SixTrackLib: A Library for GPU Accelerated Single-Particle Tracking

Martin Schwinzerl, Riccardo De Maria

July 17th, 2020

HSS Section Meeting, CERN

Massive Thanks:

Hannes Bartosik (CERN), Massimo Giovannozzi (CERN),
Giovanni Iadarola (CERN), Carlo Emilio Montanari (Univ. di Blogna),
Adrian Oeftiger (GSI/FAIR), Konstantinos Paraschou (AUTH, CERN)

Supported by the Austrian Doctoral Program @ CERN

Summary from "Introduction to GPU Programming"

- GPUs are massively parallel systems with significant single-precision and (ideally: proportionally) reduced double-precision performance
- Currently two main competing frameworks: OpenCL and CUDA
- But: highly dynamic landscape (ROCm, OneAPI, SPIR-V/PTX, ...)
- OpenCL and CUDA are similar $\rightarrow$ abstract away the differences
- $3D$ grid of threads, execute programs in lock-stop as "teams"
- Segmented memory, Host $\leftrightarrow$ Device
- Parallel Performance:
  - Sequential portion of the run-time t_s is a major limiting factor
  - Especially pertinent for GPUs: thread-deference, branching
  - Grow problem size N faster than t_s $\rightarrow$ weak scaling

Introducing SixTrackLib

`SixTrackLib` is a parallel, single particle tracking library used for simulating the trajectories of the particles forming the beam(s) in an accelerator

- Single particle: particles P_i and P_j with i do not interact.
- Tracking: via symplectic, discrete, (thin-lens) maps M_m, representing the beam-element at position m in the accelerator and updating the particle's state: $P_i\left(m+1\right) \leftarrow M_m\left(P_i\left(m\right)\right)$
- Parallel: For $N_{particles} \gg 1$: "Embarrassingly" parallel problem
- Library: independent of application, low barrier of entry, reusable, embeddable, extensible, future backend for `SixTrack`.

https://github.com/SixTrack/SixTrackLib

SixTrackLib: HPC & Design Requirements

Use-Cases: Ranging from dedicated HPC simulation tool for studies to being a potential backend for `SixTrack` in the context of the LHC@Home project

1. Single code-base (User API: C, C++, Python 3)
2. Work across wide pool of hardware, different parallel back-ends
3. Good scalability towards high number of particles on parallel processors and GPUs
4. High code efficiency for small(er) number of Particles using vectorised code / single thread implementation
5.
6. Numerical accuracy, stability & reproducibility

SixTrackLib: HPC & Design Requirements

Use-Cases: Ranging from dedicated HPC simulation tool for studies to being a potential backend for `SixTrack` in the context of the LHC@Home project

1. Single code-base (User API: C, C++, Python 3)
2. Work across wide pool of hardware, different parallel back-ends
3. Good scalability towards high number of particles on parallel processors and GPUs
4. High code efficiency for small(er) number of Particles using vectorised code / single thread implementation
5.
6. Numerical accuracy, stability & reproducibility

SixTrackLib: HPC & Design Requirements

Use-Cases: Ranging from dedicated HPC simulation tool for studies to being a potential backend for `SixTrack` in the context of the LHC@Home project

1. Single code-base (User API: C, C++, Python 3)
2. Work across wide pool of hardware, different parallel back-ends
3. Good scalability towards high number of particles on parallel processors and GPUs
4. High code efficiency for small(er) number of Particles using vectorised code / single thread implementation
5.
6. Numerical accuracy, stability & reproducibility

SixTrackLib: HPC & Design Requirements

Use-Cases: Ranging from dedicated HPC simulation tool for studies to being a potential backend for `SixTrack` in the context of the LHC@Home project

1. Single code-base (User API: C, C++, Python 3)
2. Work across wide pool of hardware, different parallel back-ends
3. Good scalability towards high number of particles on parallel processors and GPUs
4. High code efficiency for small(er) number of Particles using vectorised code / single thread implementation
5.
6. Numerical accuracy, stability & reproducibility

SixTrackLib: HPC & Design Requirements

Use-Cases: Ranging from dedicated HPC simulation tool for studies to being a potential backend for `SixTrack` in the context of the LHC@Home project

1. Single code-base (User API: C, C++, Python 3)
2. Work across wide pool of hardware, different parallel back-ends
3. Good scalability towards high number of particles on parallel processors and GPUs
4. High code efficiency for small(er) number of Particles using vectorised code / single thread implementation
5. From the users perspective: single program with minimal changes
6. Numerical accuracy, stability & reproducibility

SixTrackLib: HPC & Design Requirements

Use-Cases: Ranging from dedicated HPC simulation tool for studies to being a potential backend for `SixTrack` in the context of the LHC@Home project

1. Single code-base (User API: C, C++, Python 3)
2. Work across wide pool of hardware, different parallel back-ends
3. Good scalability towards high number of particles on parallel processors and GPUs
4. High code efficiency for small(er) number of Particles using vectorised code / single thread implementation
5. From the users perspective: single program with minimal changes
6. Numerical accuracy, stability & reproducibility

SixTrackLib: HPC & Design Requirements

Use-Cases: Ranging from dedicated HPC simulation tool for studies to being a potential backend for `SixTrack` in the context of the LHC@Home project

1. Single code-base (User API: C, C++, Python 3)
2. Work across wide pool of hardware, different parallel back-ends
3. Good scalability towards high number of particles on parallel processors and GPUs
4. High code efficiency for small(er) number of Particles using vectorised code / single thread implementation
5. From the users perspective: single program with minimal changes
6. Numerical accuracy, stability & reproducibility

Currently supported backends: OpenCL 1.2, CUDA, Single-Instruction, Multiple-Data (Auto)Vectorised Code (SIMD)

$\Rightarrow$ "Least Common Denominator Coding" (i.e. C99 for the shared code-base, abstractions using macros, High-Level API)

Implementation & Basic Usage

Follow along the `introduction_to_sixtracklib.ipynb` for a more detailed and interactive introduction

⏩ Skip Ahead to End of Section

Modelling the Particle State

- We keep track of **21** attributes in total $\sim$ 168 Bytes / particle
- 6 degrees of freedom for tracking: $x, p_x, y, p_y, \zeta, \delta$
- 4 logical coordinates (`particle_id`, `at_element`, `at_turn`, `state`)
- 11 attributes that are not strictly needed:
  - 5 attributes of the reference particle: $q_0, m_0, \beta_0, \gamma_0, (P_0 \cdot c)$
  - 6 auxiliary coordinates: s, $p_\sigma = (E - E0)/(\beta_0 \cdot P_0 \cdot c)$, $r_{pp} = P_0/P$, $r_{vv} = \beta/\beta_0$, `charge_ratio` $= (q/q_0)$, $\chi = (q/q_0)/(m/m_0)$
- A `Particles` instance can store the state of a single particle:

```
p = st.Particles(num_particles=1, p0c=6.5e12, q0=1)
```

Note: the state of the simulation is encapsulated in the `Particles` instance

Modelling the Particle State

- A `Particles` instance can also store the state of many particles:

```
beam = st.ParticlesSet()
p = beam.Particles(num_particles=10, p0c=6.5e12, q0=1)
print( p )
```

```
<Particles at 128
  num_particles:10
  q0:[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
  mass0:[9.38272081e+08 9.38272081e+08 9.38272081e+08 9.38272081e+08
 9.38272081e+08 9.38272081e+08 9.38272081e+08 9.38272081e+08
 9.38272081e+08 9.38272081e+08]
  beta0:[0.99999999 0.99999999 0.99999999 0.99999999 0.99999999 0.99999999
 0.99999999 0.99999999 0.99999999 0.99999999]
  gamma0:[6927.62813396 6927.62813396 6927.62813396 6927.62813396 6927.62813396
 6927.62813396 6927.62813396 6927.62813396 6927.62813396 6927.62813396]
  p0c:[6.5e+12 6.5e+12 6.5e+12 6.5e+12 6.5e+12 6.5e+12 6.5e+12 6.5e+12 6.5e+12
 6.5e+12]
  s:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  x:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  y:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  px:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  py:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  zeta:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  psigma:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  delta:[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  rpp:[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
  rvv:[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
  chi:[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
  charge_ratio:[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
  particle_id:[0 0 0 0 0 0 0 0 0 0]
  at_element:[0 0 0 0 0 0 0 0 0 0]
  at_turn:[0 0 0 0 0 0 0 0 0 0]
  state:[1 1 1 1 1 1 1 1 1 1]
>
```

Lattice & Beam Elements

In General: Similar to `SixTrack`

- `Drift`, `DriftExact`
- `Multipole` (incl. Dipoles, Quadrupoles, Sextupoles, etc.)
- `Cavity`
- `RFMultipole`
- `XYShift`: transversal shift
- `SRotation`: rotation in the transversal plane
- `BeamMonitor`: programmable dump of particle state
- `BeamBeam4D`, `BeamBeam6D`
- `SpaceChargeCoasting`, `SpaceChargeBunched`[1]
- `DipoleEdge`
- `LimitRect`, `LimitEllipse`, `LimitRectEllipse`: aperature checks

[1], `SpaceChargeBunched` → `SpaceChargeQGaussian`

Lattice & Beam Elements

There are different methods to get a lattice for `SixTrackLib`

1. Build manually, element by element
2. Load from binary dump
3. Import from `pysixtrack`
4. Import from `MAD-X` (via `pysixtrack` and `cpymad`)
5. Import from `SixTrack` (via `pysixtrack` and `sixtracktools`)

SideBar: pysixtrack

- pysixtrack: https://github.com/SixTrack/pysixtrack
- Minimal & and straight forward particle tracking implementation written in Python 3
- Independent of SixTrackLib (and SixTrack)
- Idea: prototyping physics, easy to understand, duck-type-able
- External Reference / Verification for SixTrackLib

```python
import pysixtrack as py6tr

# We can use any iterable object to store a lattice:
seq = [ py6tr.elements.Drift(length=0.2), py6tr.elements.Multipole(knl=[0.0, 0.01]) ]

# But if we use pysixtrack's "Line" container ....
other_lattice = py6tr.Line( seq )
print( other_lattice )
```

```
Line(elements=[Drift(length=0.2), Multipole(knl=[0.0, 0.01], ksl=[0], hxl=0, hyl=0, length=0)], element_names=())
```

Lattice & Beam Elements

1. Build manually, element by element

```
# We can use a st.Elements() class to organize individual beam-elements into a lattice

lattice = st.Elements()
drift = lattice.Drift(lenth=0.2) # length in [m]
quad  = lattice.Multipole( knl=[0.0, 0.01] ) # ksl ... skew multipole parameters

lattice.get_elements()
```

```
[<Drift at 128
   length:0.0
 >,
 <Multipole at 136
   order:1
   length:0.0
   hxl:0.0
   hyl:0.0
   bal:[0.   0.   0.01 0.  ]
 >]
```

Lattice & Beam Elements

2. Load from binary dump

```
lattice = st.Elements().fromfile( "./demo_lattice.bin" )

lattice.get_elements()
```

```
[<Drift at 128
   length:0.0
 >,
 <Multipole at 136
   order:1
   length:0.0
   hxl:0.0
   hyl:0.0
   bal:[0.   0.   0.01 0.  ]
 >]
```

- **Note**: For users of C/C++ APi, this is the recommended way to consume an imported lattice from `pysixtrack`, `MAD-X`, or `SixTrack`
- **Question**: What format does the binary dump have? $\Rightarrow$ Later

Lattice & Beam Elements

❸ Import from `pysixtrack`

```python
import pysixtrack as py6tr

# We can use any iterable object to store a lattice:
seq = [ py6tr.elements.Drift(length=0.2), py6tr.elements.Multipole(knl=[0.0, 0.01]) ]

# Use pysixtrack.Line object to store the sequence of beam-elements
other_lattice = py6tr.Line( seq )

# ensure we start from scratch with lattice
del lattice

# import other_lattice as a SixTrackLib lattice:
lattice = st.Elements().from_line( other_lattice )

print( lattice.get_elements() )
```

```
[<Drift at 128
  length:0.2
>, <Multipole at 136
  order:1
  length:0.0
  hxl:0.0
  hyl:0.0
  bal:[0.   0.   0.01 0.  ]
>]
```

- `pysixtrack` can use pretty much any iterable object to store a sequence of beam-elements (e.g. a `list` environment).
- By using a `pysixtrack.Line`, we can export directly to `SixTrackLib`

Lattice & Beam Elements

4. Import from `MAD-X` via `pysixtrack` (and `cpymad`)

```python
# requires cpymad -> https://github.com/hibtc/cpymad
from cpymad.madx import Madx
import sixtracklib as st
import pysixtrack as py6tr

from scipy.constants import e, m_p, c
import numpy as np

# Note: pysixtrack / SixTrackLib and MAD-X use different default units for energies!
p0c = 4.7e9 # p0c = P0 * c ; [p0c] = 1 eV
Etot_in_GeV = np.sqrt( p0c * p0c + ( m_p / e ) ** 2 * c ** 4 ) * 1e-9 # [Etot] = 1 GeV

mad = Madx(stdout=False)
mad.call( file="./demo_lattice.madx" )
mad.command.beam(particle='proton', energy=str(Etot_in_GeV))
mad.use(sequence="DEMO_LATTICE")

# Use the from_madx_sequence() method of pysixtrack's Line to import DEMO_LATTICE
# Also, use the remove_zero_length_drifts() and merge_consequentive_drifts()
# helpers to optimize the lattice for SixTrackLib's preferred way of torage :

imported_lattice = py6tr.Line.from_madx_sequence(
    mad.sequence.DEMO_LATTICE, ).remove_zero_length_drifts(
    inplace=True).merge_consecutive_drifts(inplace=True)

# As before, use imported_lattice to build the SixTrackLib lattice
# that we are actually interested in:
lattice = st.Elements().from_line( imported_lattice )
print( lattice.get_elements() )
```

```
[<Drift at 128
  length:0.2
>, <Multipole at 136
  order:1
  length:0.0
  hxl:0.0
  hyl:0.0
  bal:[0.   0.   0.01 0.  ]
>]
```

Lattice & Beam Elements

5. Import from `SixTrack` via `pysixtrack` (and `sixtracktools`[2])

```python
import sixtracktools
import pysixtrack as py6tr
import sixtracklib as st

six = sixtracktools.SixInput("./sixtrack_lhc_no_bb_example" )
import_lattice = py6tr.Line.from_sixinput( six ).remove_zero_length_drifts(
    inplace=True).merge_consecutive_drifts(inplace=True)

print( f"import_lattice has {len(import_lattice)} elements" )

# Same procedure as usual -> convert import_lattice to a SixTrackLattice
lattice = st.Elements().from_line( import_lattice )

print( "print first 5 elements in the SixTrackLib lattice:" )
print( lattice.get_elements()[0:5] )
```

```
import_lattice has 18403 elements
print first 5 elements in the SixTrackLib lattice:
[<Drift at 128
  length:23.3918
>, <Multipole at 136
  order:1
  length:0.0
  hxl:0.0
  hyl:0.0
  bal:[ 0.          0.         -0.00096448  0.        ]
>, <Multipole at 208
  order:13
  length:0.0
  hxl:0.0
  hyl:0.0
  bal:[ 0.00000000e+00  0.00000000e+00  1.94834134e-06  0.00000000e+00
 -1.96742097e-04  3.13114838e-05 -1.93729857e-03 -2.89009571e-04
 -1.06106285e-01 -1.73539234e-02  1.17440379e+00 -1.55894308e-01
  1.09295835e+02 -2.30954531e+00 -3.99575313e+00  2.47736694e+02
  0.00000000e+00  0.00000000e+00 -1.40058751e+07  2.48870437e+05
```

[2] `sixtracktools` is a helper library. Cf. `https://github.com/SixTrack/sixtracktools`

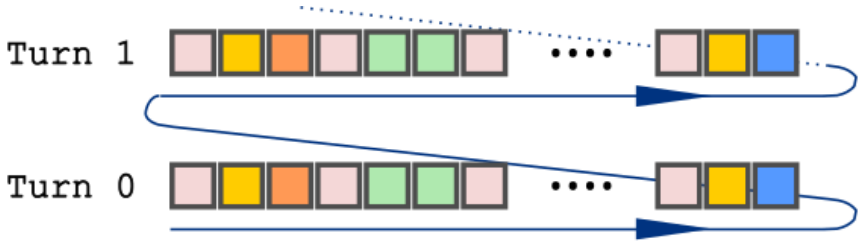
Example: Tracking Code Working On CPUs & GPUs (With Minimal Changes)

```
beam = st.ParticlesSet()
p = beam.Particles(num_particles=10, p0c=6.5e12)
p.x[:] = np.linspace(-1e-6, +1e-6, p.num_particles)
lattice = st.Elements().fromfile("./lhc_no_bb_lattice.bin")

#device=None  # Or:
device="opencl:0.0" #for GPU

job = st.TrackJob( lattice, beam, device=device )
print( f"Architecture of the track job: {job.arch_str}")

job.track_until( 100 )
job.collect_particles()

p.state[0] = 0 # Mark particle 0 explicitly as lost
job.push_particles()

job.track_until( 200 )
job.collect_particles()

if p.num_particles <= 16:
    print( f"at_element after 200 turns : {p.at_element}" )
    print( f"at_turn    after 200 turns : {p.at_turn}" )
    print( f"state      after 200 turns : {p.state}" )
```

```
Architecture of the track job: opencl
at_element after 200 turns : [0 0 0 0 0 0 0 0 0 0]
at_turn    after 200 turns : [100 200 200 200 200 200 200 200 200 200]
state      after 200 turns : [0 1 1 1 1 1 1 1 1 1]
```

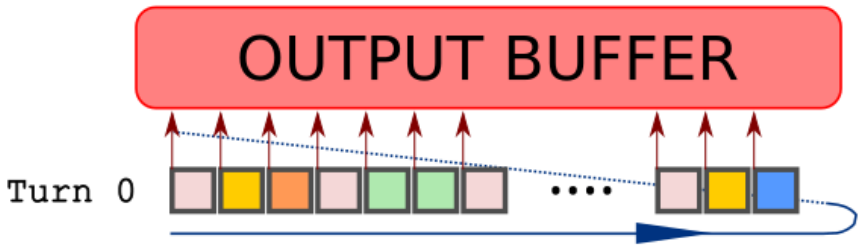
Different Tracking Modes

1. `track_until` Mode:



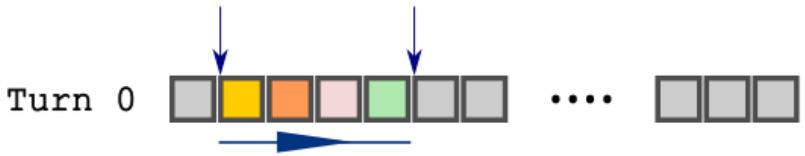
Track all active particles
until they reach at_turn N
`job.track_until( N )`

2. `track_elem_by_elem` Mode:



Like track_until(), but dump
(i.e. copy) the particle state
to an external buffer before
each beam-element
`job.track_elem_by_elem( N )`

3. `track_line` Mode:



Track over subset of lattice
[begin, end)

```
job.track_line( begin, end,
    end_turn=False )
```

Implementation & Basic Usage

Back from the interactive
`introduction_to_sixtracklib.ipynb`

⏩ Return to Start of Previous Section

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

1. Use `track_until`, `collect_*`, `push`
2. Use custom Kernel + Runtime Compilation + `track_line` (Currently only OpenCL, C99)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into SixTrackLib (C99 & C++)
5. Directly link your application against the header-only subset of SixTrackLib (C99)

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

1. Use `track_until`, `collect_*`, `push`
2. Use custom Kernel + Runtime Compilation + `track_line` (Currently only OpenCL, C99)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into SixTrackLib (C99 & C++)
5. Directly link your application against the header-only subset of SixTrackLib (C99)

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

1. Use `track_until`, `collect_*`, `push`
2. Use custom Kernel + Runtime Compilation + `track_line` (Currently only OpenCL, C99)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into `SixTrackLib` (C99 & C++)
5. Directly link your application against the header-only subset of `SixTrackLib` (C99)

Usage & Integration Strategies For SixTrackLib

Sorted in the order "easily accessible" to "complex & invasive"

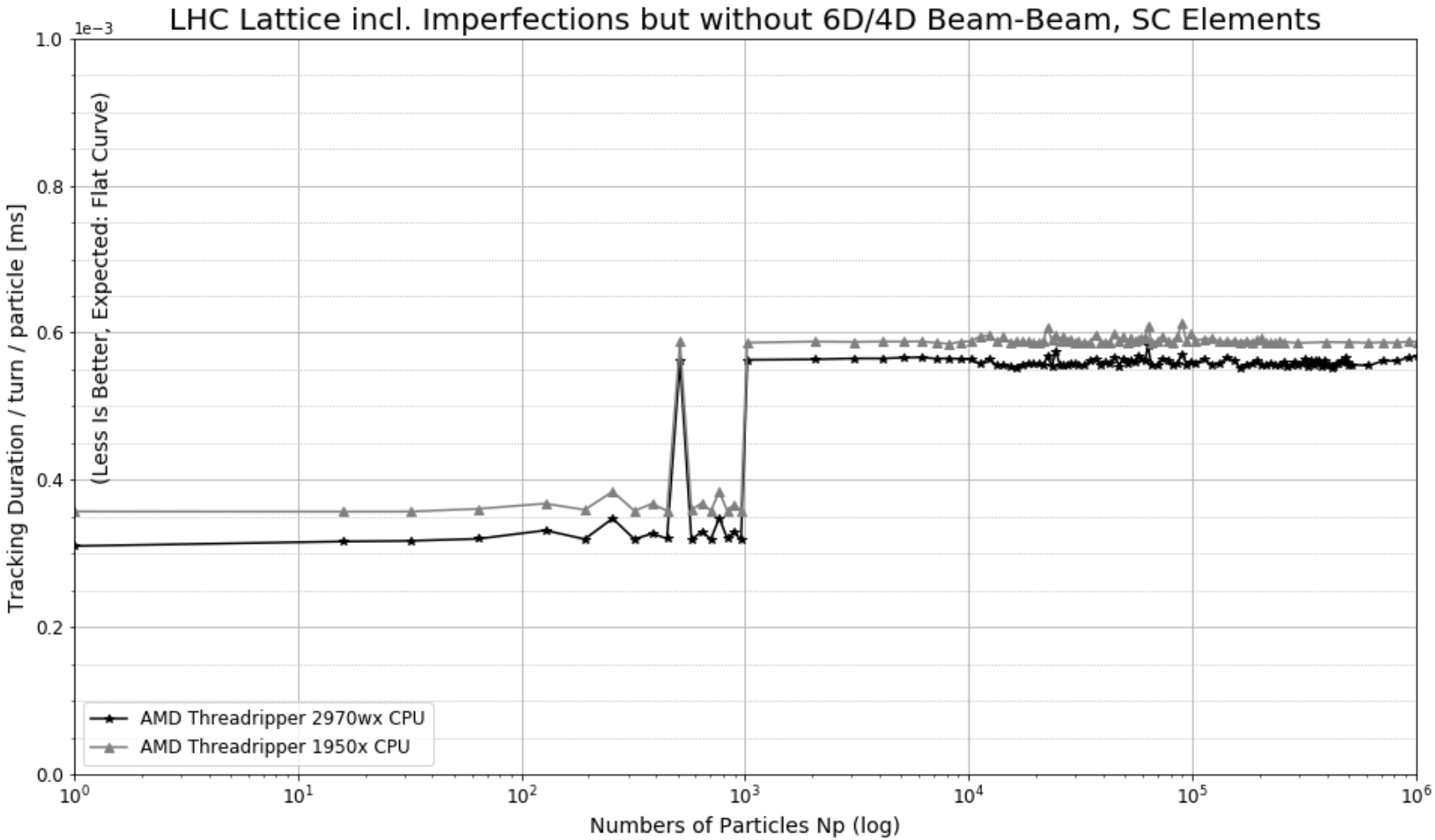
1. Use `track_until`, `collect_*`, `push`
2. Use custom Kernel + Runtime Compilation + `track_line` (Currently only OpenCL, C99)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into `SixTrackLib` (C99 & C++)
5. Directly link your application against the header-only subset of `SixTrackLib` (C99)

Usage & Integration Strategies For SixTrackLib

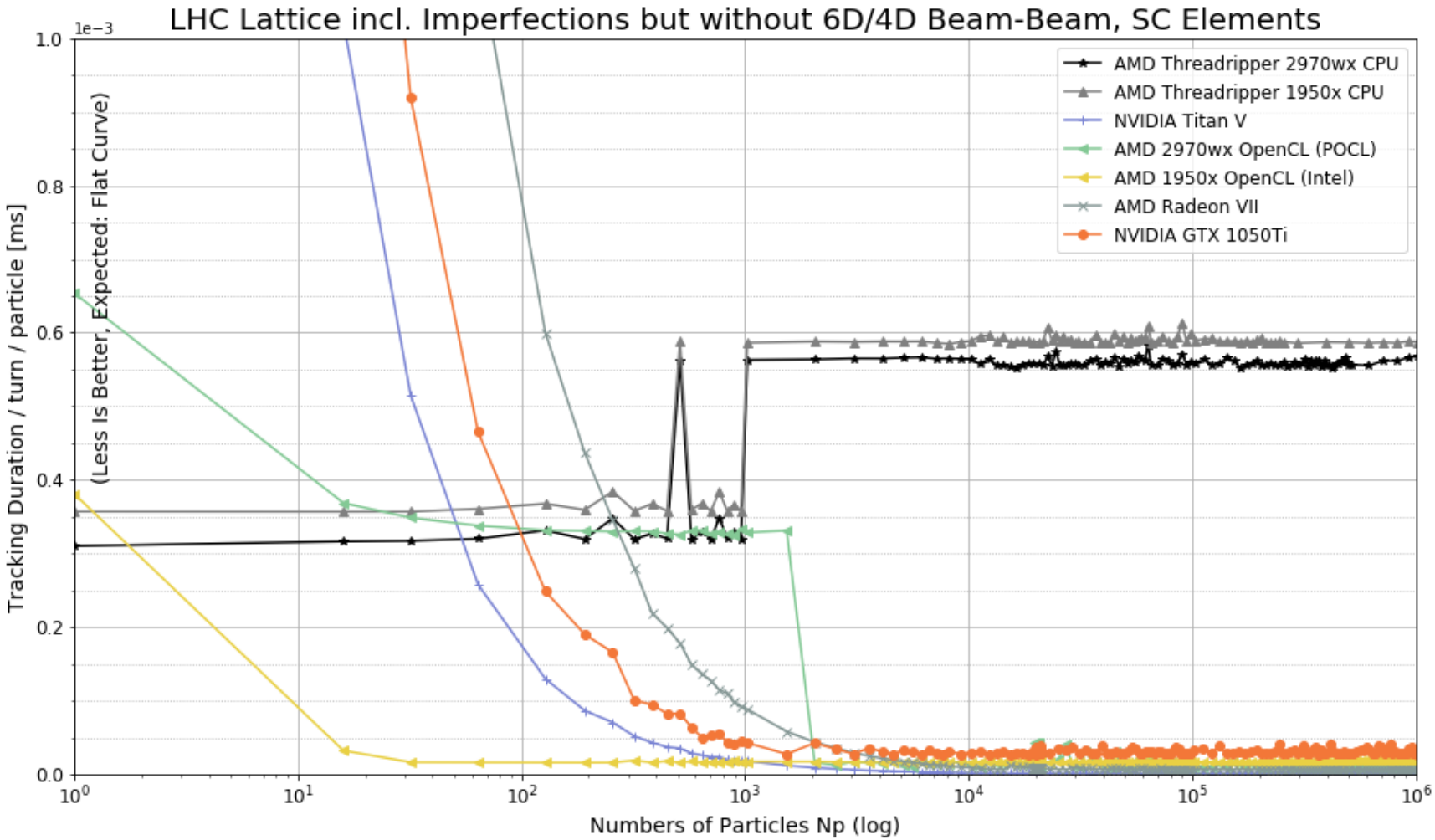
Sorted in the order "easily accessible" to "complex & invasive"

1. Use `track_until`, `collect_*`, `push`
2. Use custom Kernel + Runtime Compilation + `track_line` (Currently only OpenCL, C99)
3. Share particles state "in-place" with other applications (zero-copy) together with `track_line` (Currently only CUDA, C++ or Python)
4. Implement the required functionality (e.g. "beam-elements") into `SixTrackLib` (C99 & C++)
5. Directly link your application against the header-only subset of `SixTrackLib` (C99)

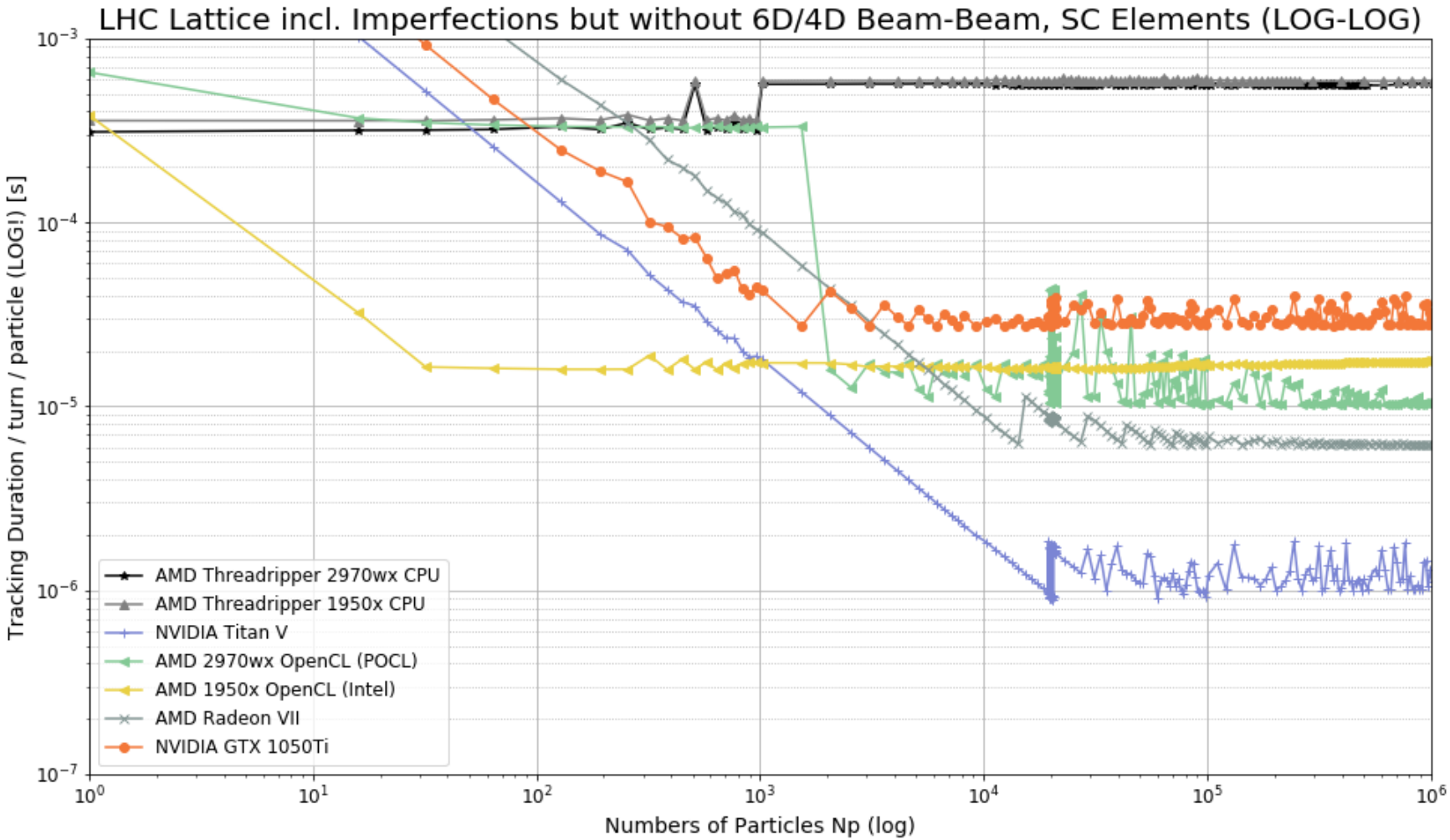
Performance Analysis (CPU, Single Threaded)



Performance Analysis - Lin/Log Plot



Performance Analysis - Log/Log Plot



A Selection Of Usage Examples

1. Studying Particle Losses
   Carlo Emilio Montanari (Università di Bologna), Massimo Giovannozzi
2. Symplectic Kicks From An Electron Cloud
   Konstantinos Paraschou (AUTH,CERN), Giovanni Iadarola, et al
3. Simulating Beam-Beam Interactions & Space-Charge Effects
   Hannes Bartosik, Giovanni Iadarola, et al
4. Integrating SixTrackLib with PyHEADTAIL
   Adrian Oeftiger (GSI/FAIR)

1 Dynamic Aperture (DA), Beam-Stability, Resonances

- Study uses `SixTrackLib` directly to perform tracking for N turns
- Performs analysis and evaluation between turns on the host
- "Simple" use case - no extension and customisation was required

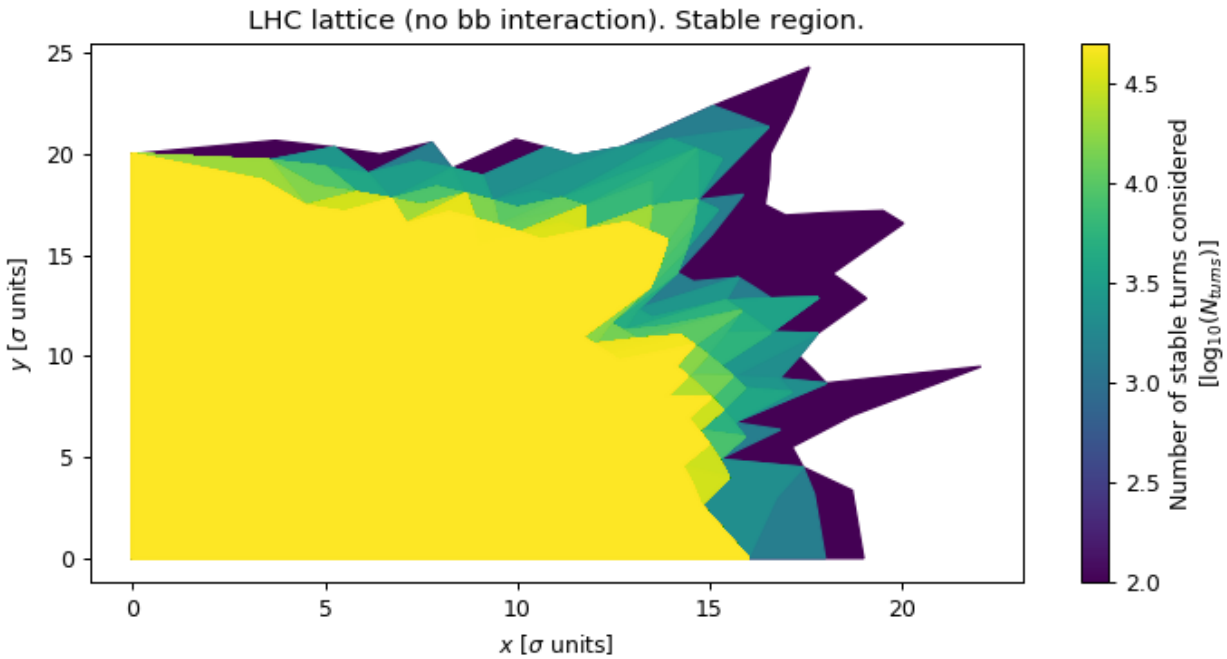


Figure: Sampling stable region via radial scans over N_{turns}

1 Dynamic Aperture (DA), Beam-Stability, Resonances

- Visualising $4D$ space ($r, \alpha, \Theta_1, \Theta_2$ is challenging - `SixTrackLib` helps with creating interactive views by being embeddable into parameterised visualisations

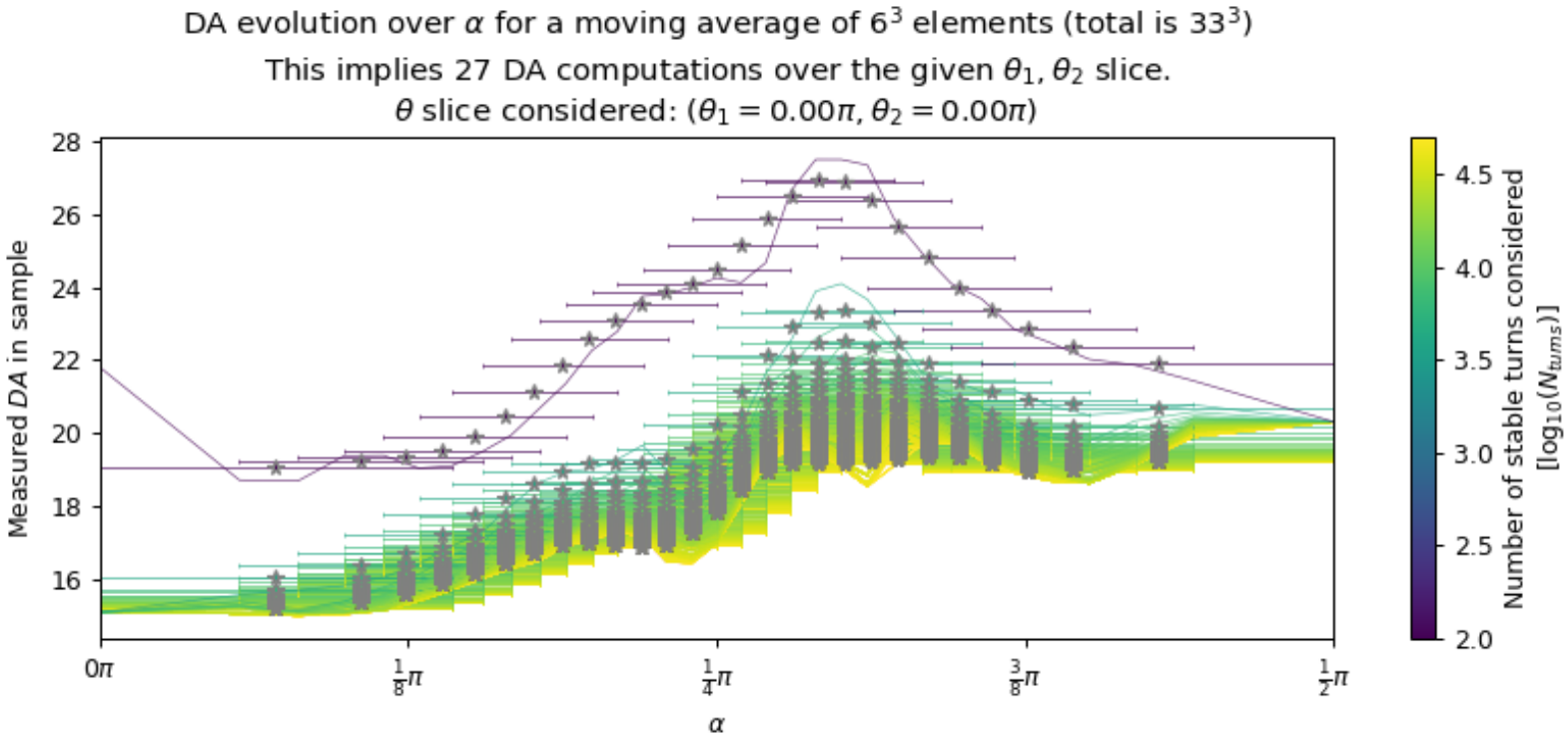


Figure: Evolution of r over α for a given Θ_1,Θ_2 slice over N_{turns}

1 Dynamic Aperture (DA), Beam-Stability, Resonances

- Visualising $4D$ space ($r, \alpha, \Theta_1, \Theta_2$ is challenging - `SixTrackLib` helps with creating interactive views by being embeddable into parameterised visualisations

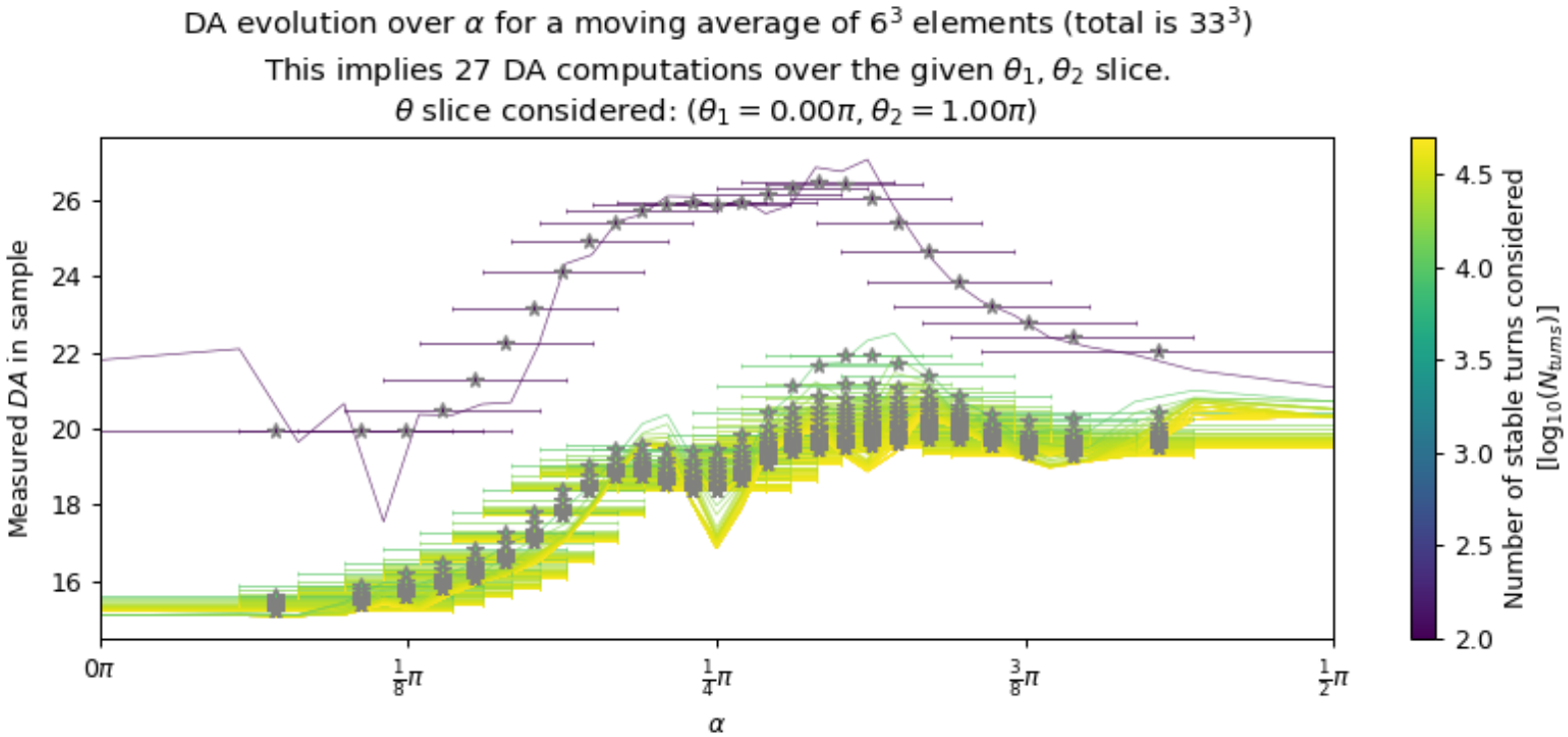


Figure: Evolution of r over α for a given Θ_1,Θ_2 slice over N_{turns}

1 Dynamic Aperture (DA), Beam-Stability, Resonances

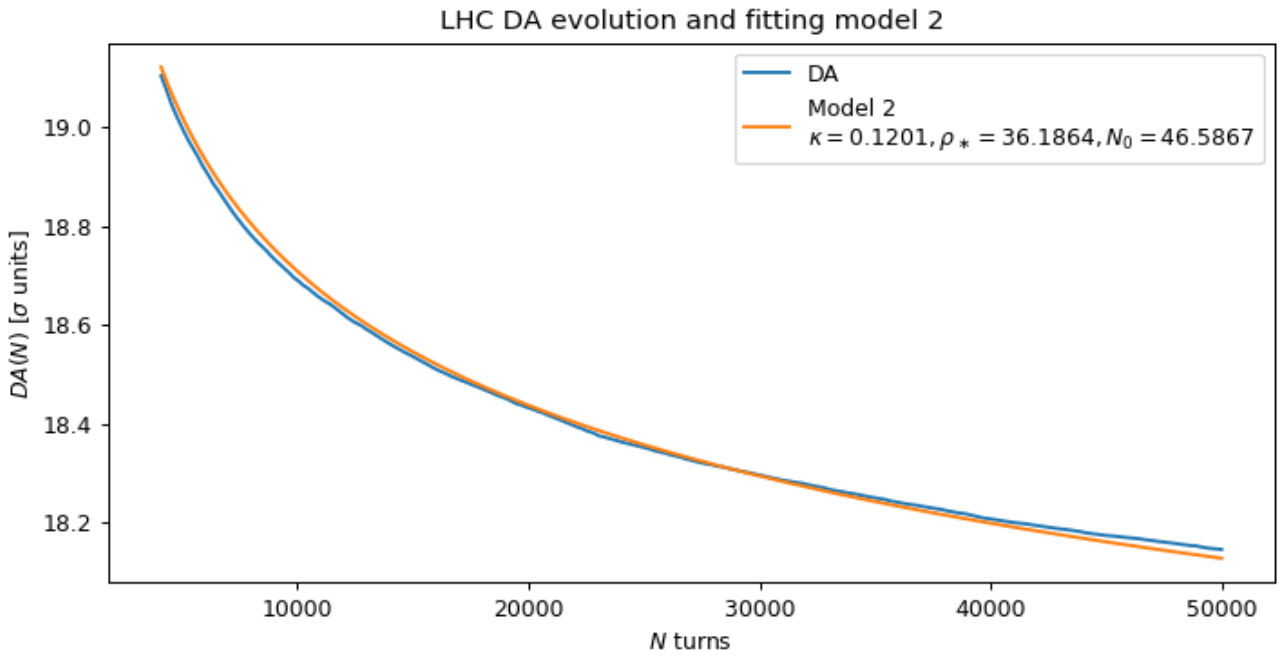


Figure: Comparison of simulated $DA(N)$ with fit $D(N) = \rho_* \left(\frac{\kappa}{2e}\right)^{\kappa} \frac{1}{\ln^{\kappa} \frac{N}{N_0}}$

1 Dynamic Aperture (DA), Beam-Stability, Resonances

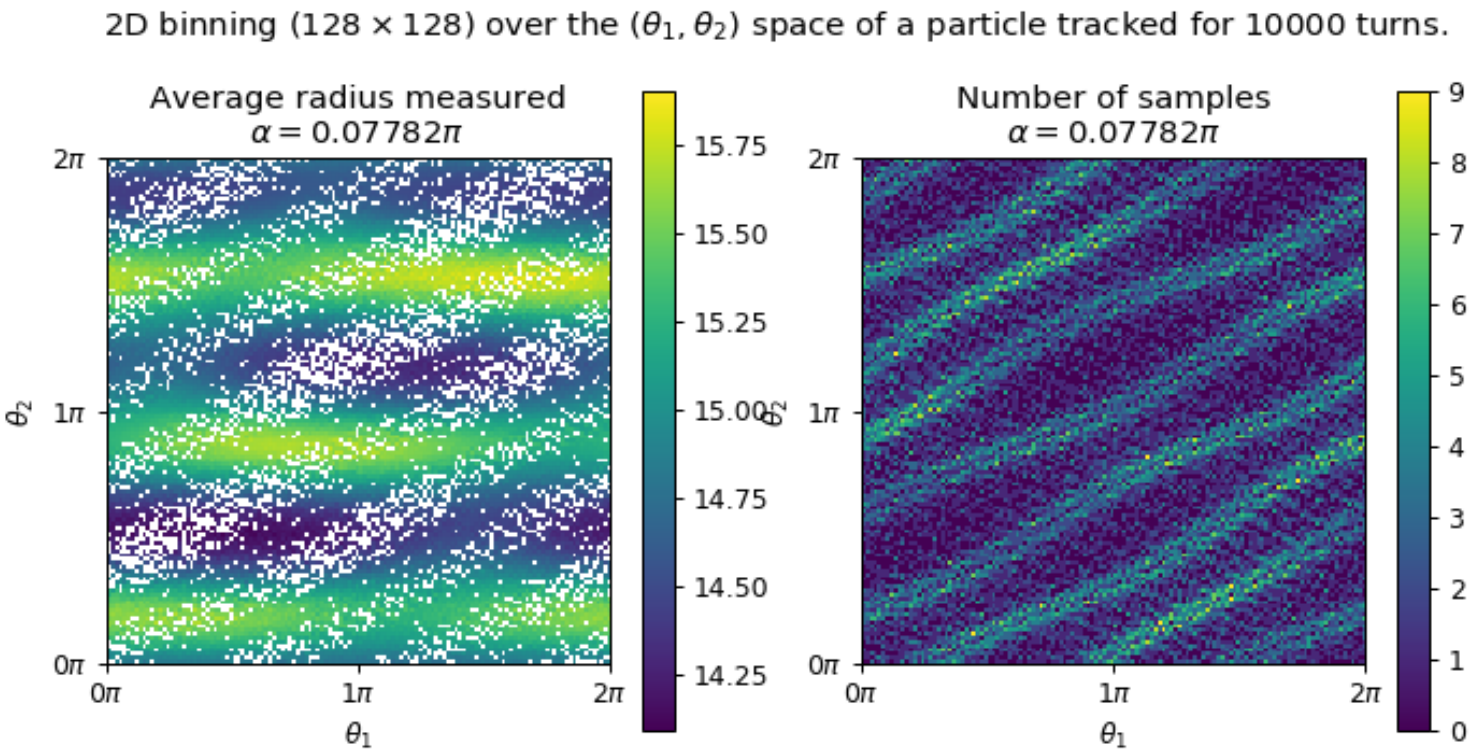


Figure: Histogram and average measured r over $_1$, Θ_2 plane in dependence of initial value for α

1 Dynamic Aperture (DA), Beam-Stability, Resonances

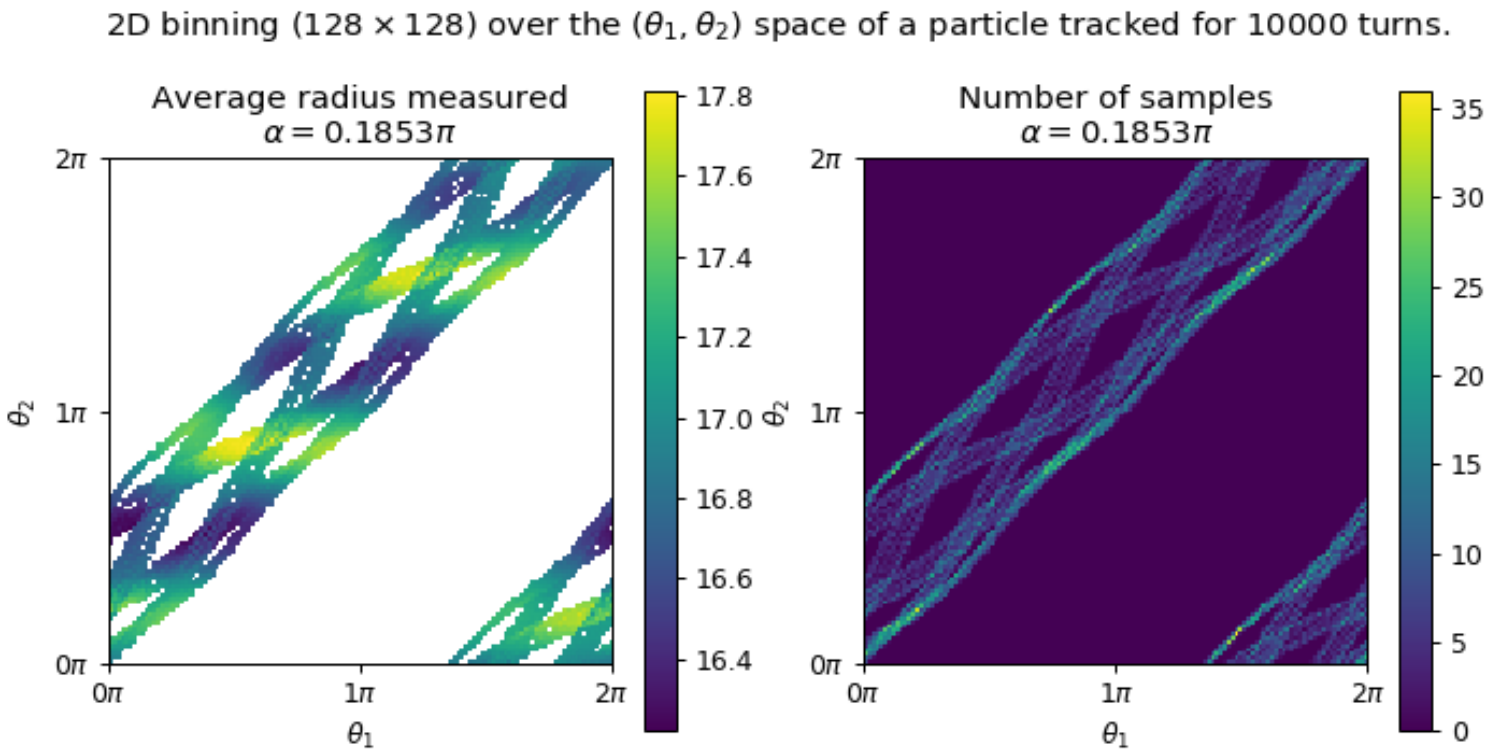
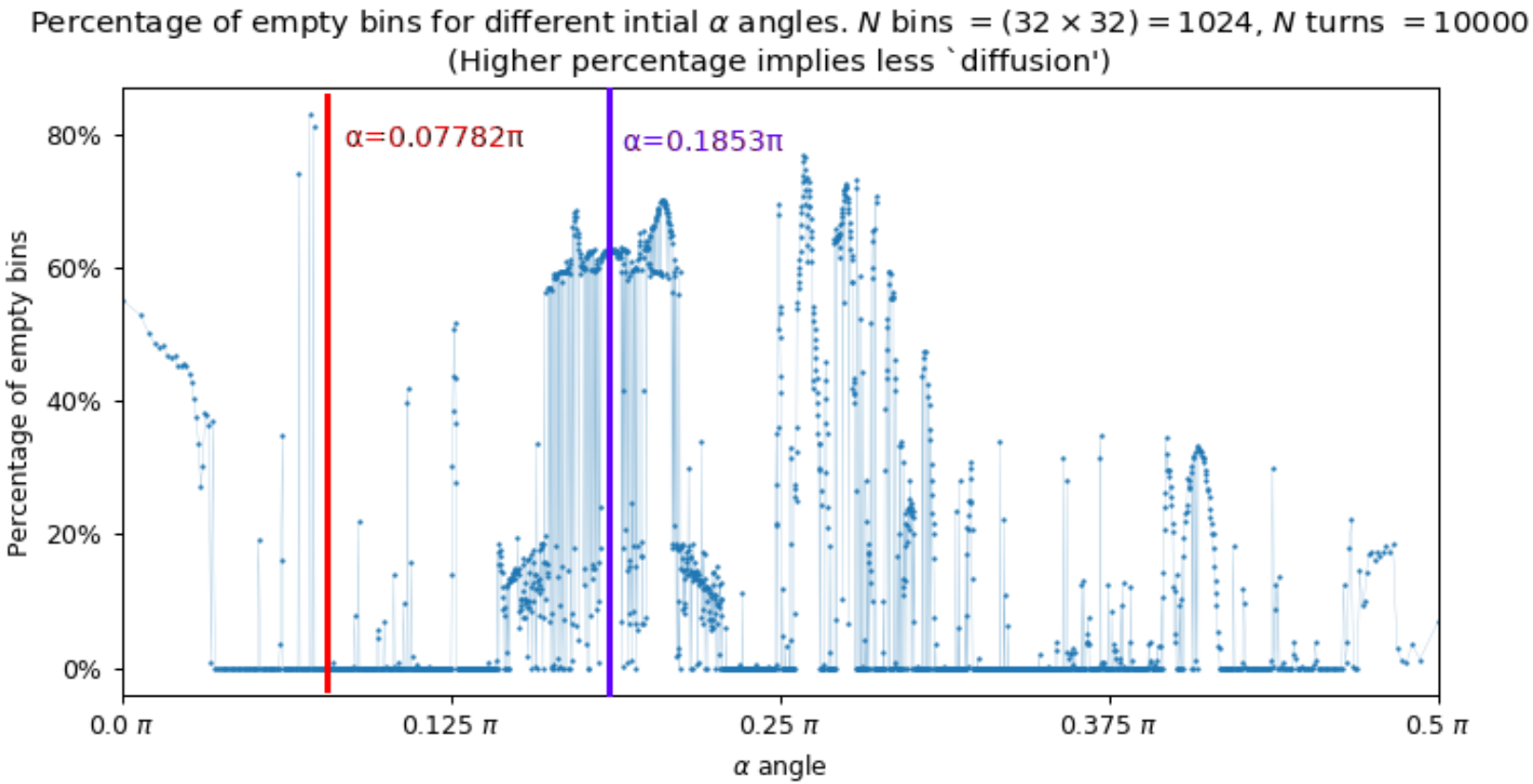


Figure: Histogram and average measured r over $_1$, Θ_2 plane in dependence of initial value for α

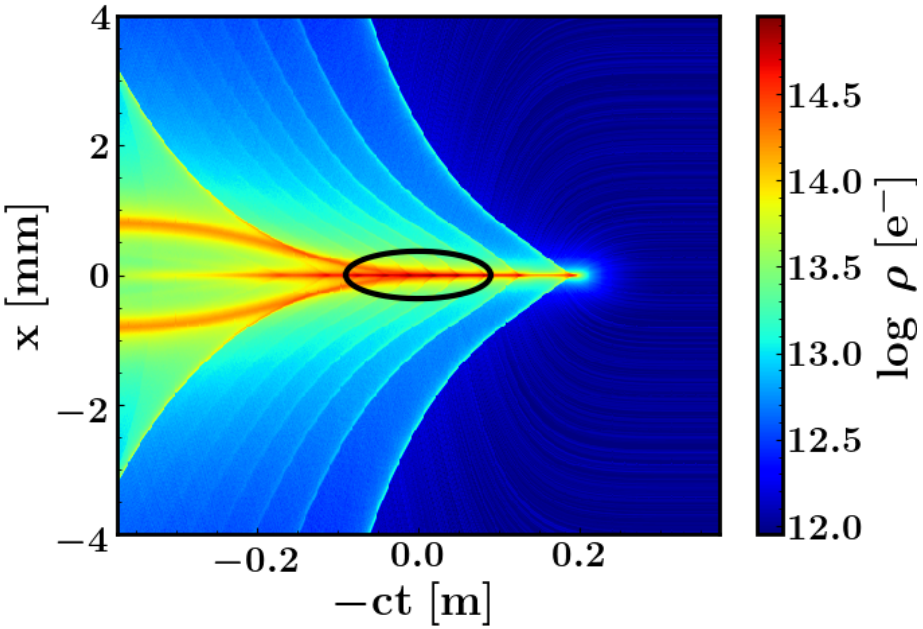
1 Dynamic Aperture (DA), Beam-Stability, Resonances



2 Symplectic Kicks From An Electron Cloud

For various reasons and under certain conditions (fulfilled in the LHC), there exists a complex distribution of electrons within the vacuum chamber that interacts with the beam called **"Electron Cloud"**.
Distribution strongly depends on x, y and time! (as bunch passes through the electron cloud)

Example PyECLOUD simulation:



Particles with an amplitude of 1 beam-σ oscillate within the black line

Under usual approximations[0] the interaction can be written as a thin-lens through the Hamiltonian:

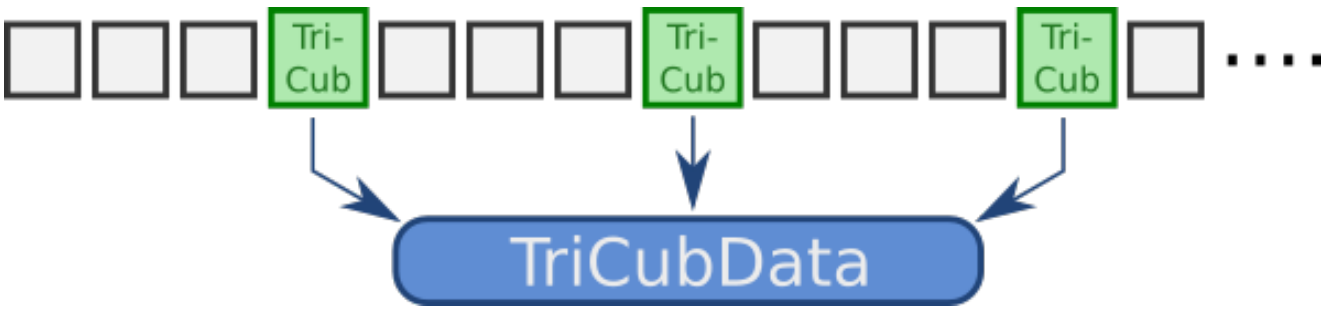
$$H(x, y, \tau; s) = \frac{qL}{\beta_0 P_0 c} \phi(x, y, \tau)\, \delta(s)$$

where ϕ is the scalar potential describing the electron cloud.

[0] see G. Iadarola, CERN-ACC-NOTE-2019-0033.

2 Symplectic Kicks From An Electron Cloud

- PyECLOUD would produce ϕ on a discrete grid (x, y, time)
  $\rightarrow \phi$ should be **interpolated**
- To study slow effects, interpolation should produce symplectic kicks
  $\rightarrow$ Tricubic Interpolation: $\phi(x, y, \tau) = \sum_{i,j,k=0}^{3} a_{ijk} x^i y^j \tau^k$
- Add custom beam-element `TriCub` to implement the map
- N^3 coefficients with typically $N \sim \mathcal{O}(10^2)$ per `TriCub` element
  $\Rightarrow \mathcal{O}(10^3)$ MByte of data for each `TriCub`
- But: interpolation data can be shared between many beam-elements (e.g. All focusing quadrupole magnets have similar Electron Cloud)
- **Idea:** implement infrastructure to store data externally from `TriCub` elements and assign & share coefficient data



2 Symplectic Kicks From An Electron Cloud

- In principle, TriCub element general enough to describe any interaction whose Hamiltonian can be discretized on a grid of (x,y,τ)
- GPUs: large global memory (4-16 GByte), adequate memory bandwidth $\rightarrow$ perfect environment for simulations with TriCub beam-elements.

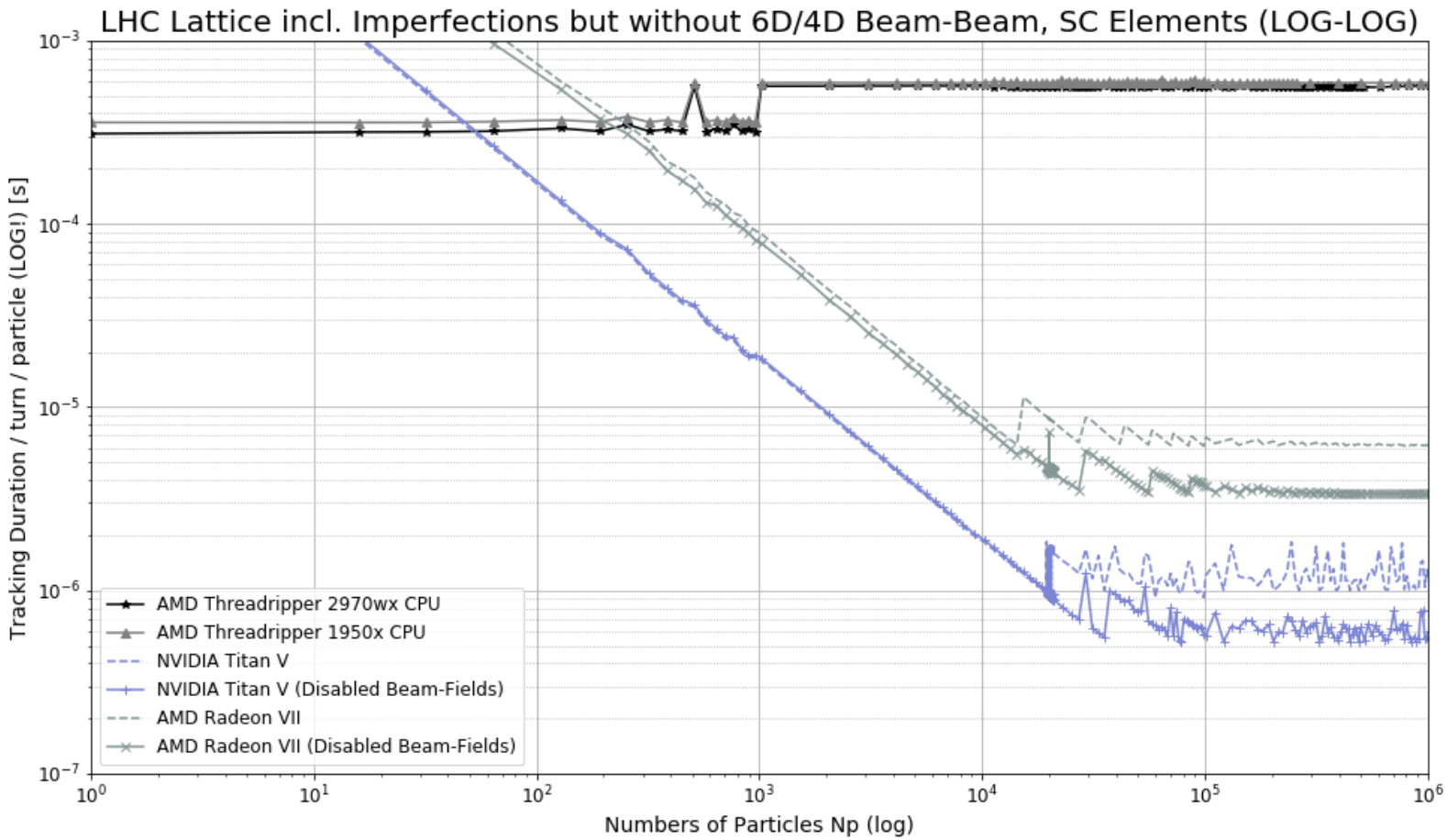
3 Beam-Beam Interactions & Space-Charge Effects

- `SixTrackLib` implements $4D$ and $6D$ beam-beam (BB) interactions using a weak-strong beam formulation[3]
- Frozen Space-Charge (SC) beam-elements share infrastructure with the BB implementation
  - Coasting `SpaceChargeCoasting`
  - Bunched `SpaceChargeQGaussianProfile`
  - Bunched `SpaceChargeInterpolatedProfile` using linear and cubic spline longitudinal interpolation (under development)
- `SpaceChargeInterpolatedProfile` uses API to assign external data to a number of beam-elements to share profile samples and interpolation parameters between SC elements

[3] G. Iadarola et al. CERN-ACC-NOTE-2018-0023 "6D beam-beam interaction step-by-step

3 Beam-Beam Interactions & Space-Charge Effects

- A closer look on the performance data $\Longrightarrow$
- BB and SC implementations impact run-time performance even if no elements are present in the lattice!



3 Beam-Beam Interactions & Space-Charge Effects

- Calculation of field components (according to a Gaussian distribution) and the complex error function (Faddeeva function) is shared between BB and SC elements

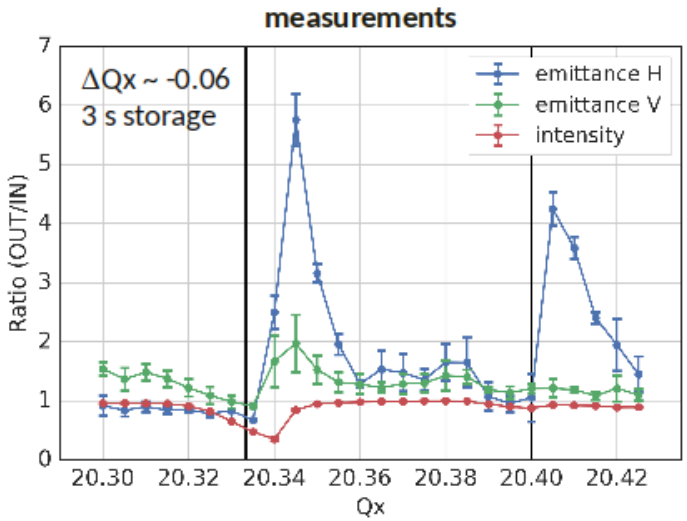
```
/* From: be_beamfields/faddeeva_cern.h */
SIXTRL_INLINE void cerrf( SIXTRL_REAL_T in_real, SIXTRL_REAL_T in_imag,
    SIXTRL_ARGPTR_DEC SIXTRL_REAL_T* SIXTRL_RESTRICT out_real,
    SIXTRL_ARGPTR_DEC SIXTRL_REAL_T* SIXTRL_RESTRICT out_imag )
{
    /* This function calculates the SIXTRL_REAL_T precision complex error fnct.
    based on the algorithm of the FORTRAN function written at CERN by K. Koelbig
    Program C335, 1970. See also M. Bassetti and G.A. Erskine, "Closed
    expression for the electric field of a two-dimensional Gaussian charge
    density", CERN-ISR-TH/80-06; */

    int n, nc, nu;
    SIXTRL_REAL_T a_constant = 1.12837916709551;
    SIXTRL_REAL_T xLim = 5.33;
    SIXTRL_REAL_T yLim = 4.29;
    SIXTRL_REAL_T h, q, Saux, Sx, Sy, Tn, Tx, Ty, Wx, Wy, xh, xl, x, yh, y;
    SIXTRL_REAL_T Rx [33];
    SIXTRL_REAL_T Ry [33]; } !!!

    x = fabs(in_real);
    y = fabs(in_imag);

    if (y < yLim && x < xLim){
        q = (1.0 - y / yLim) * sqrt(1.0 - (x / xLim) * (x / xLim));
        h  = 1.0 / (3.2 * q);
        nc = 7 + (int) (23.0 * q);
        xl = pow(h, (SIXTRL_REAL_T) (1 - nc));
        xh = y + 0.5 / h;
        yh = x;
        nu = 10 + (int) (21.0 * q);
        Rx[nu] = 0.;
        Ry[nu] = 0.;
        for (n = nu; n > 0; n--){
            Tx = xh + n * Rx[n];
            Ty = yh - n * Ry[n];
            Tn = Tx*Tx + Ty*Ty;
            Rx[n-1] = 0.5 * Tx / Tn;
            Ry[n-1] = 0.5 * Ty / Tn;
            }
        /* .... */
```

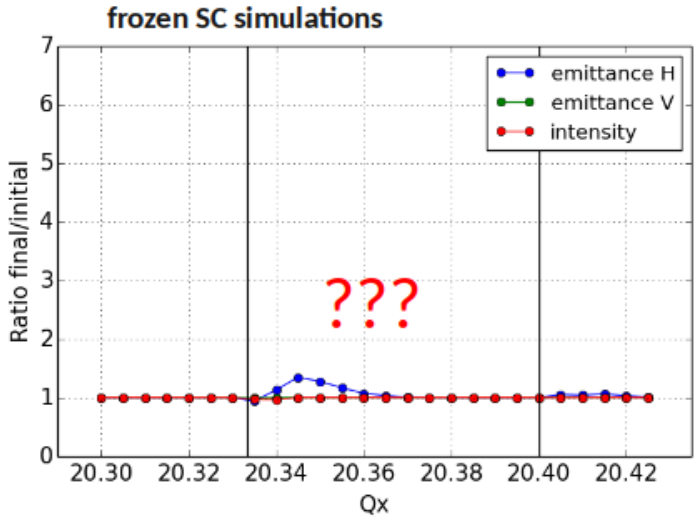
Large amount of thread-local data required → Effects Execution of Kernel even if the function is not actually called!

Data-Dependent Branching

Observations from CERN SPS experiment

- **Benchmark experiment**
  - Horizontal 3rd order resonance at Qx = 20.33 deliberately excited
  - Additional resonance observed at Qx = 20.40 (space charge driven)

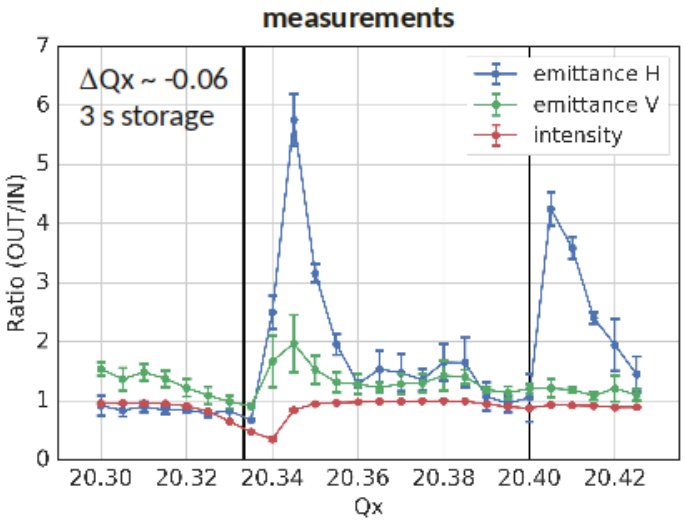




[2] H. Bartosik, F. Schmidt "Studies on Tune Ripple", 4th ICFA Mini-Workshop on SpaceCharge 2019, https://indico.cern.ch/event/828559/contributions/3528378

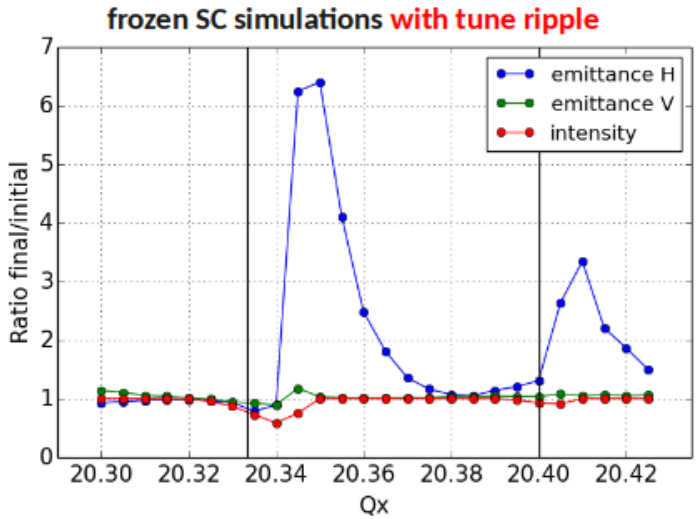
Observations from CERN SPS experiment

- **Benchmark experiment**
  - Horizontal 3rd order resonance at Qx = 20.33 deliberately excited
  - Additional resonance observed at Qx = 20.40 (space charge driven)
  - Simulations with **frozen potential** far from experiment unless SPS **tune ripple** from quadrupole power converters is taken into account



With PyORBIT 4: 5000 Teilchen ~ 4 Days

With SixTrackLib: 20e3 Teilchen ~ 4 Hours



Simulation over 130000 turns

After each turn: collect, update quadrupoles, push!

[2] H. Bartosik, F. Schmidt "Studies on Tune Ripple", 4th ICFA Mini-Workshop on SpaceCharge 2019, https://indico.cern.ch/event/828559/contributions/3528378

4 Integrating SixTrackLib with PyHEADTAIL

Beyond the single-particle treatment within SixTrackLib, model collective effects as "true" interaction between macro-particles via PyHEADTAIL[4]:

- accelerated on the GPU via (Py)CUDA
- self-consistent models for (e.g. 3D PIC/particle-in-cell) space charge, wake fields and feedback systems

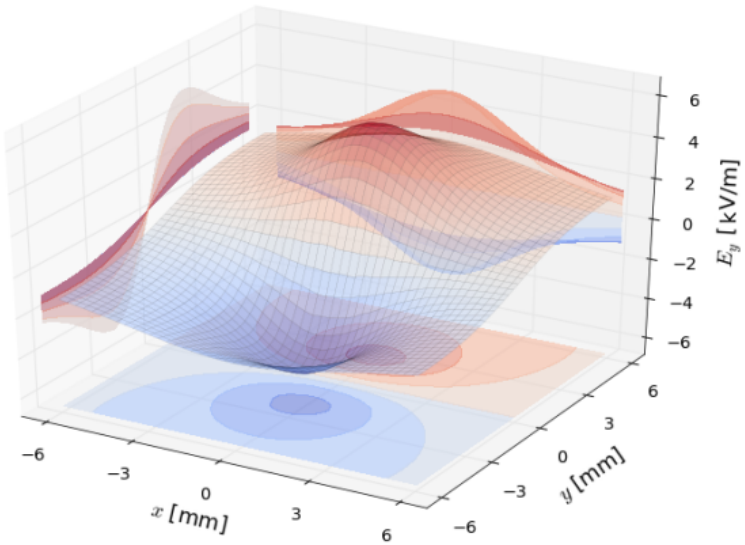


Figure: PIC space charge

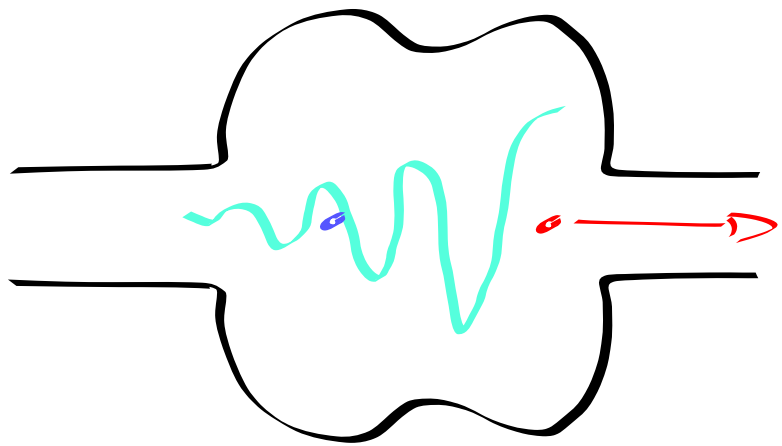


Figure: wake fields

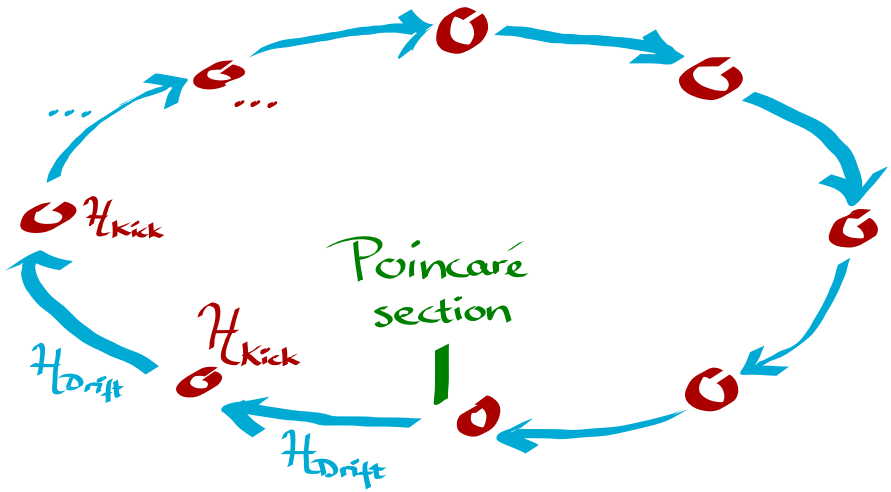
[4] https://github.com/PyCOMPLETE/PyHEADTAIL

4 Integrating SixTrackLib with PyHEADTAIL

Share particle memory between SixTrackLib and PyHEADTAIL:

1. use SixTrackLib's `track_line` API to advance particles through parts of accelerator lattice
2. expose particle coordinates on GPU via SixTrackLib's `get_particle_addresses` interface to apply kick in PyHEADTAIL

$\Longrightarrow$ alternating single- and multi-particle physics while *remaining* on GPU device memory!



Applications of SixTrackLib + PyHEADTAIL

90 deg stop-band

Interplay of coherent vs. incoherent resonances driven by space charge

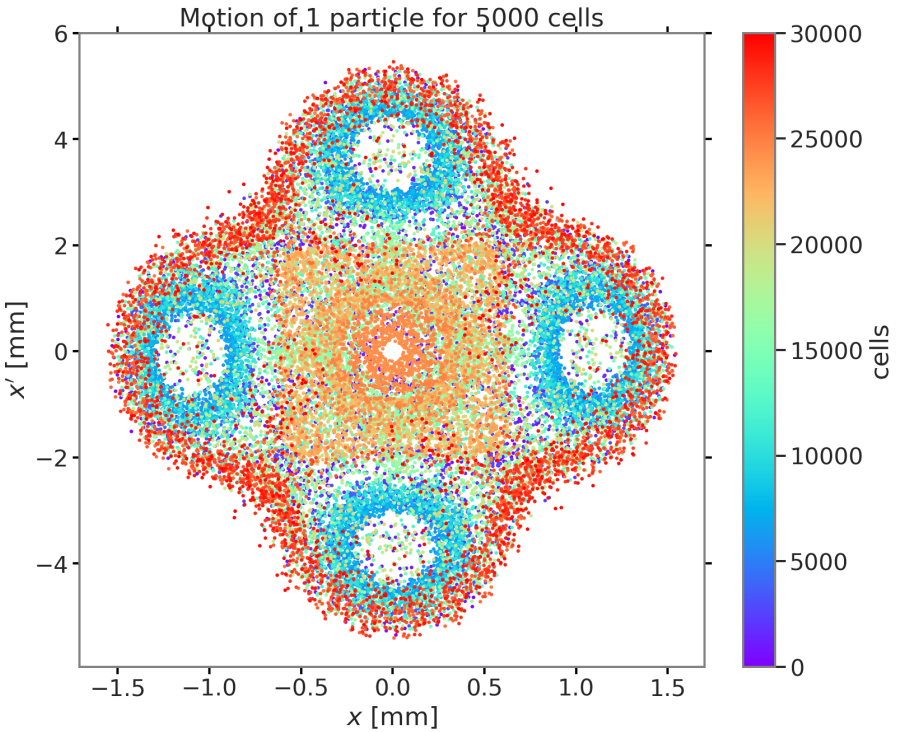


Figure: running 3D PIC in FODO

FAIR synchrotron SIS100

Beam loss studies with space charge and nonlinear magnet imperfections

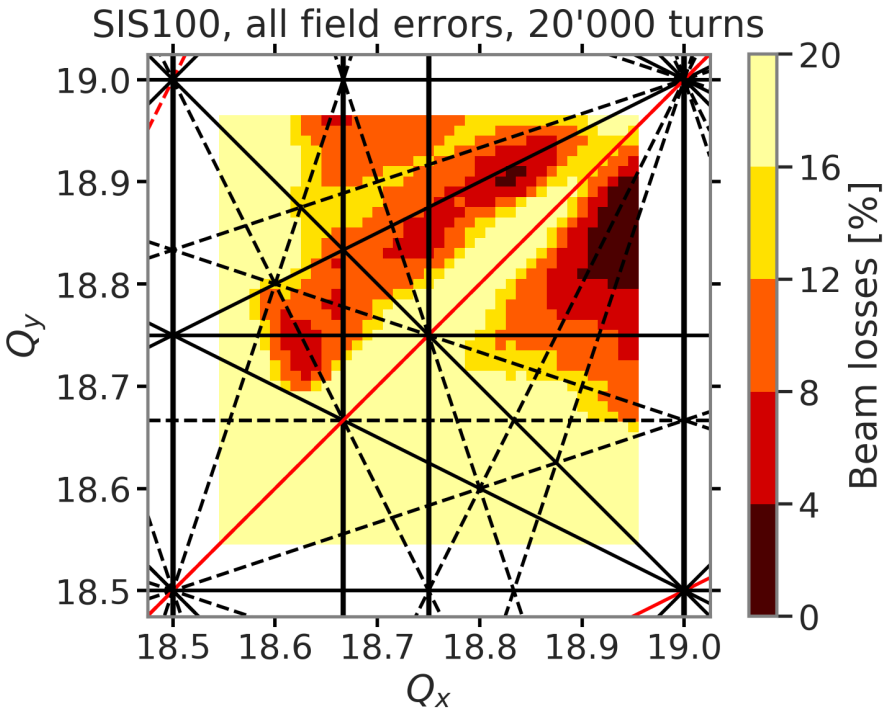
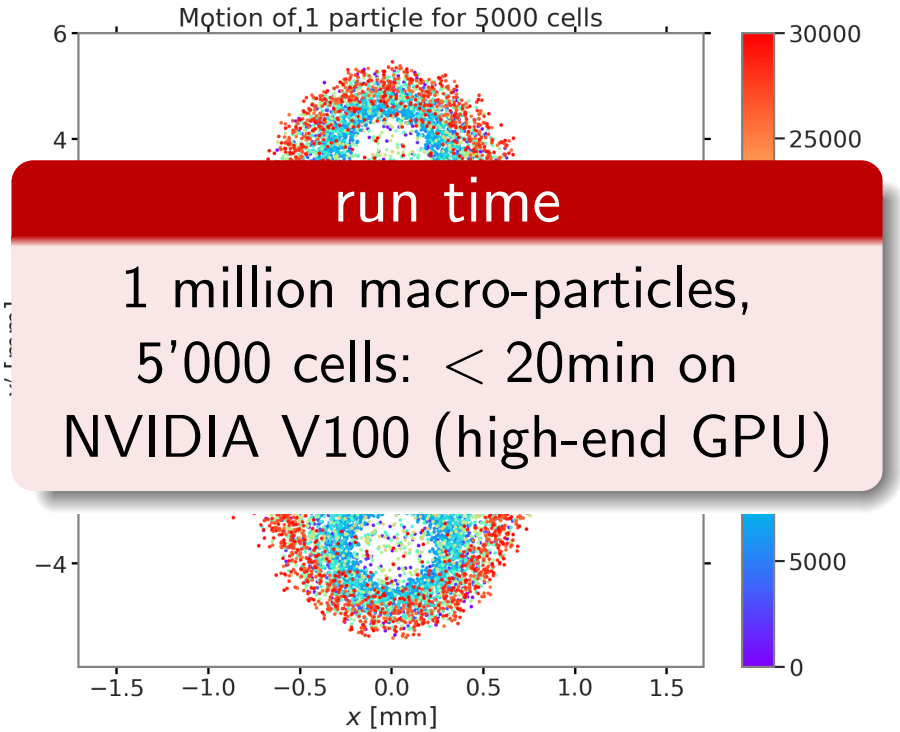


Figure: frozen SC in SIS100 lattice

Applications of SixTrackLib + PyHEADTAIL

90 deg stop-band

Interplay of coherent vs. incoherent resonances driven by space charge



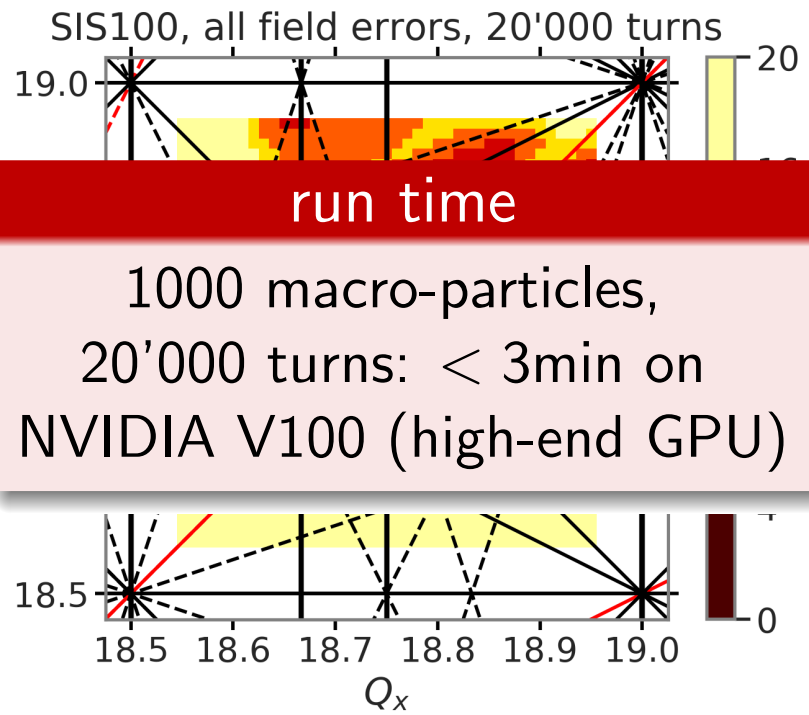
run time

1 million macro-particles, 5'000 cells: $<$ 20min on NVIDIA V100 (high-end GPU)

Figure: running 3D PIC in FODO

FAIR synchrotron SIS100

Beam loss studies with space charge and nonlinear magnet imperfections



run time

1000 macro-particles, 20'000 turns: $<$ 3min on NVIDIA V100 (high-end GPU)

Figure: frozen SC in SIS100 lattice

Thank You For Your Attention!

- `SixTrackLib` available from `https://github.com/SixTrack/sixtracklib`
- Presentation, Jupyter-Notebooks, Images, Data available via `https://github.com/martinschwinzerl/sixtracklib-presentations/tree/master/20200617_hss_section_meeting_sixtracklib`
- **Info:** Friday, June 19th, 2020, 14:30 s.t.: BE Seminar Talk about `SixTrackLib` $\longrightarrow$ `https://indico.cern.ch/event/929467/`