GPU Computing via Python’s Context Management for Beam Dynamics Simulations

Adrian Oeftiger
16 Oct 2019, PyHEP 2019
Motivation

Numerical simulations on beam dynamics...
- follow **long-term** motion of beam particles in a synchrotron
- demand **iterative development**: frequent update of models
- require heavy **number crunching**
  - in particular for collective effects (particle-to-particle interaction)
- often rely on **high-performance computing** (HPC)
HPC and Python?

HPC vs. Python?!

Pure python:

• reputation of being slow

• libraries and tools
HPC and Python?

- **rapid prototyping**
- **iterative optimisation**
- **glue: low-level languages**
- **numpy, cython, numba, pycuda, CuPy, ...**
The General Problem...

Ingredients:

- 50k lines (smoothly working) CPU simulation code
- 5 dashes new numerical challenges
- few nice GPUs in the corner...
The General Problem...

Ingredients:

50k lines  (smoothly working) CPU simulation code
5 dashes   new numerical challenges
few        nice GPUs in the corner...

Recipe:

■ translate into CUDA, ...

Figure: Mai Tai, postprohibition.com
The General Problem...

Ingredients:

- 50k lines  (smoothly working) CPU simulation code
- 5 dashes  new numerical challenges
- few   nice GPUs in the corner...

Recipe:

- translate into CUDA, ...
- fiddling with GPU libraries

Figure: Mai Tai, postprohibition.com
The General Problem...

**Ingredients:**

- *50k lines* (smoothly working) CPU simulation code
- *5 dashes* new numerical challenges
- *few* nice GPUs in the corner...

**Recipe:**

- translate into CUDA, ...
- fiddling with GPU libraries
- achieve promising speed-ups

**Figure:** Mai Tai, postprohibition.com
The General Problem...

Ingredients:

- \textit{50k lines} (smoothly working) CPU simulation code
- \textit{5 dashes} new numerical challenges
- \textit{few} nice GPUs in the corner...

Recipe:

- translate into CUDA, ...
- fiddling with GPU libraries
- achieve promising speed-ups
- this one impressive GPU cluster simulation

\textbf{Figure:} Mai Tai, postprohibition.com
The General Problem...

Ingredients:

- 50k lines (smoothly working) CPU simulation code
- 5 dashes new numerical challenges
- few nice GPUs in the corner...

Recipe:

- translate into CUDA, ...
- fiddling with GPU libraries
- achieve promising speed-ups
- this one impressive GPU cluster simulation
- ... maintenance kills the project

Figure: Mai Tai, postprohibition.com
The General Problem...

Ingredients:

- 50k lines (smoothly working) CPU simulation code
- 5 dashes new numerical challenges
- few nice GPUs in the corner...

maintenance problems...

- “oh this new feature.. yes, that’s only in the CPU version for the moment..”
- “did we fix that physics bug also in the GPU version?”
- ...

⇒ typically at some point, GPU version lags behind CPU version

classic simulation

- ... maintenance kills the project

---

**Figure:** Mai Tai, postprohibition.com
... how to solve the maintenance problem?
One implementation to rule them all

Lessons learned from past experiences:
- implement the physics **once**
- separate architecture-specific back-end from physics

Approaches to separate backend from physics:
- Python’s duck typing
- templating (⇒ cf. next talk by M. Schwinzerl)
- just-in-time (JIT) compilation
One implementation to rule them all

Lessons learned from past experiences:

- implement the physics **once**
- separate architecture-specific back-end from physics

Approaches to separate backend from physics:

- Python’s duck typing
- templating (⇒ cf. next talk by M. Schwinzerl)
- just-in-time (JIT) compilation

“If it walks like a duck and it quacks like a duck, then it must be a duck.”
⇒ dynamic typing
Gains

Separate back-end via duck typing =

- less code, less bugs, less maintenance
- more readable physics
- simplify code extensibility:
  1. **user**: use fixed script with simulation library, adapt input values
  2. “proactive” **user**: easily extend simulation library with more physics
  3. **developer**: maintain back-ends, optimise new extensions
Example of PyHEADTAIL

Implemented this strategy in beam dynamics simulation tool PyHEADTAIL.

⇒ Let’s play! Find a concept jupyter notebook in this github repo:

**Figure:** PyHEADTAIL concept jupyter notebook
Implement the physics once – a synchrotron model consists of many consecutive *accelerator elements*:

*Figure*: arrange the synchrotron like duplo (image courtesy Kevin Li)
Implement the physics once – a synchrotron model consists of many consecutive *accelerator elements*:

**accelerator element:**

```python
from abc import ABCMeta, abstractmethod

class Element(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def track(self, beam):
        pass

⇒ track method implements the physics for a given Element
```
The dynamical state of the physical system (in our case the beam particles) is stored in arrays (e.g. numpy):

```
class Particles(object):
    def __init__(
        self, x, xp, y, yp, z, dp,
        intensity, gamma, circumference,
        charge=e, mass=m_p, *args, **kwargs):

    # arrays, each entry = one macro-particle:
    self.x = x
    self.xp = xp
(....)
```
Dynamical State

The dynamical state of the physical system (in our case the beam particles) is stored in arrays (e.g. `numpy`):

```python
class Particles(object):
    (...)

    def mean_x(self):
        return pm.mean(self.x)  # imagine pm to be numpy for now
    (...)

    def sigma_x(self):
        return pm.std(self.x)
    (...)
```
Example: Track through RF Cavity

Example: a radio-frequency cavity

Figure: CERN Control Centre Animations, 09 “LHC accelerating cavities”
Example: Track through RF Cavity

Example: a radio-frequency cavity

simple accelerator element example:

class RFCavity(Element):
    def __init__(self, voltage):
        self.voltage = voltage

    def track(self, beam):
        amplitude = (beam.charge * self.voltage /
                     (beam.p0 * beam.beta * c))
        phi = 2 * np.pi * beam.z / beam.circumference
        beam.dp += amplitude * pm.sin(phi)

- track doesn’t know about the back-end!
  ⇒ just assume that pm.sin can deal with beam.z,dp array!
A typical simulation structure may look like so:

**simulation script:**

```python
beam = Particles(...)
one_turn_map = [SomeElement(...), AnotherElement(...),
                YetAnotherElement(...), ...]
n_turns = 1000

for i in range(n_turns):
    for el in one_turn_map:
        el.track(beam)
```

- `one_turn_map` represents mapping through synchrotron
- Each element therein transports particles to next element
A typical simulation structure may look like so:

```
simulation script:

beam = Particles(...)  
one_turn_map = [SomeElement(...), AnotherElement(...), YetAnotherElement(...), ...]  
n_turns = 1000  
with CPU(beam):
    for i in range(n_turns):
        for el in one_turn_map:
            el.track(beam)
```

- use context management to specify back-end (and corresponding libraries) for `el.track(beam)`
Define Context Manager

So what are the pm math library and CPU context manager?

**pm math library for the CPU:**

```python
import numpy as np

cpu_dict = dict(
    mean=np.mean,
    std=np.std,
    (...)
    sin=np.sin,
    exp=np.exp,
    (...)
)
```

- `cpu_dict` redirects to `numpy` functions as default for CPU
Define Context Manager

So what are the \texttt{pm} math library and CPU context manager?

\textbf{pm} math library for the CPU:

```python
class pmath(object):
    default = cpu_dict
    
def __init__(self):
        self.update(self.default)

    def update(self, func_dict):
        for func in func_dict:
            setattr(self, func, func_dict[func])

pm = pmath()
```

- here, global state \texttt{pm} can update active function dictionary
Define Context Manager

So what are the \texttt{pm} math library and \texttt{CPU} context manager?

**CPU context manager:**

```python
class CPU(object):
    def \_\_init\_\_(self, beam):
        self.beam = beam
        self.to_move = ['x', 'xp', ...] # all arrays in Particles
```

(...)

FAIR GmbH | GSI GmbH
Adrian Oeftiger
16 Oct 2019
Define Context Manager

So what are the $\text{pm}$ math library and CPU context manager?

### CPU context manager:

```python
class CPU(object):
    (...)

    def __enter__(self):
        # “move” data to CPU RAM:
        for attr in self.to_move:
            coord = getattr(self.beam, attr)
            transferred = np.asarray(coord)
            setattr(self.beam, attr, transferred)

        # redirect math library correctly to numpy:
        pm.update(cpu_dict)

        return self

    (...)
```
So what are the \texttt{pm} math library and \texttt{CPU} context manager?

\textbf{CPU context manager:}

```python
class CPU(object):
    (...)
    def __exit__(self, exc_type, exc_value, traceback):
        # potentially move data back to host

        # default math library
        pm.update(pm.default)
```

Remember the 3 types of interaction with the simulation library?

1. **user:**
   - use fixed script with simulation library, adapt input values
     → can easily switch back-end via context (CPU ⇝ GPU)

2. **“proactive” user:**
   - easily extend simulation library with more physics
     → super easy to add more physics in RFCavity or add own implementation based on pm math functions
     (⇒ no knowledge of back-end required!)

3. **developer:**
   - maintain back-ends, optimise new extensions
     → can add functionality in math function dictionaries behind pm
     (⇒ can provide new dictionaries + context managers)
     (i.) use low-level languages (C, Cython, ...)
     (ii.) exploit new architectures like the GPU

Python's duck typing
→ separate physics from back-end (based on numpy array API)
→ implement physics once
→ can change back-end easily
→ can provide more context managers addressing new back-ends
Simplifying Extensibility!

Remember the 3 types of interaction with the simulation library?

1. **user:**
   use fixed script with simulation library, adapt input values
   \[\rightarrow\text{can easily switch back-end via context (CPU} \rightsquigarrow \text{GPU)}\]

2. **“proactive” user:**
   easily extend simulation library with more physics
   \[\rightarrow\text{super easy to add more physics in RFCavity or add own implementation based on pm math functions}\]
   \[\Rightarrow\text{no knowledge of back-end required!}\]
Simplifying Extensibility!

Remember the 3 types of interaction with the simulation library?

1. **user:**
   - use fixed script with simulation library, adapt input values
   - can easily switch back-end via context (CPU ⇔ GPU)

2. **“proactive” user:**
   - easily extend simulation library with more physics
   - super easy to add more physics in `RFCavity` or add own implementation based on `pm` math functions
   - no knowledge of back-end required!

3. **developer:**
   - maintain back-ends, optimise new extensions
   - can add functionality in math function dictionaries behind `pm`
   - can provide new dictionaries + context managers
     - (i.) use low-level languages (C, Cython, ...)
     - (ii.) exploit new architectures like the GPU!
Simplifying Extensibility!

Remember the 3 types of interaction with the simulation library?

1. **user:**
   - use fixed script with simulation library, adapt input values
   - → can easily switch back-end via context (CPU ⇝ GPU)

**Python’s duck typing**

- → separate physics from back-end (based on `numpy` array API)
- → implement physics once
- → can change back-end easily
- → can provide more context managers addressing new back-ends

(i.) use low-level languages (C, Cython, ...)
(ii.) exploit new architectures **like the GPU!**
... as developers, we’re now interested in how to speed things up, right?!

1. new function dicts for pm
2. new GPU context
Collective Effects

In the previously mentioned jupyter notebook, you find more sophisticated Element types representing **collective effects**:

→ e.g. particle-to-particle interaction via the metal vacuum tube ("wakefields / impedances")

**Figure**: source particle impacting trailing witness particle via wakefield (induced mirror current)
Collective Effects

In the previously mentioned jupyter notebook, you find more sophisticated Element types representing collective effects:

→ e.g. particle-to-particle interaction via the metal vacuum tube ("wakefields / impedances")
→ imply calculating beam statistics, histogramming etc.
   → heavy computations and memory-intensive algorithms
Speed up with Cython!

Typically, timing bottleneck in a simulation boils down to one (statistics) function.

Suppose e.g. `beam.sigma_x()` is the bad guy:

- based on `np.std` via with previous `cpu_dict` and CPU context:

  In: `%timeit with CPU(beam): beam.sigma_x()`
  Out: 100 loops, best of 3: 6.63 ms per loop
Speed up with Cython!

⇒ Typically, timing bottleneck in a simulation boils down to one (statistics) function.

Suppose e.g. `beam.sigma_x()` is the bad guy:

- based on `np.std` via with previous `cpu_dict` and CPU context:

  In: %timeit with CPU(beam): beam.sigma_x()
  Out: 100 loops, best of 3: 6.63 ms per loop

⇒ use Cython to speed up beam size computation by $5 \times$

  In: %timeit with CPU_Cython(beam): beam.sigma_x()
  Out: 1000 loops, best of 3: 1.37 ms per loop
What's that Cython Magic?

Cython low-level implementation:

```python
%%cython --compile-args=-fopenmp --link-args=-fopenmp -n cython_functions

cimport libc.math as cmath
cimport cython.boundscheck
cimport cython.cdivision

@cython.boundscheck(False)
@cython.cdivision(True)
cpdef double cov(double[::1] a, double[::1] b):
    (...)
    for i in xrange(n):
        a_sum += a[i] - shift_a
        b_sum += b[i] - shift_b
        ab_sum += (a[i] - shift_a) * (b[i] - shift_b)
    return (ab_sum - a_sum * b_sum / n) / (n - 1)

@cython.boundscheck(False)
@cython.cdivision(True)
cpdef double std(double[::1] u):
    return cmath.sqrt(cov(u, u))
```
import cython_functions

cython_dict = cpu_dict.copy()
cython_dict.update(dict(
    cov=cython_functions.cov,
    std=cython_functions.std,
))

class CPU_Cython(CPU):
    def __enter__(self):
        # moving data as in parent CPU class
        (...)

        # replace functions in general.math.py
        pm.update_active_dict(cython_dict)
        return self

→ Full Cython implementation cf. jupyter notebook
Thanks to \texttt{numpy} array API

\begin{itemize}
  \item (a) Cython on CPU
  \item (b) CuPy on NVIDIA GPUs
  \item (c) PyCUDA on NVIDIA GPUs
\end{itemize}

\textbf{Figure:} Python libraries with \texttt{numpy} array API

Based on duck typing approach:

\begin{itemize}
  \item use other libraries implementing \texttt{numpy} array API to provide \texttt{func_dict} rebindings and context managers
  \item \textit{completely transparent to users and “proactive” users extending the physics, just need to support}
    \begin{itemize}
      \item math functions \texttt{sin}, \texttt{cos}, \texttt{exp}, \texttt{sqrt} etc.
      \item \texttt{numpy} array arithmetics: \texttt{a += b * c - d**2}
    \end{itemize}
\end{itemize}
Example for CuPy


gpu_dict with CuPy

```python
import cupy

gpu_dict = dict(
    mean=cupy.mean,
    std=cupy.std,
    (...)  
    sin=cupy.sin,
    exp=cupy.exp,
    (...)  
    )
```
Example for CuPy

**GPU context manager:**

class GPU(object):
    (...)

    def __enter__(self):
        # moving data to device
        for attr in self.to_move:
            coord = getattr(self.beam, attr)
            transferred = cupy.asarray(coord)
            setattr(self.beam, attr, transferred)

        # replace functions in general.math.py
        pm.update_active_dict(gpu_dict)

        return self

    (...)

FAIR GmbH | GSI GmbH
Adrian Oeftiger
16 Oct 2019
Example for CuPy

GPU context manager:

class GPU(object):
    (...)
    def __exit__(self):
        # moving data back to host
        for attr in self.to_move:
            coord = getattr(self.beam, attr)
            transferred = coord.get()
            setattr(self.beam, attr, transferred)
        pm.update_active_dict(pm._default_function_dict)

.
this concept makes it easy to include GPUs!
Real PyHEADTAIL...

As outlined, this concept is implemented in the actual beam dynamics simulation tool PyHEADTAIL.

Typical realistic simulations with self-consistent space charge (direct particle-to-particle Coulomb interaction \(\rightarrow\) heavily memory-constrained):

\[\text{Table: Full Timing for Space Charge Node}^1\]

<table>
<thead>
<tr>
<th>hardware</th>
<th>cores</th>
<th>time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GPU Tesla P100</td>
<td>3584</td>
<td>53</td>
</tr>
<tr>
<td>NVIDIA GPU Tesla C2075</td>
<td>448</td>
<td>694</td>
</tr>
<tr>
<td>CPU Intel Xeon E5</td>
<td>1</td>
<td>1349</td>
</tr>
</tbody>
</table>

\(^1\text{timings based on } 1 \times 10^6 \text{ macro-particles on } 256 \times 256 \times 100 \text{ grid}\)
beam dynamics with self-consistent beam fields $\leadsto$ HPC

self-field driven a) resonances and b) coherent instabilities
Lessons learned:

- separate physics from back-end implementation
- utilise duck typing and `numpy` API to provide sandwich layer: `context management` and `function redirection`
- can introduce speed-up via specialised Cython etc., exploit GPU via `CuPy` and `PyCUDA`

→ back-end details transparent to users/high-level developers
Summary

Lessons learned:

- separate physics from back-end implementation
- utilise duck typing and numpy API to provide sandwich layer: context management and function redirection
- can introduce speed-up via specialised Cython etc., exploit GPU via CuPy and PyCUDA

→ back-end details transparent to users/high-level developers

... and, based on this concept, we could enjoy the CPU / GPU cocktail again, and again, and again² ... 

² in 2015 PyHEADTAIL introduced the context management for GPU usage, many library extensions for more physics since then profited from running on the GPU!
Thank you for your attention!

Acknowledgements:
Stefan Hegglin, Riccardo de Maria, Martin Schwinzerl, and collaborators from NVIDIA
(notably Andreas Hehn, Bai-Cheng (Ryan) Jeng, Miguel Martinez, Vishal Mehta, Akira Naruse)
Timing Profile for Table 1

Line_profiler output on the P100 GPU for space charge node:

Timer unit: 1e-06 s

Total time: 0.052965 s
File: PyPIC/GPU/pypic.py
Function: pic_solve at line 675

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
</table>
| 675    |      |        |         |        | def pic_solve(self, *mp_coords, **kwargs):
| 676    |      |        |         |        |     '''Encapsulates the whole algorithm to determine the
| 677    |      |        |         |        |     fields of the particles on themselves.
| 678    |      |        |         |        |     The keyword argument charge=e is the charge per macro
| 679    |      |        |         |        |     Further keyword arguments are
| 680    |      |        |         |        |     mesh_indices=None, mesh_distances=None, mesh_weights=
| 681    |      |        |         |        |     The optional keyword arguments lower_bounds=False and
| 682    |      |        |         |        |     upper_bounds=False trigger the use of sorted_particles
| 683    |      |        |         |        |     which assumes the particles to be sorted by the node
| 684    |      |        |         |        |     mesh. (see further info there.)
| 685    |      |        |         |        |     This results in particle deposition to be 3.5x quicker
| 686    |      |        |         |        |     mesh to particle interpolation to be 0.25x quicker.
| 687    |      |        |         |        |     (Timing for 1e6 particles and a 64x64x32 mesh includes
| 688    |      |        |         |        |     The optional keyword argument state=None gets rho, phi
| 689    |      |        |         |        |     mesh_e_fields assigned as members if provided.
| 690    |      |        |         |        |     Return as many interpolated fields per particle as
| 691    |      |        |         |        |     dimensions in mp_coords are given.
| 692    |      |        |         |        |     ''' |
Timing Profile for Table 1

Line_profiler output on the P100 GPU for space charge node:

Timer unit: 1e-06 s

Total time: 0.052965 s
File: PyPIC/GPU/pypic.py
Function: pic_solve at line 675

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>696</td>
<td>1</td>
<td>2</td>
<td>2.0</td>
<td>0.0</td>
<td>charge = kwargs.pop(&quot;charge&quot;, e)</td>
</tr>
<tr>
<td>697</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>if not self.optimize_meshing_memory:</td>
</tr>
<tr>
<td>698</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>kwgars[&quot;mesh_indices&quot;], kwgars[&quot;mesh_weights&quot;] =</td>
</tr>
<tr>
<td>699</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>self.get_meshing(kwgars, *mp_coords)</td>
</tr>
<tr>
<td>700</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>701</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>lower_bounds = kwgars.pop('lower_bounds', None)</td>
</tr>
<tr>
<td>702</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>upper_bounds = kwgars.pop('upper_bounds', None)</td>
</tr>
<tr>
<td>703</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>1</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>state = kwgars.pop('state', None)</td>
</tr>
<tr>
<td>705</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>706</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>if lower_bounds is not None and upper_bounds is not None:</td>
</tr>
<tr>
<td>707</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mesh_charges = self.sorted_particles_to_mesh(</td>
</tr>
<tr>
<td>708</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*mp_coords, charge=charge,</td>
</tr>
<tr>
<td>709</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lower_bounds=lower_bounds, upper_bounds=upper_bounds</td>
</tr>
<tr>
<td>710</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>711</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>else: # particle arrays are not sorted by mesh node ids</td>
</tr>
<tr>
<td>712</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>mesh_charges = self.particles_to_mesh(</td>
</tr>
<tr>
<td>713</td>
<td>1</td>
<td>894</td>
<td>894.0</td>
<td>1.7</td>
<td>*mp_coords, charge=charge, **kwgars</td>
</tr>
<tr>
<td>714</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>)</td>
</tr>
<tr>
<td>715</td>
<td>1</td>
<td>139</td>
<td>139.0</td>
<td>0.3</td>
<td>rho = mesh_charges / self.mesh.volume_elem</td>
</tr>
<tr>
<td>716</td>
<td>1</td>
<td>4</td>
<td>4.0</td>
<td>0.0</td>
<td>if getattr(self.poissonsolver, 'is_25D', False):</td>
</tr>
</tbody>
</table>
Timing Profile for Table 1

Line_profiler output on the P100 GPU for space charge node:
Timer unit: 1e-06 s
Total time: 0.052965 s
File: PyPIC/GPU/pypic.py
Function: pic_solve at line 675

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>717</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>rho *= self.mesh.dz</td>
</tr>
<tr>
<td>718</td>
<td>1</td>
<td>48153</td>
<td>48153.0</td>
<td>90.9</td>
<td>if state: state.rho = rho.copy()</td>
</tr>
<tr>
<td>719</td>
<td>1</td>
<td>1974</td>
<td>1974.0</td>
<td>3.7</td>
<td>phi = self.poisson_solve(rho)</td>
</tr>
<tr>
<td>720</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>if state: state.phi = phi</td>
</tr>
<tr>
<td>721</td>
<td>1</td>
<td>1800</td>
<td>1800.0</td>
<td>3.0</td>
<td>mesh_e_fields = self.get_electric_fields(phi)</td>
</tr>
<tr>
<td>722</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>self._context.synchronize()</td>
</tr>
<tr>
<td>723</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>if state: state.mesh_e_fields = mesh_e_fields</td>
</tr>
<tr>
<td>724</td>
<td>1</td>
<td>5</td>
<td>5.0</td>
<td>0.0</td>
<td>self._context.synchronize()</td>
</tr>
<tr>
<td>725</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>fields = self.field_to_particles(*mesh_fields_and_mp_coords, **kwargs)</td>
</tr>
<tr>
<td>726</td>
<td>1</td>
<td>3</td>
<td>3.0</td>
<td>0.0</td>
<td>self._context.synchronize()</td>
</tr>
<tr>
<td>727</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>0.0</td>
<td>return fields</td>
</tr>
</tbody>
</table>

⇒ ≈ 90% of time spent inside low-level cuFFT library
(hidden behind poisson_solve, uses > 95% there)