Preliminary draft 13:31 11 June 2019

11 June 2019
adrian.oeftiger@cern.ch

Space Charge Parallelisation on the GPU and Timing

Adrian Oeftiger
CERN, CH-1211 Geneva, Switzerland

Keywords: beam dynamics, collective effects, particle-in-cell algorithm

Summary

The particle-in-cell algorithm is the well established algorithm of choice to self-consistently model space charge, the direct electromagnetic interaction between beam particles. The particle positions are interpolated onto a regular mesh, where the discrete Poisson equation is solved for the electric fields, and eventually these are interpolated back to the particles. The algorithm heavily relies on memory interaction which makes computational parallelisation a non-trivial task.

We investigate the computational timing efficiency for the full 3D space charge model implemented in PyPIClib, which is used by the collective effects simulation software PyHEADTAIL. Here we compare the timing of a space charge kick for a CPU to modern NVIDIA GPUs (graphics processing units) with thousands of cores. We find that the algorithm strongly profits from the GPU architecture, even when comparing to an openMP accelerated version on the 12-core CPU. In particular, the initial particle to mesh interpolation step as well as the final mesh to particle interpolation step benefit from the large memory bandwidth on the GPUs in contrast to the openMP accelerated CPU implementation. Therefore, we conclude that the GPU is a natural choice of hardware to run heavy and/or long-term simulations with self-consistent space charge on.

Contents

1 Introduction

CERN's accelerator and beam physics group carries out computationally highly demanding physics studies based on space charge. In this context, there is an ongoing performance evaluation of simulation codes and frameworks in order to identify the ideal tools and computing hardware architectures. The present document intends to provide a reference for the timing of self-consistent space charge kicks as they are implemented in the CERN developed tools `PyPIClib` and `PyHEADTAIL`. In particular, we investigate the benefit from exploiting graphics processing units (GPUs) compared to traditional CPU computing.

The structure of this document is as follows: we first describe the `PyHEADTAIL` library in some detail along with a brief introduction to how it has been parallelised on GPU hardware. Next we describe the particle-in-cell algorithm, reviewing some peculiarities of its GPU acceleration and the corresponding implementation in `PyPIClib`. Eventually, we present the timing profiles on modern NVIDIA GPUs compared to CPUs.

2 Physics Context: the `PyHEADTAIL` Library

`PyHEADTAIL`[1] is a collective beam dynamics simulation library written in Python, which simulates the interaction of beam macro-particles among each other as well as with the machine environment during their storage time in a circular accelerator. Hence, the library offers models of both direct interaction via the beam self-fields (space charge) as well as indirect electromagnetic interaction via the machine environment (beam coupling impedance).

The `PyHEADTAIL` library is actively developed within the accelerator and beam physics group at CERN and involves a dynamic developer community. The use cases vary often and address the continuously changing demands from the physics scenarios. To address these circumstances, the code was chosen to be written in the Python language: this approach (i.) enables simple, fast and flexible implementation of new physics, (ii.) enhances code legibility and maintenance, and (iii.) allows for rapid prototyping while profiting from dynamic interaction with the code. To minimise the performance loss due to abstraction overhead, the central and more static parts of `PyHEADTAIL` (having been identified as bottlenecks via code profiling, e.g. the computation of the distribution statistics) are implemented in low-level languages, mainly C and derivates.

The motion of beam macro-particles in `PyHEADTAIL` is modelled by alternate single- and multi-particle dynamics in a *drift-kick* approach. Drifting refers to tracking the linear betatron motion of all single macro-particles across a given machine segment. The collective forces between the macro-particles are integrated over the current segment and are applied as a lumped kick in the interaction point after each drift.

In the case of beam coupling impedances, these collective interaction points model the generation and interaction with wake fields left by leading particles (i.e. effects such as indirect space charge, resistive walls, resonating structures, etc.) and comprise a convolution algorithm of the macro-particle distribution with a wake function. To this end, a 1D particle-in-cell algorithm is usd to discretise the distribution for the convolution. Typically, one such wake field kick per revolution is a good approximation to model the beam coupling

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

impedances. In terms of performance, the computation of trigonometric functions within the tracking matrices for each macro-particle dominates over the computation of the impact of impedances for a single bunch.

In contrast, in the case of direct space charge one needs many collective interaction points during a revolution (on the order of hundreds to thousands of integration steps). A simple space charge model may assume a fixed beam field map for a bunch, which is not self-consistently calculated from the actual macro-particle distribution. This makes the simulation effectively embarrassingly parallel and single-particle-like. For a self-consistent treatment, however, at each time step the macro-particle distribution is discretised and interpolated onto a regular grid. Subsequently, the Poisson equation is solved for the potential on the grid (e.g. via Fourier transform and free-space integrated Green's functions) and the solution is interpolated back to the macro-particle locations. This so-called *particle-in-cell algorithm* yields space charge kicks for each macro-particle while self-consistently resolving the entire distribution. It quickly becomes expensive in terms of computing power, as high resolutions (i.e. number of grid cells, macro-particles and integration steps per revolution) are needed to minimise numerical noise problems.

Accelerating `PyHEADTAIL` simulations with the aid of NVIDIA GPUs has been facilitated by the `PyCUDA` library, making NVIDIA's `CUDA` driver available on the Python level. `PyCUDA`'s `GPUArrays` satisfy the `NumPy` API, which is the standard scientific computing library for Python. Hence, many of `PyHEADTAIL`'s physics algorithms such as the tracking could largely avoid code duplication (such as translation to `CUDA`). A context management developed by Ref. [1] makes the usage of GPUs almost transparent to `PyHEADTAIL` users and basic physics developers: depending on the CPU or GPU context, math function calls are redirected to the corresponding library (`numpy` and `scipy` for the CPU or `PyCUDA.GPUArray` and `scikit-cuda`) or sometimes specific kernel implementations in the GPU case. This approach reflects the need for flexibility, legibility and maintainability of the code as explained in the outset.

As a result of the parallelisation approach, speed-up factors of around 50 have been achieved using NVIDIA Tesla C2075 cards with respect to a single CPU core when simulating direct space charge. This memory-constrained multi-particle scenario is to be compared to speed-up factors of 400 for embarrassingly parallel single-particle physics (programmed in plain CUDA without performance loss due to intermediate Python layers) based on the same hardware set-up [3]. This promising start made larger simulation campaigns possible, such as e.g. investigating the interplay of direct space charge and beam coupling impedances for instabilities in CERN's Large Hadron Collider (LHC) at injection energy. These simulations run for up to hundreds of thousands of revolutions, while some of the beam and machine parameters have been scanned in fine intervals in separate runs. Without the high-performance computing approach exploiting NVIDIA K20 and K40 GPUs, this study would not have been possible due to maximum wall time and storage space constraints at CERN's current single-nodes cluster, `lxplus`.

Currently, we investigate to extend `PyHEADTAIL` with advanced non-linear tracking as provided by `SixTrackLib`[2]. `SixTrackLib` is parallelised for both multi-core CPU and GPU environments. A first step has been made with establishing an interface to

[2] https://github.com/rdemaria/sixtracklib

SixTrackLib from PyHEADTAIL in Ref. [2]. With the non-linear tracking supporting realistic machine lattices, PyHEADTAIL will be able to simulate accelerators much more accurately – this is particularly important for resonance studies with space charge requiring a precise model of the non-linear behaviour of the machine.

3 Particle-in-cell Algorithm

bla

3.1 Parallelised Particle to Mesh Interpolation on the GPU

The initial charge deposition on the grid discretises the beam particles onto a regular mesh. This particle to mesh interpolation (p2m) step comprises the calculation of interpolated weights for each mesh vertex at the corner of a cell for a given particle. These weights are then readily added for each particle yielding the charge distribution on the mesh (which can then subsequently be Poisson solved for the potential in a next step).

In PyPIClib we implemented two different algorithms for this p2m step: in the most straightforward approach implemented in the PyPIC_GPU.particles_to_mesh method, a GPU thread identifies a particle. In such a thread, the particle position gives (1.) the cell ID with the adjacent mesh nodes as the cell corners, (2.) the distance to the mesh nodes is computed, which (3.) determines the linearly interpolated weights for each adjacent mesh node, and which (4.) are each directly added to a mesh memory entry. As many particles can contribute to a given memory entry representing the local charge at the discretised mesh node location, several threads may access this memory entry in parallel. To solve this race condition it is necessary to use atomic operations – in this case atomicAdd: a given thread accessing the mesh memory entry will thus lock it, read the value, add the corresponding particle weight amount and write this new value to the mesh memory entry before releasing the lock for another thread to access it.

Before NVIDIA's Maxwell architecture, the atomic operations were only hardware accelerated for single precision floating numbers. However, single precision is prohibitive for the required accuracy: for a given mesh cell in the beam centre there can easily be hundreds to thousands of macro-particles summed up for the mesh nodes charge. This is repeated for thousands of space charge kicks per revolution around the accelerator ring and for hundreds of thousands of revolutions – hence, $\mathcal{O}(10^{-7})$ single precision error for a single operation rapidly sums up to non-negligible impact on the particle dynamics which is not tolerable. PyPIClib allows to test this as PyPIC_GPU.particles_to_mesh accepts a dtype keyword (np.float32 or np.float64). A simulation comparing both precisions over 50 000 turns for the CERN Large Hadron Collider using 2000 space charge kicks per revolution clearly demonstrates the impact on the centroid motion: not only does the amplitude of the centroid oscillation change by a factor for a given time instant as shown in Fig. 1a (this qualitatively different behaviour can impact stability thresholds for instability simulations), but also the frequency spectrum of the centroid changes as shown in Fig. 1b (potentially rendering the beam reaction to external excitation qualitatively quite different). We conclude that single precision is not acceptable and we therefore have to live with double

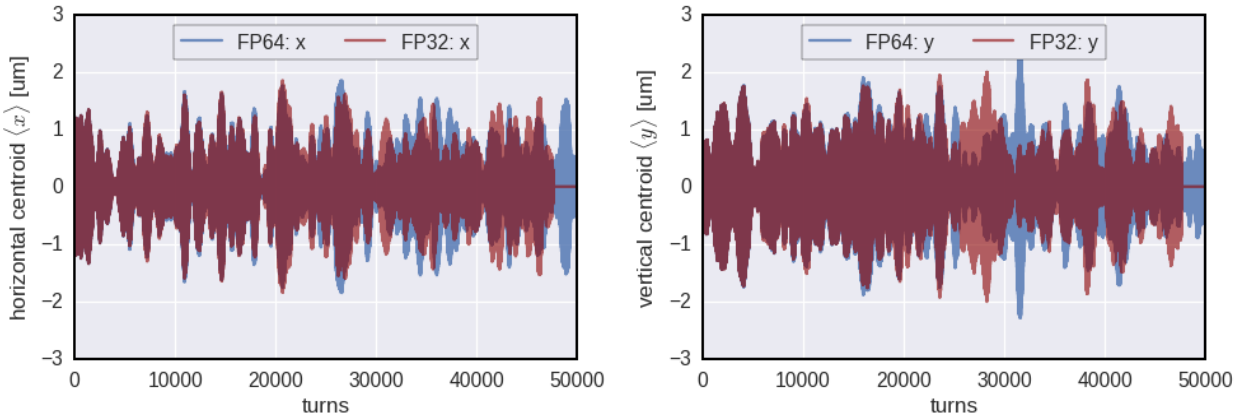
precision atomic operations which are software emulated on pre-Maxwell GPUs.

The emulated `atomicAdd` leads to long memory access times for threads: these thread memory stalls in turn resulted in suboptimal performance of this first algorithm. A way out has been found at the time via a second approach, which we implemented in the `PyPIC_GPU.sorted_particles_to_mesh` method. The approach relies on sorting the particles by mesh cell and then identifies threads with mesh cells (and not particles like in the first approach). The overhead of sorting can be compensated by the efficient memory accessing: this algorithm performed up to 3.5 times faster than the software emulated `atomicAdd` approach on the Kepler architecture, as presented in Ref. [4]. One thus avoids the memory stalls which typically happen when many atomic operations are accessing the same mesh memory entry with the slow software emulated version.

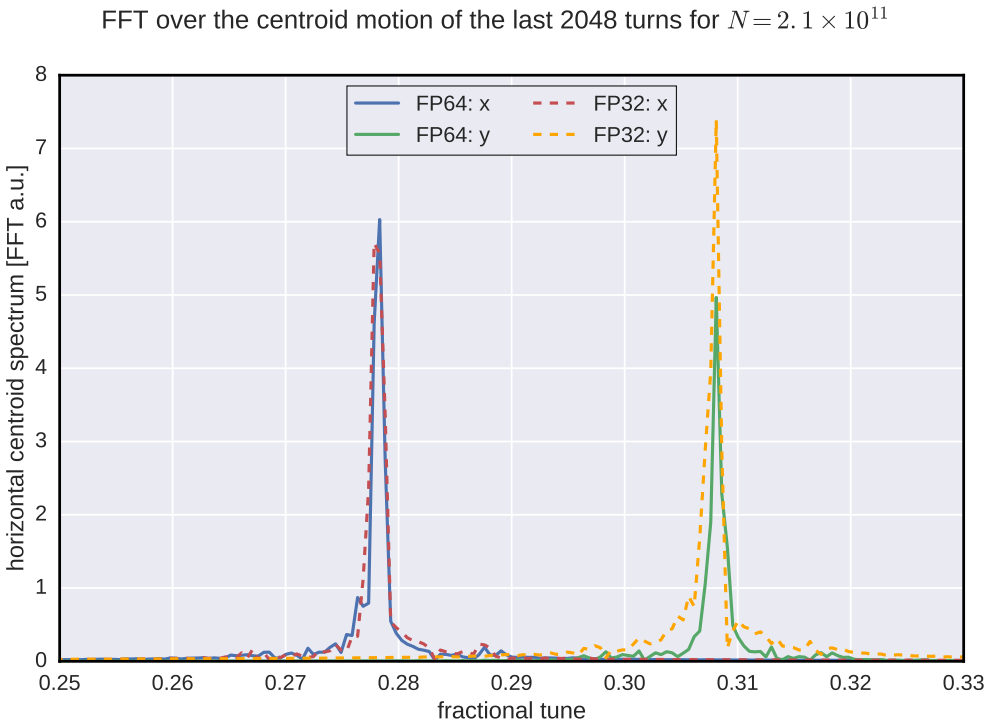
On modern NVIDIA GPUs beyond the Kepler series (i.e. Maxwell, Pascal and the latest Volta architecture) the atomic operations such as `atomicAdd` are also hardware accelerated for double floating point precision[3]. Therefore we can avoid the software emulation of double precision `atomicAdd` in the particle to mesh interpolation and threads do not stall anymore for usual numerical parameters. Eventually the overhead of sorting renders the second approach (with cell $\leftrightarrow$ thread identification) performing less well than the first approach (with particle $\leftrightarrow$ thread identification) for the `p2m` step.

On a final note, if one wants to optimise the approach on a multi-core CPU, a similar guard cell strategy as in the sorted GPU algorithm can be beneficial to avoid similar thread stalls in `openMP` atomic operations, cf. Ref. [4].

[3]See https://docs.nvidia.com/cuda/maxwell-tuning-guide/index.html#fast-shared-memory-atomics as well as https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#atomic-ops



(a) Average of all macro-particle positions in the horizontal x and vertical y plane.



(b) Spectra of average position evolution over last 2048 turns.

Figure 1: Difference of single precision (FP32) and double precision (FP64) particle to mesh interpolation with its impact on beam dynamics.

4 Timings

For the timing we compare an NVIDIA Tesla P100 GPU, an older NVIDIA Tesla C2075 and a Intel Xeon E5 (v1) CPU, cf. Tables 1 and 2.

Table 1: Relevant CPU Machine Specifications

CPU	2× Intel Xeon E5-2630 (v1)
CPU cores	2 × 6
RAM	256 GB DDR3
CPU clock rate	2.30 GHz
CPU L3 cache	15 MB
instruction set	Intel AVX
32bit floating-point performance	0.1 TFLOPS

Table 2: Relevant GPU Machine Specifications

	CERN BE-ABP	CNAF	
GPU	NVIDIA Tesla C2075	NVIDIA Tesla K20	NVIDIA Tesla K40
GPU devices	4	7	8
GPU DDR5 RAM (per device)	5.3 GB	5.1 GB	12.3 GB
GPU clock rate	1.15 GHz	0.7 GHz	0.75 GHz
CUDA cores per device	448	2496	2880
max. no of threads per block	1024 × 1024 × 64	1024 × 1024 × 64	1024 × 1024 × 64
CUDA computing capability	2.0	3.5	3.5
32bit floating-point performance	1.0 TFLOPS	3.5 TFLOPS	4.3 TFLOPS

Table 3: Full Timing for Space Charge Node

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

76ms single core vs 48ms `openMP`

4.1 Line Profiles of the GPU functions

The line-by-line timing profiles[4] of the particle-in-cell algorithm indicate in which parts the algorithm spends most time. Here we present the results for the GPU accelerated

[4]based on the Python tool `line_profiler`, https://github.com/rkern/line_profiler

Table 4: Timing of Particle to Mesh Interpolation

hardware	**cores**	**time [ms]**
NVIDIA GPU Tesla P100	3584	< 1
NVIDIA GPU Tesla C2075	448	19
CPU Intel Xeon E5	1	206

Table 5: Timing of Poisson Solve (FFT)

hardware	**cores**	**time [ms]**
NVIDIA GPU Tesla P100	3584	48
NVIDIA GPU Tesla C2075	448	648
CPU Intel Xeon E5	1	441

version of the algorithm, run on the NVIDIA Tesla P100. The test scenario comprises 1×10^6 macro-particles on a mesh of dimension $256 \times 256 \times 100$. The discrete Poisson equation on this regular grid is readily solved with an open boundary integrated Green's function approach, using the Fast Fourier Transform algorithm via Hockney's trick of cyclic domain extension [4]. The source files of `PyPIClib` can be found in the github repository[5].

Fig. 2 shows the profile of the entire algorithm, which is encapsulated in the function `PyPIC_GPU.pic_solve` within the `PyPIClib` library. The 4 relevant algorithmic steps are

1. `p2m` in lines 712 to 714 with 1.7%,
2. Poisson solve in line 720 with 90.9%,
3. gradient in line 723 with 3.7% and
4. `m2p` in line 728 and 729 with 3.3% of the time spent.

This clearly reveals the Poisson solve step to be the bottleneck on modern GPU architectures (here the NVIDIA P100, compared to Ref. [4] showing `p2m` to be the bottleneck for pre-Maxwell architectures, cf. the discussion in section 3.1).

Fig. 3 then reveals that most of the time during the FFT open boundary Poisson solve is spent on the Fourier transformations (convolution in frequency domain), namely a fraction of 98.6% in the `CUDA` FFT library routines (`cuFFT`) in lines 168 to 170. These results show a highly optimised implementation with regard to the embedding of a high performance language in a slow, interpreted top level language: the overhead from Python remains marginal.

[5] https://github.com/PyCOMPLETE/PyPIC/tree/master/GPU

Figure 2: `Line_profiler` output on the P100 GPU for `PyPIClib`'s `pic_solve` function

```
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-particle.
   679                                               Further keyword arguments are
   680                                               mesh_indices=None, mesh_distances=None, mesh_weights=None .
   681
   682                                               The optional keyword arguments lower_bounds=False and
   683                                               upper_bounds=False trigger the use of sorted_particles_to_mesh
   684                                               which assumes the particles to be sorted by the node ids of the
   685                                               mesh. (see further info there.)
   686                                               This results in particle deposition to be 3.5x quicker and
   687                                               mesh to particle interpolation to be 0.25x quicker.
   688                                               (Timing for 1e6 particles and a 64x64x32 mesh includes sorting.)
   689
   690                                               The optional keyword argument state=None gets rho, phi and
   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                                               '''
   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_bounds
   710                                                       )
   711                                               else: # particle arrays are not sorted by mesh node ids
   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):
   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), list(mp_coords))
   728         1          175    175.0      0.3          fields = self.field_to_particles(*mesh_fields_and_mp_coords, **kwargs)
   729         1         1607   1607.0      3.0          self._context.synchronize()
   730         1            1      1.0      0.0          return fields
```

Table 6: Timing of Mesh to Particle Interpolation

hardware	**cores**	**time [ms]**
NVIDIA GPU Tesla P100	3584	2
NVIDIA GPU Tesla C2075	448	17
CPU Intel Xeon E5	1	103

Figure 3: `Line_profiler` output on the P100 GPU for the `poisson_solve` function of `PyPIClib`'s 3D FFT solver

```
Timer unit: 1e-06 s

Total time: 0.048126 s
File: PyPIC/GPU/poisson_solver/FFT_solver.py
Function: poisson_solve at line 155

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   155                                           def poisson_solve(self, rho):
   156                                               ''' Solve the poisson equation with the given charge distribution
   157                                               Args:
   158                                                   rho: Charge distribution (same dimensions as mesh)
   159                                               Returns:
   160                                                   Phi (same dimensions as rho)
   161                                               '''
   162         1          114    114.0      0.2          rho = rho.astype(np.complex128)
   163         1            6      6.0      0.0          self._cpyrho2tmp.set_src_device(rho.gpudata)
   164         1            3      3.0      0.0          self._cpytmp2rho.set_dst_device(rho.gpudata)
   165                                                   # set to 0 since it might be filled with the old potential
   166         1           30     30.0      0.1          self.tmpspace.fill(0)
   167         1           20     20.0      0.0          self._cpyrho2tmp()
   168         1           87     87.0      0.2          cu_fft.fft(self.tmpspace, self.tmpspace, plan=self.plan_forward)
   169         1         1647   1647.0      3.4          cu_fft.ifft(self.tmpspace * self.fgreentr, self.tmpspace,
   170         1        45831  45831.0     95.2                      plan=self.plan_backward)
   171                                                   # store the result in the rho gpuarray to save space
   172         1           18     18.0      0.0          self._cpytmp2rho()
   173                                                   # scale (cuFFT is unscaled)
   174         1          292    292.0      0.6          phi = rho.real/(2**self.mesh.dimension * self.mesh.n_nodes)
   175         1           78     78.0      0.2          phi *= self.mesh.volume_elem/(2**(self.mesh.dimension-1)*np.pi*epsilon_0)
   176         1            0      0.0      0.0          return phi
```

References

[1] Stefan Hegglin. Simulating collective effects on gpus. Master's thesis, D-MATH/D-PHYS Dep., ETH Zürich, 2016.

[2] Meghana Madhyastha. GPGPU Accelerated Beam Dynamics Interfacing PyHEADTAIL with SixTrackLib. 2018.

[3] Adrian Oeftiger. Parallelisation of PyHEADTAIL, a Collective Beam Dynamics Code for Particle Accelerator Physics. 2016.

[4] Adrian Oeftiger and Stefan Hegglin. Space Charge Modules for PyHEADTAIL. In *ICFA Advanced Beam Dynamics Workshop on High-Intensity and High-Brightness Hadron Beams (HB 2016)*, page MOPR025, 2016.