Domain-Specific Languages to High Performance: Code Generation and Transformation in Python
Part 1: Introduction

Andreas Klöckner

Computer Science
University of Illinois at Urbana-Champaign

Outline

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code actually fast.

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code ~~actually fast.~~

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code ~~actually fast.~~
achieve the **best** performance possible on a given machine.

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code ~~actually fast.~~
achieve the **best** performance possible on a given machine.

- **NO:** I made my code 300,000x faster.
- **YES:** My code achieves 37% of the achievable floating point capability of my machine.

High Performance: What?

What is. . . High Performance Computing?

The *science* of making code ~~actually fast.~~
achieve the **best** performance possible on a given machine.

- **NO:** I made my code 300,000x faster.
- **YES:** My code achieves 37% of the achievable floating point capability of my machine.

Performance: Measure → Understand → Improve → Measure → Understand → Improve → $\cdots$

Setting

High-performance code is **challenging**:

- designed to push machines, models, and methods to the limits of their capabilities
- often repurposed → high demands on flexibility

Goals

Recipe: Split '**math work**' from '**performance work**'

- Build Mathematically-oriented mini-languages ('DSLs')
- Apply domain-specific optimizations and transformations
- Leverage tools to generate GPU/multi-core code from DSL
- Create glue that ties components together

Goals

Recipe: Split '**math work**' from '**performance work**'

- Build Mathematically-oriented mini-languages ('DSLs')
- Apply domain-specific optimizations and transformations
- Leverage tools to generate GPU/multi-core code from DSL
- Create glue that ties components together

Necessary consequence:
The computation itself is now *data* that we will manipulate programmatically.

- Introduction
  - IPython
  - Python
  - `numpy`
- Building languages
  - Syntax trees
  - Expression languages
  - Operations on expression trees
  - A first glimpse of code generation
- OpenCL as a vehicle for code generation
  - Execution model
  - OpenCL + Python
  - High-performance primitives
- Case studies
  - `numpy`: broadcasting
  - `numpy`: `einsum`
  - UFL
- Generating C
  - Using templating engines
  - Types and hybrid code
  - Structured code generation (ASTs)
- Code generation via Loopy
  - Loop polyhedra
  - Instructions and ordering
  - Loop transformation, and data layout
  - Generating instructions from DSLs

- Introduction
  - IPython
  - Python
  - `numpy`
- Building languages
  - Syntax trees
  - Expression languages
  - Operations on expression trees
  - A first glimpse of code generation
- OpenCL as a vehicle for code generation
  - Execution model
  - OpenCL + Python
  - High-performance primitives
- Case studies
  - `numpy`: broadcasting
  - `numpy`: `einsum`
  - UFL
- Generating C
  - Using templating engines
  - Types and hybrid code
  - Structured code generation (ASTs)
- Code generation via Loopy
  - Loop polyhedra
  - Instructions and ordering
  - Loop transformation, and data layout
  - Generating instructions from DSLs

Outline

1 Outline

2 Software Overview

Getting the software

Core packages:

- Python: `https://www.python.org`
- numpy: `https://www.numpy.org`
- pymbolic: `https://github.com/inducer/pymbolic`
- PyOpenCL: `https://github.com/pyopencl/pyopencl`
- loopy: `https://github.com/inducer/loopy`

All open-source under MIT/BSD licenses.

DEMO TIME