MAX-PLANCK-INSTITUT
FÜR PLASMAPHYSIK

Struphy

A Python Framework for HPC Plasma Simulations

Max Lindqvist
Max Planck Institute for Plasma Physics, Germany

EUROfusion

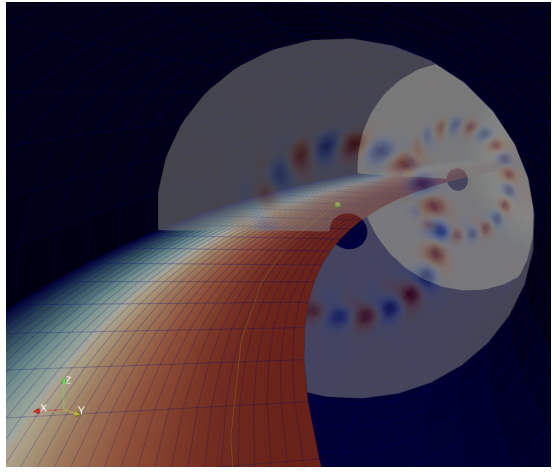
This work has been carried out within the framework of the EUROfusion Consortium, funded by the European Union via the Euratom Research and Training Programme (Grant Agreement No 101052200 — EUROfusion). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them.

Struphy: Structure-Preserving Hybrid Codes

STRUPHY

Plasma Physics Models in Python

- Solving PDEs using **Python**
- Open-source
- For teaching and HPC applications
- From prototyping to production **in one framework**



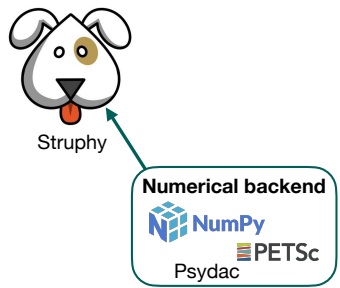
Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**

Struphy

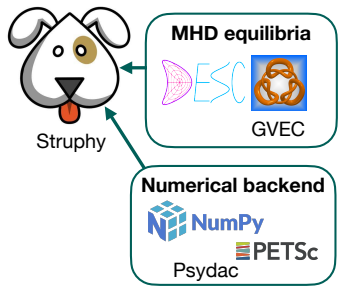
Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**



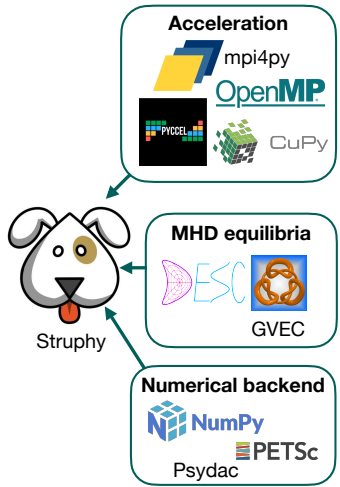
Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**



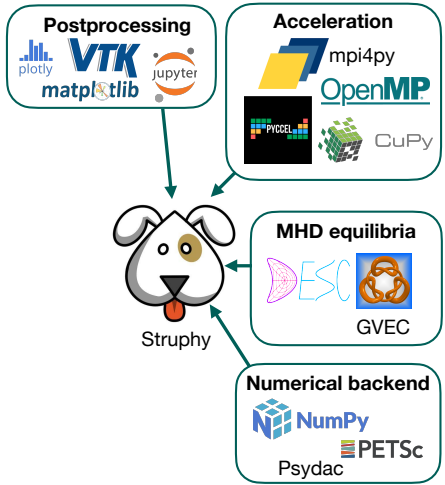
Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**



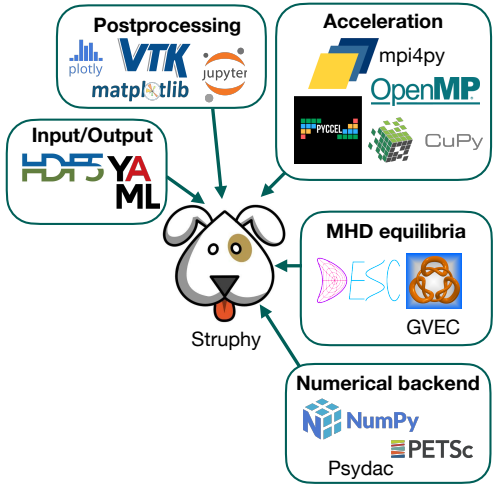
Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**



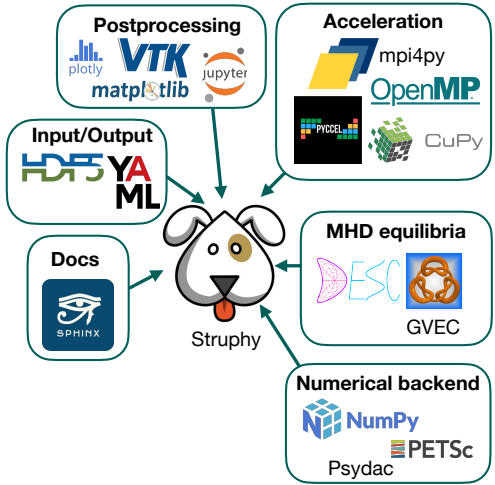
Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**



Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**



Why Python?

1. **Clarity, usability and flexibility first**.
2. Fast development
3. Be easily accessible
4. Integration with `Python` ecosystem
   - `psydac`: High-order spline library
   - `pyccel`: Transpile in C/Fortran
5. Provide a framework for quickly adding **new physics models**

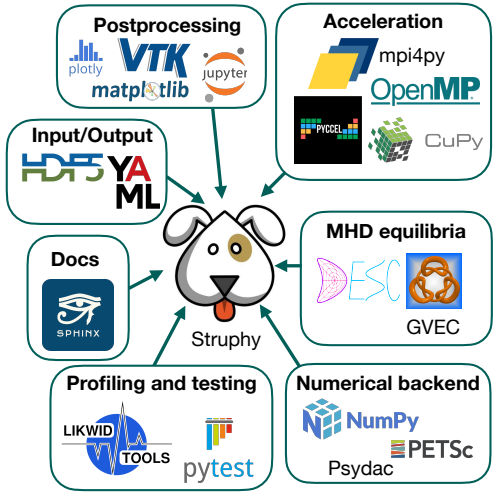


Table of Contents

Table of Contents

Progress in 2025

Progress in 2025

Technical

- Closed ∼100 issues, merged ∼100 PRs into `devel`, and published 9 releases to PyPI
- Migrated project from GitLab to GitHub
- Performed major API refactoring (shifting from CLI toward API usage where possible)
- Enforced consistent code formatting using `ruff`
- Published interactive Jupyter notebook tutorials:
  `https://mybinder.org/v2/gh/struphy-hub/struphy-tutorials/main`
- Compatibility with `numpy >= 2.0`, `mpi4py >= 4.0`, and `pyccel >= 2.0`, ...
- Significant speedup of CI pipelines

Progress in 2025

Technical

- Closed ∼100 issues, merged ∼100 PRs into `devel`, and published 9 releases to PyPI
- Migrated project from GitLab to GitHub
- Performed major API refactoring (shifting from CLI toward API usage where possible)
- Enforced consistent code formatting using `ruff`
- Published interactive Jupyter notebook tutorials:
  `https://mybinder.org/v2/gh/struphy-hub/struphy-tutorials/main`
- Compatibility with `numpy >= 2.0`, `mpi4py >= 4.0`, and `pyccel >= 2.0`, ...
- Significant speedup of CI pipelines

Scientific

- Added new Smooth Particle Hydrodynamics (SPH) and Hasegawa–Wakatani models

Progress in 2025

Technical

- Closed ∼100 issues, merged ∼100 PRs into `devel`, and published 9 releases to PyPI
- Migrated project from GitLab to GitHub
- Performed major API refactoring (shifting from CLI toward API usage where possible)
- Enforced consistent code formatting using `ruff`
- Published interactive Jupyter notebook tutorials:
  `https://mybinder.org/v2/gh/struphy-hub/struphy-tutorials/main`
- Compatibility with `numpy >= 2.0`, `mpi4py >= 4.0`, and `pyccel >= 2.0`, ...
- Significant speedup of CI pipelines

Scientific

- Added new Smooth Particle Hydrodynamics (SPH) and Hasegawa–Wakatani models

Performance

- Improved CPU–GPU portability via `numpy` and `cupy`
- Introduced Pyccelized kernels with OpenMP acceleration

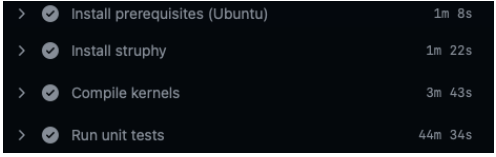
A few comments on CI pipelines

- Lots of work on restructuring CI pipelines, both on Gitlab and Github
- Multiple OSs, Python versions, Pyccelizing to C/Fortran, with/without MPI/OpenMP, ...
- Unit tests, model tests, tutorial tests, code formatting, ...
- **Results**: Many errors/bugs were caught quickly!

A few comments on CI pipelines

- Lots of work on restructuring CI pipelines, both on Gitlab and Github
- Multiple OSs, Python versions, Pyccelizing to C/Fortran, with/without MPI/OpenMP, ...
- Unit tests, model tests, tutorial tests, code formatting, ...
- **Results**: Many errors/bugs were caught quickly!

But also:

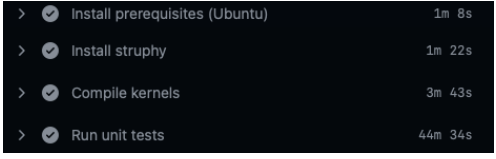


Running 20+ of these jobs/push is unneccesary

A few comments on CI pipelines

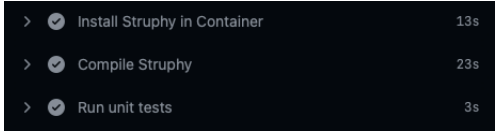
- Lots of work on restructuring CI pipelines, both on Gitlab and Github
- Multiple OSs, Python versions, Pyccelizing to C/Fortran, with/without MPI/OpenMP, ...
- Unit tests, model tests, tutorial tests, code formatting, ...
- **Results**: Many errors/bugs were caught quickly!

But also:



Running 20+ of these jobs/push is unneccesary

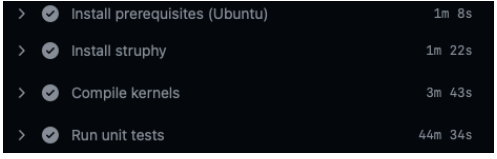
After streamlining:



A few comments on CI pipelines

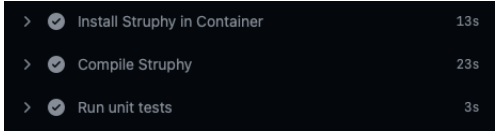
- Lots of work on restructuring CI pipelines, both on Gitlab and Github
- Multiple OSs, Python versions, Pyccelizing to C/Fortran, with/without MPI/OpenMP, ...
- Unit tests, model tests, tutorial tests, code formatting, ...
- **Results**: Many errors/bugs were caught quickly!

But also:



Running 20+ of these jobs/push is unneccesary

After streamlining:



- Build containers with dependencies preinstalled, and kernels compiled
- Only run unit tests which touch changed code using `testmon`
- Move the huge testing matrix to scheduled pipelines

Table of Contents

`Struphy` Models

- Objects combine a variety of plasma "models" for different physics scenarios:
- Particles, FEEC fields, operators, solvers, mappings, equilibria, code interfaces

Struphy Models

- Objects combine a variety of plasma "models" for different physics scenarios:
- Particles, FEEC fields, operators, solvers, mappings, equilibria, code interfaces

Toy models:

DeterministicParticleDiffusion,
GuidingCenter,
Maxwell,
Poisson,
PressureLessSPH,
RandomParticleDiffusion,
ShearAlfven,
TwoFluidQuasiNeutralToy,
VariationalBarotropicFluid,
VariationalCompressibleFluid,
VariationalPressurelessFluid,
Vlasov

Kinetic models:

DriftKineticElectrostaticAdiabatic,
LinearVlasovAmpereOneSpecies,
LinearVlasovMaxwellOneSpecies,
VlasovAmpereOneSpecies,
VlasovMaxwellOneSpecies

Fluid models:

ColdPlasma,
EulerSPH,
HasegawaWakatani,
LinearExtendedMHDuniform,
LinearMHD,
ViscoResistiveDeltafMHD,
ViscoResistiveDeltafMHD_with_q,
ViscoResistiveLinearMHD,
ViscoResistiveLinearMHD_with_q,
ViscoResistiveMHD,
ViscoResistiveMHD_with_p,
ViscoResistiveMHD_with_q,
ViscousFluid

Hybrid models:

ColdPlasmaVlasov,
LinearMHDDriftkineticCC,
LinearMHDVlasovCC,
LinearMHDVlasovPC

Vlasov-Maxwell Model for One Species

Description: Solves Vlasov-Maxwell equations for a single particle species (electrons/ions).
Normalized Equations:

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla f + \frac{1}{\varepsilon}(\mathbf{E} + \mathbf{v} \times (\mathbf{B} + \mathbf{B}_0)) \cdot \frac{\partial f}{\partial \mathbf{v}} = 0,$$
$$-\frac{\partial \mathbf{E}}{\partial t} + \nabla \times \mathbf{B} = \frac{\alpha^2}{\varepsilon} \int \mathbf{v} f \, d^3\mathbf{v},$$
$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \times \mathbf{E} = 0.$$

Model Components:

- **EM-Fields:** electric field $\mathbf{E}$, magnetic field $\mathbf{B}$, potential ϕ
- **Kinetic ions:** particles in 6D phase space
- **Propagators:** `Maxwell`, `PushEta`, `PushVxB`, Vlasov-Ampere coupling

Initial Condition: Weak Poisson solve ensures Gauss's law

Refactoring of `Struphy`

Goal: Align the console interface and Python API to reduce user confusion.

Key Changes:

- Shift towards API usage; remove certain console commands (e.g., `struphy run`)
- Introduce an auto-generated `.py` parameter file for main user interaction
  - File imports necessary classes to set up models and simulations
- IDE support (e.g., VSCode) allows inspecting model options via hover/click
- Replace `dicts` with `Classes` to make code more Pythonic and easier to document

Outcome: More consistent, maintainable, and user-friendly API

What a `StruphyModel` looks like

1. Inherit abstract base class

```
class VlasovMaxwellOneSpecies(StruphyModel):
    r"""Vlasov-Maxwell equations for one species.
```

What a `StruphyModel` looks like

1. Inherit abstract base class
2. Define species (i.e variables) and corresponding data structures

```
class EMFields(FieldSpecies):
      def __init__(self):
          self.e_field = FEECVariable(space="Hcurl")
          self.phi = FEECVariable(space="H1")
          self.init_variables()

class KineticIons(ParticleSpecies):
    def __init__(self):
        self.var = PICVariable(space="Particles6D")
        self.init_variables()
```

What a `StruphyModel` looks like

1. Inherit abstract base class
2. Define species (i.e variables) and corresponding data structures
3. Add **propagators** (push variables $t \to t + \Delta t$)

```
class Propagators:
    def __init__(self, with_B0: bool = True):
        self.push_eta = propagators_markers.PushEta()
        if with_B0:
            self.push_vxb = propagators_markers.PushVxB()
        self.coupling_va = propagators_coupling.VlasovAmpere()
```

Example propagator: PushEta will be shown later

What a `StruphyModel` looks like

1. Inherit abstract base class
2. Define species (i.e variables) and corresponding data structures
3. Add **propagators** (push variables $t \to t + \Delta t$)
4. Initialize a light-weight instance of model

```python
def __init__(self, with_B0: bool = True):
    # 1. instantiate all species
    self.em_fields = self.EMFields()
    self.kinetic_ions = self.KineticIons()

    # 2. instantiate all propagators
    self.propagators = self.Propagators(with_B0=with_B0)

    # 3. assign variables to propagators
    self.propagators.push_eta.variables.var = self.kinetic_ions.var
    if with_B0:
        self.propagators.push_vxb.variables.ions = self.kinetic_ions.var
    self.propagators.coupling_va.variables.e = self.em_fields.e_field
    self.propagators.coupling_va.variables.ions = self.kinetic_ions.var
```

What a `StruphyModel` looks like

1. Inherit abstract base class
2. Define species (i.e variables) and corresponding data structures
3. Add **propagators** (push variables $t \to t + \Delta t$)
4. Initialize a light-weight instance of model
5. When starting the simulation, allocate all arrays

```
model.setup_domain_and_equil(domain, equil)     # domain and fluid background
model.allocate_feec(grid, derham_opts)          # allocate feec
model.setup_equation_params(units=model.units) # equation parameters
model.allocate_variables()                      # allocate model variables
model.allocate_helpers()                        # allocate helpers functions
model.allocate_propagators()                    # allocate propagators
model.compute_plasma_params()                   # compute plasma parameters
```

Old vs. New Model Structure

Old Approach (dict-based):

```
class VlasovMaxwellOneSpecies(StruphyModel):
    @staticmethod
    def species():
        dct = {"em_fields": {}, "fluid": {}, "kinetic": {}}
        dct["em_fields"]["e_field"] = "Hcurl"
        dct["em_fields"]["b_field"] = "Hdiv"
        dct["kinetic"]["species1"] = "Particles6D"
        return dct
```

New Approach (class-based):

```
class VlasovMaxwellOneSpecies(StruphyModel):
    class EMFields(FieldSpecies):
        def __init__(self):
            self.e_field = FEECVariable(space="Hcurl")
            self.b_field = FEECVariable(space="Hdiv")
            self.phi = FEECVariable(space="H1")
            self.init_variables()
    class KineticIons(ParticleSpecies):
        def __init__(self):
            self.var = PICVariable(space="Particles6D")
            self.init_variables()
```

Propagators - the building blocks of models

- Advance a model's variables by one time step
- Combine to create new models with arbitrary splitting schemes

Particle propagators:

`PushDeterministicDiffusion`,
`PushEta`,
`PushEtaPC`,
`PushGuidingCenterBxEstar`,
`PushGuidingCenterParallel`,
`PushRandomDiffusion`,
`PushVinEfield`,
`PushVinSPHpressure`,
`PushVinViscousPotential2D`,
`PushVinViscousPotential3D`,
`PushVxB`

Field propagators:

`AdiabaticPhi`, `CurrentCoupling5DDensity`,
`CurrentCoupling6DDensity`,
`FaradayExtended`, `Hall`, `HasegawaWakatani`,
`ImplicitDiffusion`, `JxBCold`, `Magnetosonic`, `MagnetosonicUniform`, `Maxwell`, `OhmCold`, `Poisson`, `ShearAlfven`, `ShearAlfvenB1`,
`ShearAlfvenCurrentCoupling5D`,
`TimeDependentSource`,
`TwoFluidQuasiNeutralFull`,
`VariationalDensityEvolve`,
`VariationalEntropyEvolve`,
`VariationalMagFieldEvolve`,
`VariationalMomentumAdvection`,
`VariationalPBEvolve`, `VariationalQBEvolve`,
`VariationalResistivity`,
`VariationalViscosity`

Particle-field coupling propagators:

`CurrentCoupling5DCurlb`,
`CurrentCoupling5DGradB`,
`CurrentCoupling6DCurrent`,
`EfieldWeights`,
`PressureCoupling6D`,
`VlasovAmpere`

Table of Contents

Profiling of `Struphy`

Profiling of `Struphy`

- `cProfile`: Built-in Python profiler for function-level timings

Profiling of `Struphy`

- `cProfile`: Built-in Python profiler for function-level timings
- **Other Python ecosystem tools**: `line_profiler`, `memory_profiler`, ...

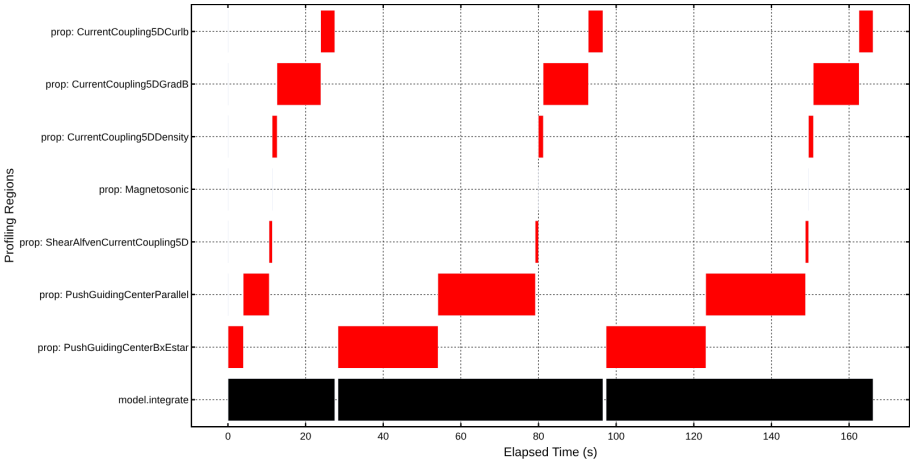
Profiling of `Struphy`

- `cProfile`: Built-in Python profiler for function-level timings
- **Other Python ecosystem tools**: `line_profiler`, `memory_profiler`, ...
- `LIKWID`: performance monitoring and benchmarking tool for multi-core processors
- `pylikwid`: Python wrapper for LIKWID, access marker API directly in Python

Profiling of `Struphy`

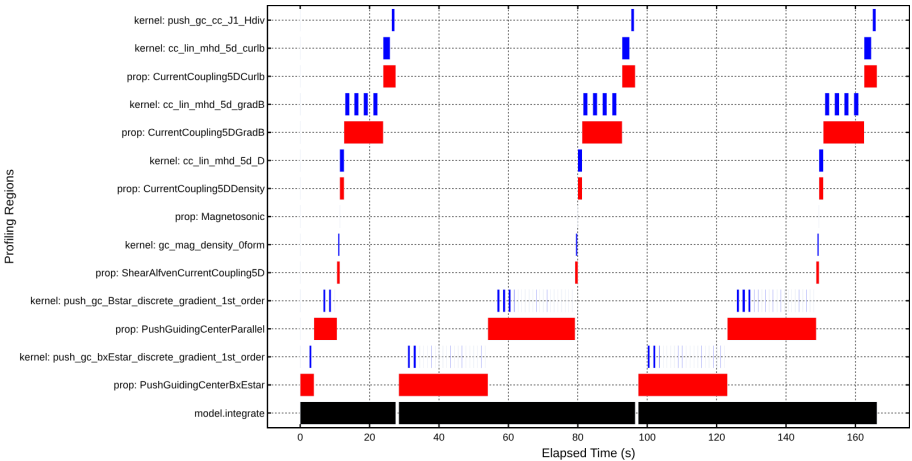
- `cProfile`: Built-in Python profiler for function-level timings
- **Other Python ecosystem tools**: `line_profiler`, `memory_profiler`, ...
- `LIKWID`: performance monitoring and benchmarking tool for multi-core processors
- `pylikwid`: Python wrapper for LIKWID, access marker API directly in Python
- **Custom region profiler (`scope-profiler` on PyPI)**:
  - Allows profiling of specific regions of interest in the code
  - Integrates with `LIKWID` to access hardware performance counters
  - Generates time traces
  - **Auto-profiles all propagators**

Time trace example



Red: Propagators

Time trace example



Red: Propagators, Blue: Pyccelized kernels

Table of Contents

Options for GPU porting of Python code

Goal: GPU porting with minimal changes to the code
A far from exhaustive list...

- `JAX` - AD + XLA-accelerated numerical computing
- `Pytorch` - Deep learning library
- `TensorFlow` - Machine learning library with GPU support
- `Numba` - JIT compilation for GPU
- `PyCUDA` - Direct CUDA kernel access
- `PyKokkos` - Python interface to Kokkos
- `Taichi Lang` - JIT compilation for GPU
- `CuPy` - NumPy-like GPU array computations
- Interface Fortran/C with `OpenMP` target directives using `Pyccel`
- ... - and more

GPU porting - CuPy example

```python
import numpy as np
import cupy as cp

# Create NumPy arrays
a_np = np.random.rand(3, 3)
b_np = np.random.rand(3, 3)

# Transfer NumPy -> CuPy
a_cp = cp.asarray(a_np)
b_cp = cp.asarray(b_np)

# Matrix multiplication on GPU
c_cp = cp.matmul(a_cp, b_cp)

# Transfer CuPy -> NumPy
c_np = cp.asnumpy(c_cp)
```

GPU porting - CuPy example

```python
import numpy as np
import cupy as cp

# Create NumPy arrays
a_np = np.random.rand(3, 3)
b_np = np.random.rand(3, 3)

# Transfer NumPy -> CuPy
a_cp = cp.asarray(a_np)
b_cp = cp.asarray(b_np)

# Matrix multiplication on GPU
c_cp = cp.matmul(a_cp, b_cp)

# Transfer CuPy -> NumPy
c_np = cp.asnumpy(c_cp)
```

Next: Switch between CuPy and NumPy using a wrapper.

CuPy/Numpy backend

- For the most part, CuPy can be used as a drop-in replacement for NumPy
- Therefore, replace `import numpy` with `import cupy`
- This is handled with an `ArrayBackend` class from `cunumpy` (lightweight wrapper)

Simple example, `export ARRAY_BACKEND=cupy`:

```
import cunumpy as xp
arr = xp.array([1,2])
```

Using this method, almost all NumPy/CuPy operations are interchangable.

CuPy/Numpy backend

- For the most part, CuPy can be used as a drop-in replacement for NumPy
- Therefore, replace `import numpy` with `import cupy`
- This is handled with an `ArrayBackend` class from `cunumpy` (lightweight wrapper)

Simple example, `export ARRAY_BACKEND=cupy`:

```
import cunumpy as xp
arr = xp.array([1,2])
```

Using this method, almost all NumPy/CuPy operations are interchangable.
A few exceptions:

- Universal Functions in CuPy only work with CuPy array or scalar (not Python lists)
- NumPy's reduction functions return scalars, CuPy's return zero-dimensional arrays
- A few more similar exceptions, all are very easy to accomodate

CuPy - Pyccel interface

- **Problem**: Cupy arrays are not supported by pyccel
- **Solution**: Wrap kernels in `PyccelKernel` class

CuPy - Pyccel interface

- **Problem**: Cupy arrays are not supported by pyccel
- **Solution**: Wrap kernels in `PyccelKernel` class

```python
class PyccelKernel:
    def __init__(self, kernel: Callable[..., Any], use_cupy: bool = False) -> None:
        self._kernel = kernel
        self._use_cupy = use_cupy

    def __call__(self, *args: Any, **kwargs: Any) -> Any:
        if not self._use_cupy:
            return self._kernel(*args, **kwargs)

        # Convert all inputs to NumPy before calling kernel
        np_args = tuple(to_numpy(a) for a in args)
        np_kwargs = {k: to_numpy(v) for k, v in kwargs.items()}

        # Call kernel
        result = self._kernel(*np_args, **np_kwargs)

        # Convert result to CuPy
        if isinstance(result, tuple):
            return tuple(to_cupy(r) for r in result)
        return to_cupy(result)
```

CuPy - Pyccel interface

- **Problem**: Cupy arrays are not supported by pyccel
- **Solution**: Wrap kernels in `PyccelKernel` class

Example:

```
import cunumpy as xp
from struphy.linalg_kernels import matmul
from struphy.utils.pyccel import Pyccelkernel

# Create matrices
A = xp.random.random((10, 10))
B = xp.random.random((10, 10))

kernel = PyccelKernel(matmul, use_cupy=True)
result = kernel(A, B)
```

CuPy - Pyccel interface

- **Problem**: Cupy arrays are not supported by pyccel
- **Solution**: Wrap kernels in `PyccelKernel` class

Example:

```
import cunumpy as xp
from struphy.linalg_kernels import matmul
from struphy.utils.pyccel import Pyccelkernel

# Create matrices
A = xp.random.random((10, 10))
B = xp.random.random((10, 10))

kernel = PyccelKernel(matmul, use_cupy=True)
result = kernel(A, B)
```

Next: Run the kernel itself on GPU

GPU porting Pyccelized kernels with OpenMP

```python
def matmul_gpu(A: "float[:,:]", B: "float[:,:]", C: "float[:,:]"):
    N: int = shape(A)[0]
    s: float = 0.0
    # omp target teams distribute parallel for collapse(2)
    for i in range(N):
        for j in range(N):
            s = 0.0
            for k in range(N):
                s += A[i, k] * B[k, j]
            C[i, j] = s
```

GPU porting Pyccelized kernels with OpenMP

```python
def matmul_gpu(A: "float[:,:]", B: "float[:,:]", C: "float[:,:]"):
    N: int = shape(A)[0]
    s: float = 0.0
    # omp target teams distribute parallel for collapse(2)
    for i in range(N):
        for j in range(N):
            s = 0.0
            for k in range(N):
                s += A[i, k] * B[k, j]
            C[i, j] = s
```

Custom compiler flags can be specified in a .json file:

```
pyccel matmult.py --language fortran --openmp --compiler-config nvidia.json
```

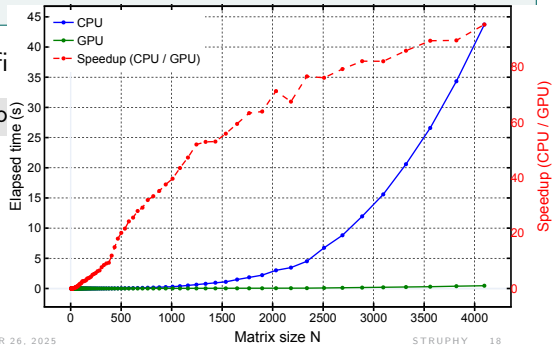
GPU porting Pyccelized kernels with OpenMP

```python
def matmul_gpu(A: "float[:,:]", B: "float[:,:]", C: "float[:,:]"):
    N: int = shape(A)[0]
    s: float = 0.0
    # omp target teams distribute parallel for collapse(2)
    for i in range(N):
        for j in range(N):
            s = 0.0
            for k in range(N):
                s += A[i, k] * B[k, j]
            C[i, j] = s
```

Custom compiler flags can be specified in a .json fi

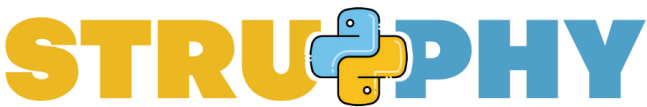
```
pyccel matmult.py --language fortran --o
```

Note: 1 GPU compared to 1 CPU core:
(proof of concept)



Lessons learned

- Struphy offers to be a `Python` hub for geometric PDEs
- The modularity of the has allowed for students to quickly add new plasma models
- Pylikwid integration provides detailed performance metrics
- Porting numpy operations to GPU with CuPy is trivial
- Porting Pyccelized kernels to GPU using OpenMP requires some effort, but works!
- Next steps:
  - Continue adding OpenMP pragmas to the remaining kernels
  - Multi-node simulations on GPU



Plasma Physics Models in Python

About Struphy

Struphy offers to be a `Python` hub for geometric PDEs:

- **Prototyping and production in one place**
- Lots of abstractions, parallel data structures
- Automated testing and I/O
- Documentation, maintainance, code improvement

How to use Struphy:

- Start with the notebook tutorials:
  `https://mybinder.org/v2/gh/struphy-hub/struphy-tutorials/main`
- Play around with Struphy objects and write a first toy model
- Follow Struphy on LinkedIn: `https://www.linkedin.com/company/struphy/`
- Follow the development of the repo: `https://github.com/struphy-hub/struphy`