NHR FAU

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Introduction to Code Generation Techniques

Sebastian Kuckuk

EXA2PRO-EoCoE joint workshop, Feb 23rd, 2021

Motivation

Motivation – 3P

- Performance
  - Optimal usage of (expensive) hardware
  - Minimizing energy expenditure
  - Assessment via profiling and performance modelling

Motivation – 3P

- Performance

- Productivity
  - = programmer productivity
  - LOC per hour
  - Minimize effort to implement a certain
    - Algorithm
    - Feature
    - Optimization
    - Parallelization
    - …

Motivation – 3P

- Performance
- Productivity
- Portability
  - Execution on varying target hardware
    - Intel CPU -> AMD CPU
    - CPU -> GPU
    - CPU -> CPU + GPU
    - …
  - Portable does not automatically include performance portable

Motivation – 3P

- Performance
- Productivity
- Portability
- Issue
  - Increasing diversity in hardware and software

Motivation – 3P

- Issue: increasing diversity in hardware …
  - CPU – AMD, Intel, IBM, Fujitsu, …
  - GPU – NVIDIA, AMD, Intel
  - FPGA
- … and software
  - Vector instruction sets, OpenMP
  - CUDA, ROCm
  - OpenMP 4+, OpenACC, SYCL, DPC++
- MPI+X is predominant

Motivation – Specialized Implementations

- Standalone applications (typically)
  - For a single purpose
  - Running on a single target
  - Implemented and maintained by a very small group of developers

- Can be highly optimized
- (Performance) portability challenging
- Full control for the developer
- Low productivity for larger applications

Motivation – Remedy

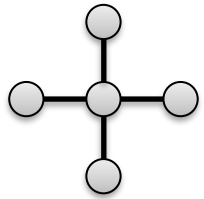
- Remedy: raising the level of abstraction
  - Describe what you want to compute not how it is to be computed

- Possible abstractions in this scope
  - Data (structures and handling)
  - Computations performed on the data
  - Parallelization
  - + Higher level: physics, math, …

- This talk
  - View of application scientists (‘the person doing applied science’)
  - View of computer scientists (‘the person enabling others to do applied science’)

Motivation – Running Example

- Running example: stencil code
  - Laplace equation $-\Delta u = 0$
  - Finite differences (2D) $-u_{i-1,j} + 2u_{i,j} - u_{i+1,j} - u_{i,j-1} + 2u_{i,j} - u_{i,j+1} = 0$
  - Jacobi iterations (undamped) $u_{i,j}^{new} = \left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\right)/4$
  - Implementation in C++



```
for (int j = 1; j < ny - 1; ++j) {
  for (int i = 1; i < nx - 1; ++i) {
    u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                             + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
  }
}
```

Motivation – Running Example

- Two versions for CPU and GPU

```
# pragma omp parallel for collapse(2)
for (int j = 1; j < ny - 1; ++j) {
  for (int i = 1; i < nx - 1; ++i) {
    u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                             + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
  }
}

#pragma acc parallel loop collapse(2)
for (int i = 1; i < nx - 1; ++i) {
  for (int j = 1; j < ny - 1; ++j) {
    u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                             + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
  }
}
```

Approaches

Approach – HPC framework

- Traditionally HPC frameworks/ libraries are used as a go-to solution

- Examples
  - xSDK (https://xsdk.info/) with popular packages such as Trilinos, PETSc, hypre, deal.II, …
  - waLBerla (https://walberla.net/) for massively parallel LBM and multiphysics applications

- Typical challenges
  - Introducing new programming models
  - Porting to novel hardware

Approach – HPC framework

- Users
  - Increase in productivity …
  - … after an initial cost to learn about the framework
  - Limited control over (performance) portability

- Developers
  - Can be highly optimized
  - (Performance) portability challenging
  - Full control

Approach – Abstraction Layers

- (Performance) portability can be addressed by adding another abstraction layer in a framework

- Example: Kokkos (https://github.com/kokkos/kokkos)
  - Since 2012
  - C++
  - CPU and GPU
  - Main focus: parallelism within one compute node
  - DOE national labs with dedicated developers
  - 100 projects, 500 developers

Approach – Abstraction Layers

- Example: Laplace with Kokkos

```
parallel_for (RangePolicy<>(1, ny-1), KOKKOS_LAMBDA (int j) {
  for (int i = 1; i < nx - 1; ++i) {
    u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                             + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
  }
});

parallel_for (MDRangePolicy<Rank <2> >({1, 1}, {nx-1, ny-1}),
              KOKKOS_LAMBDA (int i, int j) {
  u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                           + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
});
```

Approach – Abstraction Layers

- Users
  - Need to implement in a potentially very restrictive abstraction layer …
  - … which is still C++
  - Implementations need to be designed with target hardware in mind
- Developers
  - More complicated integration into workflow
  - Only one implementation for multiple back ends
- Possible alternative: code generation

Approach – Code Generation

- C++ templates provide one type of code generation

```
template<typename T, int nx, int ny>
void jacobi(T* u, T* u_new) {
  for (int j = 1; j < ny - 1; ++j) {
    for (int i = 1; i < nx - 1; ++i) {
      u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                               + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
}}}
```

⇨
```
void jacobi(float* u, float* u_new) {
  for (int j = 1; j < 258 - 1; ++j) {
    for (int i = 1; i < 258 - 1; ++i) {
      u_new[j*258 + i] = /* ... */
}}}
```

⇨
```
void jacobi(double* u, double* u_new) { /* ... */ }
```

⇨
```
void jacobi(std::complex<double>* u, std::complex<double>* u_new) { /* ... */ }
```

Approach – Code Generation

- Users
  - Build-in language feature
  - Can increase compile times (by a lot)
  - Increases code complexity
- Developers
  - As above
  - Ensuring that all possible variants work correctly can be challenging
- Alternative: source code generation with an external tool

Code Generation – String-Based

- Reminder: Two versions for CPU and GPU

```
# pragma omp parallel for collapse(2)
for (int j = 1; j < ny - 1; ++j) {
  for (int i = 1; i < nx - 1; ++i) {
    u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                             + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
  }
}

#pragma acc parallel loop collapse(2)
for (int i = 1; i < nx - 1; ++i) {
  for (int j = 1; j < ny - 1; ++j) {
    u_new[j*nx + i] = 0.25 * ( u[    j*nx + i-1] + u[    j*nx + i+1]
                             + u[(j-1)*nx + i  ] + u[(j+1)*nx + i  ] );
  }
}
```

Code Generation – String-Based

- Most basic approach: simply use strings

- Example: Python generate function using f-strings

```
def generate_jacobi(use_openmp, use_openacc, nx, ny):
  def pragma():
    if use_openmp:
      return "#pragma omp parallel for collapse(2)\n"
    if use_openacc:
      return "#pragma acc parallel loop collapse(2)\n"
    return ""

  return f"""{pragma()}
for (int j = 1; j < {ny} - 1; ++j) {'{'}
  for (int i = 1; i < {nx} - 1; ++i) {'{'}
    u_new[j*{nx} + i] = ...
  {'}'}
{'}'}"""
```

Code Generation – String-Based

- Users
  - Low complexity
  - Still close to the final generated code
  - Extra step in development workflow
  - Debugging is not straight-forward
- Developers
  - Easy to conceptualize and implement
  - Limited range of features – not suitable for most code transformations
  - Validation of generated code challenging
- Issue: code generator does not understand the code it generates

Code Generation – IR

- Issue: code generator does not understand the code it generates
- Solution: compose an abstract representation that allows analysis and manipulation
- Usually called intermediate representation (IR) or abstract syntax tree (AST)
- Requires pretty-printer
- Enables code transformations …
  - … that can ideally be composed iteratively

Code Generation – IR

- Example: class hierarchy in Python

```
class Node:
  def __init__(self):
    pass

  def apply_rec(self, transformation):
    raise NotImplementedError

  def pp(self, printer, indent=""):
    raise NotImplementedError
```

Code Generation – IR

- Example: class hierarchy in Python

```
class ForLoop(Node):
  def __init__(self, iterator, begin, end, body):
    super().__init__()

    self.iterator = iterator
    self.begin = begin
    self.end = end
    self.body = body
```

Code Generation – IR

- Example: class hierarchy in Python

```
class ForLoop(Node):
  def __init__(self, iterator, begin, end, body):
    # ...

  def pp(self, printer, indent=""):
    it = printer.print(self.iterator)
    begin = printer.print(self.begin)
    end = printer.print(self.end)

    ret = f"{indent}for (auto {it} = {begin}; {it} != {end}; ++{it}) {'{'}\n"
    ret += f"{printer.print(self.body, indent + '  ')}\n"
    ret += f"{indent}{'}'}\n"

    return ret
```

Code Generation – IR

- Example: class hierarchy in Python

```
class ForLoop(Node):
  def __init__(self, iterator, begin, end, body):
    # ...

  def pp(self, printer, indent=""):
    # ...

  def apply_rec(self, transformation: Transformation):
    self.iterator = transformation.apply(self.iterator)
    self.begin = transformation.apply(self.begin)
    self.end = transformation.apply(self.end)

    self.body = [transformation.apply(stmt)
                 for stmt in self.body]

    return self
```

Code Generation – IR

- Loop order is different

```
# pragma omp parallel for collapse(2)
for (int j = 1; j < ny - 1; ++j) {
  for (int i = 1; i < nx - 1; ++i) {
    /* ... */
  }
}

#pragma acc parallel loop collapse(2)
for (int i = 1; i < nx - 1; ++i) {
  for (int j = 1; j < ny - 1; ++j) {
    /* ... */
  }
}
```

Code Generation – IR

- Transformation example: loop order inversion

```
class ReverseLoopOrder(Transformation):
  def __init__(self):
    super().__init__()

  def apply_internal(self, node):
    if isinstance(outer := node, ForLoop):
      if 1 == len(outer.body) and isinstance(inner := outer.body[0], ForLoop):
        inner.body, outer.body = [outer], inner.body
        return inner

      else:
        # handle loops not perfectly nested

    else:
      return node.apply_rec(self)
```

Code Generation – IR

- Users
  - Powerful tool …
  - … that needs a suitable front end
- Developers
  - More control than any other approach
  - Abstractions need to be designed carefully
  - High implementation effort
  - Optimizations previously difficult or impossible are now viable
- Implementation effort can be lowered by building on established tools such as LLVM, MLIR, etc.

Code Generation Front End – DSLs

- Users (usually) avoid programming directly in the code generator
- Custom front end programming languages are a viable remedy – so called domain-specific languages (DSLs)
  - Specific to one class of problems (= domain)
  - Concepts/ syntax familiar to potential users
  - High level of abstraction
- DSLs can be internal or external

Code Generation Front End – DSLs

- Internal DSLs
  - Extension or restriction of a host general purpose language (GPL)

- Example: pystencils (https://i10git.cs.fau.de/pycodegen/pystencils)
  - Code generation front end for the waLBerla multiphysics framework
  - Can be used as stand-alone from Python
  - Serves as a back end for lbmpy (https://i10git.cs.fau.de/pycodegen/lbmpy)

Code Generation Front End – DSLs

- Example: pystencils

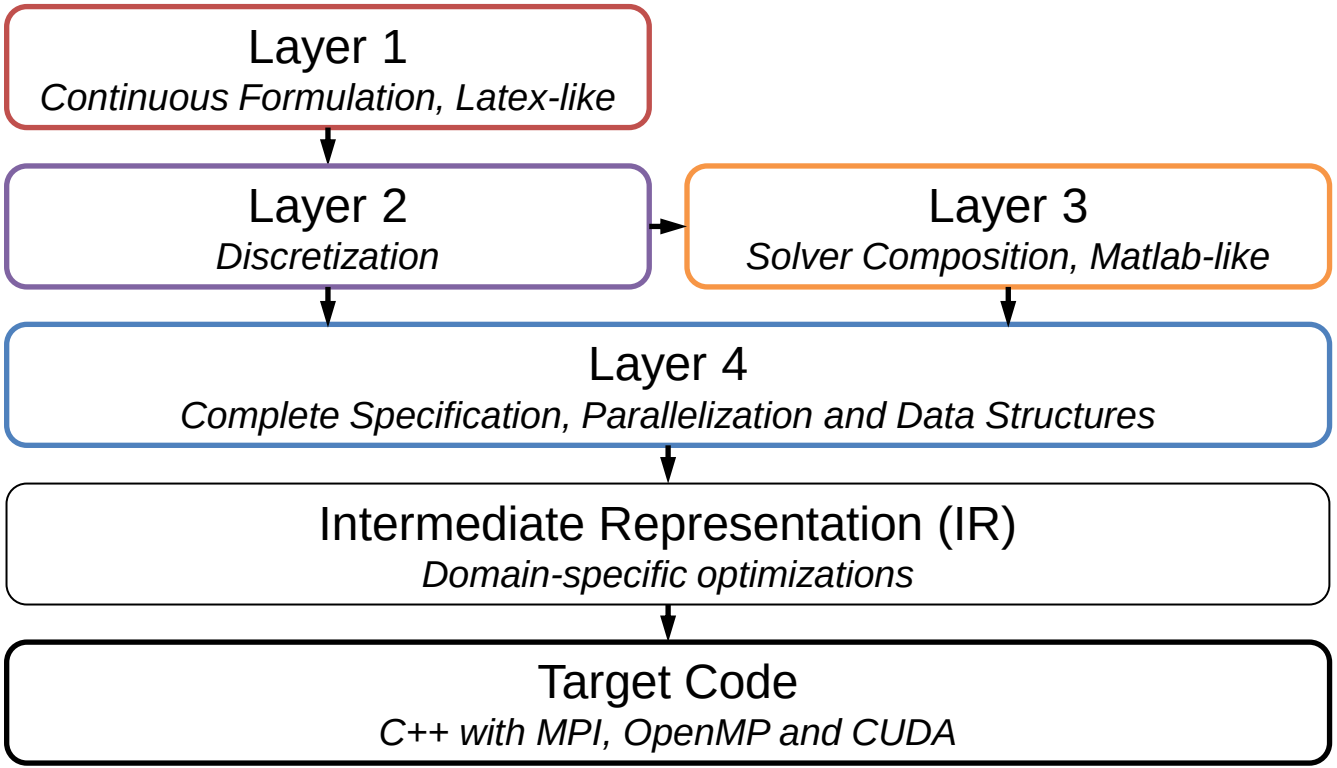
```
src, dst = ps.fields("u, u_new : [2D]")

stencil = ps.Assignment(dst[0, 0],
                        (src[1, 0] + src[-1, 0] + src[0, 1] + src[0, -1]) / 4)
kernel = ps.create_kernel(stencil).compile()

# or

@ps.kernel
def kernel_func():
  dst[0, 0] @= (src[1, 0] + src[-1, 0] + src[0, 1] + src[0, -1]) / 4

ps.generate_sweep(ctx, ‘Jacobi', kernel_func)
```

DSLs – Internal

- Users
  - Perceived as part of a familiar GPL
  - Potentially difficult to write code that matches the DSLs expectations
- Developers
  - Less effort to design language
  - Must be able to handle arbitrary code from the host language
  - Some optimizations may be blocked

DSLs – External

- Example: ExaStencils and its DSL ExaSlang for multigrid PDE solvers



DSLs – External

- Example: ExaStencils and ExaSlang (https://www.exastencils.fau.de/)

```
Ω = (0, 256) × (0, 256)

u ∈   Ω
u ∈ ∂ Ω = x + y

op = - Δ
eq: op * u = 0
```

```
Stencil Laplace {
  [ 0,  0] =>  4.0
  [-1,  0] => -1.0
  [ 1,  0] => -1.0
  [ 0, -1] => -1.0
  [ 0,  1] => -1.0
}
```

```
loop over u {
  solve locally with Jacobi {
    Laplace * u = 0.0
  }
}
```

```
loop over u {
  u[next] = ( Laplace:[-1, 0] * u@[-1, 0] + Laplace:[1, 0] * u@[1, 0]
            + Laplace:[0, -1] * u@[0, -1] + Laplace:[0, 1] * u@[0, 1]
            ) / Laplace:[0, 0]
}
```

DSLs – External

- Distinct programming language with its own grammar and syntax

- Users
  - High productivity using familiar concepts …
  - … after learning a whole new language
  - Very limited control

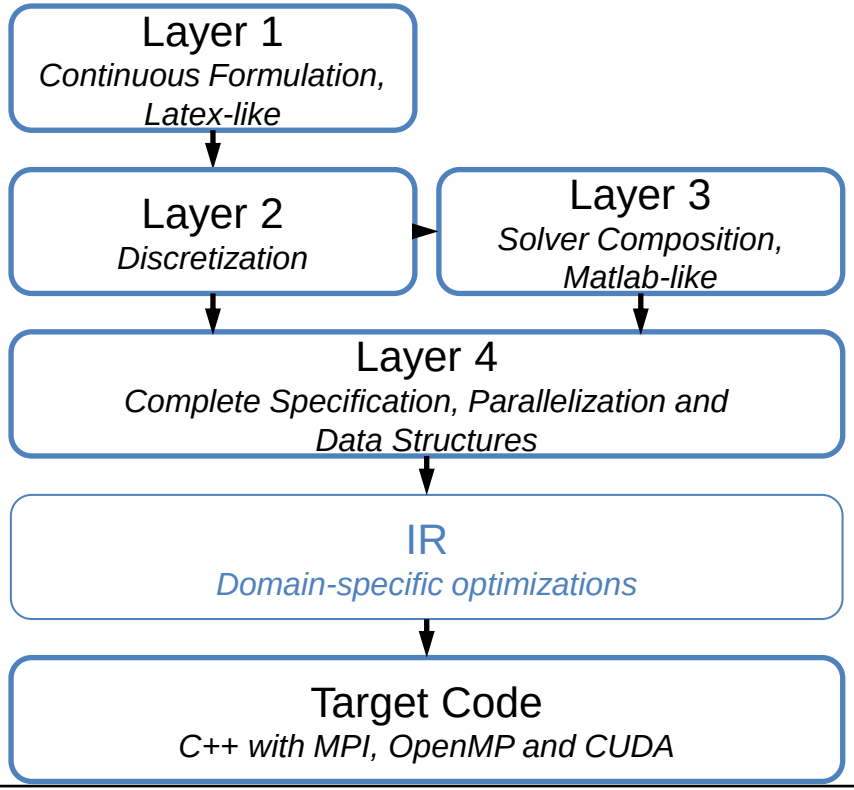
- Developers
  - Full control over every aspect
  - More effort required
    - language definition
    - lexer, parser, IR, etc.

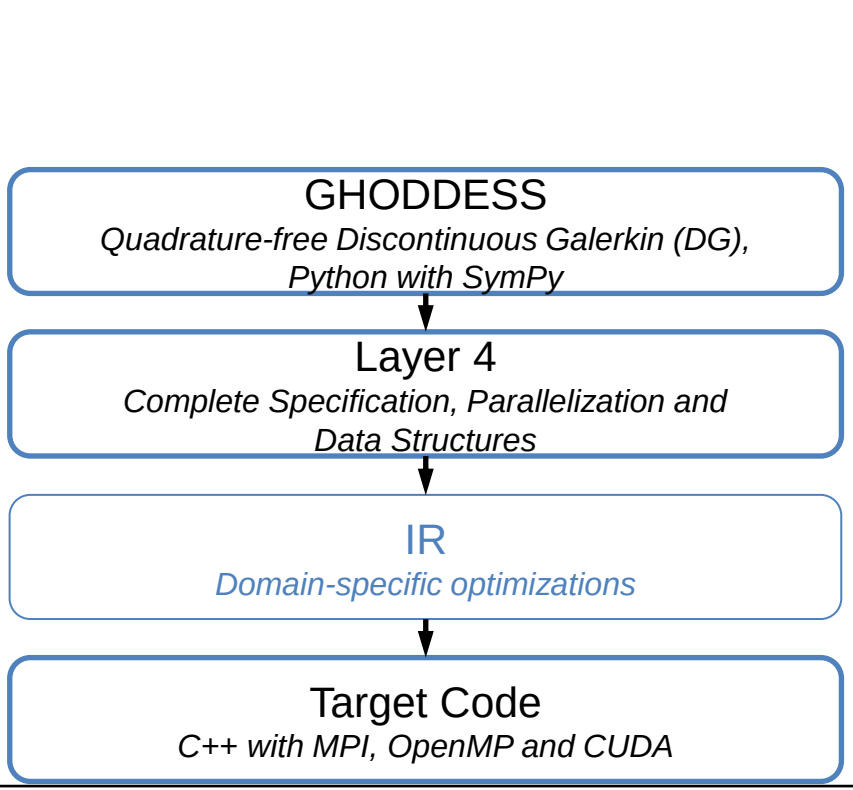
Code Generation – Back End

- At this point kernels can be pretty-printed from the IR
- Variants
  - Kernels are generated for a driver code/ an underlying framework
  - Kernels are generated to use a given runtime
  - Kernels are generated alongside data structures
- Data communication (MPI, host-device, …) can be generated as well

Approach – Composability

- Example: ExaStencils front end for higher order DG discretizations





Summary

Summary

Approach	Performance	Productivity	Portability
Specialized implementation	Tailored optimizations possible	Good for sufficiently small domains	Generally very low
Framework	General optimizations possible	Good even for multiple applications	Esp. performance portability limited
Framework + abstraction layer	Small deterioration possible	As above	High
Framework + generated kernels	Very specific optimizations possible	Extra step in workflow	High
String-based generation	Difficult to implement some optimizations	Issues with debugging likely	Medium
Internal DSL	Some optimizations might be blocked	High	High
External DSL	Very specific optimizations possible	Very high after initial learning effort	Very high

Challenges

- Finding the right level of abstraction (physics, math, computational operations, parallelization, …)
- Getting users/ developers on board
  - Address loss of control
  - Provide debugging support, and tool support in general
- Ensuring code maintainability
- Preparing for the next generation of hardware and programming models
- Upgrading legacy code / coupling with existing code

Thank you for your attention!

Questions?