GPGPU Accelerated Beam Dynamics
Interfacing PyHEADTAIL with SixTrackLib

BE-ABP-HSC

Wednesday 23rd January, 2019

AUTHOR:

Meghana Madhyastha

SUPERVISORS:

Adrian Oeftiger

Haroon Rafique

CERN openlab

Abstract

Simulations of beam dynamics vastly profit from parallelisation with high performance computing techniques. The two simulation libraries SixTrackLib and PyHEADTAIL are GPGPU accelerated. The former models non-linear particle tracking while the latter models wakefields and space charge using linear tracking. In this project the goal is to explore concepts to bridge the gap between the two simulation codes, thus a potential user can use them as a single simulation code while being abstracted from the interfacing details. Advanced non-linear particle tracking capability is introduced into PyHEADTAIL via SixTrackLib. Therefore collective effects such as space charge and wakefield interactions can be simulated more accurately in view of resonance dynamics and beam instability applications. As an introductory task wakefield calculations are ported to the GPU in order to reduce total simulation time. A major part of the project is to interface PyHEADTAIL with SixTrackLib so that a user of PyHEADTAIL can utilise the non-linear tracking functionality of SixTrackLib via an API in python. This interfacing mechanism works on both the CPU and GPU and the user has the option to switch between the two.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the fundamental guiding principles of CERN is to advance fundamental research in particle physics and accelerator physics. Particles when accelerated to high enough energies exhibit interesting properties and interactions. In high energy physics, a beam of particles is accelerated to high energies. This calls for interesting analyses on these particles to ensure that they behave as they should be. In order to be able to test new theories and to further deepen understanding of some of these physical phenomena, one can leverage computing technology to run simulations. For accelerator physics, this is where beam tracking codes come in. There are many codes that allow physicists to simulate various aspects of accelerated particle beams. A few examples are PyECloud for electron cloud simulation, PyORBIT for space charge modelling, MADX, Sixtrack and PTC for particle tracking and design codes to name a few.

There is a plethora of beam tracking codes written in different languages, with varying design principles. Moreover, these codes operate under different energy ranges and sometimes different unit conventions for physical quantities. This is furthermore complicated by the fact that they have different functionalities which some overlap at times. The users of these codes often need to use functionalities present in different codes. Thus, as a user, having to use multiple codes for different parts of a sample simulation case can become a complex task because they might not work well together. This necessitates the development of a unified framework for beam tracking codes which enables interoperability for users. This is the ultimate and broad final goal this project builds a foundation for. Such a framework would require consolidation of the individual beam tracking codes which are currently used. In this internship project, SixTrackLib is interfaced with PyHEADTAIL so that a user of PyHEADTAIL can leverage non-linear tracking from SixTrackLib. This is described in detail in chapter 3. Such an interface allows users of PyHEADTAIL to use SixTrackLib.

PyHEADTAIL is a numerical n-body simulation code for modelling macro-particle beam dynamics with collective effects of charged beams [5, 3, 7]. It has been written primarily in python but also uses cython and CUDA. For more information on collective effects, one can refer to [8]. PyHEADTAIL was started by Dr Kevin Li and extensively developed by Dr Adrian Oeftiger as part of his PhD Thesis [6]. For more information about PyHEADTAIL and its usage, the reader is referred to [6]. The specific module in PyHEADTAIL which has been extensively used for the first part of the project is the wakefield. Wakefields occur due to delayed currents in the accelerator aperture, where a trailing electromagnetic field is induced

by a charge traveling down a pipe with finite conductivity and/or geometrical discontinuities. PyHEADTAIL supports parallelisation using the GPU to speed up simulations. It uses a context manager class [4] to switch between the CPU and GPU usage as a form of modularisation and to ensure that processor specific details are abstracted from the rest of the source code. Parallelisation over the particles is based on an object-of-arrays data structure: the coordinates and momenta of each particle correspond to an index within an array of a length equal to the number of particles.

SixTrackLib is a particle tracking library written primarily in C [2]. SixTrackLib's development arose from the need to refactor Sixtrack for reasons of maintainability and to extend its hardware support to GPU architectures. Sixtrack is a beam tracking code written in fortran with a long and diverse history of development. SixTrackLib supports GPU parallelisation. Here, parallelisation over the particles is based on an array-of-objects data structure due to the distributed computing requirements (e.g. on BOINC [1]): particles are thus separate instances of structs containing the coordinates and momenta values. SixTrackLib offers advanced non-linear tracking algorithms but lacks a python interface yet.

PyHEADTAIL, although fairly comprehensive and quite extensive in the features it provides, has some limitations. One limitation is that it only supports linear transverse betatron tracking. The purpose of simulation codes is to be able to model real physical systems as accurately as possible. Modelling resonant beam behaviour accurately with PyHEADTAIL's space charge suite requires non-linear tracking. SixTrackLib can provide this functionality. The major part of the present project is to provide users with an API to use SixTrackLib's non-linear tracking from PyHEADTAIL. This poses a number of challenges; the data structures with the memory addressing of particles are quite different and parallelisation is implemented differently according to the data structures.

Chapter 2

Speeding up Wakefields

The first task of the project was to get familiarized with PyHEADTAIL. Simulating a bunch of particles in a Proton Synchroton machine with wakefields on the GPU was the first task. The simulations were profiled using cProfile and nvprof to identify bottlenecks. We use the NVIDIA Tesla V100 (16GB) GPU. Detailed specifications can be viewed at `https://www.nvidia.com/en-us/data-center/tesla-v100/`. The following are the simulation parameters:

- Number of macro-particles $= 10^6$
- Number of slices for the wakefield $= 10^4$
- Number of turns $= 10^4$

Each turn consists of 1 transverse (linear) tracking node (i.e. one segment), 1 longitudinal non-linear tracking node and 1 wakefield interaction node.

The particle beam is sliced by uniform bin slicing (`UniformBinSlicer` in PyHEADTAIL). The wakefield effect is generated via a circular broad-band resonator. Before the tracking computations, the particles are generated on the CPU. Next, the context manager is invoked, the particles are transferred to the GPU and henceforth the rest of the tracking computations are performed on the GPU.

Profiling is a form of dynamic program analysis that measures the space (memory) or time complexity of a program, the usage of particular instructions or the frequency and duration of function calls. Profiling information serves to aid program optimisation. Using nvprof for CUDA GPU profiling and cProfile for python profiling (from the CPU perspective), the simulations are profiled and bottlenecks are identified.

In the rest of the section, the profiling results of wakefield simulations are displayed and analysed. First, the simulations (particle tracking and wakefield) are run with the master branch of PyHEADTAIL. This is then compared to a recent pull request [1] aimed at making the central convolution part of the wakefield interaction more efficient. Finally, it is compared to my pull request [2] which is the result of the first task in the internship. In this branch, convolution and all other wakefield calculations are done on the GPU, implemented in a custom kernel.

The total timings of the simulation run time are displayed in the caption of the tables.

[1] `https://github.com/PyCOMPLETE/PyHEADTAIL/pull/82`
[2] `https://github.com/megh1241/PyHEADTAIL/commit/c914a7dc92489fc511d31b691f3ca75b8b087acc`

2.1 Running the master branch

Wakefield calculations for a beam of particles with parameters mentioned above were run and profiled with the current PyHEADTAIL master branch[3]. Table 2.1 shows the results of cProfile. Here, 'ncalls' is the number of times a function is called.

ncalls	cumulative time(sec)	filename:lineno(function)
10000	111.203	wakes.py:119(track)
2070033	25.309	gpuarray.py:162(__init__)
20000	6.518	gpu_wrap.py:780(convolve)

Table 2.1: cProfile Log: Master branch
Total Simulation Time: 176.09s

For more detailed logs, one can refer to my repository[4].

2.2 Wakefield on GPU I: FFT and Inverse FFT

This experiment will involve a modified version of the wakefield where a part of the computations are carried out on the GPU. Specifically, convolution of the wake function with the slice centroid positions is done on the GPU by computing a Fourier transform of each of them, multiplying these and then computing the inverse Fourier transform of the result. This has already been implemented using the cuFFT library (CUDA's library for performing Fast Fourier Transforms) in a PyHEADTAIL pull request[5].

ncalls	cumulative time(sec)	filename:lineno(function)
10000	199.872	wakes.py:119(track)
2070033	43.434	gpuarray.py:162(__init__)
20000	41.344	gpu_wrap.py:780(convolve)
20045	22.585	cufft.py:206(cufftPlanMany)

Table 2.2: cProfile Log: FFT approach
Total Simulation Time: 210.56s

As we can see from the above table, convolve takes 43.4 seconds which is an increase from 6.5 seconds on the master branch. The reason for this is that the time taken to set up cuFFT is an additional cost. For small arrays (array lengths of the order 10^4), the FFT approach does not offer significant improvements in terms of time taken to run the simulations and rather slows down the total simulation time.

[3] https://github.com/PyCOMPLETE/PyHEADTAIL/commit/f9c94b19398148f7bf3d9c6c96adc96b930990b9
[4] https://github.com/megh1241/CERN2018
[5] https://github.com/PyCOMPLETE/PyHEADTAIL/pull/82

2.3 Wakefield on GPU II: time-domain convolution on the GPU with one-time slicing

The simulations were profiled, this time using regular time-domain convolution on the GPU. For implementation details please refer to the github commit[6].

ncalls	cumulative time(sec)	filename:lineno(function)
10000	78.637	wakes.py:119(track)
2070033	20.504	gpuarray.py:162(__init__)
20000	1.150	gpu_wrap.py:780(convolve)

Table 2.3: cProfile Log: Convolution on GPU
Total Simulation Time: 145.93 seconds

We see here that we achieve a speedup of around 20% in terms of time taken for the entire simulation to occur.

2.4 Results at a glance

In the previous sections, the profiling results pertaining to a single simulation case were displayed and analyzed. It is also useful to see a comparison across simulation cases. That is, it is useful to observe how the new implementation compares with the current master branch as can be seen in Figure 2.1. It is clear that we obtain a decrease in simulation times, both for the entire tracking time and for the individual wake functions. To be precise, we see a relative performance gain (in terms of time taken to execute) by 17% in the overall simulation of a bunch of particles on the PS with wakefields and 29.2% gain in the time taken for the computation of the wakefields only, given the convolution array length of 10^4 slices.

[6] https://github.com/megh1241/PyHEADTAIL/commit/c914a7dc92489fc511d31b691f3ca75b8b087acc

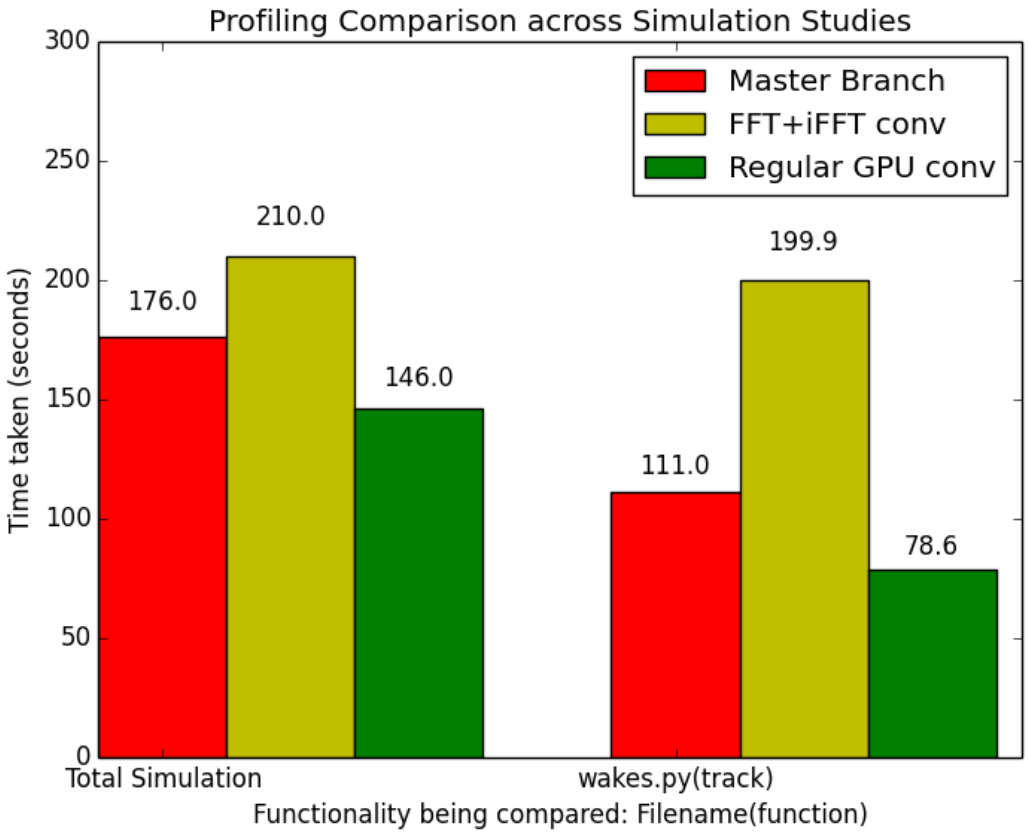

Figure 2.1: cProfile Results

Chapter 3

Bridging PyHEADTAIL with SixTrackLib

As briefly mentioned in the previous sections, SixTrackLib provides non-linear tracking. PyHEADTAIL is used to simulate collective effects. However, the simulations would be more accurate with non-linear tracking.

SixTrackLib has a format for storing particles in a structure called st_Blocks. This structure stores pointers to the particle attributes in a non trivial way. They store pointers which themselves contain pointers to the actual location of the particle attributes. In the current setup, these particle buffers must be created on the CPU.

3.1 Comparing physical quantities and variables

There are differences in naming physical quantities in PyHEADTAIL and SixTrackLib. The identification of variables is shown in Table 3.1. While SixTrackLib allows each macro-particle to have an individual Lorentz β_i, PyHEADTAIL assumes a rigid beam in paraxial approximation. Hence all macro-particles assume the beam restframe Lorentz β. With this assumption, the longitudinal coordinate $s - \beta c t$ (with s the path length around the accelerator and t the independent time parameter) becomes the same in both codes. Positions in the horizontal plane x and the vertical plane y are measured in meters in both codes. Transverse momenta are defined by $x' = P_x / p_0$ where P_x denotes the canonically conjugate momentum to x and $p_0 = m\beta\gamma c$ the total momentum (introducing the Lorentz γ, the mass m of the particle and the speed of light c). Note that under the above approximations, p_0 is the same for all particles and equal to the restframe momentum. Hence, the horizontal momenta px and xp denote the same quantity in both codes (and likewise for the vertical plane).

SixTrackLib	PyHEADTAIL
particle_id	id
$q0$	charge
mass0	mass
beta0	beta
gamma0	gamma
p0c	p0*c
x	x
px	xp
y	y
py	yp
delta	dp
sigma	z
psigma	$\frac{(E-E_0)}{\beta_0 p_0 c}$
rpp	$\frac{P}{P_0}$
rvv	$\frac{\beta}{\beta_0}$

Table 3.1: Comparison of physical quantities in PyHEADTAIL vs. SixTrackLib

3.2 Overview

Figure 3.1 shows a block diagram to visualize the basic interfacing prototype.

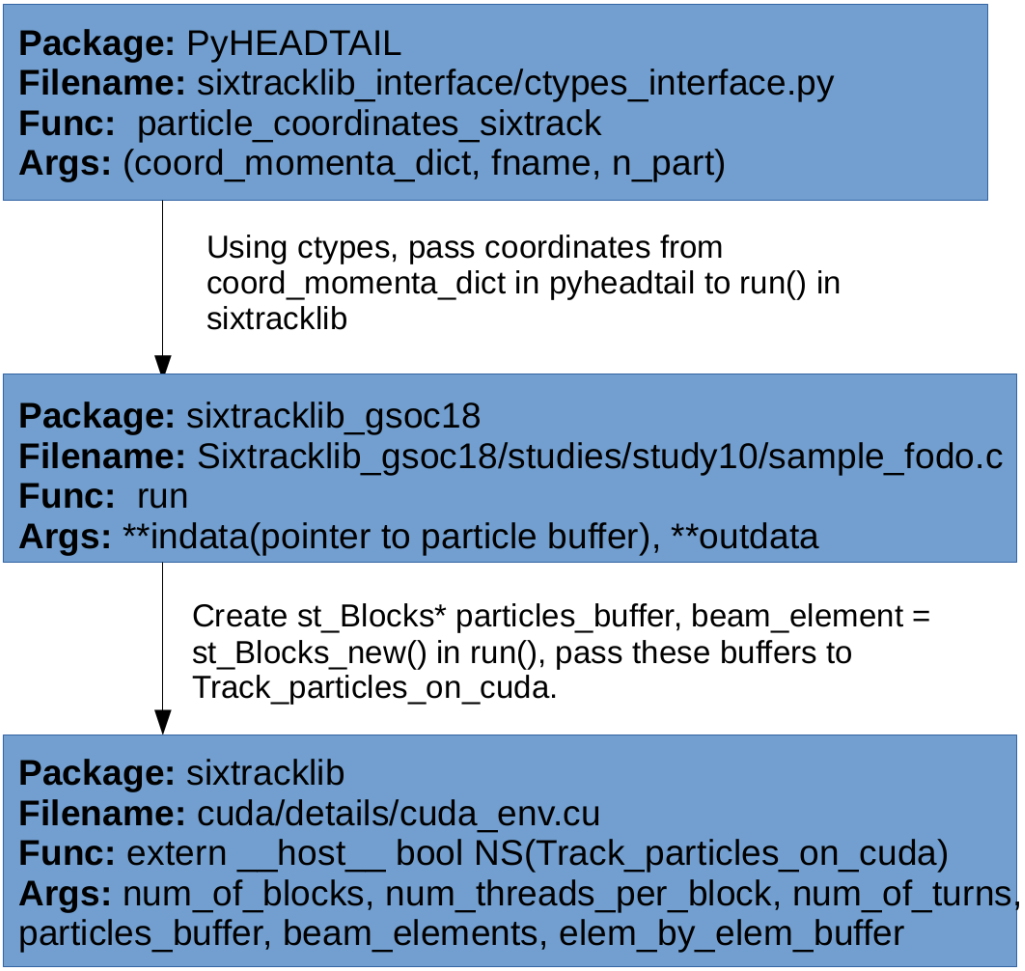

Figure 3.1: PyHEADTAIL-SixTrackLib bridge

The first step is creation of a particle bunch in PyHEADTAIL (not shown in the figure). Every particle bunch in PyHEADTAIL has a dictionary containing position and momentum information about the particles. This information is stored in a dictionary format and is called coord_momenta_dict. It can be procured via a function call on a Particles object in PyHEADTAIL. The position and momentum arrays are then passed to the interface method "particle_coordinates_sixtrack". This uses ctypes to convert the momenta and position attributes to ctypes pointers which are then passed to the C function "run" in SixTrackLib_gsoc18/studies/study10/sample_fodo.c. This in turn calls the SixTrackLib tracking function Track_particles_on_CUDA.

We investigate two cases:

- **CASE 1: PYHEADTAIL particle on the CPU**
  SixTrackLib particle buffers are created. Particle coordinates from pyheadtail are transferred to the particle buffers using ctypes, call "st_Track_particles_on_CUDA" in SixTrackLib which essentially takes these particle buffers, transfers them to the device and performs a memory remapping. Once the tracking has finished, the buffers are once again transferred to a CPU array and returned.

- **CASE 2: PYHEADTAIL particle on the GPU**
  Empty particle buffers still have to be created on the CPU because there is currently no API available to directly create store the attributes in the structured memory format on the GPU. Thus, one should create an empty particle buffer and transfer it to "st_Track_particles_on_CUDA" in SixTrackLib. Along with this a pointer to the gpuarray containing particle coordinates is passed from PyHEADTAIL. In "st_Track_particles_on_CUDA" the contents of the gpuarray pointer are copied to a structured buffer on the device. One must create a custom CUDA kernel for the memory copy.

These steps are better understood with sample pseudocode.

3.3 CASE 1: PYHEADTAIL particles on the CPU

The sequence of functions that are called on the CPU and GPU are described below. The source code containing these functions can be viewed on github[1].

3.3.1 Sequence of events on the CPU

The following occurs in "run" in sample_fodo.c.

1. Create st_Blocks* particles_buffer

2. st_Particles* particles = st_Blocks_add_particles(Set particles→ x , particles→ px,etc)

3. call st_Blocks_serialize. This results in structured memory with a double level of pointer indirection.

4. Call "st_Track_particles_on_CUDA" passing particles_buffer which is on the CPU.

```
bool __host__ NS(Track_particles_on_CUDA)(
    int const num_of_blocks,
    int const num_threads_per_block,
    SIXTRL_UINT64_T const num_of_turns,     NS(Blocks)* SIXTRL_RESTRICT particles_buffer,
    NS(Blocks)* SIXTRL_RESTRICT beam_elements,
    NS(Blocks)* SIXTRL_RESTRICT elem_by_elem_buffer )
```

[1] https://github.com/rdemaria/SixTrackLib_gsoc18/tree/master/studies/study10

3.3.2 Sequence of events on the GPU

The source code containing these functions can be viewed at another github repository [2]. The following occurs in st_Track_particles_on_CUDA.

1. Transfer contents from structured particles_buffer to the host buffer.

```
host_particles_data_buffer = NS(Blocks_get_data_begin)(  particles_buffer )
```

2. The CUDA buffer is created on the GPU and the contents of the host buffer are transferred to this CUDA buffer.

```
CUDAMalloc( ( void** )&CUDA_particles_data_buffer, particles_buffer_size );
CUDAMemcpy( CUDA_particles_data_buffer, host_particles_data_buffer, particles_buffer_size
    , CUDAMemcpyHostToDevice );
```

3. Memory is remapped to make a structure out of the device buffer.

```
Track_remap_serialized_blocks_buffer<<< num_of_blocks, num_threads_per_block >>>
```

4. The FODO is computed.

```
Track_particles_kernel_CUDA<<< num_of_blocks, num_threads_per_block >>>(num_of_turns,
    CUDA_particles_data_buffer, CUDA_beam_elements_data_buffer,
    CUDA_elem_by_elem_data_buffer, CUDA_success_flag );
```

5. The buffer contents now on the CUDA buffer are copied back into the host buffer.

```
CUDAMemcpy( host_particles_data_buffer, CUDA_particles_data_buffer, particles_buffer_size
    , CUDAMemcpyDeviceToHost );
```

6. The contents of this buffer are unserialized on the GPU.

```
NS(Blocks_unserialize)(particles_buffer, host_particles_data_buffer )
```

[2] https://github.com/martinschwinzerl/SixTrackLib/blob/master/SixTrackLib/CUDA/details/CUDA_env.cu

3.4 CASE 2: PYHEADTAIL particles on the GPU

The sequence of functions that are called on the CPU and GPU are described below. The newly introduced actions by me to avoid the GPU-CPU-GPU transfer of the particles data are indicated in red.

3.4.1 Sequence of events on the CPU

The following occurs in "run" in sample_fodo.c.

1. Create st_Blocks* particles_buffer
2. st_Particles* particles = st_Blocks_add_particles(Set particles→ x , particles→ px,etc)
3. call st_Blocks_serialize. This results in structured memory with a double level of pointer indirection.
4. Call "st_Track_particles_on_CUDA" passing particles_buffer which is on the CPU. We note line 7 is the extra parameter (which is different from the CPU particle case)

```
1 bool __host__ NS(Track_particles_on_CUDA)(
2     int const num_of_blocks,
3     int const num_threads_per_block,
4     SIXTRL_UINT64_T const num_of_turns,     NS(Blocks)* SIXTRL_RESTRICT particles_buffer,
5     NS(Blocks)* SIXTRL_RESTRICT beam_elements,
6     NS(Blocks)* SIXTRL_RESTRICT elem_by_elem_buffer ),
7     double* pyheadtail_ptr[]
```

3.4.2 Sequence of events on GPU

The source code containing these functions can be viewed on another github repository[3]. The following occurs in st_Track_particles_on_CUDA.

1. Transfer contents from structured particles_buffer to the host buffer.

```
1 host_particles_data_buffer = NS(Blocks_get_data_begin)(  particles_buffer )
```

2. The CUDA buffer is created on the GPU and the contents of the host buffer are transferred to this CUDA buffer.

```
1 CUDAMalloc( ( void** )&CUDA_particles_data_buffer, particles_buffer_size );
2 CUDAMemcpy( CUDA_particles_data_buffer, host_particles_data_buffer, particles_buffer_size
      , CUDAMemcpyHostToDevice );
```

3. Memory is remapped to make a structure out of the device buffer.

```
1 Track_remap_serialized_blocks_buffer<<< num_of_blocks, num_threads_per_block >>>
```

[3] https://github.com/martinschwinzerl/SixTrackLib/blob/master/SixTrackLib/CUDA/details/CUDA_env.cu

4. The particle buffer on the gpu is copied from pyheadtail to SixTrackLib buffer

```
Copy_buffer_pyheadtail_SixTrackLib<<<num_of_blocks, num_threads_per_block >>>(
    CUDA_particles_data_buffer, pyheadtail_ptr, CUDA_success_flag )
```

5. The FODO is computed.

```
Track_particles_kernel_CUDA<<< num_of_blocks, num_threads_per_block >>>(num_of_turns,
    CUDA_particles_data_buffer, CUDA_beam_elements_data_buffer,
    CUDA_elem_by_elem_data_buffer, CUDA_success_flag );
```

6. The particle buffer is copied from SixTrackLib buffer on gpu to pyheadtail buffer on gpu

```
Copy_buffer_pyheadtail_sixtracklib<<<num_of_blocks, num_threads_per_block >>>(
    CUDA_particles_data_buffer, pyheadtail_ptr, CUDA_success_flag )
```

The source code for the above two copy kernels is given below.

- Copy_buffer_pyheadtail_SixTrackLib

```
__global__ void Copy_buffer_pyheadtail_SixTrackLib(
    unsigned char* __restrict__ particles_data_buffer,
    double*  __restrict__ py_particles_buffer[],
    int64_t*       __restrict__ ptr_success_flag
){
        NS(Blocks) particles_buffer;
        NS(Blocks_preset)( &particles_buffer );
        NS(Blocks_unserialize_without_remapping)( &particles_buffer,
    particles_data_buffer );

        NS(BlockInfo)* ptr_info  = NS(Blocks_get_block_infos_begin)( &particles_buffer );
        NS(Particles)* particles = NS(Blocks_get_particles)( ptr_info );
        size_t num_of_particles  = NS(Particles_get_num_particles)( particles );

        memcpy( NS(Particles_get_x)( particles ),
             py_particles_buffer[0],
             num_of_particles * sizeof( double )
             );
        memcpy( NS(Particles_get_px)( particles ),
             py_particles_buffer[1],
             num_of_particles * sizeof( double ));
}
```

- Copy_buffer_SixTrackLib_pyheadtail

```
__global__ void Copy_buffer_SixTrackLib_pyheadtail(
    unsigned char* __restrict__ particles_data_buffer,
    double*  __restrict__ py_particles_buffer[],
    int64_t*  __restrict__ ptr_success_flag
){
        NS(Blocks) particles_buffer;
        NS(Blocks_preset)( &particles_buffer );
        NS(Blocks_unserialize_without_remapping)( &particles_buffer,
    particles_data_buffer );

        NS(BlockInfo)* ptr_info  = NS(Blocks_get_block_infos_begin)( &particles_buffer );
        NS(Particles)* particles = NS(Blocks_get_particles)( ptr_info );
        size_t num_of_particles  = NS(Particles_get_num_particles)( particles );
        memcpy( py_particles_buffer[0], NS(Particles_get_const_x)( particles ),
             num_of_particles * sizeof( double )
             );
        memcpy( py_particles_buffer[1], NS(Particles_get_const_px)( particles ),
             num_of_particles * sizeof( double )
```

Conclusion and Discussions

In the first part of this internship, the wakefield computations of PyHEADTAIL have successfully been ported to the GPU. Specifically, a custom kernel has been implemented with a speed-up of the wakefield computation by 29.2% for 10^4 slices. This involves fixing constant slice positions at the beginning of the simulation.

During profiling of PyHEADTAIL's wakefield and linear transverse tracking simulations, I observed that the overall GPU throughput is only around 10% for my numerical values. There seem to be at least three ingredients responsible for this finding, they are briefly discussed in order of descending impact.

First of all, the transverse tracking consumes quite some time as most of the instructions are in python: correspondingly, there is considerable overhead on the creation of PyCUDA's GPUArray instances as well as every single instruction such as $a+b$ triggers a kernel call. For a more efficient GPU utilisation, the inner loop statements of the transverse tracking should be carried out in a single kernel. In order to avoid duplicate code for the implemented physics, a similar macro embedding set-up as implemented in SixTrackLib could be devised.

Secondly, functions which are part of the CUDA Thrust library show up with quite some impact in the profiling results. They are related to the slicing algorithm used in the wakefields. This part could be investigated more carefully for potential bottlenecks, as the Thrust library is addressed via the ctypes library.

Finally, from the nvprof logs it becomes obvious that some of PyHEADTAIL's pmath calculations such as pow are not very efficient on the GPU as they consume quite some time in the profiling results.

Nevertheless, the GPU results still run considerably faster than PyHEADTAIL on the CPU.

In the second part of the internship, the tracking of a FODO cell in SixTrackLib has been integrated into PyHEADTAIL. The concepts explored demonstrate how the advanced non-linear tracking capabilities of SixTrackLib can be integrated into PyHEADTAIL. This enables simulations to cover simultaneously collective effects and non-linear tracking on both CPU and GPU architectures.

In order to fully integrate SixTrackLib into PyHEADTAIL, there are a few considerations to be made. Currently, the memory structure in both codes is very different. By unifying the particle memory description in both codes one could avoid costly memory copies which can be quite inefficient.

In order to make use of memory striding, PyHEADTAIL's coordinate and momentum arrays are more useful on a single computing device (GPU), while SixTrackLib's single particle instances with coordinate and momentum attributes are more suited for the distributed computing context it is often employed in. For our purposes, a new initialisation of SixTrackLib's particle structures could be benefitial, where the respective attributes just point to continuous entries in PyHEADTAIL's allocated particle position and momentum arrays.

The basic goal behind the present project is to explore approaches to interface PyHEADTAIL and SixTrackLib. Based on such an interface, a user of PyHEADTAIL should be able to use the SixTrackLib functionality in a simulation script seamlessly without understanding the implementation details of SixTrackLib. We achieved a step towards this by being able to run the FODO lattice within SixTrackLib from a PyHEADTAIL script. A natural next step after the present project would be to extend this functionality to a general machine layout, making use of the foreseen availability of a SixTrackLib-MADX interface.

Bibliography

[1] David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004. 2

[2] Riccardo de Maria et al. CERN SixTrackLIB: a 6D single particle symplectic tracking code for the computation of the trajectories of individual relativistic charged particles in circular accelerators. *https://github.com/rdemaria/sixtracklib/*, 2017. 2

[3] Kevin Li et al. Code Development for Collective Effects. In *ICFA Advanced Beam Dynamics Workshop on High-Intensity and High-Brightness Hadron Beams (HB 2016)*, page WEAM3X01, 2016. 1

[4] Stefan Hegglin. Simulating Collective Effects on GPUs. Master's thesis, D-MATH/D-PHYS Dep., ETH Zürich, 2016. 2

[5] Kevin Li and Adrian Oeftiger et al. CERN PyHEADTAIL: numerical n-body simulation code for simulating macro-particle beam dynamics with collective effects. *https://github.com/PyCOMPLETE/PyHEADTAIL/*, 2014. 1

[6] Adrian Oeftiger. *Space Charge Effects and Advanced Modelling for CERN Low Energy Machines*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 2016. 1

[7] Adrian Oeftiger, Steven Hancock, and Giovanni Rumolo. Space charge mitigation with longitudinally hollow bunches. In *ICFA Advanced Beam Dynamics Workshop on High-Intensity and High-Brightness Hadron Beams (HB 2016)*, page MOPR026, 2016. 1

[8] Frank Zimmermann. Introduction to Collective Effects in Particle Accelerators. *ICFA Beam Dyn. Newslett.*, 69:8–17, 2016. 1