CERN

# Introducing SixTrackLib

## A Versatile, Hardware-Accelerated Single-Particle Tracking Library

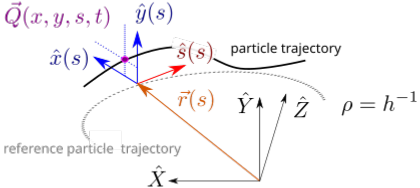
M. Schwinzerl[1], Riccardo De Maria[1], Giovanni Iadarola[1], Adrian Oeftiger[2]

[1]CERN, BE Department, ABP-HSS, [2]GSI/FAIR

PyHEP 2019 :: Abingdon, Uk

## Introduction :: A $6D$ Single Particle Tracking (Parallel) Library

1. $6D$: Particle motion through phase space $(x, p_x, y, p_y, \zeta, \delta)$




2. Single-Particle: non-interaction particles $p_i, p_k$ with $i \neq k < N_p$
3. Tracking: via symplectic (thin-lens) map $f_j$ for the beam-element at position $j$ in the lattice: $p_i\left(j+1\right) \leftarrow f_j\left(p_i\left(j\right)\right)$
4. Parallel: For $N_p \gg 1$: "embarrassingly" parallel problem
5. Library: independent of application, low barrier of entry, reusable, embedable, extensible (in contrast to established application SixTrack: `https://github.com/SixTrack/SixTrack`)

## Introduction :: Usage Scenarios For Such A Library

- Building-block for applications (i.e. user-generated simulations or even frameworks) that require tracking (for example `PyHEADTAIL`)
- Usable on PCs and Laptops with limited or no parallel computing capabilities (development & debugging!)
- Large-scale simulations (i.e. many particles, many turns) on dedicated HPC infrastructure
- Optimal usage of donated computing time (GPU and CPU) via LHC@Home volunteer project
  `http://lhcathome.web.cern.ch/projects/sixtrack`
- **Across these: same API/syntax**
  (i.e. without having to rewrite any user-code)
- **Regular users should not need any GPU/HPC knowledge**
  (but allow advanced users to tweak things)

# Introduction :: Current SixTrackLib Status

- Available from `https://github.com/SixTrack/sixtracklib`
- **Early stage**, intended for advanced users & selected studies
- In development for 18 months+ now
- Supports C99, C++11, and Python 3 → consistent API
- Supports Single threaded CPU (Auto-Vec), OpenCL 1.2 and Cuda
- Part of a larger ecosystem of libraries especially targeting Python:
  - cobjects: Specialised binary serialisation buffer/protocol `https://github.com/SixTrack/cobjects`
  - pysixtrack: Rapid prototyping Python-only implementation `https://github.com/SixTrack/pysixtrack`
  - sixtracktools: Access SixTrack IO files from Python `https://github.com/SixTrack/sixtracktools`

# Examples :: Track All Particles Until Turn On CPU

```
In [1]: import sixtracklib as st

        # Load particle data and the accelerator machine description from
        # a binary dump:

        particles = st.ParticlesSet.fromfile("./particles.bin")
        lattice = st.Elements.fromfile("./elements.bin")

        # Create a track-job instance
        job = st.TrackJob( lattice, particles )

        # Track all particles until they are in turn 100
        job.track_until( 100 )

        # Collect the particle state -> not strictly required on the CPU
        job.collect_particles()

        # particles now contains the updated state after tracking for 100 turns
        # ....
```

# Examples :: Track All Particles Until Turn On GPU (OpenCL)

```
In [1]: import sixtracklib as st

        # Like before, lead particles and machine description from binary dumps
        particles = st.ParticlesSet.fromfile( "./particles.bin" )
        lattice = st.Elements.fromfile( "./lattice.bin" )

        # We want to track now using OpenCL; get a list of supported devices
        !clinfo -l

Platform #0: NVIDIA CUDA
 +-- Device #0: TITAN V
 `-- Device #1: GeForce GT 1030
Platform #1: Intel(R) CPU Runtime for OpenCL(TM) Applications
 `-- Device #0: AMD Ryzen Threadripper 1950X 16-Core Processor
Platform #2: AMD Accelerated Parallel Processing
 `-- Device #0: Hawaii


In [2]: # Initialize the TrackJob to use the NVidia GT 1030 card
        job = st.TrackJob(lattice, particles, device="opencl:0.1")
        job.track_until( 100 )

        # Collect the particles -> now required
        job.collect_particles()
```

# Examples :: Lattice from MAD-X Sequence (via pysixtrack)

```
In [1]: from cpymad.madx import Madx
        import sixtracklib as st
        import pysixtrack
        from scipy.constants import e, m_p, c
        import numpy as np

        p0c = 6.0 * 1e9 # (P0 * c) [eV]
        Etot = np.sqrt( p0c**2 + ( m_p / e )**2 * c**4 ) * 1e-9 # [GeV]

In [2]: mad = Madx()
        mad.call(file="fodo.madx")
        mad.command.beam(particle='proton',energy=str(Etot))
        mad.use(sequence="FODO")

        # Setup the lattice from the MAD-X Sequence
        sis18 = mad.sequence.FODO
        ps_line, _ = pysixtrack.Line.from_madx_sequence(sis18)
        elements = st.Elements()
        elements.append_line(ps_line)

        # Setup the particles consistent with (P0*c)
        particles = st.Particles.from_ref(1000, p0c=p0c)

In [3]: # From here on, continue as before
        job = st.TrackJob(lattice, particles, device="opencl:0.1")
        job.track_until( 100 )
        job.collect_particles()
```

## Design Principles :: Overview

- Design Goal: separate the technical details (CPU, OpenCL, Cuda, parallel computing, HPC, etc.) from the physics
- $\Rightarrow$ Keep SixTrackLib extensible wrt. physics (even by end-users)
- $\Rightarrow$ Have a single implementation of the physics models (header only, C99, heavily abstracted & limited $\approx$ DSL)
- $\Rightarrow$ Keep SixTrackLib extensible wrt. supported architectures
- Design Goal: consistent API across languages & architectures
- Challenge: limit externally facing API surface, stability promises
- $\Rightarrow$ Focus SixTrackLib on tracking
- $\Rightarrow$ Move auxiliary components out of the library

# Implementation :: External Library `pysixtrack`

- Idea: minimal, pythonic (Python-only!) tracking implementation
- Place for prototyping & implementing new physics models
- Collect also I/O helper routines (cf. MAD-X example above)
- Strong focus on numerical precision and correctness (`mp.math`!)
- Slightly reduced focus on performance and scalability

```
In [1]: import pysixtrack as pyst
        particle_1=pyst.Particles()
        particle_1.x=1
        particle_1.y=1

        particle_2=particle_1.copy()

        belem1=pyst.elements.RFMultipole(knl=[.5,2,.2],ksl=[.5,3,.1])
        belem2=pyst.elements.Multipole(knl=el1.knl,ksl=el1.ksl)

        belem1.track(particle_1)
        belem2.track(particle_2)

        assert particle_1.compare(particle_2,abs_tol=1e-15), "should have same effect!"
```

# Implementation :: External Library `cobjects`

- Particles, beam-elements, output
  → need to be serialized, stored/restored, exchanged (Host↔Device)
- `cobjects` : binary protocol & buffer (Python3, `numpy`)
- C/C++ implementation available as part of SixTrackLib
- Allows objects to have nested structured members and vectors
- Allows for user-contributed (structured) data-types

```
In [1]: from cobjects import CField, CObject, CBuffer
        import numpy as np

        class VecObj(CObject):
            _typeid  = 1024 # to be coordinated across stored obj
            x = CField(0, 'real', default=1.0, alignment=8) # Scalar
            y = CField(1, 'real', default=0.0, length=4, alignment=8) # Fixed Size Vec
            # z ... dynamically sized vector of 3 * z_length elements
            z_length = CField(2, 'uint64', const=True, alignment=8)
            z = CField(3,'real', default=0.0, pointer=True, length='3*z_length', alignment=8)

            def __init__(self, z_length=None, **kwargs):
                if z_length is None:
                    z_length = 1
                super().__init__(z_length=z_length, **kwargs)
```

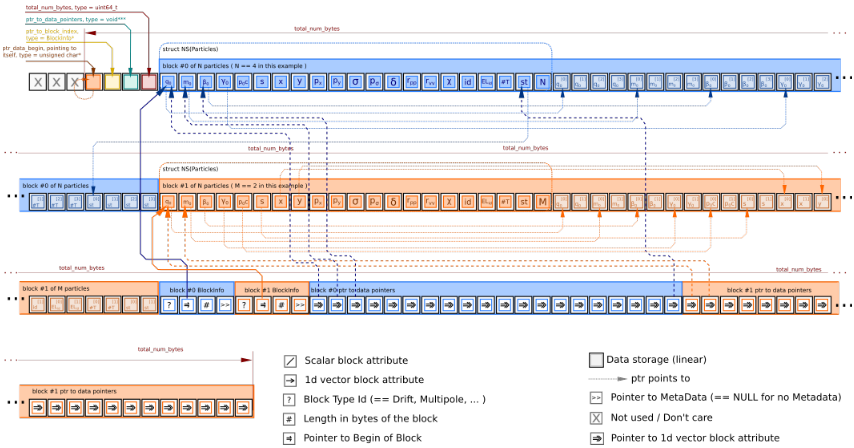
# Implementation :: Externally Library `cobjects` (cont.)

- Allows light-weight, zero-copy, in-place access to stored objects

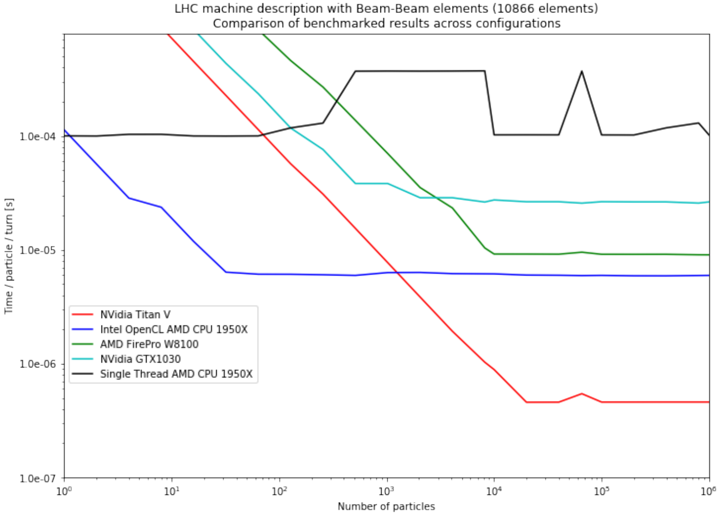
```
In [2]: buffer = CBuffer()
        obj1 = VecObj(cbuffer=buffer, z_length=10)
        obj2 = VecObj(cbuffer=buffer)

        assert len(obj1.z) == 30
        assert len(obj2.z) == 3

        alias_to_obj2 = buffer.get_object(1)
        alias_to_obj2.x = 5.0
        assert np.isclose(alias_to_obj2.x, obj2.x, atol=1e-16)
```

- Challenge: if a buffer of `cobjects` is moved, all stored pointers have to be "remapped"
- → `cobjects` buffers are intended to be used "sequentially"

# Implementation :: `cobjects` Storage Memory Layout



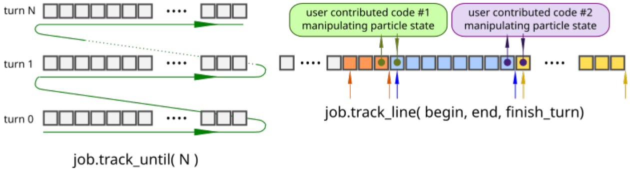
# Performance Analysis :: Full LHC lattice, OpenCL Backend



# Integrating SixTrackLib Into Programs :: Overview

SixTrackLib supports several different integration strategies. Ranging from "easily accessible" to "complex & invasive":

- Use `track_until()` & `collect_particles()` via TrackJob
- Use `track_line()` & manipulate particle state in-place




- Run-time compile and execute custom kernel function written in the header-only subset (currently only OpenCL, requires C99 interface)
- Integrate required functionality into SixTrackLib (C99,C++)
- Directly use the C99 header-only subset of SixTrackLib

# Example: Integration of SixTrackLib With PyCUDA

```
In [1]: import sixtracklib as st
        import numpy as np
        import pycuda
        from pycuda import gpuarray, driver, autoinit

        particles = st.ParticlesSet.fromfile("./particles.bin")
        lattice = st.Elements.fromfile("./lattice.bin")
        job = st.CudaTrackJob(lattice, particles)

        # Initialize the particles to some values on the host
        job.particles_buffer.get_object(0).x[:] = np.array( [-1.0, -2.0 ])

        # push particle state to the device
        job.push_particles()

In [2]: job.fetch_particle_addresses()
        ptr_particles_addr = job.get_particle_addresses(0) #0 .. only one particle set
        particles_addr = ptr_particles_addr.contents # particles_addr contains addr *on the device*

        print("num_particles   = {0:8d}".format(particles_addr.num_particles))
        print("x     begin at = {0:16x}".format(particles_addr.x))

num_particles   =        2
x     begin at =     7f6ca5000190
```

# Example: Integration of SixTrackLib With PyCUDA (cont.)

```
In [3]: # Create a PyCUDA GPUArray coinciding with our memory range
        cuda_x = pycuda.gpuarray.GPUArray(
                particles_addr.num_particles, np.float64, gpudata=particles_addr.x)

        # Modify the values via the cuda_x array
        new_values = np.array( [ 5.0, 6.0 ] )
        cuda_x[:] = new_values

In [4]: # Verify that the particle state has been modified
         job.collect_particles()
         print( job.particles_buffer.get_object(0).x )

[5. 6.]
```

- Allows in-place particle manipulation between calls to `track_line()`
- Similar implementation also available with `CuPY`
- This way of integration is an additional motivation to support both OpenCL and CUDA!
- But: corresponding integration with PyOpenCL tbd/wip

## What Could Possibly Go Wrong? (Cuda Edition)

- Cuda High-Level API context management is implicit ⟶ sharing contexts relies on conventions (Alternative: Driver API)
- PyCUDA, CuPY: context initialized & destroyed via Python
  SixTrackLib objects created and destroyed in-between
  - Inputs (particles, lattices): used by SixTrackLib
  - Device buffers, Output buffer: managed by SixTrackLib
- Coordinated device selection PyCUDA / CuPY ↔ SixTrackLib
- If selected devices mismatch, PyCUDA may try to be "helpful" ⟶ very slow device↔device mem cpy
- SixTrackLib currently only exposes default stream
  everything else: explicit device synchronisation
- ...
- **Beyond entry-level usage ⟶ Platform-level integration!**

## Conclusion & Outlook

- Writing a cross-platform, multi-language library that behaves like a proper Python module is challenging
- Approach chosen for SixTrackLib is feasible & performance numbers and feedback from users are encouraging
- Programming to least-common denominator & heavily relying on abstractions has its limitations $\longrightarrow$ Look into reflection & automatic code-generation from `pysixtrack`
- Goal: Using SixTrackLib as tracking backend to `SixTrack` and within the context of LHC@Home
- Goal: Simplify installation and deployment for Python users
- Goal: Optimise run-time performance and scalability
- Goal: Continue integration efforts with `PyHEADTAIL` et al

# Thank You For Your Attention!

# Backup Slides

# Example :: Tracking Map (Drift)

```
SIXTRL_INLINE NS(track_status_t) NS(Track_particle_drift)(
    SIXTRL_PARTICLE_ARGPTR_DEC NS(Particles)* SIXTRL_RESTRICT p,
    NS(particle_num_elements_t) const ii,
    SIXTRL_BE_ARGPTR_DEC const NS(Drift) *const SIXTRL_RESTRICT drift )
{
    typedef NS(particle_real_t) real_t;

    real_t const rpp    = NS(Particles_get_rpp_value)( p, ii );
    real_t const xp     = NS(Particles_get_px_value )( p, ii ) * rpp;
    real_t const yp     = NS(Particles_get_py_value )( p, ii ) * rpp;
    real_t const length = NS(Drift_get_length)( drift );
    real_t const dzeta  = NS(Particles_get_rvv_value)( p, ii ) -
                          ( ( real_t )1 + ( xp*xp + yp*yp ) / ( real_t )2 );

    NS(Particles_add_to_x_value)( p, ii, xp * length );
    NS(Particles_add_to_y_value)( p, ii, yp * length );

    SIXTRL_ASSERT( NS(Particles_get_beta0_value)( p, ii ) > ( real_t )0 );

    NS(Particles_add_to_s_value)( p, ii, length );
    NS(Particles_add_to_zeta_value)( p, ii, length * dzeta );

    return SIXTRL_TRACK_SUCCESS;
}
```

# Example :: C++ API Example

```cpp
#include "sixtracklib/sixtracklib.hpp"

int main()
{
    namespace st = sixtrack;

    st::Buffer particle_set( "./particles.bin" );
    st::Buffer lattice( "./lattice.bin" );

    // We have to be explicit about using the CPU for tracking
    st::TrackJobCpu job( particle_set, lattice );

    // Track until turn 100
    job.trackUntil( 100 );

    // Collect the particle state -> would not be needed for the CPU TrackJob
    job.collectParticles();

    // particle_set now contains the tracked data
    // ... Do something with the particles ...

    return 0;
}
```

# Example :: C99 API Example

```c
#include "sixtracklib/sixtracklib.h"

int main()
{
    NS(Buffer)* particle_set = NS(Buffer_new_from_file)( "./particles.bin" );
    NS(Buffer)* lattice      = NS(Buffer_new_from_file)( "./lattice.bin" );

    /* We have to be explicit about using the CPU for tracking */
    NS(TrackJobCpu)* job = NS(TrackJobCpu_new)( particle_set, lattice );

    /* Track until turn 100 */
    NS(TrackJob_track_until)( job, 100 );

    /* Collect the particle state ->
     * would not be needed for the CPU TrackJob */
    NS(TrackJob_collect_particles)( job );

    /* particle_set now contains the tracked data */
    /* ... Do something with the particles ... */

    /* Cleaning-Up */
    NS(TrackJob_delete)( job );
    NS(Buffer_delete)( particle_set );
    NS(Buffer_delete)( lattice );

    return 0;
}
```