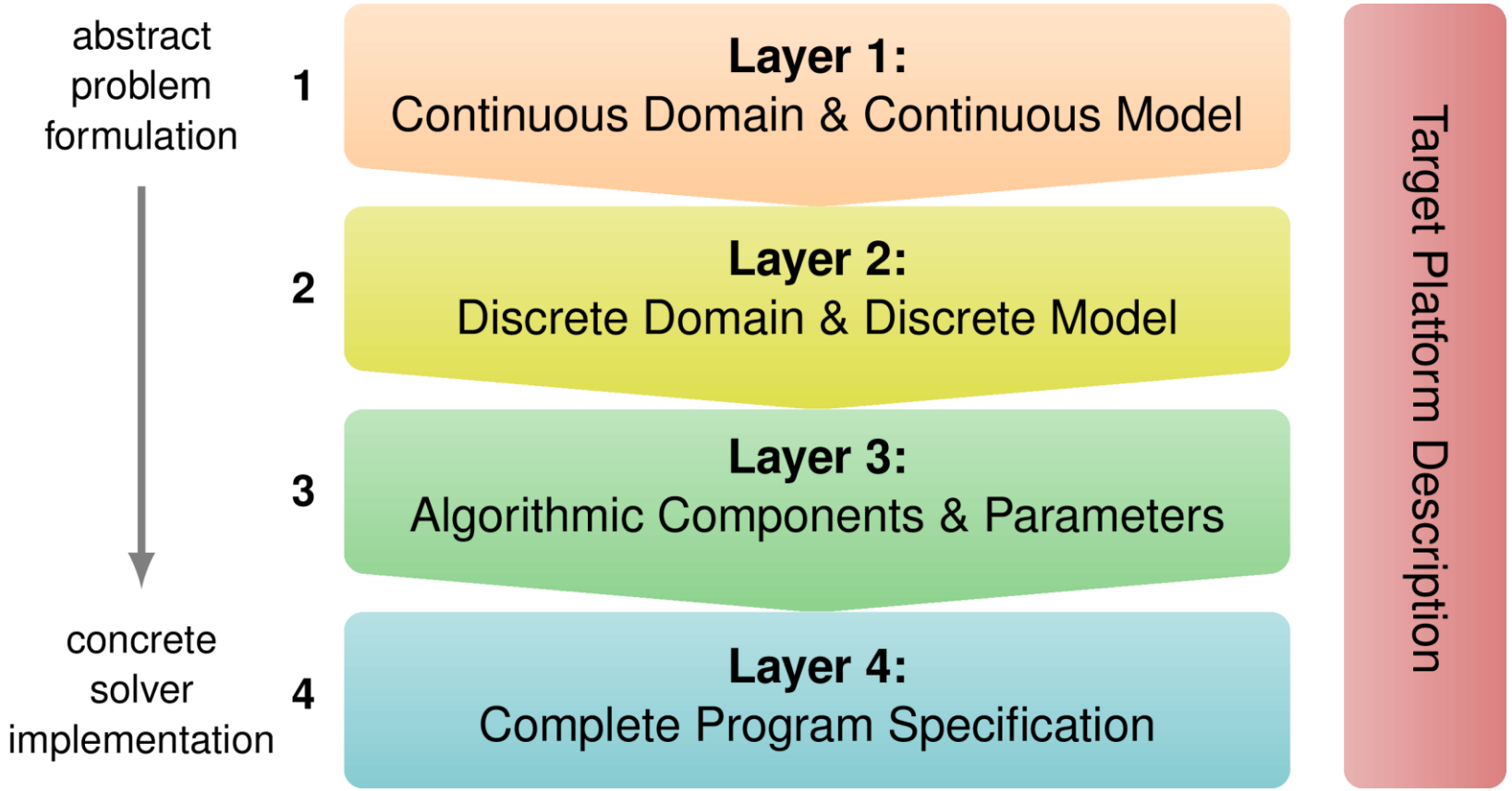
High variability leads to many solution approaches!

Specialized Solutions are required!

In-house Solutions sometimes reasonable!

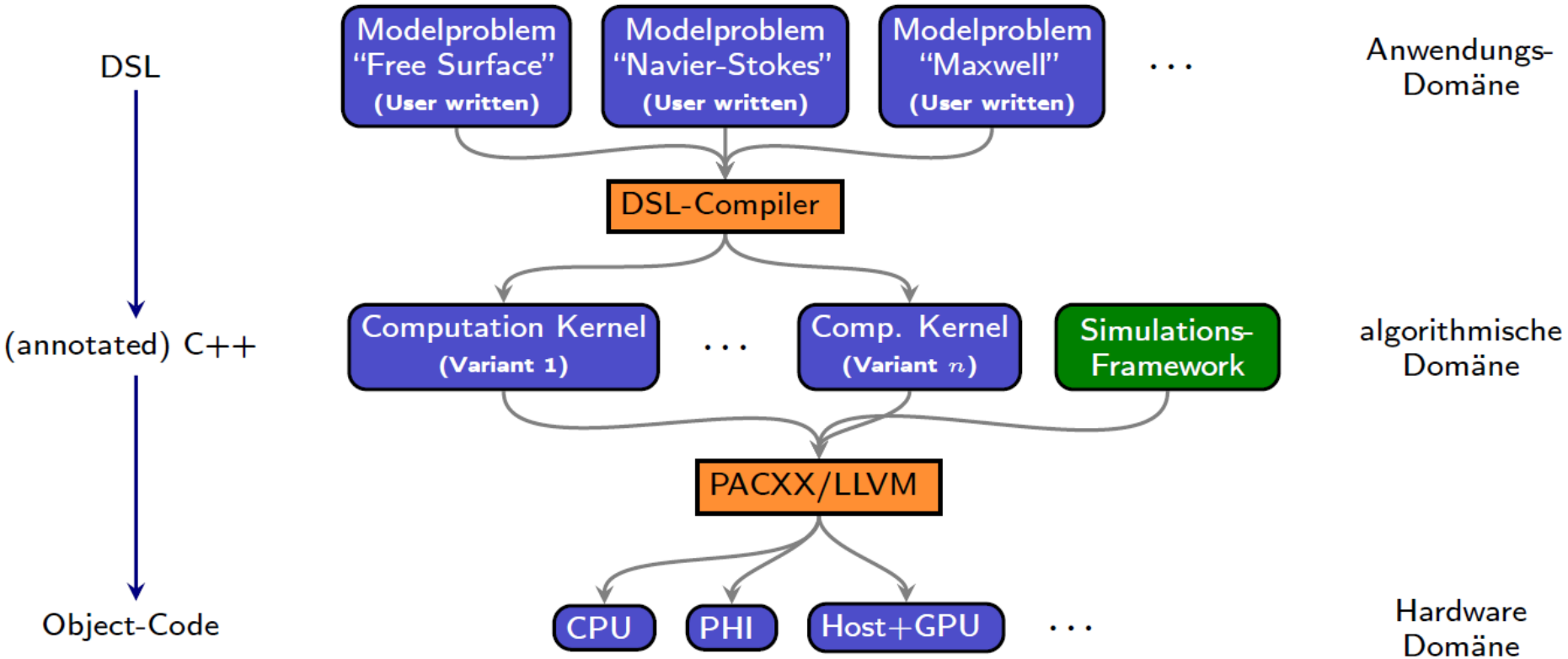
Example: ExaStencils (SPPEXA)

- We propose a multilayered, external DSL



Schmitt, Kuckuk, Hannig, Köstler, Teich. **ExaSlang: A Domain-Specific Language for Highly Scalable Multigrid Solvers**.

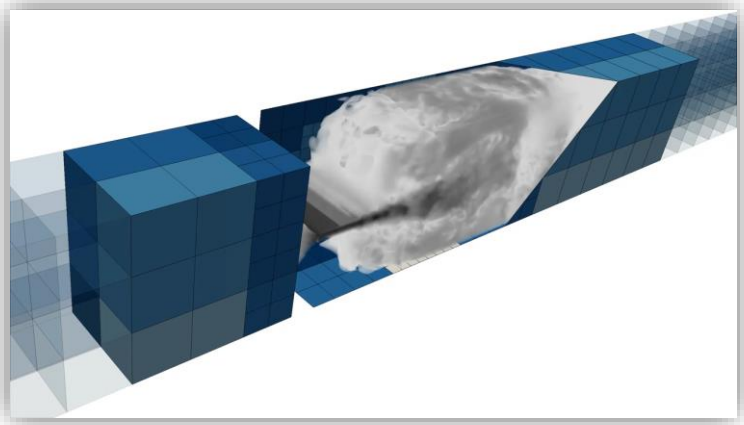
HPC²SE: Code Generation Pipeline



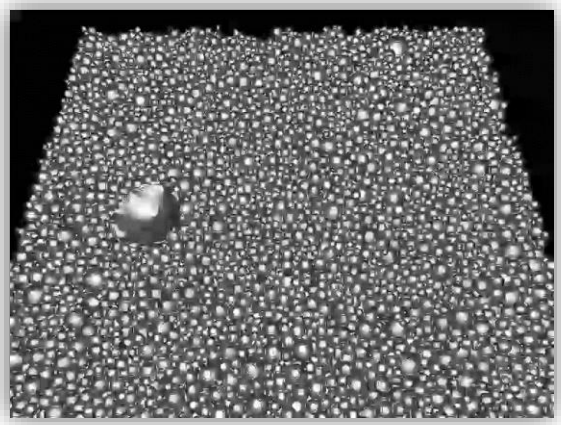
LBM Code Generation

Introduction: waLBerla

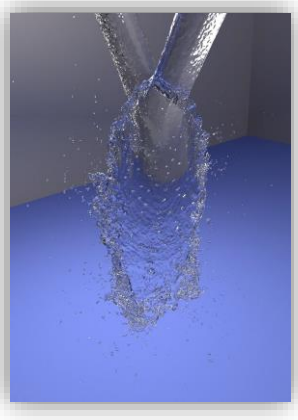
- written in C++ with high level Python interface
- main application: CFD with the lattice Boltzmann method
- massively parallel stencil framework with block structured domain partitioning and adaptive load balancing
- open source: **www.walberla.net**



Adaptivity and dynamic load balancing



Additive Manufacturing



Free Surface Flows

Introduction: waLBerla (SKALB Project)

- focus of the framework: high performance computing (HPC)

- support of accelerator hardware required (GPUs, XeonPhi)
- besides scalability, a high **single node performance** is important
- Optimization techniques for LBM kernels
  - (manual) or guided vectorization (AVX2, AVX512, QPX)
  - inner loop splitting to improve prefetching due to lower number of load/store streams
  - sparse (list-based) kernels for domains with many boundary cells
  - data layout: simple two grid stream-collide, AABB pattern, EsoTwist

LBM Kernels (related to SKAMPY project)

- lattice representation
  - direct addr.: full 3-D array
  - indirect addr.: 1-D array + adj. list
- data layouts
  - array of structures (AoS)
  - structures of arrays (SoA)
- one step (OS) [1]
  - push/pull
  - pull with nontemporal stores and loop splitting
- AA pattern [2] (indirect addr. only)
  - stand. implementation SoA/AoS
  - … + reduced indirect addressing (RIA)
  - … + partial vectorization (PV) [3]
- parallelization: OpenMP parallel for work sharing
  - manual work distribution: blocked OS push/pull
  - automatic work distribution: remaining kernels
- most kernels support loop blocking

[1] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger und U. Rüde: Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. Parallel Processing Letters, 13(4):549–560, 2003. doi:10.1142/S0129626403001501.
[2] P. Bailey, J. Myre, S. Walsh, D. Lilja und M. Saar: Accelerating lattice Boltzmann fluid flow simulations using graphics processors. In: International Conference on Parallel Processing 2009 (ICPP'09), Seiten 550–557. 2009. doi:10.1109/ICPP.2009.38.
[3] M. Wittmann, T. Zeiser, G. Hager und G. Wellein: Modeling and analyzing performance for highly optimized propagation steps of the lattice Boltzmann method on sparse lattices. CoRR, abs/1410.0412v2, 2015. Version 2: arXiv:1410.0412v2

Overview of Implemented Kernels

kernel name	prop. step	data layout	addr.	parallel	blocking	B_l [B/FLUP]
push-soa	OS	SoA	**D**	**x**	**-**	456
push-aos	OS	AoS	**D**	**x**	**-**	456
pull-soa	OS	SoA	**D**	**x**	**-**	456
pull-aos	OS	AoS	**D**	**x**	**-**	456
blk-push-soa	OS	SoA	**D**	**x**	**x**	456
blk-push-aos	OS	AoS	**D**	**x**	**x**	456
blk-pull-soa	OS	SoA	**D**	**x**	**x**	456
blk-pull-aos	OS	AoS	**D**	**x**	**x**	456
list-push-soa	OS	SoA	**I**	**x**	**x**	528
list-push-aos	OS	AoS	**I**	**x**	**x**	528
list-pull-soa	OS	SoA	**I**	**x**	**x**	528
list-pull-aos	OS	AoS	**I**	**x**	**x**	528
list-pull-split-nt-1s	OS	SoA	**I**	**x**	**x**	376
list-pull-split-nt-2s	OS	SoA	**I**	**x**	**x**	376
list-aa-soa	AA	SoA	**I**	**x**	**x**	340
list-aa-aos	AA	AoS	**I**	**x**	**x**	340
list-aa-ria-soa	AA	SoA	**I**	**x**	**x**	304-342
list-aa-pv-soa	AA	SoA	**I**	**x**	**x**	304-342

Introduction

Models / Features

- stencils
- moment-based methods (MRT)
  - efficient SRT and TRT implementations
  - moment basis construction
  - various equilibria
  - forcing approaches
- different collision space: cumulant method
- entropic stabilization
- locally varying relaxation rates e.g. to include turbulence models
- coupling of multiple kernels

Hardware / Optimization

- GPU/CUDA support
- (manual) or guided vectorization (AVX2, AVX512, QPX)
- inner loop splitting to improve prefetching due to lower number of load/store streams
- sparse (list-based) kernels for domains with many boundary cells
- data layout: simple two grid stream-collide, AABB pattern, EsoTwist

too many combinations

code generation

Code Generation Technology

pystencils

Benefits of Code Generation

- LBM derivation in symbolic computer algebra system: toolbox for model development
- flexibility: each parameter (relaxation rate, force) can either be
  - constant: additional simplifications possible
  - symbolic expression: computed using available quantities, e.g. determine relaxation rate from shear rates for turbulence models
  - array access: different value in each cell; used e.g. for coupling of multiple schemes
- LB model and kernel available in symbolic form
  - automatic Chapman-Enskog analysis possible
  - automatic performance modelling possible (roofline, ECM model)

pystencils

- transforms a list of equations that describe stencil update rule into efficient code
- based on *sympy* package
- *sympy* is a Python computer algebra system (similar to Maple or mathematica)
- why *sympy*?
  - open source
  - easily extensible – has flexible tree representation
  - waLBerla already has a Python interface
  - C/C++ printer already available

Example: Jacobi Kernel

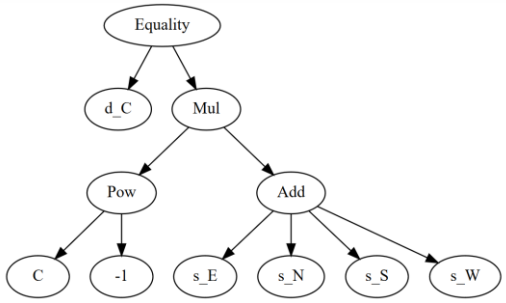
Symbolic Stencil Update Rule

- list of equations with field accesses and symbolic constants
- represented by syntax tree

Automatic Simplifications and Optimizations

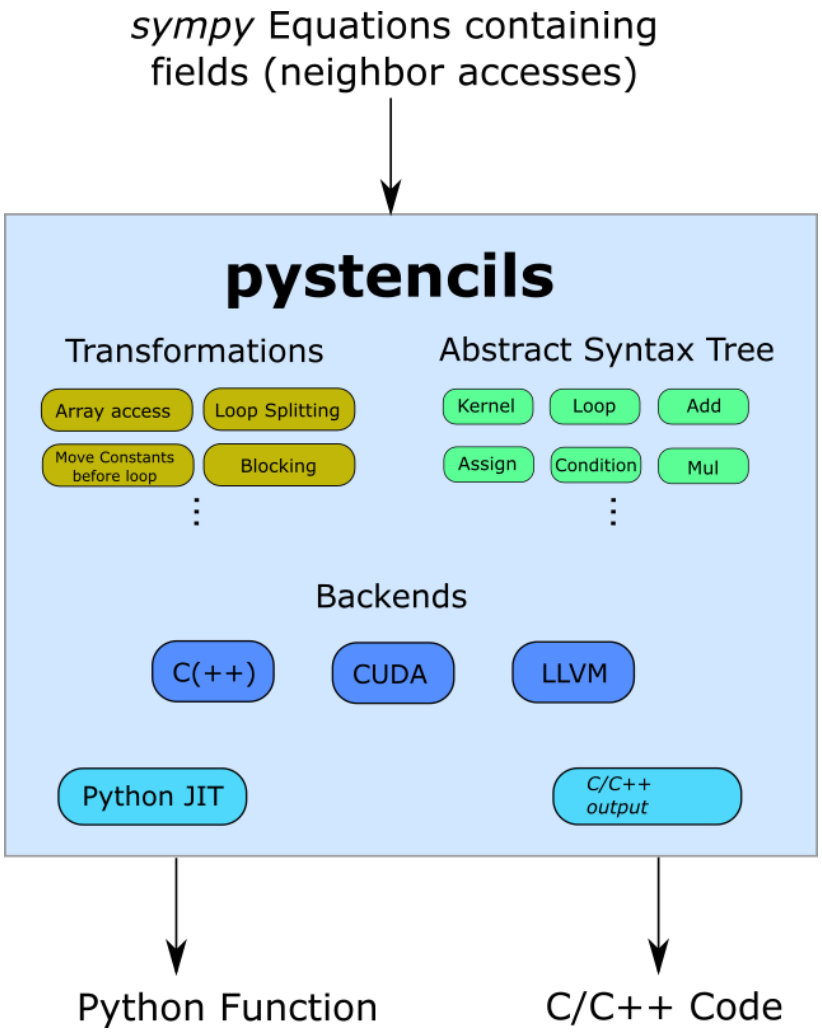
C / CUDA / LLVM IR

$$d_C = \frac{1}{C}(s_E + s_N + s_S + s_W)$$

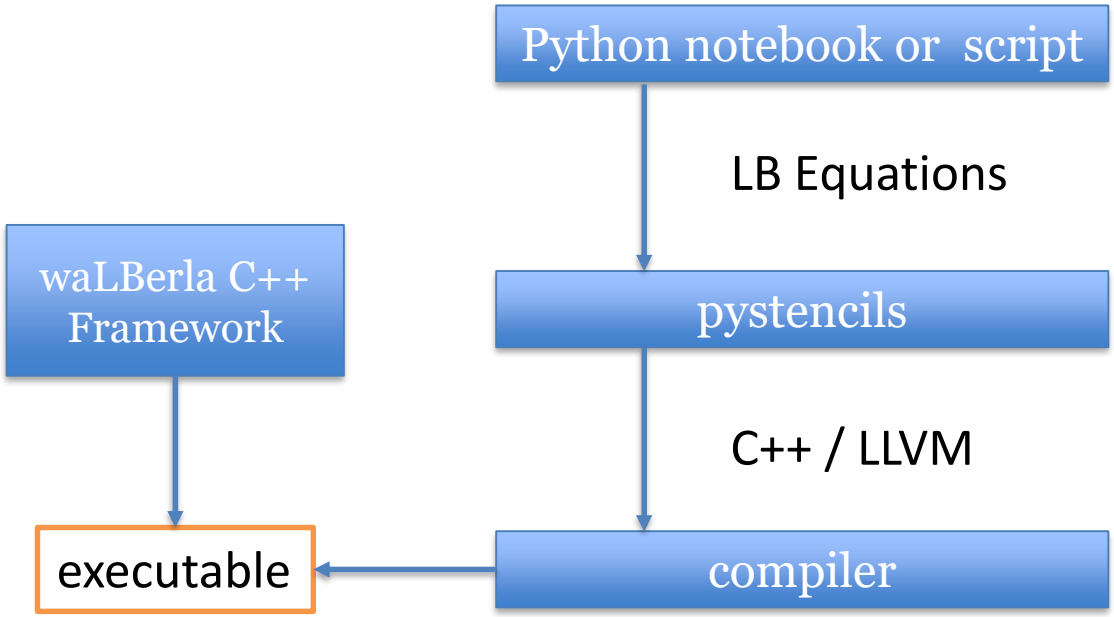



```
FUNC_PREFIX void kernel(double * RESTRICT  const  fd_s, double * RESTRICT  fd_d, double C)
{
   for (int ctr_0 = 1; ctr_0 < 19; ctr_0 += 1)
   {
      double * RESTRICT   fd_d_C = 30*ctr_0 + fd_d;
      double * RESTRICT  const   fd_s_E = 30*ctr_0 + fd_s + 30;
      double * RESTRICT  const   fd_s_C = 30*ctr_0 + fd_s;
      double * RESTRICT  const   fd_s_W = 30*ctr_0 + fd_s - 30;
      for (int ctr_1 = 1; ctr_1 < 29; ctr_1 += 1)
      {
          fd_d_C[ctr_1] = (fd_s_C[ctr_1 - 1] + fd_s_C[ctr_1 + 1] + fd_s_E[ctr_1] + fd_s_W[c
      }
   }
}
```

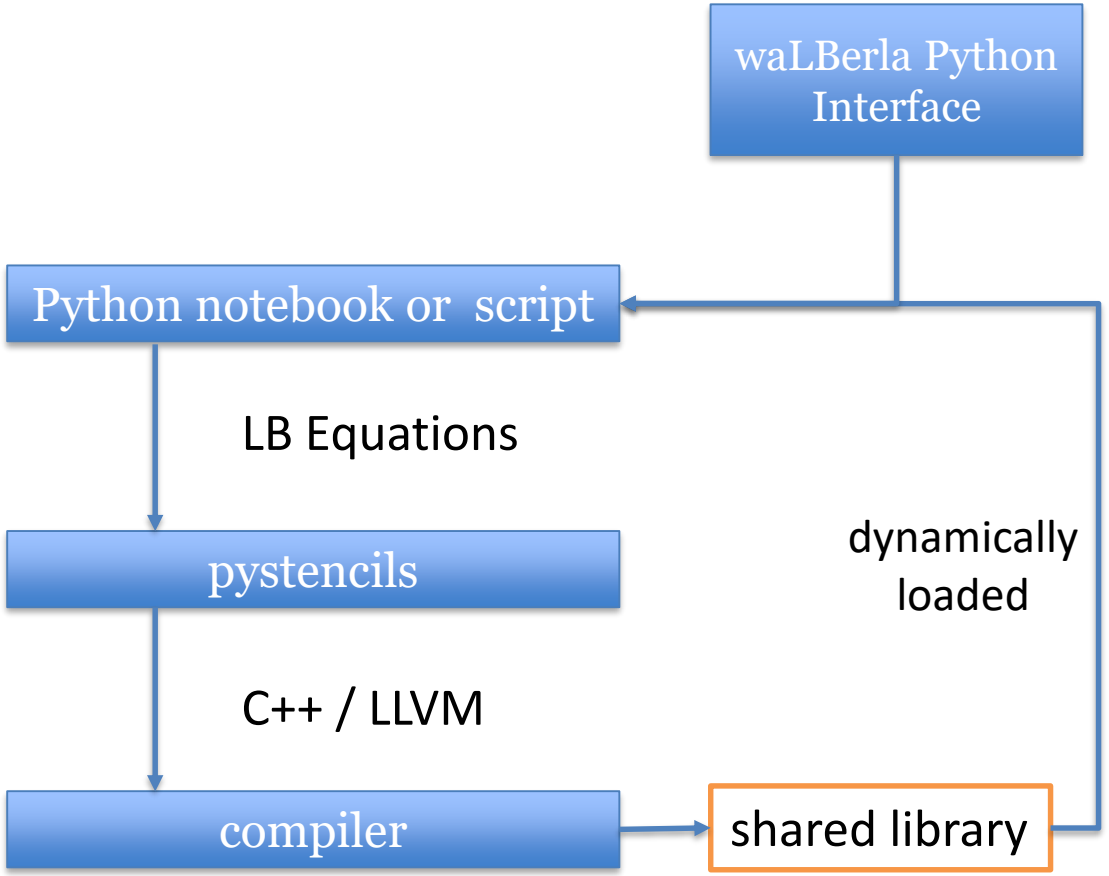
pystencils


sympy Equations containing fields (neighbor accesses)
pystencils
Transformations
Array access
Loop Splitting
Move Constants before loop
Blocking
Abstract Syntax Tree
Kernel
Loop
Add
Assign
Condition
Mul
Backends
C(++)
CUDA
LLVM
Python JIT
C/C++ output
Python Function
C/C++ Code

pystencils: Static Compilation



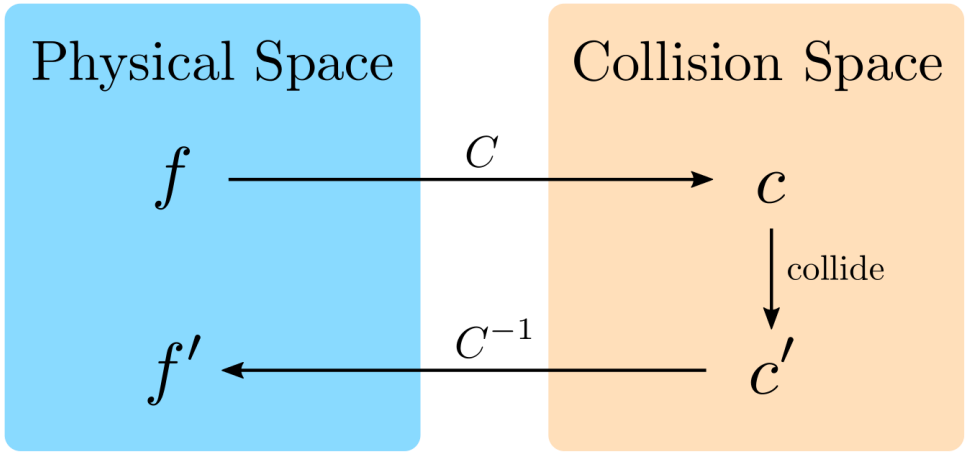
pystencils: Just-in-Time compilation



Benefits of JIT: array sizes and parameter values known at compile time

Generic Lattice Boltzmann Method

$$f_q(\mathbf{x} + \mathbf{c}_q \delta t, t + \delta t) = K\left(f_q(\mathbf{x}, t)\right)$$



$$c := C(f)$$
$$c^{(eq)} := C\left(f^{(eq)}\right)$$
$$c' := (\mathbb{1} - S)c + Sc^{(eq)}$$
$$S := diag(\omega_1, \omega_2, ..., \omega_q)$$

Moment-based methods

Example: Default D2Q9 MRT:

```
createLatticeBoltzmannMethod(stencil='D2Q9',
                             method='mrt',
                             equilibriumAccuracyOrder=3)
```

Moment	Eq. Value	Relaxation Rate
1	ρ	ω_0
y	u_1	ω_1
x	u_0	ω_1
$3y^2 - 2$	$-\rho + 3u_1^2$	ω_2
xy	$u_0 u_1$	ω_2
$3x^2 - 2$	$-\rho + 3u_0^2$	ω_2
$3xy^2 - 2x$	$3u_0 u_1^2 - u_0$	ω_3
$3x^2 y - 2y$	$3u_0^2 u_1 - u_1$	ω_3
$9x^2y^2 - 6x^2 - 6y^2 + 4$	$\rho - 3u_0^2 - 3u_1^2$	ω_4

Method is fully defined by these quantities – this table can be freely edited

Moment-based methods

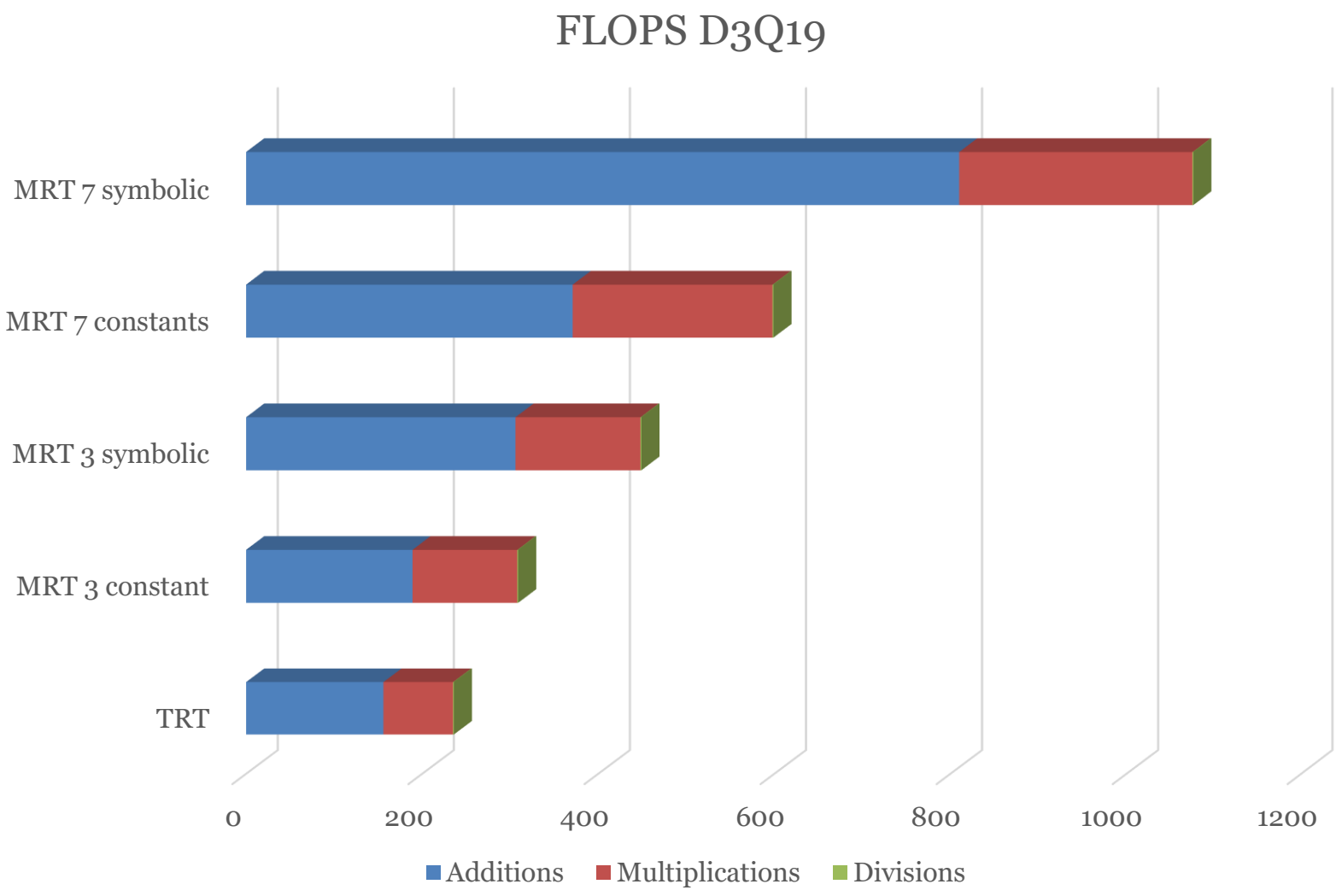
```
s.createSimplificationReport(method.getCollisionRule())
```

Name	Runtime	Adds	Muls	Divs	Total
OriginalTerm	-	256	165	0	421
expand	54.25 ms	79	165	0	244
replaceSecondOrderVelocityProducts	22.81 ms	89	173	0	262
expand	19.68 ms	81	165	0	246
factorRelaxationRates	16.44 ms	81	110	0	191
replaceDensityAndVelocity	12.56 ms	81	110	0	191
replaceCommonQuadraticAndConstantTerm	21.77 ms	69	86	0	155
factorDensityAfterFactoringRelaxationTimes	16.21 ms	69	86	0	155
subexpressionSubstitutionInMainEquations	17.54 ms	65	82	0	147
addSubexpressionsForDivisions	6.26 ms	65	82	0	147

greedy method wouldn't work

- FLOPS reduced from 421 to 147
- matches best manual SRT/TRT implementation in waLBerla
- no special handling required since automatic simplification is good enough

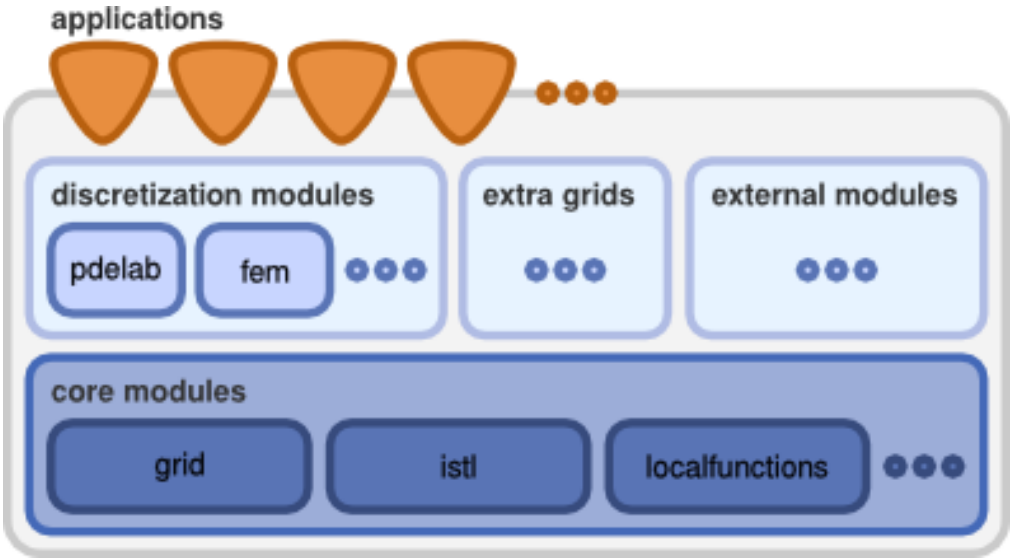
Moment-based methods



Code Generation for DG

DUNE

DUNE, the Distributed and Unified Numerics Environment is a modular toolbox for solving partial differential equations (PDEs) with grid-based methods.



Code generation dune-pdelab (group Prof. Bastian)

EXADUNE: Making **DUNE** ready for exascale computing:

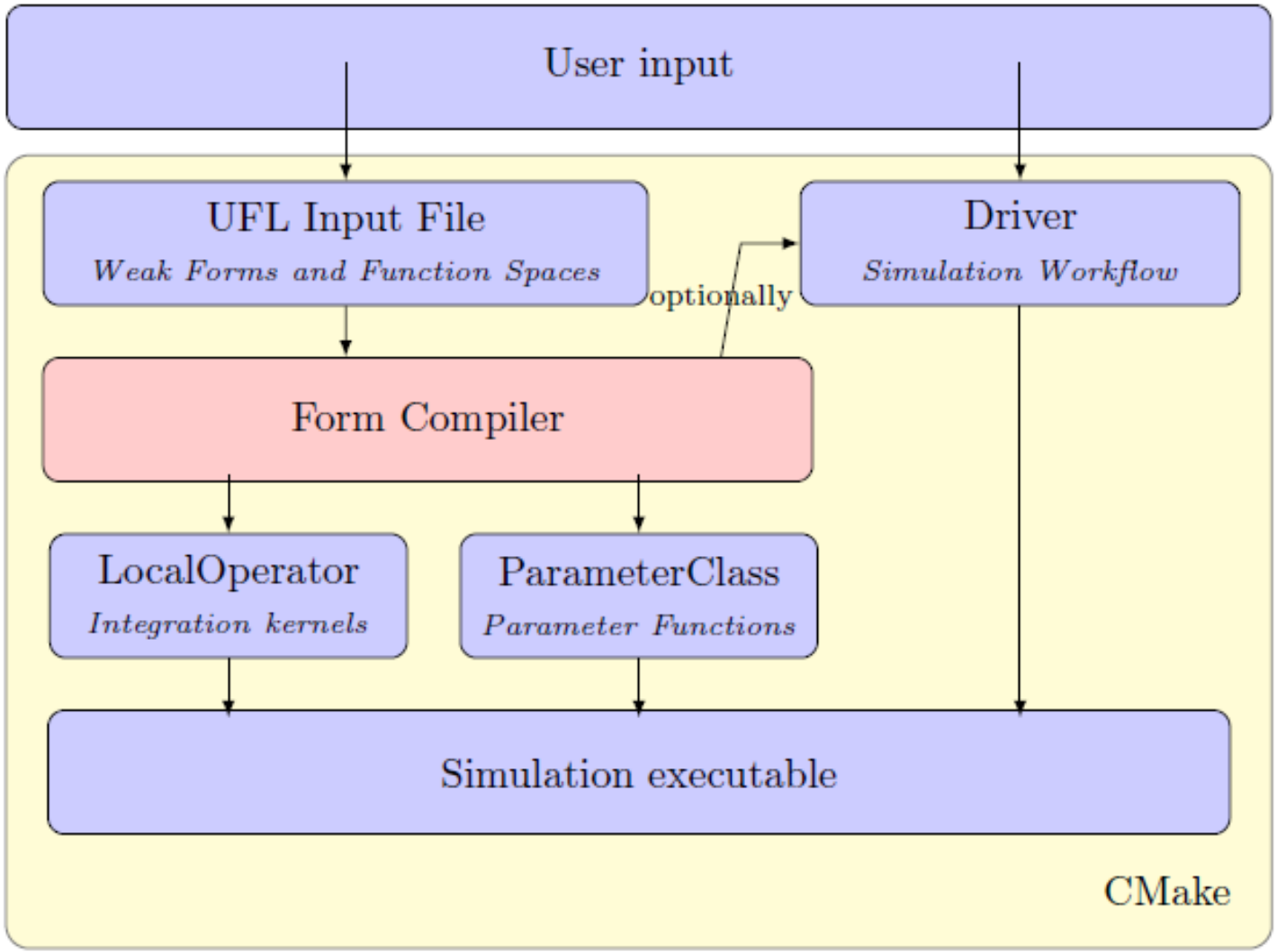
- high order DG methods ⇒ Overcoming memory boundedness
- optimizing SIMD throughput
- algorithmic optimization: exploiting tensor product structure of finite elements
- multi-threaded grid traversal
- Algebraic Multigrid

Some conclusions:

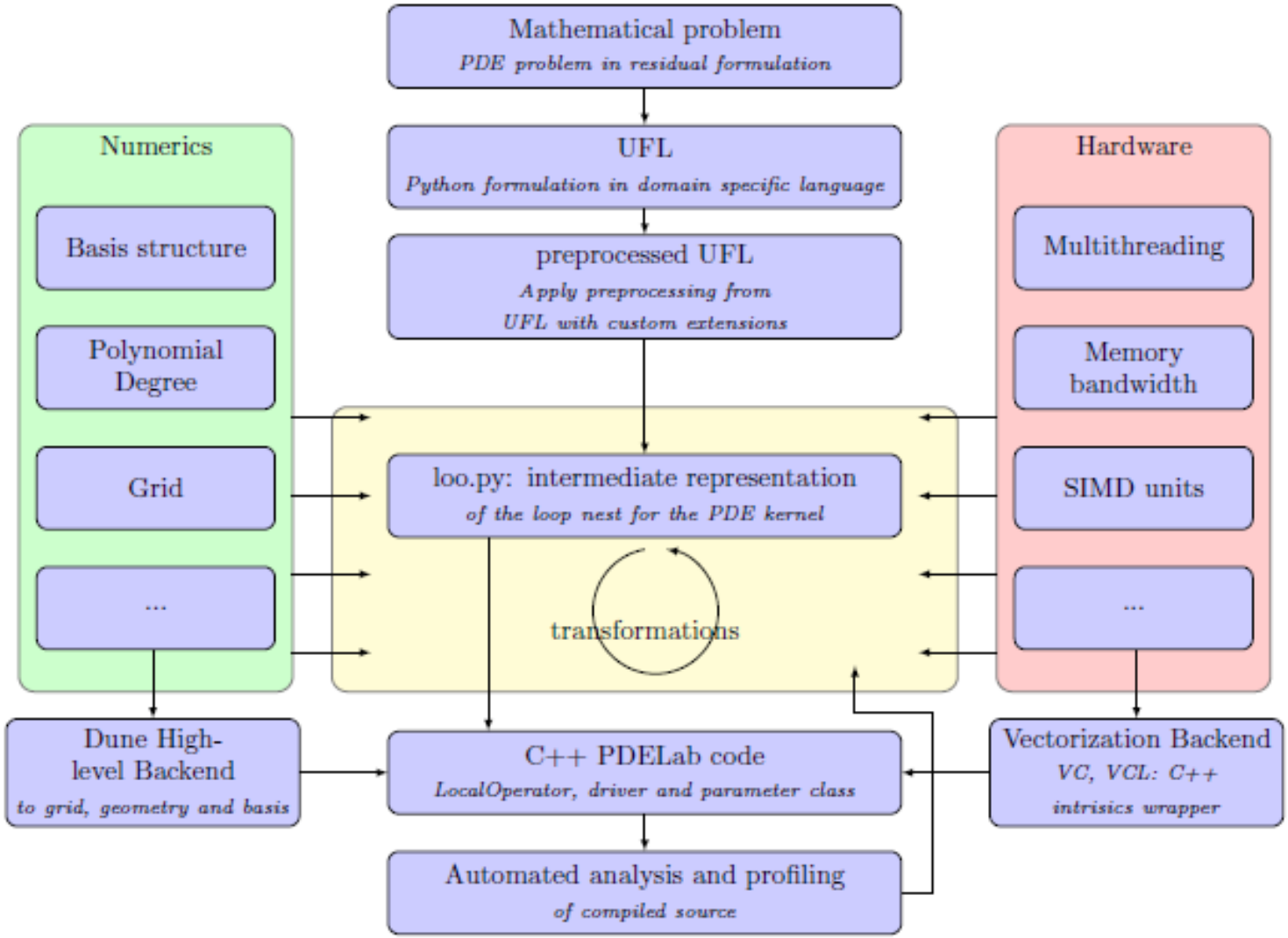
- Automatic vectorization is not satisfactory
- Optimization depends heavily on the PDE model

⇒ How can **performance portability** be achieved?

Code generation workflow



Form compiler approach



Choice of intermediate representation: loopy

- developed by Andreas Klöckner (UIUC)
- Python data structure of a loop kernel with
  - polyhedral loop domain as ISL set
  - instructions using simple symbolic language *pymbolic*
- Transformations targetting
  - fusion, splitting, unrolling etc.
  - memory layout
  - vectorization
  - common subexpression elimination
- Code generation for several targets
  - C, OpenCL, CUDA etc.
  - C++, *dune-pdelab*

Benchmark Setup

Problem:

- Diffusion reaction problem with full permeability tensor
- SIPG DG discretization on structured, axiparallel grid
- MPI-parallel on 16 processes (=whole processor)
- 100MB of DOFs per core

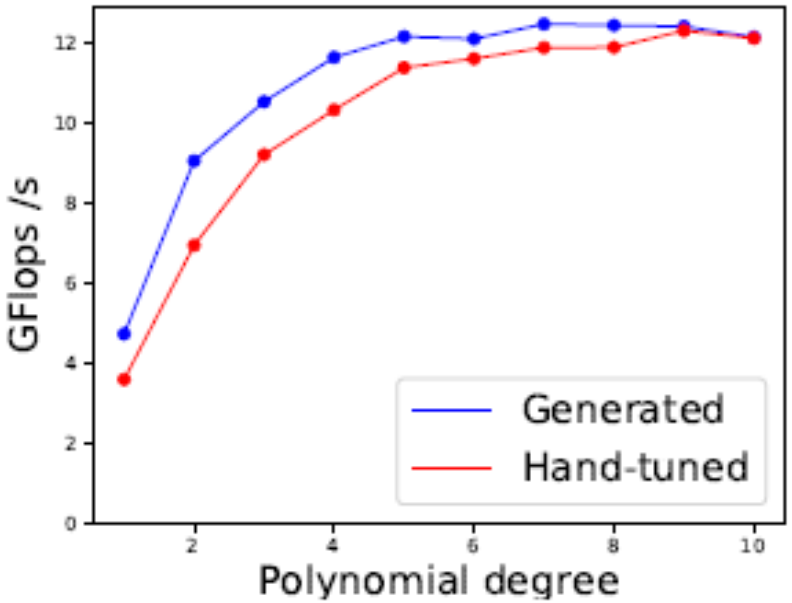
Hardware:

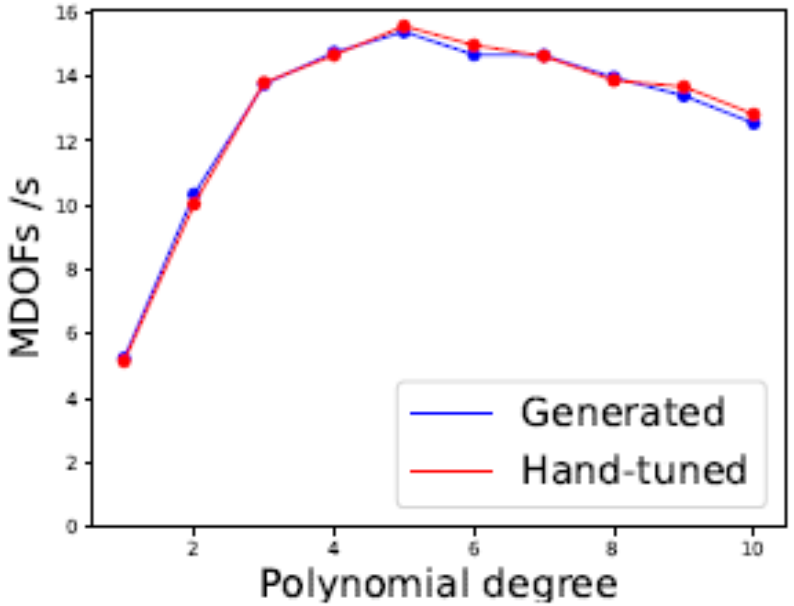
- Intel Xeon Processor E5-2698 v3
- 16 cores
- 2.3 GHz, AVX2: 1.9 GHz
- Theoretical peak performance per core: 30 GFlops/s

Measuring GFlops/s:

- Counting operations: Instrumented C++ floating point type
- Separate executables for time and FLOP measurements

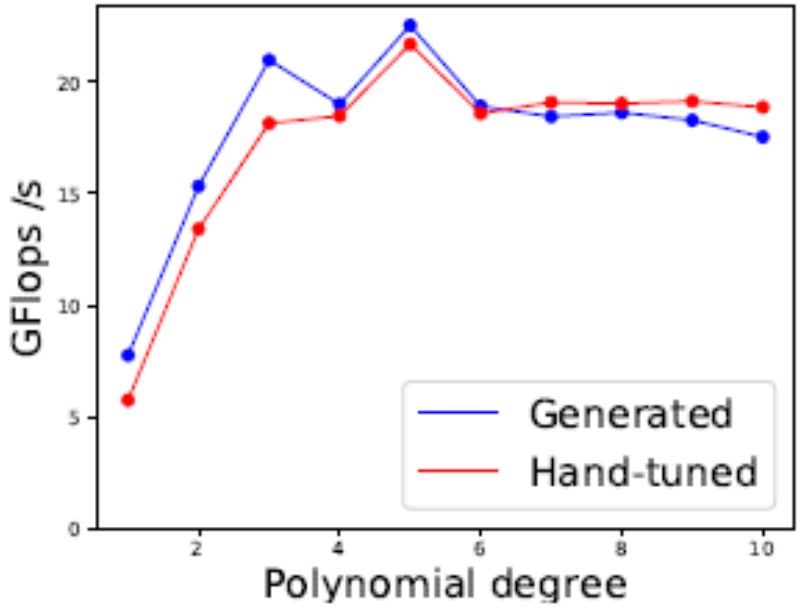
Performance: Residual evaluation

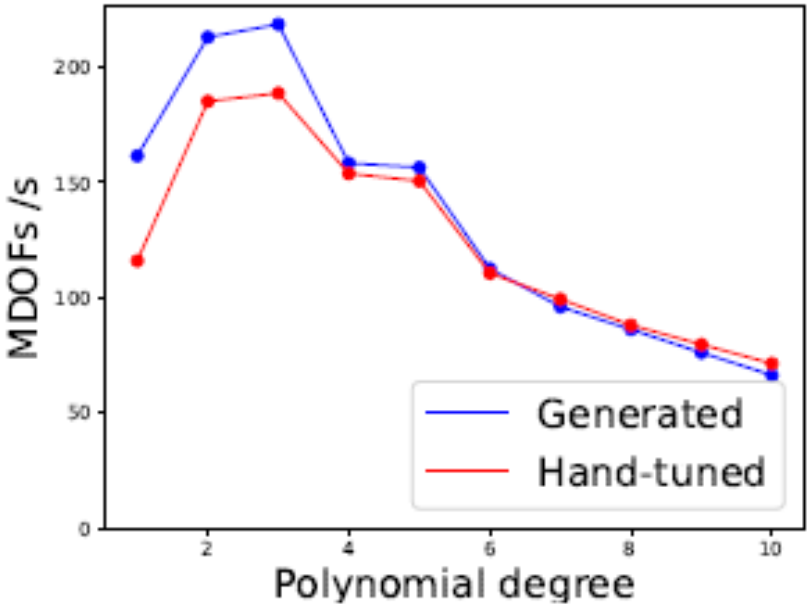




Performance: Sumfactorization kernels (I)

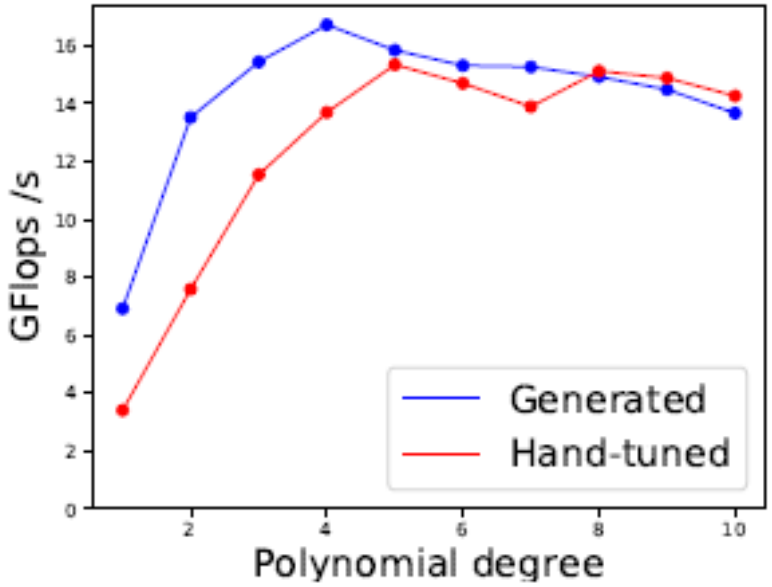
Cell integrals, calculating u and ∇u simultaneously.

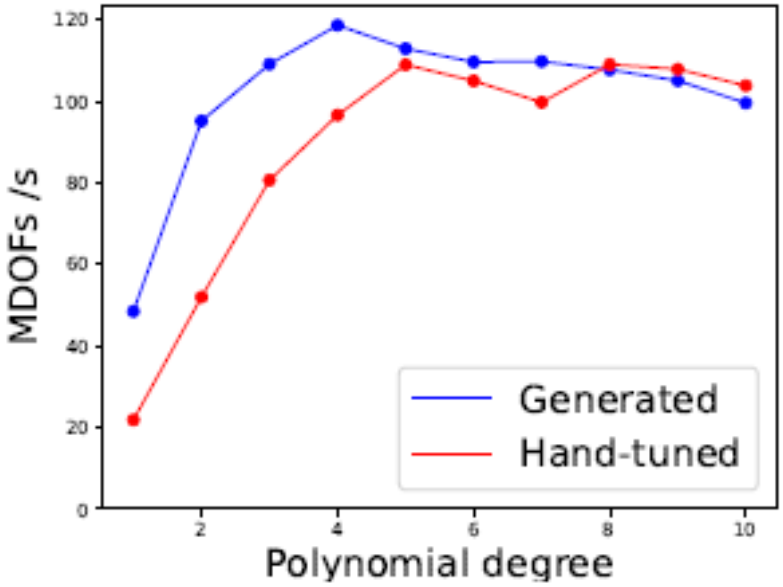




Performance: Sumfactorization kernels (II)

Facet integrals, calculating u and ∇u simultaneously.





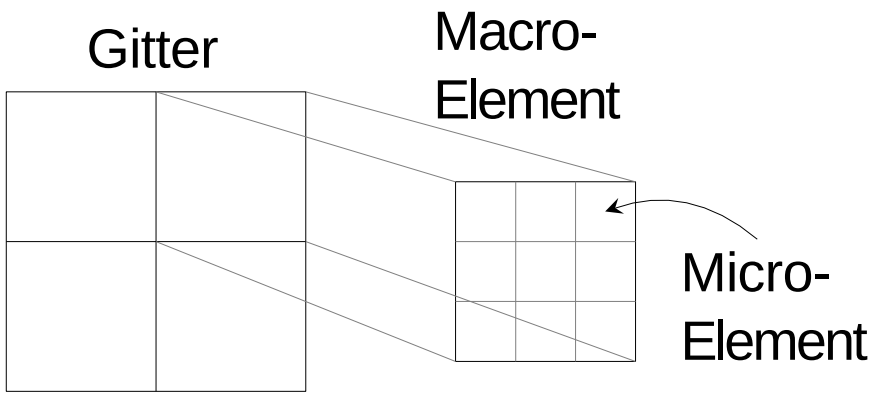
Workload is constant per DOF (matches theory).

Additional Data Structures for DG

Locally block-structured FEM (group Prof. Engwer)

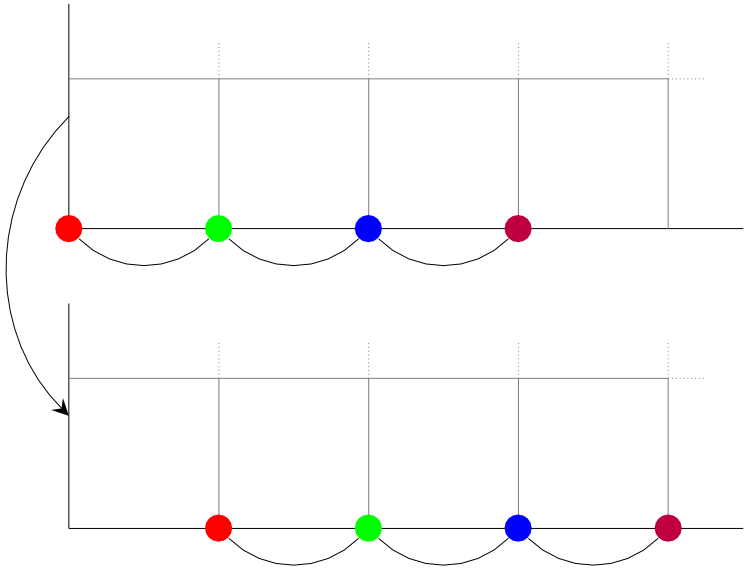
Goal: Generate efficient lower order FEM code

Idea:
- Coarse grid of macro elements
- Refine macro-Elements in B Micro-Elements
- In local kernels higher arithmetic intensity



Vectorization

- Vectorize neighboring micro-elements
- Vector length 4 → local Kernel speedup ~ 3.5



PACXX Compiler as Back-end

GPU mit PACXX (group Prof. Gorlatch)

- PACXX: Programming Accelerators with C++
  - Programming model for accelerators: massively parallel > 10^8 virtual Threads
  - An extended C++ Compiler for Code Generation
  - A runtime system to run kernels on different target architectures
- Macro-Element corresponds to Thread-Block
- Micro-Element corresponds to one Thread

Programming Accelerators with C++ (PACXX)

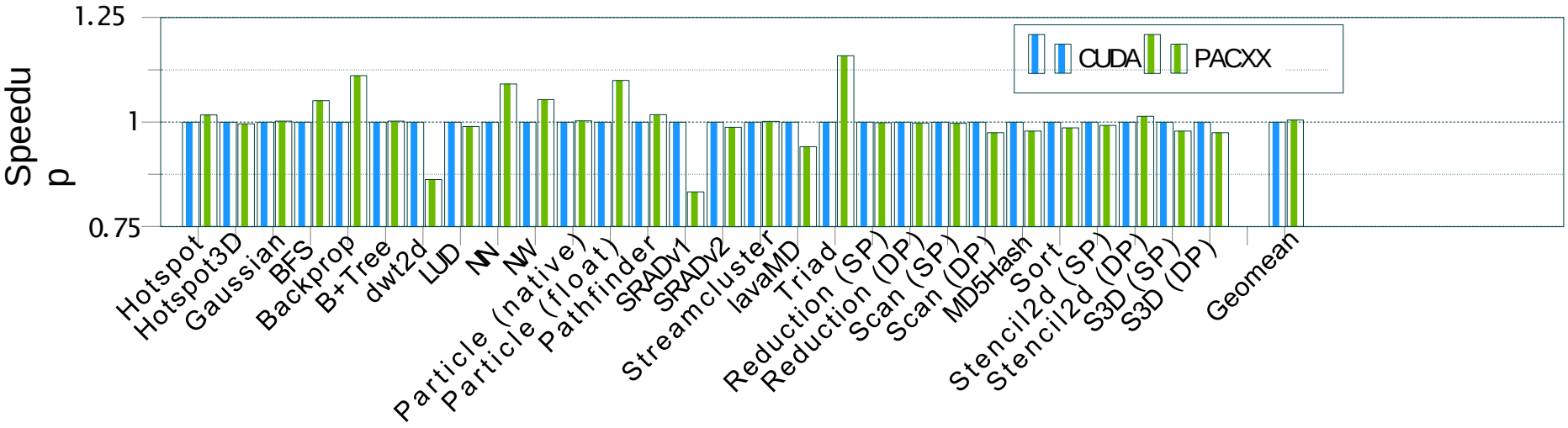
- Supports C++11/14/17/2a
- Integration of accelerator code in C++ (single-source).
- Based on Clang/LLVM
- Integrated Just-In-Time Compiler
- Various Back-ends
- Open Source: https://github.com/pacxx/pacxx-llvm

Easier to program accelerators

```
class DeviceBuffer
  { public:
    void *get () { return _buffer; }
  private:
    void *[[pacxx::device_memory]] _buffer;
};
```

- PACXX adds attributes to C++ (e.g. `[[pacxx::device_memory]]`).
- Clang Front-End ensures that kernel accesses only memory that has been allocated
- Less errors in memory management and less debugging
- Runtime optimizations in the kernels are possible

Performance on Nvidia GPUs



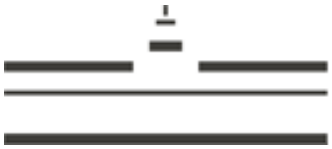
Performance of PACXX comapred to Nvidia CUDA 9.0 on an Nvidia 940m (Maxwell Architecture). Benchmarks from Rodinia and SHOC Benchmark suits.

Next Steps in the project

- Fuse code generation approaches (sympy vs. pymbolic, pystencils vs. loop.py)
- Integrate PACXX backend
- Test inter-framework generation of kernels for waLBerla and DUNE

Thank you for your Attention!

Questions ?



Westfälische Wilhelms-Universität Münster

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT