Recent advances for code generation in the $HPC^2 SE$ 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

```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 < fieldData_LaplCoeff;)
for (int y = iterationOffsetBegin[fragmentIdx]; y < iterationOffsetEnd[fragmentIdx] + 1; y++)
for (int x = iterationOffsetBegin[fragmentIdx]; x < iterationOffsetEnd[fragmentIdx] + 1; x++)

  slottedFieldData_Solution[fragmentIdx][x][y][0] =
      fieldData_LaplCoeff[fragmentIdx][x][y] +
      fieldData_RHS[fragmentIdx][x][y];
```

![Diagram of model problem and triangulation](image)

Engwer, et al.
**HPC\(^2\) SE code generation approach**

1. **DSL**
   - Modelproblem “Free Surface” (User written)
   - Modelproblem “Navier-Stokes” (User written)
   - Modelproblem “Maxwell” (User written)

2. **DSL-Compiler**
   - DSL-Zwischenrepräsentation
   - Mathematische Optimierung

3. **Computation Kernel** (Variant 1)
   - . . .

4. **Comp. Kernel** (Variant \(n\))

5. **Simulations-Framework**

6. **PACXX C++-Compiler**
   - LLVM-Zwischenrepräsentation
   - Algorithmische Optimierung

7. **Object-Code**
   - CPU
   - PHI
   - Host + GPU

8. **Anwendungs-Domäne**

9. **Algorithmische Domäne**

10. **Hardware Domäne**

---

Engwer, et al.
Outline

1. Our Code Generation Approach
2. Higher-order DG
3. Block-structured meshes
4. Stencil code generation
5. Platform portability?!
6. Discussion
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

loo.py: intermediate representation of the loop nest for the PDE kernel

C++ PDELab code
LocalOperator, driver and parameter class

Automated analysis and profiling of compiled source

Numerics
Basis structure
Polynomial Degree
Grid
Dune High-level Backend
to grid, geometry and basis

Hardware
Multithreading
Memory bandwidth
SIMD units
Vectorization Backend
VC, VCL: C++ intrinsics wrapper

Numérics
Basis structure
Polynomial Degree
Grid
Dune High-level Backend
to grid, geometry and basis

HPC
2
SE

Engwer, et al.
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
Current State

Low-level optimization

- Performance optimizations on intermediate representation (e.g., 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)
Higher-order DG
Evaluation of $\hat{\delta}_0 \hat{u}$ in 3D using sum factorization

$$
\hat{\delta}_0 \hat{u}(\xi_{i_1,i_2,i_3}) = \sum_{j_3 \in J^{(3)}} A_{i_3,j_3}^{(3)} \left( \sum_{j_2 \in J^{(2)}} A_{i_2,j_2}^{(2)} \left( \sum_{j_1 \in J^{(1)}} A_{i_1,j_1}^{(1)} x_{j_1,i_2,j_3} y_{j_3,i_2,i_3} \right) y_{j_2,j_3,i_1} \right) y_{j_2,i_2,i_3}
$$

$A_{i,j}^{(1)} = \hat{\theta}'_i(\xi_j^{(1)})$: Derivatives of the 1D Basis at the 1D quadrature points. $A_{i,j}^{(k)} = \hat{\theta}_i(\xi_j^{(k)})$, $k > 1$: Evaluations of the 1D Basis at the 1D quadrature points.
Vectorization by Loop Fusion

How can we adopt this to a 512 bit SIMD width?

Engwer, et al.
Parallelism through Loop Splitting

Idea: Find parallelism by splitting loops: e.g. two times half the quadrature points.

Engwer, et al.
Performance on Intel Haswell

- Intel Xeon processor E5-2698 v3
- 2×16 cores, 2.3 GHz
- Theoretical node peak performance: 1.17 TFlops/s

![Graph](image1)

![Graph](image2)
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
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

$\rightarrow$ 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 ~ 3.5x
## Performance - Matrix free operator application

<table>
<thead>
<tr>
<th>Problem</th>
<th>Variant</th>
<th>MDof/s</th>
<th>Speedup vs $k = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear elasticity (LE)</td>
<td>$k = 8$</td>
<td>2.1</td>
<td>2.2x</td>
</tr>
<tr>
<td></td>
<td>$k = 8 + \text{SIMD4}$</td>
<td>7.1</td>
<td>7.8x</td>
</tr>
<tr>
<td>Poisson (P)</td>
<td>$k = 16$</td>
<td>18.4</td>
<td>5.6x</td>
</tr>
<tr>
<td></td>
<td>$k = 16 + \text{SIMD4}$</td>
<td>41.3</td>
<td>12.6x</td>
</tr>
</tbody>
</table>

Engwer, et al.
# Performance - Matrix free operator application

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

Engwer, et al.
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
- Heterogeneous computation
DG Stencil for ExaStencils
Scaling on SuperMUC-NG
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

```cpp
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

```cpp
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 similar to Map/Reduce
- Split into two separate kernels

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

Refinement=6 (24833 cells)

- Device-scatter kernel:
  - Host: $4.86 \cdot 10^{-4}$
  - NORV: $8.41 \cdot 10^{-4}$
  - RV: $9.86 \cdot 10^{-4}$

- Device-host:
  - Host: $6 \cdot 10^{-3}$
  - NORV: $6.88 \cdot 10^{-3}$
  - RV: $6.9 \cdot 10^{-3}$

- Host:
  - NORV: $1.22 \cdot 10^{-2}$

Sekunden pro Iteration
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?