HPC$^2$SE: Hardware- and Performance-aware Codegeneration for Computational Science and Engineering

7. HPC-Status-Konferenz der Gauß-Allianz
Stuttgart, December 5$^{th}$ 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
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

```c
#include "MultiGrid/MultiGrid.h"

void Smoother_4() {
  exchsolutionData_4((0);
  #pragma omp parallel for schedule(static) num_threads(8)
  for (int fragmentIdx = 0; ++fragmentIdx)
    if (isValidForSubdomain[fragmentIdx][0]) {
      for (int y = iterationOffsetBegin[fragmentIdx][0]; y += i)
        for (int x = iterationOffsetBegin[fragmentIdx][0]; x += i)
          slottedFieldData_Solution[fragmentIdx][0] (((y*19)+(x*17)))) =
            (fieldData_LaplCoeff[fragmentIdx][0] (((y*17)+(x+1))))*
            (fieldData_RHS[fragmentIdx][0] (((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

1. Layer 1: Continuous Domain & Continuous Model
2. Layer 2: Discrete Domain & Discrete Model
3. Layer 3: Algorithmic Components & Parameters
4. Layer 4: Complete Program Specification
HPC\textsuperscript{2}SE: Code Generation Pipeline

Diagram showing the process from DSL to object code, including model problems, computation kernel variants, and simulation frameworks.
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


## Overview of Implemented Kernels

<table>
<thead>
<tr>
<th>kernel name</th>
<th>prop. step</th>
<th>data layout</th>
<th>addr.</th>
<th>parallel</th>
<th>blocking</th>
<th>( B_1 ) [B/FLUP]</th>
</tr>
</thead>
<tbody>
<tr>
<td>push-soa</td>
<td>OS</td>
<td>SoA</td>
<td>D</td>
<td>x</td>
<td></td>
<td>456</td>
</tr>
<tr>
<td>push-aos</td>
<td>OS</td>
<td>AoS</td>
<td>D</td>
<td>x</td>
<td></td>
<td>456</td>
</tr>
<tr>
<td>pull-soa</td>
<td>OS</td>
<td>SoA</td>
<td>D</td>
<td>x</td>
<td></td>
<td>456</td>
</tr>
<tr>
<td>pull-aos</td>
<td>OS</td>
<td>AoS</td>
<td>D</td>
<td>x</td>
<td></td>
<td>456</td>
</tr>
<tr>
<td>blk-push-soa</td>
<td>OS</td>
<td>SoA</td>
<td>D</td>
<td>x</td>
<td>x</td>
<td>456</td>
</tr>
<tr>
<td>blk-push-aos</td>
<td>OS</td>
<td>AoS</td>
<td>D</td>
<td>x</td>
<td>x</td>
<td>456</td>
</tr>
<tr>
<td>blk-pull-soa</td>
<td>OS</td>
<td>SoA</td>
<td>D</td>
<td>x</td>
<td>x</td>
<td>456</td>
</tr>
<tr>
<td>blk-pull-aos</td>
<td>OS</td>
<td>AoS</td>
<td>D</td>
<td>x</td>
<td>x</td>
<td>456</td>
</tr>
<tr>
<td>list-push-soa</td>
<td>OS</td>
<td>SoA</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>528</td>
</tr>
<tr>
<td>list-push-aos</td>
<td>OS</td>
<td>AoS</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>528</td>
</tr>
<tr>
<td>list-pull-soa</td>
<td>OS</td>
<td>SoA</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>528</td>
</tr>
<tr>
<td>list-pull-aos</td>
<td>OS</td>
<td>AoS</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>528</td>
</tr>
<tr>
<td>list-pull-split-nt-1s</td>
<td>OS</td>
<td>SoA</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>376</td>
</tr>
<tr>
<td>list-pull-split-nt-2s</td>
<td>OS</td>
<td>SoA</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>376</td>
</tr>
<tr>
<td>list-aa-soa</td>
<td>AA</td>
<td>SoA</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>340</td>
</tr>
<tr>
<td>list-aa-aos</td>
<td>AA</td>
<td>AoS</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>340</td>
</tr>
<tr>
<td>list-aa-ria-soa</td>
<td>AA</td>
<td>SoA</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>304-342</td>
</tr>
<tr>
<td>list-aa-pv-soa</td>
<td>AA</td>
<td>SoA</td>
<td>I</td>
<td>x</td>
<td>x</td>
<td>304-342</td>
</tr>
</tbody>
</table>
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
sympy Equations containing fields (neighbor accesses)

pystencils

Transmutations
- Array access
- Loop Splitting
- Move Constants before loop
- Blocking

Abstract Syntax Tree
- Kernel
- Loop
- Add
- Assign
- Condition
- Mul

Backends
- C(++)
- CUDA
- LLVM
- Python JIT
- 

Python Function

C/C++ Code
pystencils: Static Compilation

- Python notebook or script
  - LB Equations
  - pystencils
    - C++ / LLVM
  - compiler
  - executable

waLBerla C++ Framework
pystencils: Just-in-Time compilation

Benefits of JIT: array sizes and parameter values known at compile time
\[ f_q(x + c_q \delta t, t + \delta t) = K(f_q(x, t)) \]

- **Physical Space**
  - \( f \)
  - \( f' \)

- **Collision Space**
  - \( c \)
  - \( c' \)
  - \( c^{(eq)} := C\left(f^{(eq)}\right) \)

\[ c' := (1 - S)c + Sc^{(eq)} \]

\[ S := \text{diag}(\omega_1, \omega_2, \ldots, \omega_q) \]
# Moment-based methods

Example: Default D2Q9 MRT:

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

<table>
<thead>
<tr>
<th>Moment</th>
<th>Eq. Value</th>
<th>Relaxation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\rho$</td>
<td>$\omega_0$</td>
</tr>
<tr>
<td>$y$</td>
<td>$u_1$</td>
<td>$\omega_1$</td>
</tr>
<tr>
<td>$x$</td>
<td>$u_0$</td>
<td>$\omega_1$</td>
</tr>
<tr>
<td>$3y^2 - 2$</td>
<td>$-\rho + 3u_1^2$</td>
<td>$\omega_2$</td>
</tr>
<tr>
<td>$xy$</td>
<td>$u_0u_1$</td>
<td>$\omega_2$</td>
</tr>
<tr>
<td>$3x^2 - 2$</td>
<td>$-\rho + 3u_0^2$</td>
<td>$\omega_2$</td>
</tr>
<tr>
<td>$3xy^2 - 2x$</td>
<td>$3u_0u_1^2 - u_0$</td>
<td>$\omega_3$</td>
</tr>
<tr>
<td>$3x^2y - 2y$</td>
<td>$3u_0^2u_1 - u_1$</td>
<td>$\omega_3$</td>
</tr>
<tr>
<td>$9x^2y^2 - 6x^2 - 6y^2 + 4$</td>
<td>$\rho - 3u_0^2 - 3u_1^2$</td>
<td>$\omega_4$</td>
</tr>
</tbody>
</table>

Method is fully defined by these quantities – this table can be freely edited
## Moment-based methods

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

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

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

**greedy method wouldn’t work**
Moment-based methods

FLOPS D3Q19

- MRT 7 symbolic
- MRT 7 constants
- MRT 3 symbolic
- MRT 3 constant
- TRT

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

User input

UFL Input File
*Weak Forms and Function Spaces*

Form Compiler

Driver
*Simulation Workflow*

optionally

LocalOperator
*Integration kernels*

ParameterClass
*Parameter Functions*

Simulation executable

CMake
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 \textit{pymbolic}
- Transformations targeting
  - fusion, splitting, unrolling etc.
  - memory layout
  - vectorization
  - common subexpression elimination
- Code generation for several targets
  - C, OpenCL, CUDA etc.
  - C++, \textit{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

![Graph 1: GFlops/s vs Polynomial degree](image1)

- Blue line: Generated
- Red line: Hand-tuned

![Graph 2: MDOFs/s vs Polynomial degree](image2)

- Blue line: Generated
- Red line: Hand-tuned
Performance: Sumfactorization kernels (I)

Cell integrals, calculating $u$ and $\nabla u$ simultaneously.
Performance: Sumfactorization kernels (II)

Facet integrals, calculating $u$ and $\nabla u$ 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 $\rightarrow$ local Kernel speedup $\sim 3.5$
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](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 of PACXX compared 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?