From low to high level languages and back

Timo Betcke
t.betcke@ucl.ac.uk
University College London

Joint with S. Kailasa, M. Scroggs and many other collaborators

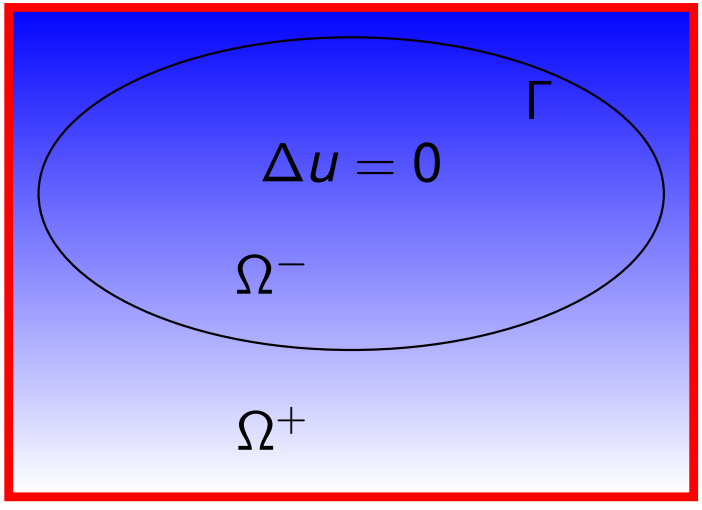
About me

- Studied Computational engineering at Hamburg University of Technology (1998 - 2002)
- DPhil at Oxford Numerical Analysis Group, (2002 - 2005, graduated 2006)
- Postdocs in Braunschweig, Manchester, Reading (2002 - 2009)
- EPSRC Career Acceleration Fellow, Reading (2009 - 20011)
- Since 20011 at UCL Department of Mathematics
- Since 2018 Professor of Computational Mathematics
- Since 2022 ARC Associate Director

What you don’t find in my CV

- I owned the original Nintendo Entertainment System and played Super Mario on a black and white TV
- First PC: 25MHz 386DX with 2MB Ram
- Favourite Youtube Channel: PBS Spacetime
- Long term Vim user, now overcome by the dark side and moved to VS Code
- I beat Elden Ring (my son still tells me I am bad at games since I never won Fall Guys)
- Best song of all times: Mad World by Tears for Fears

Some Maths to set the stage



$$\Delta u(\mathbf{x}) := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$$

Application areas: Heat diffusion, electrostatics, etc.

As boundary condition on Γ we use $u(\mathbf{x}) = f(\mathbf{x})$.

Green's fct.: The function $g(\mathbf{x}, \mathbf{y}) = \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}$ satisfies $\Delta_{\mathbf{x}} g(\mathbf{x}, \mathbf{y}) = 0$ for every $\mathbf{x} \neq \mathbf{y}$. This function is special. It is called the Green's fct. (or fundamental solution) of the PDE.

Representing the solution

We want to find a function ϕ defined on Γ such that

$$u(\mathbf{x}) = \int_\Gamma g(\mathbf{x}, \mathbf{y})\phi(\mathbf{y}), \quad \mathbf{x} \in \Omega$$

We plugin the known boundary data and arrive at the **boundary integral equation**

$$f(\mathbf{x}) = \int_\Gamma g(\mathbf{x}, \mathbf{y})\phi(\mathbf{y}), \quad \mathbf{x} \in \Gamma$$

We call V the single-layer boundary integral operator defined by

$$[V\phi](\mathbf{x}) = \int_\Gamma g(\mathbf{x}, \mathbf{y})\phi(\mathbf{y}), \quad \mathbf{x} \in \Gamma$$

How to discretise the integral equation?

We want to solve $V\phi = f$. Introduce a triangulation $\mathcal{T}$ of the domain Ω into triangles τ_j. Define basis fct.

$$\phi_j(\mathbf{x}) = \begin{cases} 1 & \mathbf{x} \in \tau_j \\ 0, & \text{otherwise} \end{cases}$$

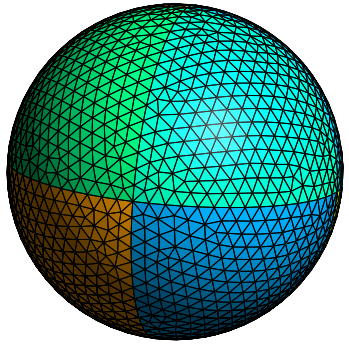
We approximate $\phi = \sum_{j=1}^{N} c_j \phi_j$. Multiplying with ϕ_i and integrating gives

$$\int_\Gamma \phi_i(\mathbf{x}) \left[V\phi\right](\mathbf{x}) ds(\mathbf{x}) = \int_\Gamma f(\mathbf{x})\phi_i(\mathbf{x}) ds(\mathbf{x}), i = 1, \ldots, N$$

Solve $\mathbf{Vc} = \mathbf{b}$

$$\mathbf{V}_{ij} = \int_{\tau_i}\int_{\tau_j} g(x,y) ds(y) ds(x)$$

$$\mathbf{b}_i = \int_\Gamma f(\mathbf{x})\phi_i(\mathbf{x}) ds(\mathbf{x})$$

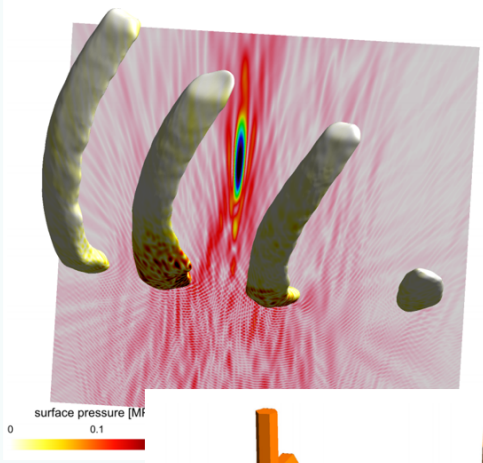


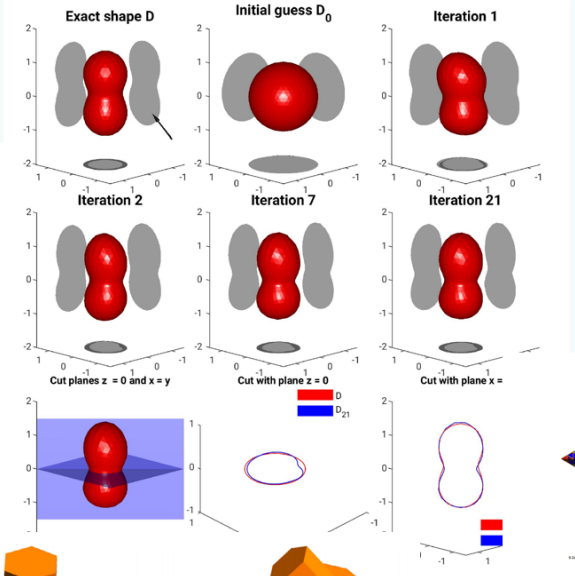
Sounds easy - or not?

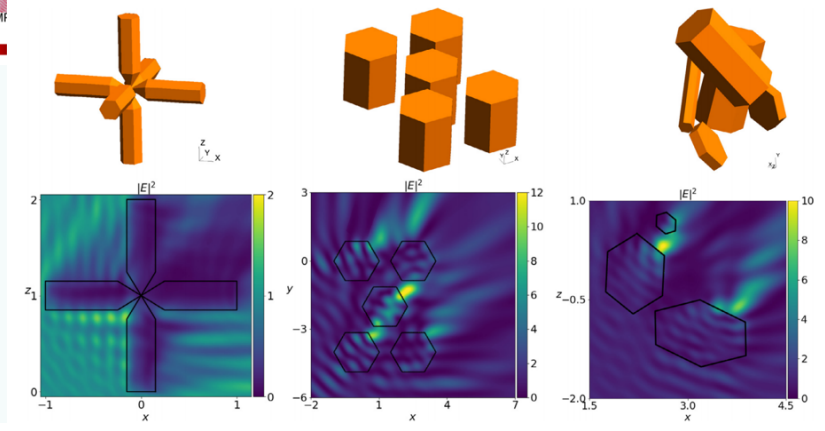
Two difficulties

- The Green's function g is singular for $\mathbf{x} = \mathbf{y}$. Each integral of a triangle with itself or and adjacent triangle requires special singularity treatment.
- The matrix $\mathbf{V}$ is dense with N^2 elements if there are N triangles. Huge assembly cost, and for large problems need linear scaling in complexity. (Can solve this by using Fast Multipole Methods for large problems, but complex and won't go into detail here)

The Bempp boundary element software







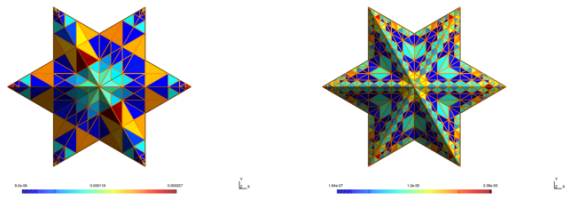


FIGURE 17. Example 3.3: Mesh $\mathcal{T}_\ell$ and distribution of the error estimator η_ℓ^2 for $\ell = 10$ (left) with 360 elements and $\ell = 15$ (right) with 1524 elements.

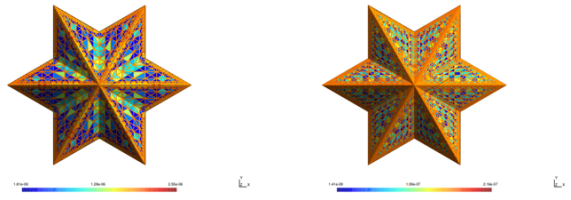


FIGURE 18. Example 3.3: Mesh $\mathcal{T}_\ell$ and distribution of the error estimator η_ℓ^2 for $\ell = 20$ (left) with 5754 elements and $\ell = 25$ (right) with 20304 elements.

Basic Objects

- Grid objects. Managing grids and geometric information.
- Grid functions. Discrete functions defined on grids.
- Boundary Operators. Definition of the various boundary operators
- Discrete boundary operators. Boundary operators after discretisation

The early C++ years of Bempp

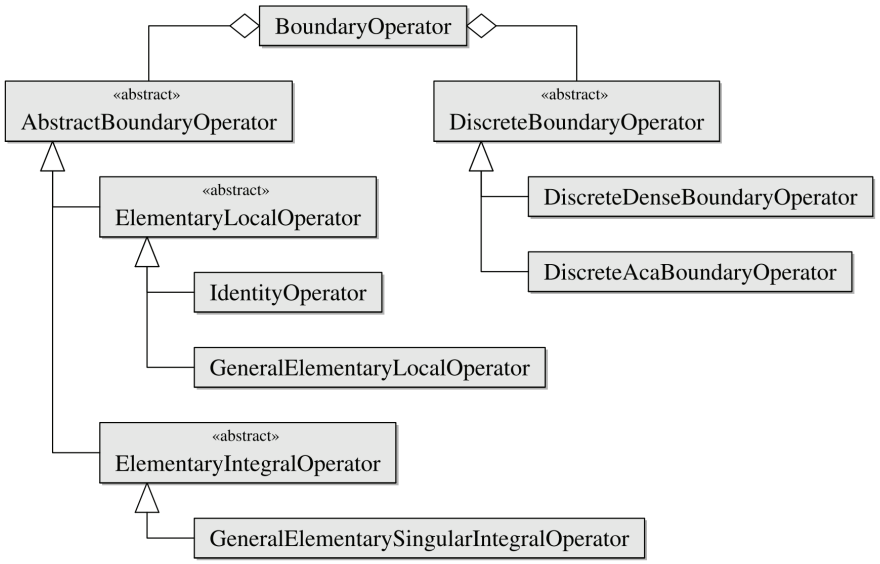


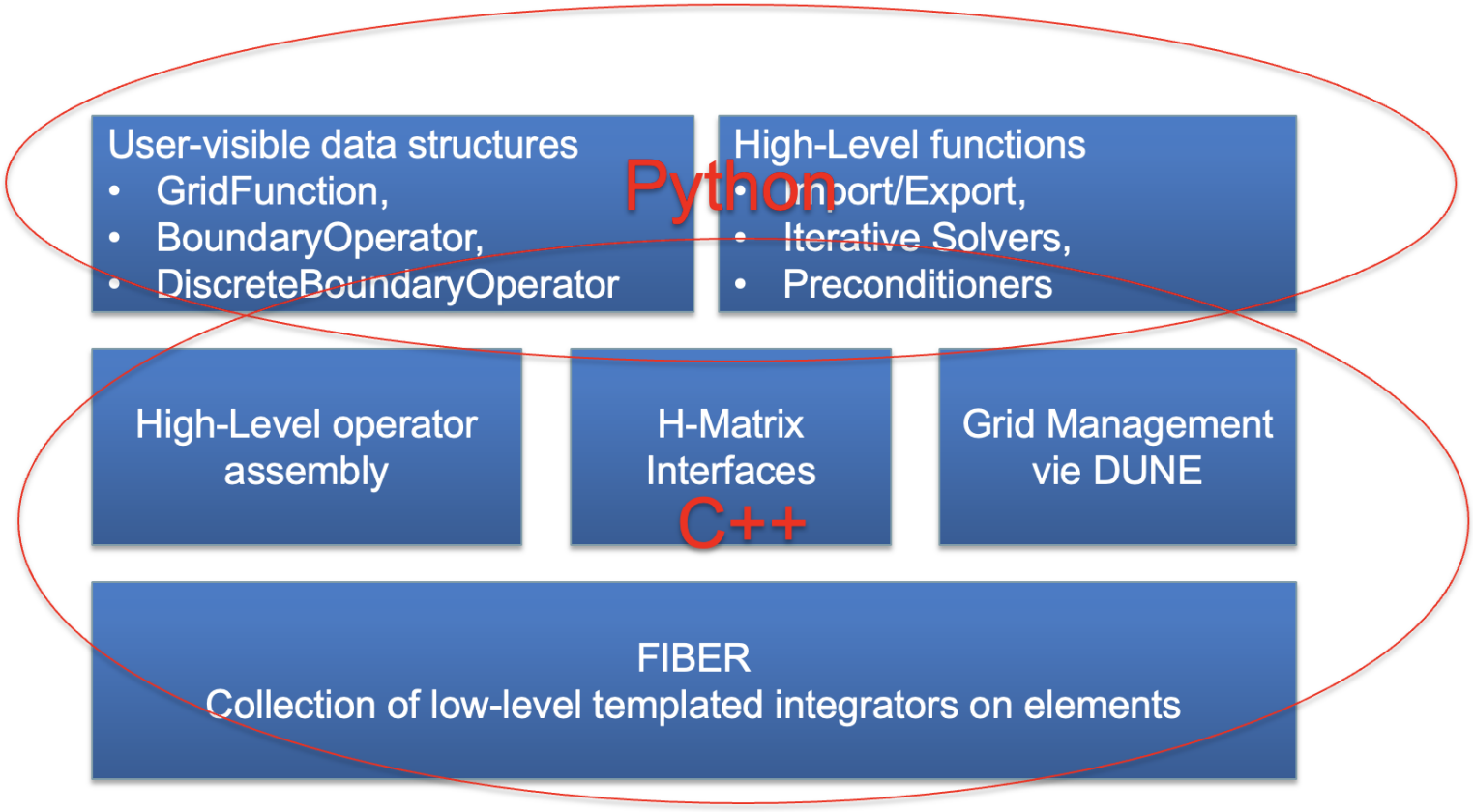
Fig. 3. Relationships between the main classes representing boundary operators.

- Complex object hierarchy
- Highly templated code
- Dependencies on complex packages (Boost, BLAS, Armadillo, Trilinos, DUNE, Intel TBB)

Python Interface

- Auto-generated with Swig
- Thin layer on C++ code
- Almost no logic in the Python layer, all handled by C++

The mixed C++/Python years



Only performance critical logic remained in C++. Python/C++ communication via Cython.

Why was I not happy with it?

- Fewer dependencies but still complicated build process and dependency hell.
- Difficult for PhD students to do low-level code work on the C++ side.
- Very tight object hierarchy on C++ code. Code not well written for modern vector registers (lots of arrays of structs instead of structs of arrays)
- Lots of attempts to shoehorn accelerator support into the code. But it was a mess.
- ...I am never happy with my codes.

> What I wanted: All code in Python, high-performance, SIMD and Accelerator Support, shallow object structure, no dependency hell.

Can we do it in Python?

```
def compute_operator_matrix():
    """Python pseudo-code for BEM evaluation."""
    A = numpy.zeros((n, n), dtype='float64')
    for test_element in test_elements:
        for trial_element in trial_elements:
            if is_adjacent(test_element, trial_element):
                local_interactions = \
                    singular_integration(test_element, trial_element)
            else:
                local_interaction = \
                    regular_integration(test_element, trial_element)
            sum_into_global_matrix(
                A, local_interaction)
    return A
```

- How should we parallelise?
- What about data structures and cache locality?
- Is the interpreter not too slow to handle these loops?

Acceleration technologies

UCL

Numba

- Grid topology computations
- Python callables
- Routines on grid functions, (L^2-norm, etc.)
- Fallback for operators

OpenCL

- All potential evaluations
- Mass matrices
- FMM Near-Field

Other technologies

- Scipy for spare matrix operations and iterative solvers
- Numpy dense solvers

Numba acceleration

```python
@_numba.njit
def _project_function_vectorized(
    function_data,
    grid_data,
    support_elements,
    local2global,
    local_multipliers,
    normal_multipliers,
    evaluate_on_element,
    shapeset_evaluate,
    points,
    weights,
    codomain_dimension,
    projections,
    function_parameters,
):
    """Project a Numba callable onto a grid."""

    npoints = points.shape[1]

    for index, element in enumerate(support_elements):

        element_vals = evaluate_on_element(
            element,
            shapeset_evaluate,
            points,
            grid_data,
            local_multipliers,
            normal_multipliers,
        )

        for local_fun_index in range(element_vals.shape[1]):
            projections[local2global[element, local_fun_index]] += (
                _np.sum(
                    _np.sum(
                        element_vals[:, local_fun_index, :]
                        * function_data[:, index * npoints : (1 + index) * npoints]
                        * weights,
                        axis=0,
                    )
                )
                * grid_data.integration_elements[element]
            )
```

- Numba just-in-time compiles Python functions to machine code (using LLVM).
- Executable code independent of the Python interpreter.
- Allows simple loop-based threading.
- Most Python features and many Numpy features supported.

Evaluating the Green's fct. in Numba

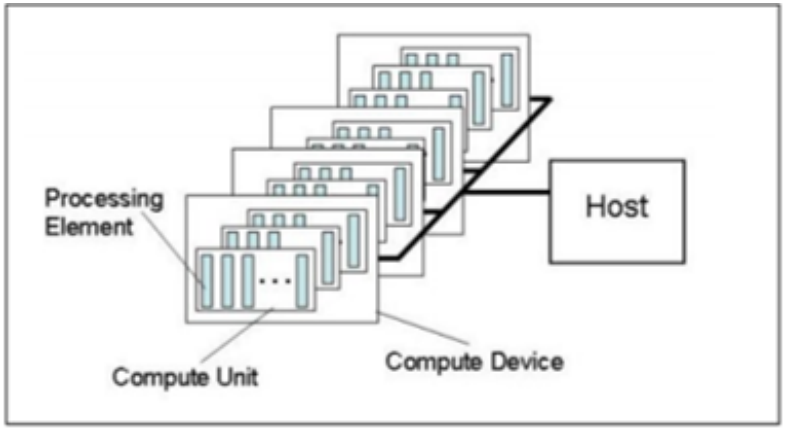
```python
@_numba.jit(
    nopython=True, parallel=False, error_model="numpy", fastmath=True, boundscheck=False
)
def laplace_single_layer_regular(
    test_point, trial_points, test_normal, trial_normals, kernel_parameters
):
    """Laplace single layer for regular kernels."""
    npoints = trial_points.shape[1]
    dtype = trial_points.dtype
    output = _np.zeros(npoints, dtype=dtype)
    m_inv_4pi = dtype.type(M_INV_4PI)
    for i in range(3):
        for j in range(npoints):
            output[j] += (trial_points[i, j] - test_point[i]) ** 2
    for j in range(npoints):
        output[j] = m_inv_4pi / _np.sqrt(output[j])
    return output
```

- Evaluate $g(x, y) = \frac{1}{4\pi|x-y|}$ for one x-value and many values for y.
- Loops written so that the LLVM autovectorizer can produce AVX code.
- Called on the element assembly for each test-quadrature point and evaluated for all trial quadrature points.
- Kernels implemented for all the standard Green's functions and their derivatives for Laplace and Helmholtz.

Global assembly in Numba

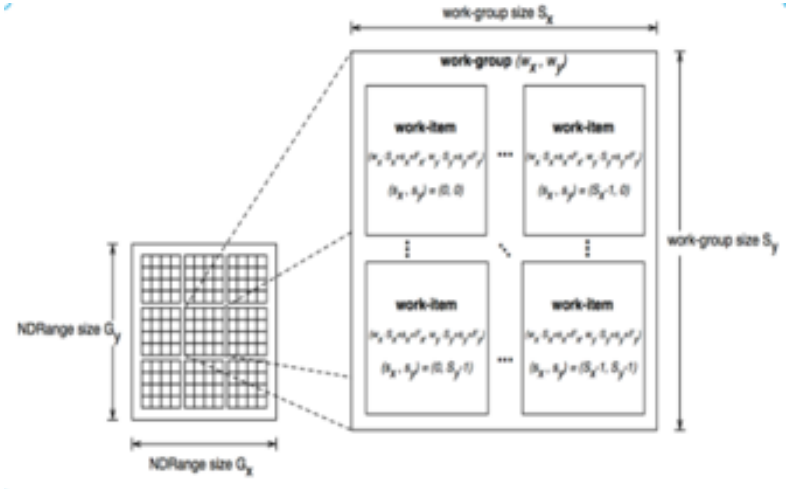
```
def global_assembly():
    """Global assembly of operator"""

    A = empty_array()

    trial_points = get_all_trial_points()
    for test_element in numba.prange(test_elements):
        local_result = zero_array()
        test_points = get_integration_points(test_element)
        for test_point in test_points:
            interactions = evaluate_kernel(test_point, trial_points)
        for trial_element in trial_elements:
            if is_adjacent(test_element, trial_element):
                continue
            sum_into_local_results(
                interactions,
                geometry_data,
                test_fun_values,
                trial_fun_values
            )
        sum_into_global_matrix(
            A, test_element, local_results,
            local2global_data
        )
```

Kernel based programming with OpenCL



- Heterogeneous compute model
- MIMD/SIMD/SIMT support
- C99 based kernel language
- Current version 2.2
- OpenCL drivers by Intel, AMD, Nvidia plus open-source

- Each device consists of compute units (e.g. CPU cores) and processing elements (e.g. SIMD lanes)
- Work items arranged into work groups
- Memory synchronization only inside a workgroup
- Excellent Python support through PyOpenCL by Andreas Kloeckner



Compute kernels for regular integrals

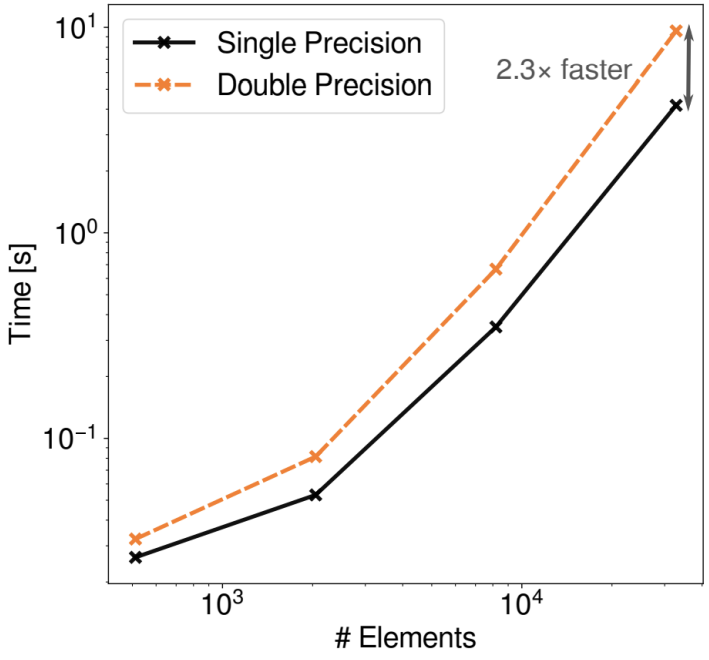
```
__kernel void kernel_function(...){

  size_t testIndex = testIndices[gid[0]];
  size_t trialIndex = trialIndices[gid[1]];

  // Compute geometric data

  // Evaluate all integral contributions for test/trial pair

  // Sum into global matrix if test/trial pair not adjacent

}
```

- Compute kernels are JIT-compiled during Python execution by the OpenCL driver
- On CPUs one test triangle coalesced with 4/8/16 trial triangles to efficiently target SIMD lanes.
- A typical computation launches millions of kernels

Performance

	single	double
PoCL	0.05s	0.08s
Numba	0.25s	0.25s
GPU	0.11s	3.00s

Boundary op. assembly times (2048 elem)



Single vs double precision assembly on CPU

Am I finally happy?

- Python + Numba + OpenCL powerful environment for multitude of tasks
- Can make use of modern CPU/GPU features
- New code much faster than our old C++ code

But...

- Need to deal with OpenCL drivers from different vendors
- Numba JIT not sufficiently performant on CPUs to replace OpenCL
- Thousands of lines of low-level OpenCL C99 kernel code
- Latency issues for pure Python operations
- Numba not a good model for very complex algorithms

Numba is great but not perfect

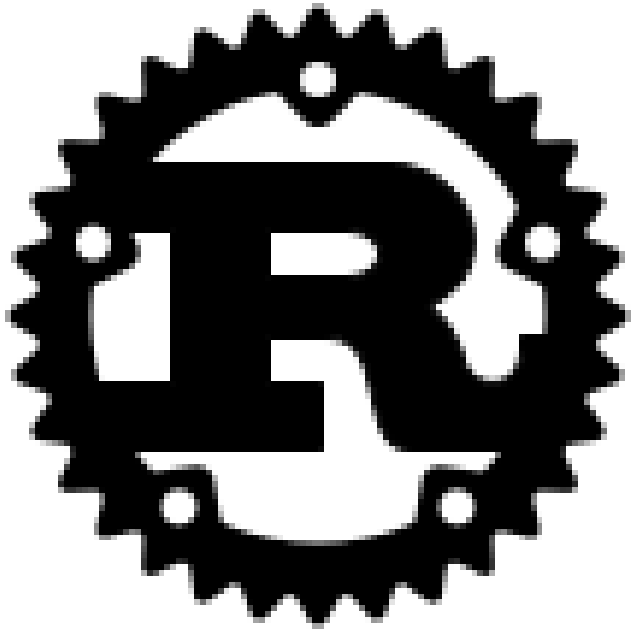
UCL

- Latency issues for small Numba functions called from Python interpreter
- Numba does not support small stack-based arrays. Everything must be heap allocated
- Numba cannot handled inlining well (e.g. if functions are separately JIT compiled)
- The more efficient a Numba program the less pythonic it looks
- Nested multithreading is a nightmare and hard to control

What now?

- Python with C++ is complex with cumbersome build systems and terrible dependency management
- Python with JIT compilation works for certain tasks but not algorithms with complex structures. Optimising the code can become complicated.

Along came Rust



- Modern low-level language
- Excellent tooling and dependency management
- Straight-forward Python integration
- Growing HPC community

Let's switch to some practical demonstrations.