



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

Adrian Oeftiger

16 Oct 2019, PyHEP 2019

Numerical simulations on beam dynamics...

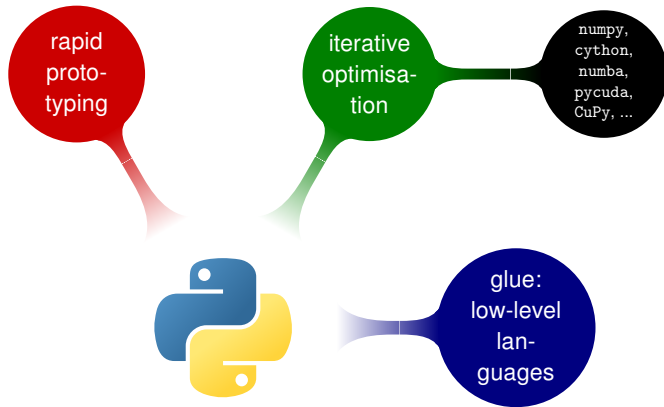
- follow **long-term** motion of beam particles in a synchrotron
- demand **iterative development**: frequent update of models
- require heavy **number crunching**
 - in particular for collective effects (particle-to-particle interaction)
- often rely on **high-performance computing** (HPC)

HPC vs. Python?!

Pure python:

~> reputation of being slow

⇒ libraries and tools



Ingredients:

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

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

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

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

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



Figure: Mai Tai, postprohibition.com

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

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
- cluster simulation
- ... *maintenance* kills the project

A close-up photograph of a Mai Tai cocktail in a glass, garnished with a lime wedge and a sprig of mint, sitting on a wooden surface.

Figure: Mai Tai, postprohibition.com

... how to solve the
maintenance problem?



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 (\Rightarrow cf. next talk by M. Schwinzerl)
- just-in-time (JIT) compilation

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

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
- 
- An orange curved arrow originates from the text "optimise new extensions" in the third list item and points back to the text "easily extend simulation library" in the second list item, indicating a feedback loop or relationship between the developer's maintenance and the user's extension capabilities.

Example of PyHEADTAIL

Implemented this strategy in beam dynamics simulation tool
PyHEADTAIL ↗.

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

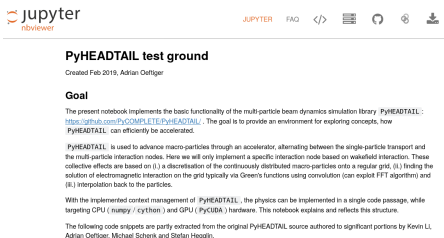


Figure: PyHEADTAIL concept jupyter notebook

Implement the physics once – a synchrotron model consists of many consecutive *accelerator elements*:

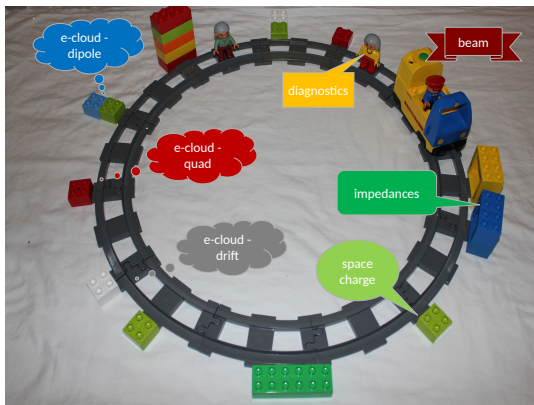


Figure: arrange the synchrotron like duplo (image courtesy Kevin Li)

Implement the physics once – a synchrotron model consists of many consecutive *accelerator elements*:

accelerator element:

```
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):

particles:

```
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  
        (...)
```

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

particles:

```
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: a radio-frequency cavity

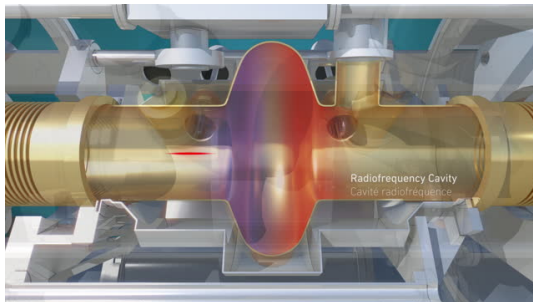


Figure: CERN Control Centre Animations, 09 “LHC accelerating cavities”

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!

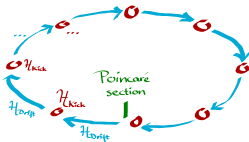
⇒ just assume that `pm.sin` can deal with `beam.z`, `dp` array!

A typical simulation structure may look like so:

simulation script:

```
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)`

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

`pm` math library for the CPU:

```
import numpy as np

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

- `cpu_dict` \rightsquigarrow redirects to numpy functions as default for CPU

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

`pm` math library for the CPU:

```
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 `pm` can update active function dictionary

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

CPU context manager:

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

(...)

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

CPU context manager:

```
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 `pm` math library and CPU context manager?

CPU context manager:

```
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 \rightsquigarrow GPU)

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 \rightsquigarrow 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!

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 \rightsquigarrow 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!**

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 \rightsquigarrow 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

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**”)

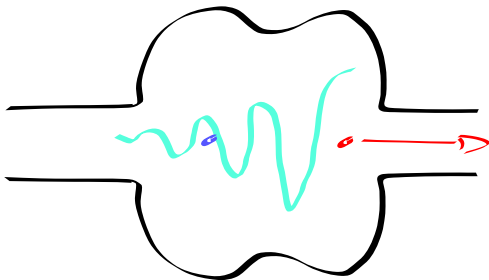


Figure: source particle impacting trailing witness particle via wakefield (induced mirror current)

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

⇒ 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
```

⇒ 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×:

```
In: %timeit with CPU_Cython(beam): beam.sigma_x()  
Out: 1000 loops, best of 3: 1.37 ms per loop
```

Cython low-level implementation:

```
%%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))
```

cython_dict and CPU_Cython

```
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](#) ↗



(a) Cython on CPU



(b) CuPy on NVIDIA GPUs



(c) PyCUDA on NVIDIA GPUs

Figure: Python libraries with `numpy` array API

Based on duck typing approach:

- use other libraries implementing `numpy` array API to provide `func_dict` rebindings and context managers
- ⇒ completely transparent to users and “proactive” users extending the physics, just need to support
 - math functions `sin`, `cos`, `exp`, `sqrt` etc.
 - `numpy` array arithmetics: `a += b * c - d**2`

gpu_dict with CuPy

```
import cupy

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

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

    (...)
```

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!

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
↪ heavily memory-constrained):

Table: Full Timing for Space Charge Node¹

hardware	cores	time [ms]
NVIDIA GPU Tesla P100	3584	53
NVIDIA GPU Tesla C2075	448	694
CPU Intel Xeon E5	1	1349

¹timings based on 1×10^6 macro-particles on $256 \times 256 \times 100$ grid

- beam dynamics with self-consistent beam fields \rightsquigarrow 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

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 ...



²in 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

Line #	Hits	Time	Per Hit	% Time	Line Contents
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					
682					The optional keyword arguments lower_bounds=False and
683					upper_bounds=False trigger the use of sorted_particle
684					which assumes the particles to be sorted by the node
685					mesh. (see further info there.)
686					This results in particle deposition to be 3.5x quicker
687					mesh to particle interpolation to be 0.25x quicker.
688					(Timing for 1e6 particles and a 64x64x32 mesh include
689					
690					The optional keyword argument state=None gets rho, ph
691					mesh_e_fields assigned as members if provided.
692					
693					Return as many interpolated fields per particle as
694					dimensions in mp_coords are given.
695					'''

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

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

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

Line #	Hits	Time	Per Hit	% Time	Line Contents
=====					
717					rho *= self.mesh.dz
718	1	1	1.0	0.0	if state: state.rho = rho.copy()
719					
720	1	48153	48153.0	90.9	phi = self.poisson_solve(rho)
721	1	1	1.0	0.0	if state: state.phi = phi
722					
723	1	1974	1974.0	3.7	mesh_e_fields = self.get_electric_fields(phi)
724	1	5	5.0	0.0	self._context.synchronize()
725	1	1	1.0	0.0	if state: state.mesh_e_fields = mesh_e_fields
726					
727	1	3	3.0	0.0	mesh_fields_and_mp_coords = zip(list(mesh_e_fields),
728	1	175	175.0	0.3	fields = self.field_to_particles(*mesh_fields_and_mp
729	1	1607	1607.0	3.0	self._context.synchronize()
730	1	1	1.0	0.0	return fields

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