Simulating Collective Effects on GPUs

CSE Master's Thesis

Stefan Hegglin

Supervision:
Dr. Kevin Li, CERN
Prof. Dr. Peter Arbenz, ETH Zürich

D-MATH/D-PHYS
ETH Zürich
Switzerland
08.02.2016

Abstract

Computer simulations are an important tool to study the dynamics of charged particles in particle accelerators, with new hardware solutions such as GPUs providing a vast increase in computing power. In the accelerator physics domain simulations are used to understand instabilities arising due to collective effects in high intensity beams which limit the accelerator performance. In this thesis PyHEADTAIL, a code to study collective effects in synchrotrons, is ported to GPUs using PyCUDA. The goal is to achieve a significant speedup while at the same time producing a simple interface for users and other developers. A speedup of 6 compared to the CPU version is achieved on a typical simulation study of instabilities in the Large Hadron Collider (LHC) at CERN.

Keywords: collective effects, beam dynamics, simulation, GPU, PyHEADTAIL, PyCUDA

Contents

List of Figures

List of source codes

1 Introduction

The main purpose of this thesis is to parallelise the collective effects simulation code PyHEADTAIL using GPUs while at the same time creating a transparent and easy-to-use interface for users and developers. The thesis was written at CERN in Geneva.

CERN is the largest particle physics research laboratory in the world. It hosts high energy physics experiments ranging from studying properties of atomic nuclei to dark matter and fundamental particles. All experiments at CERN have a need for high-energy particles (protons or ions), either to study the products of collisions of the particles with other particles/matter or using them to produce secondary particle beams such as neutrinos or anti-protons. The large particle accelerator complex at CERN (Figure 1) serves these needs, providing particle beams ranging from several MeV to TeV of energy. The Accelerator and Beam Physics Group (ABP) at CERN is responsible for beam physics research and optimisation of machine parameters covering the entire CERN accelerator complex. The Hadron Synchrotron Coherent Collective Effects Section (HSC), in which this thesis has been performed, studies and tries to mitigate beam instabilities arising due to collective effects in the CERN synchrotrons.

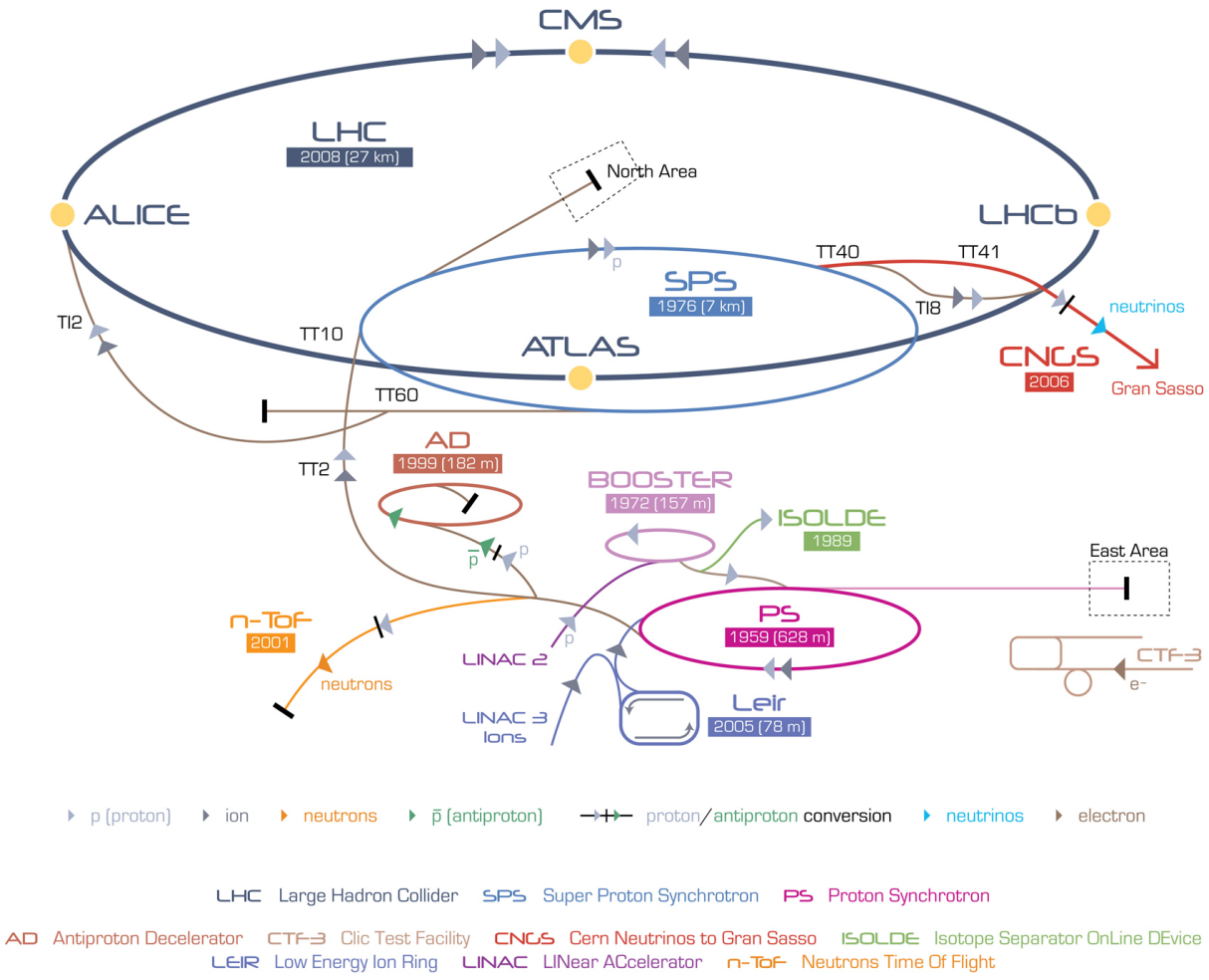

Figure 1: The CERN accelerator complex as of 2015. Protons (*ions*) are accelerated in the linear accelerator Linac 2 (*Linac 3*) before getting injected into the Booster (*LEIR*). Both particle species can further be accelerated in the PS, SPS and the LHC, where they reach energies up to 7 TeV. Most particles are not destined for the LHC but are extracted to experimental areas (e.g. North/East Area, Isolde, etc.) at lower energies. [image courtesy CERN]

Luminosity is one of the key parameters characterising the performance of a collider. It quantifies the collision rate at the interaction point. One way of increasing the luminosity is to increase the number of particles per bunch (intensity). The intensity increase leads to challenges for the beam stability, collective effects become increasingly important [13]. In order to mitigate these effects, a thorough understanding of the beam dynamics is required, and simulations are an important tool to achieve this. PyHEADTAIL is a simulation code at CERN which is used to simulate collective effects in the accelerators. The simulations, which require large scale parameter scans, are often constrained by runtime limits rendering new technologies such as GPUs 'interesting'. Making use of emerging computer architectures to speed up simulations without increasing the code complexity and usability is challenging and the main goal of this thesis. Thus, the code developed within this thesis should make GPU usage in PyHEADTAIL accessible for a wide range of non-specialised developers and users.

GPU computing has become a useful tool in scientific computing over the last decade [14]. Using GPUs is usually cumbersome and requires rewriting large parts of existing codes. PyCUDA tries to simplify the access to GPUs by providing an interface between Python and CUDA[1]. PyCUDA has been used successfully in a number of scientific projects and publications, ranging from fields as diverse as cosmology [19], biochemistry [4], to ecology [2] and numerical mathematics [7]. PyCUDA was also successfully used at CERN to write a particle-in-cell solver in 2015. The fact that PyCUDA is well established with a mature code base and stable development status was the reason to chose PyCUDA for the code development in this thesis.

Similar work trying to build a unified interface for GPUs and CPUs was independently performed at PSI [1].

This document is structured as follows: Section 2 provides an introduction to accelerator physics and terminology that is required to understand the thesis, Section 3 presents PyHEADTAIL and the numerical models. Section 4 explains the methods and implementation strategies used to port PyHEADTAIL to GPUs and Section 5 presents the quantitative and qualitative results of this thesis.

[1]CUDA is an application programming interface for GPUs by Nvidia, a major manufacturer of GPUs

2 Theory

This section provides an overview of particle accelerator physics required to understand this thesis and is unless otherwise stated based on [8]. It introduces the transverse and longitudinal equations of motion for a single particle in a synchrotron and collective effects due to particle-particle and particle-environment interactions.

2.1 Synchrotrons

A synchrotron is a ring-shaped machine to accelerate charged particles. The particles are usually contained in a vacuum tube (beam pipe) along which magnets and radio-frequency (RF) cavities are placed. Dipole magnets are used to bend the trajectory of the particles, while quadrupolar magnets are used for focusing in the transverse plane. The RF cavities are used to accelerate and bunch the particles, i.e. provide focusing in the longitudinal plane. A synchrotron obviously consists of many more parts (e.g. higher order magnetic fields, collimators, etc.) which will not be discussed here.

The particles are created in a source and accelerated in a linear accelerator before entering the synchrotron. Due to the circular/periodic structure of the synchrotron, the particles perform many revolutions. The beam energy is increased by ramping the dipole magnetic fields which causes the particles to get accelerated in the RF cavities. Once the desired beam energy is reached, the ramp is stopped and the particles are either extracted to another (higher-energy) synchrotron or brought to collision. The collision can take place between two oppositely moving beams inside the synchrotron (e.g LHC) or between the beam and a fixed target at a special target facility (e.g SPS).

An example of a modern synchrotron is the LHC located at CERN with a circumference of 27 km. 8 RF cavities per beam, operating with a maximal voltage of 2 MV, are responsible for the acceleration and bunching of the particles, while 1232 dipoles keep the particles on the desired trajectory.

Accelerator physics deals with the design of accelerators and dynamics of charged particles inside these structures, with the ultimate goal of delivering a high quality beam to the experimentalists to provide the highest possible luminosity (events per time per area).

2.2 Single Particle Beam Dynamics

The motion of charged particles in electric and magnetic fields follows Maxwell's equations. Particles in a low-intensity beam can be approximated as moving independently from another, influenced only by the electromagnetic fields of the magnets and radio-frequency cavities placed along the accelerator. In the following, the particles are assumed to have charge e. The particles occupy a six-dimensional phase space Γ and each particle is represented by a vector

$$\vec{\psi} = \begin{pmatrix} x \\ x' \\ y \\ y' \\ z \\ \delta \end{pmatrix}, \vec{\psi} \in \Gamma \tag{1}$$

where x, x', y, y' are the transverse positions and conjugate momenta offsets of the particles with respect to the design orbit, z the longitudinal offset with respect to a reference position s after a time t and $\delta = \frac{p-p_0}{p_0}$ the normalised deviation from the reference momentum, see

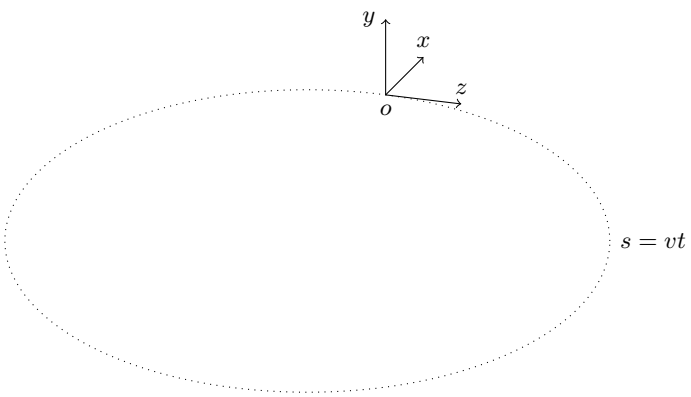

Figure 2: A schematic overview of the used co-moving coordinate system. The dotted line is the design orbit of the synchrotron. x, y and z are the offset of the particle position to the reference position o. s describes the position along the accelerator (following [3]).

Figure 2. The transverse and longitudinal motion of a single particle can be approximated as independent from another and are explained separately here.

2.2.1 Longitudinal Equations of Motion

The longitudinal dynamics of the particles are governed by RF cavities. The cavities produce a time dependent electromagnetic field which is used to accelerate the particles and keep them confined in bunches. The longitudinal equations of motion for a particle with charge e are:

$$\dot{z} = -\eta\beta c\delta \tag{2}$$

$$\dot{\delta} = \frac{eV_{RF}}{p_0 C} \sin\left(\frac{2\pi h z}{C} - \frac{\Delta E}{eV_{RF}}\right). \tag{3}$$

Hereby V_{RF} is the integrated voltage of the RF-cavity, C the circumference of the accelerator, p_0 the reference momentum, h the harmonic number (the particle revolution period divided by the device frequency), ΔE the energy gain per revolution, η the phase-slippage factor, and $\beta = \frac{v}{c}$ the velocity in units of the speed of light c. The phase-slippage factor indicates whether an increase of the momentum offset δ leads to a shorter ($\eta < 0$) or longer ($\eta > 0$) revolution period and depends on machine optics and the particle energy [8] [13] [17].

2.2.2 Transverse Equations of Motion

Due to the focusing and defocusing quadrupolar fields, the particles oscillate around the reference trajectory. This oscillation is called betatron motion. This transverse motion of a charged particle with reference momentum in dipolar and quadrupular fields is modelled by the Hill equation

$$\frac{d^2u(s)}{ds^2} + K(s)u(s) = 0, \tag{4}$$

where $u(s)$ is the transverse offset (x or y) of a particle at the longitudinal position s, and K, a measure for the magnetic field strength, is periodic: $K(s) = K(s + C)$. The solution to Hill's equation for a structure of length L is given by a pseudo-harmonic oscillation with an

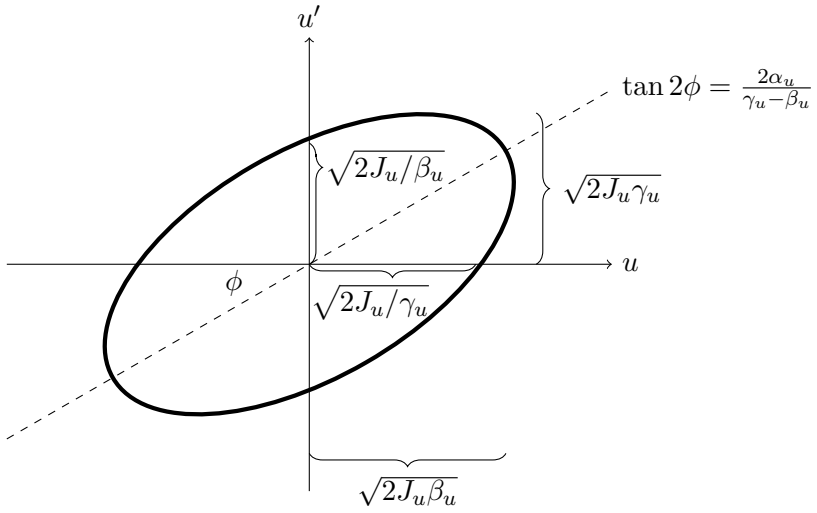

Figure 3: The phase space ellipse defined by the Twiss parameters α, β, γ. The area of the ellipse is $2\pi J_u$.

s-dependent amplitude:

$$u(s) = \sqrt{2J_u\beta_u(s)}\cos(\psi_u(s) + \psi_{u,0}) \tag{5}$$

$$u'(s) = -\sqrt{\frac{2J_u}{\beta_u(s)}}\left[\sin(\psi_u(s) + \psi_{u,0}) - \frac{\beta_u'(s)}{2}\cos(\psi_u(s) + \psi_{u,0})\right] \tag{6}$$

where

$$\psi_u(s) = \int_0^L \frac{ds}{\beta_u(s)}. \tag{7}$$

The solution is parametrised by the Courant-Snyder (Twiss) parameters $\alpha_u(s) = -\frac{\beta_u'(s)}{2}$, $\beta_u(s)$ and $\gamma = \frac{1+\alpha_u^2(s)}{\beta_u(s)}$ (whose choice will be explained in the next paragraph). The transverse action J_u and phase offset $\psi_{u,0}$ are given by the initial conditions.

It follows that each particle's trajectory moves on an ellipse in the (u, u') phase space (see Figure 3) defined by its action and parametrised by the Twiss parameters α_u, β_u and γ_u:

$$\gamma_u(s)u^2(s) + 2\alpha_u(s)u(s)u'(s) + \beta_u(s)u'^2(s) = 2J_u. \tag{8}$$

The tune Q_u of a particle is defined as the number of betatron oscillations it performs per revolution and is equal to the phase advance over one revolution:

$$Q_u = \oint \frac{ds}{\beta_u(s)}. \tag{9}$$

The (first-order) chromaticity ξ_u is a measure for the δ-dependance of the tune

$$\xi_u = \frac{\partial Q_u}{\partial \delta} \tag{10}$$

and couples the longitudinal and transverse motion. A tune shift ΔQ_u of a particle is defined as the deviation from the betatron tune defined above. One possible source of a tune shift are collective effects which are described in the next section.

2.3 Collective Effects

This section is based on Physics of Intensity Dependent Beam Instabilities [13]. In the previous section the equations of motion of a particle were treated as independent of other particles or any surroundings and solely under the influence of external fields of magnets and RF cavities. Increasing the beam intensity (the number of particles per bunch) leads to stronger self-generated electric fields perturbing the external electric fields. These field perturbations can lead to beam instabilities if the intensity is high enough. High intensity beams can therefore not be approximated as a collection of independent particles moving in external fields, but have to be described by an ensemble of mutually interacting particles [3].

Collective effects are usually divided into the direct interaction between the particles, so called space-charge, and the interaction with the charges inside the surrounding beam pipe, or other devices, induced by the beam, called impedance.

In the transverse plane, the forces F_c generated by these effects lead to a perturbation of the Hill equation,

$$\frac{d^2u}{ds^2} + K(s)u = F_c(\Psi, s), \tag{11}$$

where Ψ is the phase space distribution of the particles. This means the force acting on each particle is now also dependent on the collective distribution Ψ of all the particles in the beam. Except in special cases, F_c cannot be computed analytically and is only available through numerical means.

The most general formalism is to model the particles as a continuous distribution in phase space and formulate a partial differential equation for the time evolution of this distribution. This approach leads to the Vlaslov-equation which will not be discussed further in this thesis, since its formalism is not required for the macro-particle based methods described in section 3.

2.3.1 Impedance

In the following paragraph, the effect of wake-fields is explained in more detail. Wake fields are a way to describe the interaction between the particles in a beam and its surrounding.

A charged particle moving inside a chamber induces image charges in the surrounding chamber. In a perfectly conducting round chamber these image charges follow the beam at the same velocity as the beam itself, losing no energy and therefore creating no trailing field. In a more realistic non-perfectly conducting chamber, these image charges cannot move freely, lose energy and induce a trailing electromagnetic field perturbing the following particles and beams. These fields can be described in frequency-domain (impedance) or time-domain (wake fields) [13]. More formally, a charged particle q_0 at position (x_0, y_0, s) moving through a chamber of length L generates a trailing field which a probe charge q at a distance z $(x, y, s-z)$ will see, see Figure 4. The transverse wake function is then defined as the integral over the chamber length of the resulting force and corresponds to the Green's function of the chamber

$$W_\perp(x, y, x_0, y_0, z)[V/C] = -\frac{1}{q_0 q}\int_0^L F_\perp(x, y, s, x_0, y_0, z)ds. \tag{12}$$

The force is usually computed using numerical simulation software which solve Maxwell's equation in 3D for the corresponding accelerator chamber and compared to measurements at the device.

Given the device response to a single source particle, a continuous bunch distribution $\lambda(z)$ can be modelled as follows: A particle at position z sees the wakes left behind by all the

preceding particles of the bunch. The resulting energy change is given by by the convolution of the bunch distribution with the Green's function [21]:

$$\Delta E(z) \propto \int_{z}^{-\infty} \lambda(z') W_{\perp}(z - z') dz' \tag{13}$$

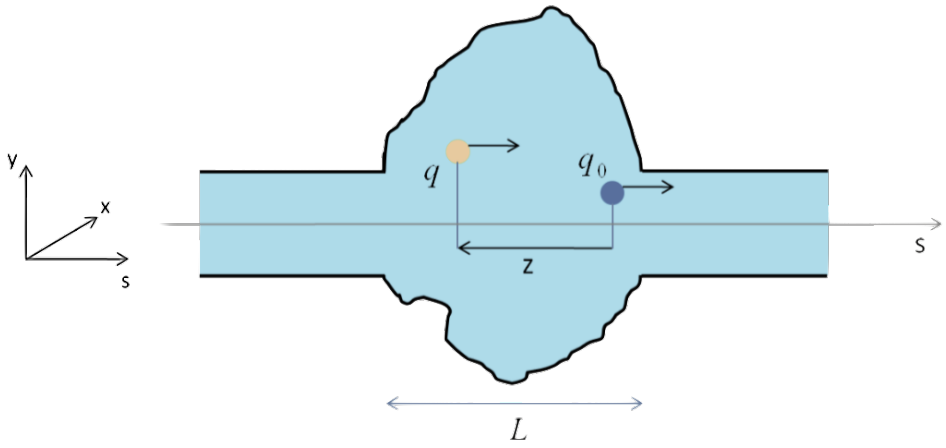

Figure 4: A schematic overview of a test charge q moving behind a source q in a generic chamber [21]

2.4 Beam Instabilities

A beam instability is a self-amplified exponential increase in one or more moments of the beam distribution and the beam emittance. Instabilities can lead to the degradation of the beam quality or beam loss. In a real machine, an instable beam may get extracted to a so called beam-dump in order to protect the machine from uncontrolled particle losses. Some instabilities can be mitigated via active feedback systems (e.g dampers) or by adjusting certain machine parameters such as the chromaticity.

2.5 GPU Computing

This section provides a short introduction to GPUs and GPU computing.
A GPU (graphics processing unit) is a specialised computing device which can be used to offload certain computing tasks from the CPU. A modern GPU consists of many (thousands) of lightweight specialised computing cores, compared to CPUs which consist of very few (tens) of general-purpose cores. This architecture allows the GPU to be more efficient than a CPU when working on highly parallel tasks. GPUs, in contrast to CPUs, can perform an instruction on an array of data at the same time, similarly to the limited SIMD capabilities of CPUs, although on a much higher scale. Nvidia, a major manufacturer of GPUs, calls this concept 'single-instruction-multiple-threads (SIMT)'. The SIMT programming model is especially well suited for so called embarrassingly parallel tasks, which are tasks requiring no interaction between the different threads. An example of such an embarrassingly parallel operation is the calculation of the color of each pixel on the screen in a computer game (therefore also the name GPU) [15].

GPU computing, traditionally used in computer graphics applications, has become a useful tool in scientific computing over the last decade [14]. In the context of accelerator physics,

the tracking of independent particles in a beam is conceptually well suited for a parallelisation using GPUs and the SIMT programming model. The simulation of collective effects requires computing the interaction between particles in a beam and a direct match to the SIMT model is not possible.

The usual way to run software on GPUs is to write specialised code (in either CUDA C or Fortran for Nvidia GPUs) for the device. To make use of GPUs, the code has to be written targeted for the device, making a rewrite of an existing code base necessary.

The data flow of a typical programme using a GPU is as follows:

1. The CPU runs the usual code not targeted for the GPU.
2. If a command to run a computation on the GPU is found, the CPU (host) copies the data to the GPU (device) and launches the GPU-code (kernel).
3. The GPU computes the result making use of the parallel computing capabilities. At the same time, the CPU can run independent parts code or wait for the result from the GPU.
4. Once the work on the GPU is completed, the results are copied back to the CPU for further processing.

Copying data between the CPU and GPU is relatively slow and should be avoided whenever possible. In newer CUDA versions, the host can also launch multiple kernels which can run independently on the GPU (streams).

3 PyHEADTAIL

PyHEADTAIL is a collective effects simulation code developed at CERN. It is based on the original HEADTAIL code [18] written in C. Due to the growing complexity of HEADTAIL, the maintenance of the non-object-oriented code proved to be increasingly difficult and inefficient. In 2013 it was decided to rewrite the HEADTAIL in Python (thus the name PyHEADTAIL) to make it more maintainable, extensible and easier for non-developers to develop and use. Instead of steering the simulation in the classic way via an input file and creating yet another syntax, it was also decided to write the simulations directly in Python, making the entire simulation scriptable [11] [12]. The code is freely available on GitHub[2].

This section will provide an introduction to the numerical models used in PyHEADTAIL to solve the equations described in Section 2 and is largely based on [9].

3.1 Current Scope

PyHEADTAIL can be used to simulate various single-bunch collective effects in synchrotrons. Currently (February 2016) space-charge, electron-cloud, impedances and damping/feedback systems are implemented. Some of these effects rely on external modules such as PyPIC[3] (space-charge) and PyECLOUD[4] (electron-cloud).

3.2 Software Design

The software design follows an object-oriented approach: All tracking elements are subclasses of a common base class providing a `track(bunch)`-method, making it easy to add new functionalities. The particles' phase space is stored in a class, providing further functionalities such as computing the statistics of the bunch distribution and sorting the particles according to a specified ordering. The dynamic nature of Python allows changing the machine parameters during the simulation (e.g. trimming the RF voltage and phases for shaping the longitudinal phase space to create hollow bunches in the PS). A testing module consisting of unit tests and interactive tests is part of PyHEADTAIL.

Due to the performance limitations of interpreted Python code, computationally intensive parts are written Cython[5] and compiled to machine code. PyHEADTAIL makes use of well-known scientific Python libraries such as NumPy[6] and SciPy[7].

3.3 Numerical Model

The numerical model of PyHEADTAIL is described in detail in [9] and CAS2015 Lectures [10].

The large number of particles in a real bunch (in the order of 10^{11}) make simulating each particle computationally unpractical. PyHEADTAIL represents the bunch as a number of macro-particles in 6 dimensional phase space, where each macro-particle represents a certain amount of real physical particles. Typical simulations require between 10^5 and 10^7 macro-particles. A `Particles` object holds six arrays corresponding to the coordinate vectors (Structure-of-Arrays) and provides various methods to compute statistics of the particle distribution. The integration of the particles position over a large number of turns requires the use of double precision. A typical particle distribution at the beginning of a simulation is shown in Figure 6.

[2] `https://github.com/PyCOMPLETE/PyHEADTAIL`
[3] `https://github.com/PyCOMPLETE/PyPIC`
[4] `https://github.com/PyCOMPLETE/PyECLOUD`
[5] `http://cython.org/`
[6] `http://www.numpy.org/`
[7] `http://www.scipy.org/scipylib/index.html`

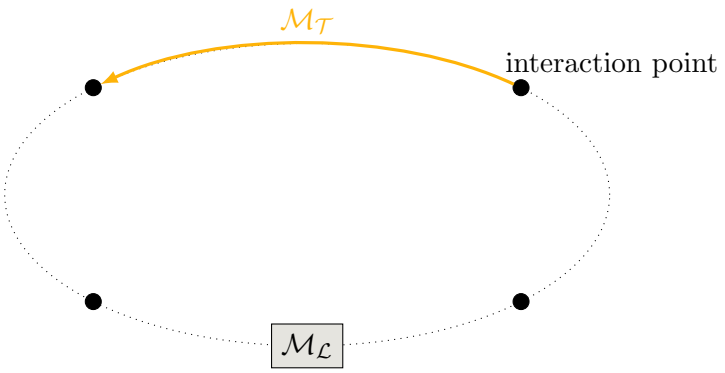

Figure 5: The PyHEADTAIL drift-kick model: The synchrotron is split into segments and interaction points. The particles are linearly tracked along the segments. Kicks (e.g. impedance, damping) are applied at the interaction points. Interaction points can also be used to compute statistics of the bunch and write them to file via monitors. Usually, the longitudinal tracking is lumped at one interaction point.

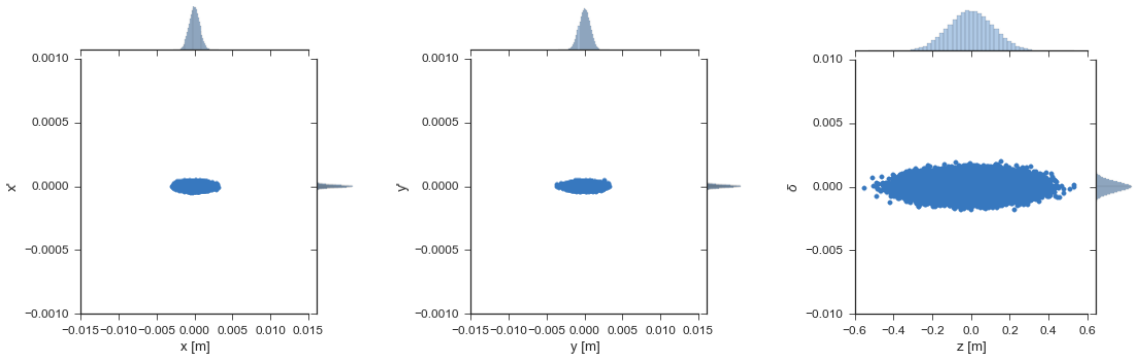

Figure 6: Typical phase space at the beginning of a simulation. The bunch has a 6-dimensional Gaussian distribution which is matched to the machine parameters.

PyHEADTAIL divides the synchrotron into a number of ring segments. Collective effects/-kicks are applied at interaction points which lie between these ring segments. The particles are tracked along the segments using linear transfer maps. A visualisation of the drift-kick model is shown in Figure 5.

The next sections describe the numerical models and implementations of PyHEADTAIL which are used to solve the equations of motion described in the theory section.

3.3.1 Transverse Tracking

The transverse position of each particle is updated on a segment-by-segment basis using a transfer-map approach:

$$\begin{pmatrix} u_i \\ u_i' \end{pmatrix}_1 = \mathcal{M}_{\mathcal{T}_i}^{0\to 1} \begin{pmatrix} u_i \\ u_i' \end{pmatrix}_0 , \tag{14}$$

where the linear transfer map $\mathcal{M}_\mathcal{T}$ evolves from the parametrised solution of Hill's equation for a segment $0 \to 1$ using the Twiss parametrisation:

$$\mathcal{M}_{\mathcal{T}i,0\to 1} = \begin{pmatrix} \sqrt{\beta_1} & 0 \\ -\frac{\alpha_1}{\sqrt{\beta_1}} & \frac{1}{\sqrt{\beta_1}} \end{pmatrix} \begin{pmatrix} \cos\Delta\psi_i & \sin\Delta\psi_i \\ -\sin\Delta\psi_i & \cos\Delta\psi_i \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{\beta_0}} & 0 \\ \frac{\alpha_0}{\sqrt{\beta_0}} & \frac{1}{\sqrt{\beta_0}} \end{pmatrix} . \tag{15}$$

α and β are the Twiss parameters at the interaction points 0 and 1, respectively, and $\Delta\psi_i = \psi_{i,1} - \psi_{i,0}$ is the phase advance of particle i between these two interaction points. While the Twiss parameters are constant and identical for all macro-particles at a certain interaction point, detuning effects such as chromaticity or amplitude detuning lead to phase advances which differ from macro-particle to macro-particle and from turn to turn. The map $\mathcal{M}_\mathcal{T}$ can be understood as a rotation of the normalised phase-space (transforming the ellipse in Figure 3 into a circle) by $\Delta\psi_i$:

$$\mathcal{M}_{\mathcal{T}i,0\to1} = \mathbf{B} \begin{pmatrix} \cos\Delta\psi_i & \sin\Delta\psi_i \\ -\sin\Delta\psi_i & \cos\Delta\psi_i \end{pmatrix} \mathbf{B^{-1}} \tag{16}$$

Detuning effects are then modelled by changing the phase advance on a per-particle (chromaticity, amplitude detuning) and per turn and segment basis (amplitude detuning) and are implemented as follows:

$$\Delta\psi_i = \Delta\psi_{0,i} + \frac{\Delta\psi_0}{Q_u} \times (\underbrace{\xi\delta_i}_{\text{Chromaticity}} + \underbrace{\alpha_{uu}J_{u,i} + \alpha_{uv}J_{v,i}}_{\text{Amplitude Detuning}}). \tag{17}$$

ξ is the (first-order) chromaticity, and the parameters α_{uu}, α_{uv} are a measure of the octupole magnet current acting on the transverse and longitudinal action J_u.

3.3.2 Longitudinal Tracking

The longitudinal tracking is done either linearly or non-linearly. The linear longitudinal tracking is performed by using the matrix

$$\mathcal{M}_\mathcal{L} = \begin{pmatrix} \cos(2\pi Q_s) & -\frac{\eta\beta c \sin(2\pi Q_s)}{\omega_s} \\ \frac{\omega_s \sin(2\pi Q_s)}{\eta\beta c} & \cos(2\pi Q_s) \end{pmatrix}, \tag{18}$$

where

$$\omega_s = \frac{2\pi\beta c Q_s}{C}, \tag{19}$$

β is the velocity in units of the speed of light c, Q_s the synchrotron tune, C the circumference of the synchrotron and η the first order slippage factor as defined in section 2. The tracking along an accelerator segment from point 0 to point 1 is then given by

$$\begin{pmatrix} z \\ \delta \end{pmatrix}_1 = \mathcal{M}_\mathcal{L} \begin{pmatrix} z \\ \delta \end{pmatrix}_0. \tag{20}$$

The more realistic non-linear longitudinal tracking is performed using the Velocity-Verlet algorithm [20] on a turn-by-turn basis ($\Delta z = C$):

$$z_{i+\frac{1}{2}} = z_i - \frac{\eta C}{2}\delta_i \tag{21}$$

$$\delta_{i+1} = \delta_i + \frac{eV_{RF}}{m\gamma\beta^2c^2} \sin\left(\frac{2\pi h}{C} z_{i+\frac{1}{2}}\right) \tag{22}$$

$$z_{i+1} = z_{i+\frac{1}{2}} - \frac{\eta C}{2}\delta_{i+1}. \tag{23}$$

This approximation is obtained by replacing the time by a spatial derivative in Equation 3 and setting $\Delta E = 0$ (no acceleration). The longitudinal motion, also called synchrotron motion, is important since many collective effects couple the longitudinal motion into the transverse plane (and vice-versa).

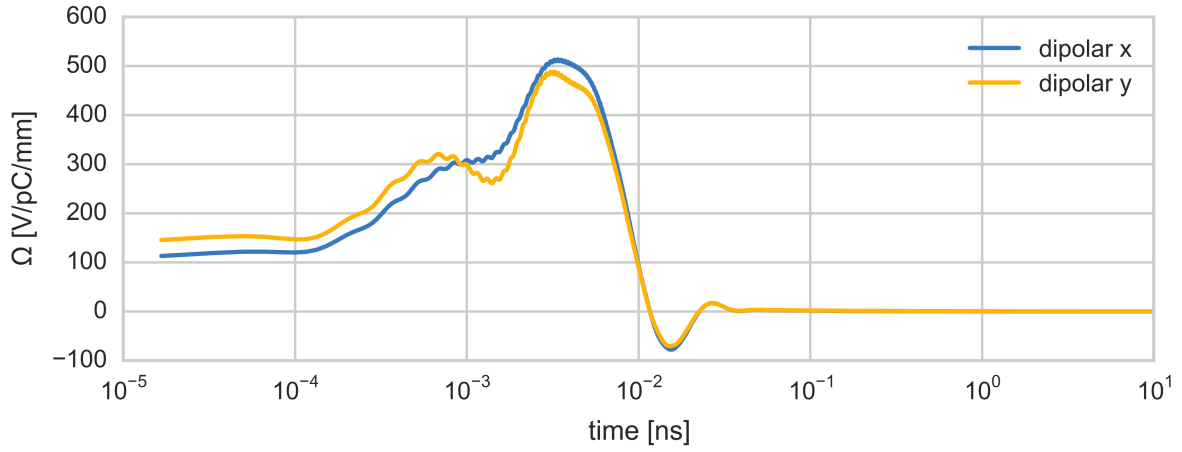

Figure 7: An example of an effective wake field used in the simulation of the LHC. Shown are the transverse dipole components for x and y. The axis is given in time from the source particle for a given velocity.

3.3.3 Feedback: Transverse Damping

A transverse damper corrects particles towards the design trajectory and can dampen certain instabilities. In the most simple approach, the momentum of each particle u'_i gets changed by an amount $\Delta u'$ proportional to average momentum of all particles $\bar{u}'$:

$$\Delta u' = -g_u \bar{u}', \tag{24}$$

where the gain g_u specifies the strength of the damping.

3.3.4 Impedance

This section follows [17] and explains how the impedance effects are computed numerically.

A very accurate model of the synchrotron would require to treat the impedance of each part/device of the accelerator individually. This is however not required to simulate most phenomena. A simpler and computationally more efficient model is to weight and sum all the device impedances to obtain an effective impedance of the whole synchrotron. This effective impedance is then applied as a single kick at one interaction point.

Except for toy-models such as resistive-wall and simple resonators, PyHEADTAIL usually imports impedance models computed and obtained with specialised software which provide the wake as a file. This file can be loaded as a wake-table into PyHEADTAIL. An example of such a wake-table is shown in Figure 7. Given equation 13 for the energy change due to the wake, the change in u' (u either x or y) for a particle at position z and charge e due to a transverse dipole wake (weighting with the offset u) $W_{D,u}(z)$ is

$$\Delta u' = -\frac{e^2}{cp_0} \int_z^{-\infty} \langle u \rangle (z') \lambda(z') W_{D,u}(z - z') dz', \tag{25}$$

where $\lambda(z')$ is the continuous bunch distribution and $\langle u \rangle (z')$ the mean offset of the particles at position z'. To compute this expression, one could approximate the momentum kick by replacing the integral over the bunch distribution with a sum over all preceding particles. To make the computation more efficient, another approach is followed: The bunch is longitudinally binned into slices of length Δz and the kicks are computed on a per-slice basis, see Figure 8.

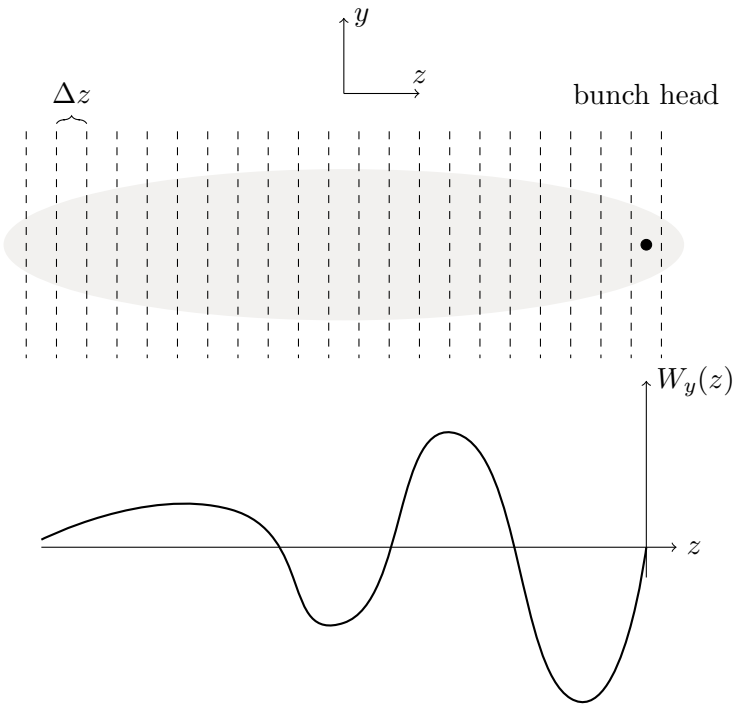

Figure 8: The bunch (in grey) and the longitudinal slicing model in PyHEADTAIL. The beginning and end point of the slices are usually chosen via a standard deviation of the bunch distribution, e.g $[-3\sigma_z, 3\sigma_z]$. The lower plot shows the (transverse dipole) wake generated by the particles in the first slice.

This is a valid approximation as long as the wake function is nearly constant within a slice, and vastly reduces the computational complexity, as the number of slices is typically $\approx 10^2$ while the number of macro-particles is $\approx 10^6$. The number of slices has to be chosen such that the wake-function can be sampled exactly enough assuming a constant wake per slice. Numbering the slices from 0 to n starting at the head of the bunch, a particle in slice $m \geq 0$ is affected by the wake $W_{D,u}(m\Delta z)$ of the N_0 particles in slice 0, $W_{D,u}((m-1)\Delta z)$ of the N_1 particles in slice 1 up to $W_{D,u}(0)$[8] of the N_m particles in the slice itself. The expression for the u'-kick of a particle in slice m is thus

$$\Delta u'_m = -\frac{e^2}{cp_0} \sum_{k=0}^{m} W_{D,u}(k\Delta z) \langle u \rangle_m N_{m-k}, \tag{26}$$

where $\langle u \rangle_m$ denotes the average u of all particles in slice m.

The expression for longitudinal or transverse quadrupolar wakes can be derived similarly.

3.3.5 Space-Charge

The space-charge effect is not part of this thesis and the model is explained only briefly. The particles within a beam feel the Coulomb force from all repulsing neighbouring particles. This force is called direct space-charge and is computed by a particle-in-cell algorithm:

1. The particles' charges are deposited to a rectangular mesh covering the desired region around the beam.

[8] $W_D(0) = 0$. $W(z > 0) = 0$ for a bunch moving with the speed of light

2. The Poisson equation is solved on the grid which yields the potential at each mesh node.
3. Numerical differentiation of the potential is used to compute the electric field at each mesh node.
4. The field is interpolated from the mesh to the particles and weighted by the charge to get the force acting on each particle.

In PyPIC an integrated Green's function approach [16] is used to solve the Poisson equation on a grid with open boundary conditions. The Poisson equation can either be solved in full 3D or on a longitudinal slice-by-slice basis, called 2.5D. PyPIC has been ported to GPUs in 2015.

3.3.6 Electron-Cloud

The electron-cloud effect is not a part of this thesis and the model is explained only briefly. This effect arises due to the build up of electron clouds in the beam pipe.

Different mechanisms, such as the ionisation of residual gas or photo-emission from the surrounding environment (e.g. beam pipe) due to synchrotron radiation emitted by the beam, can generate free electrons in the vacuum chamber. The negatively charged electrons feel an attracting Coulomb force towards the positively charged beam, are accelerated and hit the chamber's wall. Depending on their impact energy, the wall emits multiple secondary electrons, further increasing the electron density. This effect that each electron can trigger the release of multiple other electrons (multipacting) can lead to a strong multiplication of the electrons in the beam pipe. The electric field of a high density electron cloud in the beam pipe can lead to beam instabilities limiting the performance of an accelerator [5].

Electron-cloud effects can be incorporated into PyHEADTAIL via an external software, PyECLOUD. PyECLOUD computes the entire electron dynamics in the presence of a beam, including multipacting, and exports the electric field generated by the electrons for the proton beam to interact with in PyHEADTAIL. In contrast to direct space-charge, the geometry of the surrounding beam pipe is important and open boundary conditions are not a valid approximation. Therefore, a finite-difference based approach with Dirichlet boundary conditions is used for the PIC solver.

3.3.7 Monitor

Monitors provide an easy way to write the state of the system to an HDF5-file. Two often used monitors in PyHEADTAIL are the `BunchMonitor` and the `SliceMonitor`. Both monitors provide means to compute statistics of the particle distribution such as the mean, the variance and the emittance per plane and store them to a file. While the `BunchMonitor` computes these quantities on a per bunch basis, the `SliceMonitor` bins the bunch longitudinally into a variable amount of slices and computes the statistics per slice. The data generated by the `BunchMonitor` is suited to obtain the evolution of macroscopic quantities to compute the complex tuneshift of the bunch. The per-slice data generated by the `SliceMonitor` can be used to analyse intra-bunch motion and oscillations to further quantify the type of instability.

To compute the moments of the bunch distribution required by the monitor, the usual sample estimators are used. The statistical emittance (in the absence of dispersion) ϵ_u of the bunch is computed using

$$\epsilon_u = \sqrt{\sigma_u^2 \sigma_{u'}^2 - \sigma_{uu'}^2}, \tag{27}$$

where σ_u^2 denotes the sample variance of u and $\sigma_{uu'}^2$ the sample covariance of the quantities u, u' [8].

4 Implementation & Methods

Although the main goal of porting PyHEADTAIL to GPUs is achieving a speedup, there are other equally important constraints, namely that using the GPU functionality must be as easy as possible, the existing code should be changed as little as possible, future extensions should be easily integrable, and it is not desired to have two completely different codes for CPU and GPU. This section provides an overview of the libraries and the software design chosen to achieve these goals.

4.1 Programming Language & Libraries

Python was chosen as the programming language to be able to easily integrate the GPU functionality into the existing PyHEADTAIL code. The code is documented directly inside the source files using docstrings.

4.1.1 PyCUDA

Due to the requirement of minimal code changes, PyCUDA was selected as an interface to the GPU. PyCUDA provides high level Python interfaces for CUDA, an API for Nvidia GPUs. PyCUDA tries to combine the dynamic scripting approach of Python with GPU hardware. Simple arithmetic operations can be written as normal Python code and will automatically be run on the GPU. A basic example of PyCUDA is shown in Listing 1.

PyCUDA consists of two parts: One is mainly an object-oriented wrapper for the CUDA runtime library, including methods for memory allocation and (limited) garbage collection. It also contains a `GPUArray` class, similar to `numpy.ndarray`, encapsulating a contiguous block GPU memory. `GPUArray` objects also support basic math operations via the CUDA Math Library. PyCUDA also provides an easy way to write CUDA C code directly inside the Python code, thus allowing meta-programming by modifying the C code directly from Python at runtime [6]. For a more detailed discussion of PyCUDA and its capabilities, [6] or the project's web page[9] are an excellent reference.

4.1.2 scikit-cuda

scikit-cuda[10] is a Python library which provides higher level functions and interfaces to parts of the CUDA Programming Toolkit built on top of PyCUDA. In the scope of this project, only two functions (`diff` and `std`) have been used in connection with `GPUArray` objects.

4.1.3 Thrust

Thrust[11] is a library providing parallel algorithms on the GPU written in C++. The sorting of particles is implemented via Thrust routines. Thrust is installed with every CUDA installation and therefore does not complicate the setup and utilisation of the GPU module.

4.2 Software Design

The GPU functionality is provided using a context manager. The context manager is responsible for moving the data to and from the GPU and selecting the correct implementation of the algorithms and library calls. It serves as a layer between the PyHEADTAIL classes and

[9] `http://documen.tician.de/pycuda/index.html`
[10] `https://github.com/lebedov/scikit-cuda`
[11] `https://developer.nvidia.com/thrust`

```python
import pycuda.gpuarray as gpuarray
import pycuda.autoinit # initialises the GPU context
import numpy as np

# create two arrays a and b on the CPU using numpy
a = np.zeros(1000)
b = np.ones(1000)

# move the data to the GPU
a_d = gpuarray.to_gpu(a)
b_d = gpuarray.to_gpu(b)

# the computation is automatically performed on the GPU
# by launching two kernels (addition, multiplication)
c_d = a_d * b_d + 0.5

# move the result back to the CPU
c = c_d.get()
```

Listing 1: An basic example of how to use PyCUDA

the back-end implementation. While it is sufficient to redirect calls to basic mathematical routines to either NumPy or PyCUDA (e.g. `numpy.sin` vs. `pycuda.cumath.sin`) via the new context module, more complicated functionality requires a rewrite of the underlying function in Python/PyCUDA or CUDA C. Whenever possible, a rewrite in Python/PyCUDA, which runs on both CPU and GPU, is preferred over CUDA C, which will be restricted to the GPU.

The control flow of the program is the following: The user declares the intent to run PyHEADTAIL on a GPU by enclosing the main tracking-loop in his script with a `with`-statement. Upon entering this part of the code, the phase space arrays of the bunch are automatically copied onto the GPU and the global context is set to 'GPU' (the default context is 'CPU'). Inside the `track(bunch)` methods of the various trackers, all calls to specific algorithms and mathematical routines are redirected via the layer and dispatched to the correct GPU routine. See Figure 9 for a schematic overview.

On importing PyHEADTAIL modules, it is automatically checked whether the required libraries (PyCUDA, scikit-cuda) are installed and a GPU is available.

To verify the numerical consistency of the GPU and CPU implementation, more than 40 unit tests were added to the already existing PyHEADTAIL unit test module.

4.2.1 The Context Manager

```
with contextmanager.GPU(bunch) as device:
  machine.track(bunch)
```

The context manager is a class which implements the 'context management protocol' of Python[12] and thus can be used in a `with`-statement. It provides a convenient way to enclose a subregion of code and functions which will be called when entering and exiting this scope. The `contextmanager.GPU` context manager is responsible for copying the bunch phase space, con-

[12] See the corresponding PEP 343, `https://www.python.org/dev/peps/pep-0343/`

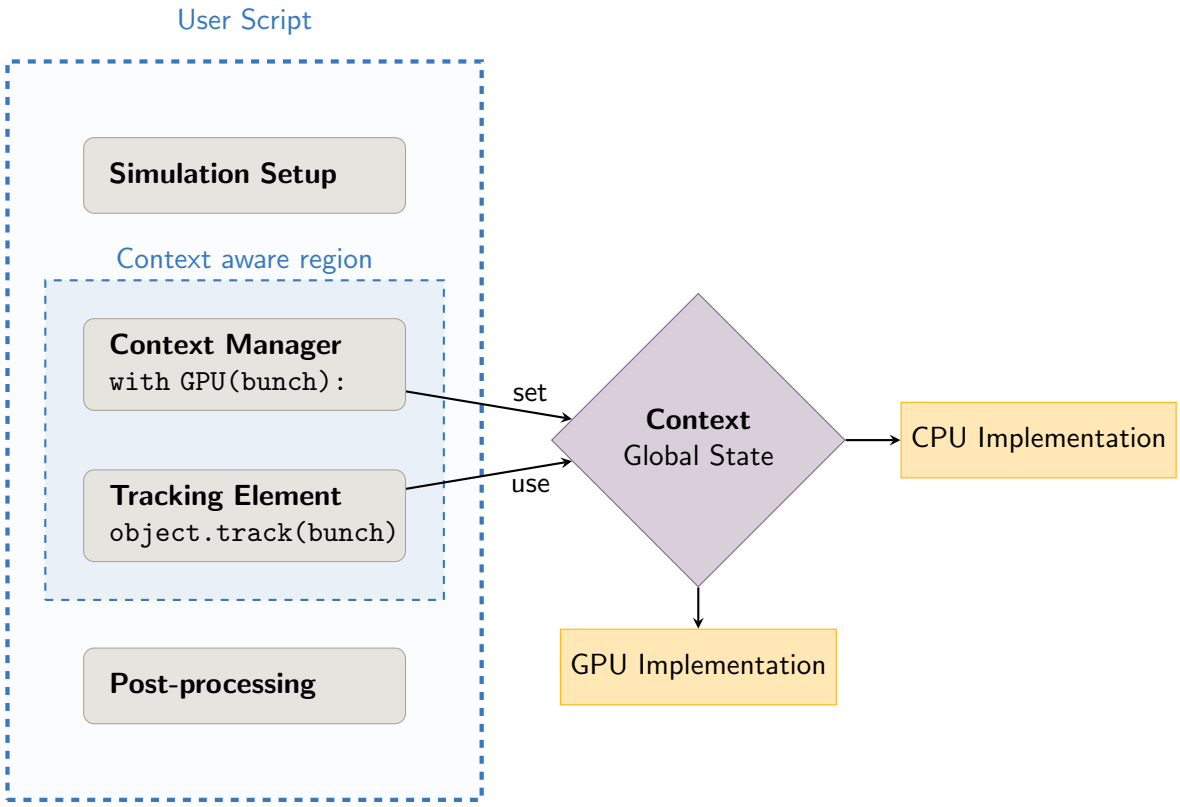

Figure 9: Flow chart of the various modules of the code. The user creates a bunch which the context manager moves to the GPU when a `with`-statement is encountered. All function calls inside the statement that have been designed to be context-dependent are directed via the context and automatically dispatched to the desired implementation.

sisting of six arrays, from the CPU to the GPU when entering and vice-versa when exiting the region. This approach has several advantages:

- The changes to existing user scripts are minimal. Besides the importing of the new `contextmanager` module it requires only one extra line of code.
- It confines the changes of the programme's state to a well-defined and local scope. From a user's perspective it is easy to locate the parts of the script where the bunch data is stored in GPU memory and cannot be directly used as input to other libraries (e.g. for plotting, analysis).
- Future modules and classes that might possibly require a special initialisation on the GPU can easily be incorporated into the existing entering and exiting functions of the statement.
- It is easily extendible to other devices or programming models (e.g. OpenCL, MPI) which require an initialisation and finalisation step. As an example, entering the context could create the MPI communicators and switch to MPI-accelerated implementations of certain functions. Upon exiting the region, the communicators could be destroyed and all bunch data copied back to the main node.

4.2.2 The Context

The context is implemented in a separate module (`pmath`) which stores the currently active context and a reference to the corresponding function implementations. The implementations themselves are grouped in two Python dictionaries (hash-tables), one each for GPU and CPU. When switching the context, the elements of the corresponding dictionary get spilled to the module-global scope and are accessible in other files via `pmath.functionname`. Switching contexts is a lightweight operation and totally transparent to the user via the context manager. All functions which should be available in a given context have to be registered in the context dictionary.

4.3 The CPU Implementation

The CPU implementation stayed the same for most parts of the code. Minor parts (< 100 lines) of the code were changed to create a code running on both CPU and GPU. These changes are completely transparent to the user and other developers since no interfaces were changed.

4.4 The GPU Implementation

This section provides a short overview of the GPU implementation strategies and most important optimisations.

The following list provides a short overview of the available tracking classes and the strategy used to run them on the GPU. The trackers below are embarrassingly parallel. The implementation is straightforward using the PyCUDA `GPUArray` class and required only very small code changes:

- Transverse tracking: `TransverseTracker`
- Longitudinal tracking: `Drift, LinearMap, RFSystems`

The following trackers require some synchronisation between the particles and are not embarrassingly parallel:

- Transverse damping: `TransverseDamper`. The damping requires the calculation of the mean particle position (see Section 3.3.3).

- Wakefield: `WakeField`. The computation of a dipolar wake kick requires the following substeps (compare with Equation 26):

  1. Compute the longitudinal slicing give the number of slices and slice width.
  2. Compute the mean per slice and the number of particles per slice.
  3. Perform the convolution of the wake with the weighted mean per slice from step 2 to get the kick per slice.
  4. Apply the the same kick for all particles within a slice.

  The computation of the mean particle position per slice and number of particles per slice is performed on the GPU with a custom kernel. The computation of the convolution is currently performed on the CPU, even though this requires moving data to and from the CPU each turn. The analysis of a typical use-case showed that the convolution does not contribute significantly to the run-time ($< 10\%$) of the wake field computation. The convolution is performed on relatively small arrays of length 100-1000 compared to the operations involving the particle arrays which have typically 10^6 elements.

- Monitors: `BunchMonitor`, `SliceMonitor`. The computation of various statistics (mean, variance, emittance) is performed using functions provided by PyCUDA (`BunchMonitor`) or custom kernels written in CUDA C (`SliceMonitor`).

With the goal to write as few GPU specific code as possible, most functionality was implemented using the PyCUDA `GPUArray` class. The drawback of this method is the large overhead of memory allocation and temporary variable creation, which can partly be overcome by using memory pools (see Section 4.4.1). All the routines to compute statistics of the particle distribution are methods of the `Particles` class and are automatically dispatched via the context. Code using these functions can be left unchanged and run on both CPU and GPU.

4.4.1 Optimisations

After the functionality on the GPU had implemented and the correctness checked, a typical use case (see section 5.3) was profiled. The profiling revealed that the embarrassingly parallel tracking and the the wake field did not contribute significantly to the run time initially ($< 25\%$), therefore no optimisation effort was spent to specifically speed up these parts. The most important optimisations are listed below:

- **Memory pool**: To decrease the time spent in GPU memory allocation, a memory pool (`pycuda.tools.PooledDeviceAllocation`) is created while entering the GPU-context and all further memory allocations are handled via this pool. Whenever GPU memory is freed, the pool holds on to this memory and tries to reuse it whenever a new chunk of memory is requested. This approach vastly decreases the amount of time spent in the initialisation of `GPUArray` objects (almost a factor 2 of the total runtime in a typical application).

- **Streams**: The computation of the standard deviation, mean and emittance are independent for each of the x, y and z planes. A global stream pool is created upon entering the GPU-context and the streams are used to compute independent statistics of the bunch. To synchronise the streams before and after a function, two decorators

(`decorators.synchronize_gpu_streams_after` and `decorators.synchronize_gpu_streams_before`) are available.

  - PyCUDA does not support specifying a stream for mathematical operators on `GPUArray` objects such as `a + b`, `a *= b` etc. In the most time-consuming parts of the code (e.g the emittance computation), these operations were exchanged with custom `ElementwiseKernels` accepting a stream as a parameter. This is transparent to the user but not to the developers.
  - Inside functions which are running in a separate stream no memory allocation or other operations using the standard stream can be used, since this implicitly forces the streams to synchronise.

- `GPUArray` **special methods**[13]: PyCUDA does not explicitly support operations of type `a -= b` where `a` is a `GPUArray` and `b` a scalar on the GPU without copying `b` to the CPU first. Because this type of operation is often required (e.g in the transverse damper or emittance computation), the `GPUArray` class is patched with custom elementwise kernels providing this functionality upon entering the context. This is a transparent way to provide this functionality for both users and other developers.

- **Sorting**: In order to speed up applying kicks depending on the longitudinal slicing of the beam (e.g. wake-field) and compute statistics on a per-slice basis, the particles get sorted along along their z-position if a slicer is applied. The sorting is split into two sub-steps, the computing of a permutation array and the sorting of the phase space arrays according to this permutation. Both steps are implemented via the Thrust library.

- **Monitor buffer**: Similar to the CPU implementation, the GPU implementation does not write the statistics computed in the monitors to a file after each turn but writes them to a buffer in the GPU memory. The data is copied to the CPU only when writing the data to a file after a user-specified number of steps.

[13] Special or magic methods in Python provide a way for operator overloading in classes, i.e. get called by a special syntax (e.g. `a + b` calls `a.__add__(b)`

5 Results

PyHEADTAIL has successfully been ported to run on Nvidia GPUs. This section presents and discusses the results of the thesis and is split into qualitative results concerning the software, quantitative results (profiling) and a benchmark study performed using the new GPU module.

5.1 Qualitative Results: Usability and Extensibility

The GPU module of PyHEADTAIL is straightforward to use for users running a script and developers creating new trackers or functionality. A user is defined as someone who just wants to run simulations and does not want to add or change anything in PyHEADTAIL. Developers add and change the functionality of PyHEADTAIL and might not necessarily know about GPU programming. Nevertheless, they should be able to develop simple trackers which run on the GPU.

5.1.1 User Perspective

The newly implemented GPU module is transparent to the users of PyHEADTAIL. All existing scripts still work correctly without any changes and simulations on the CPU are not affected by the added GPU functionality.

A user wishing to run a simulation on the GPU can easily adapt his script by enclosing the main tracking loop with the `with`-statement and a GPU context (see Listing 2) and requires no further attention from the user. It is, thus, also suited for users without knowledge of GPUs. Due to an extensive testing suite, GPU simulation is guaranteed to produce the same output as the CPU implementation within reasonable numerical limits. [14]

As a general guideline only simulations with more than 10^5 macro-particles profit from a speedup.

5.1.2 Developer Perspective

The GPU module implemented in this thesis allows for maximal flexibility and transparency for the developers working on PyHEADTAIL. A developer can use the existing framework to add new functionality which should run on a GPU. All standard mathematical operations and statistics are readily available and are - when written as correct Python code - automatically executable on a Nvidia GPU. More sophisticated functionality or modifications that require the rewrite of GPU kernels can easily be integrated in the existing framework by adding them to the GPU-context. See Listing 3 for an example of code that runs on both CPU and GPU.

If the `GPUArray` approach does not yield the desired speedup, more advanced optimisation techniques might be required (such as minimising the creation of temporary objects by writing kernels, using multiple streams, see Section[Optimisations]).

5.2 Quantitative Results: Profiling

All the trackers were timed and the speedup vs. the number of particles was measured. A typical simulation contains between $10^4 - 10^7$ macro-particles, therefore the speedup was measured for $10^3 - 10^7$ macro-particles per bunch. In this section, two representative timings are presented and discussed in detail, one each for embarrassingly parallel and non-embarrassingly parallel methods. The full suite of measurements is attached in Appendix A. A detailed analysis of

[14] Some more 'exotic' methods have not been ported to GPUs yet (e.g. non-uniform longitudinal slicing and `ParticleMonitor`)

```
        # The only additional user import to use the GPU functionality
        from PyHEADTAIL.general.contextmanager import CPU, GPU

        # Some other imports...
        from PyHEADTAIL.monitors.monitors import BunchMonitor
        import PyCERNmachines.CERNmachines as cernmachines
        ...

        # Define some simulation parameters
        n_macroparticles = 100000
        n_turns = 100000
        ...

        # Define the machine
        machine = cernmachines.LHC(n_segments=1, machine_configuration='450GeV',
        longitudinal_focusing=longitudinal_focusing,
        Qp_x=[Qp_x], Qp_y=[Qp_y], Q_s=Qs,
        beta_x=[65.9756], beta_y=[71.5255])

        # Create a bunch
        bunch = machine.generate_6D_Gaussian_bunch(
        n_macroparticles, intensity, epsn_x, epsn_y, sigma_z=sigma_z)

        with GPU(bunch) as device:
            for i in range(n_turns):
                machine.track(bunch)
                bunchmonitor.dump(bunch)
```

Listing 2: A typical user script of PyHEADTAIL making use of the new GPU functionality.

GPU and CPU

```
def track_without_dispersion(self, beam):
  amplitude = e*self.voltage / (beam.beta*c)
  phi = (self.harmonic
       * (2*np.pi*beam.z/self.circumference)
       + self.phi_offset + self._phi_lock)

  delta_p = beam.dp * beam.p0
  delta_p += (amplitude * pm.sin(phi)
              - self.p_increment)
  beam.p0 += self.p_increment
  beam.dp = delta_p / beam.p0
```

CPU

```
def track_without_dispersion(self, beam):
  amplitude = e*self.voltage / (beam.beta*c)
  phi = (self.harmonic
       * (2*np.pi*beam.z/self.circumference)
       + self.phi_offset + self._phi_lock)

  delta_p = beam.dp * beam.p0
  delta_p += (amplitude * np.sin(phi)
              - self.p_increment)
  beam.p0 += self.p_increment
  beam.dp = delta_p / beam.p0
```

Listing 3: Comparison of a typical code segment before and after the changes. Note how the only difference is on line 8. The left code runs on both CPU and GPU. The dynamic typing of Python allows the `beam` object to be either of type `numpy.array` or `pycuda.GPUArray`.

the profiling proved to be difficult: A single line of Python code might spawn several kernels on the GPU as the complexity is hidden inside PyCUDA. Furthermore, the kernels are created

dynamically by PyCUDA and cannot easily be specialised or adapted.

There is a trade off between complexity for the user and developer and possible higher speed-ups.

The code was profiled on on a workstation at CERN consisting of 24 Intel(R) Xeon(R) CPU E5-2630-0 @ 2.30GHz processors and 4 NVidia(R) Tesla C2075 GPUs. The system runs Ubuntu 14.04.3 LTS and the Linux kernel 3.13.0-73-generic. The CUDA version is 7.0. PyCUDA 2015.1.3 and scikit-cuda 0.5.1 were used together with Python version 2.7.10 (Continuum Analytics, Inc) and NumPy version 1.10.1. The CPU implementation is single-threaded, the C/Cython code were compiled using gcc 4.8.4. The initialisation time of the GPU is not taken into account, since its impact over a typical number of turns ($> 10^5$) is negligible.

5.2.1 Transverse Map

The transverse map shows a speedup of up to 27x (Figure 10), the highest of all embarrassingly parallel trackers (see Appendix A). The CPU implementation shows the $\mathcal{O}(n)$ dependency on the number of particles n associated with an independent treating of each particle. The GPU implementation shows the same asymptotic behaviour, for small n however the runtime stays constant due to the available parallelism of the not fully exploited GPU.

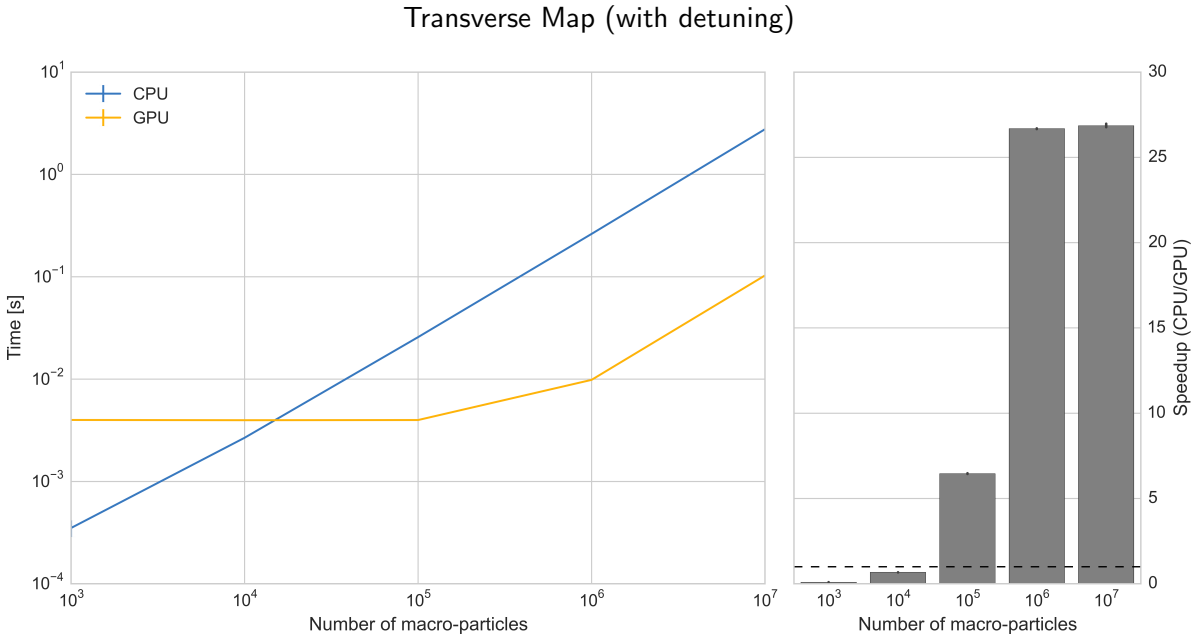

Figure 10: The timing and achieved speedup transverse map including detuning (chromaticity) (`TransverseMap` class). The time is per single invocation of the `track(bunch)` method, averaged over 20 turns.

The GPU implementation is slower than the CPU implementation for bunches consisting of less than 10'000 particles. This overhead can be explained by the use of PyCUDA `GPUArray` objects, where temporary arrays are created for every elemental operation (`+,-,...`). This effect is less important for a large number of particles, where the speedup reaches a plateau at 27x and the GPU is fully exploited.

To further analyse this issue, consider an expression like:

```
x = a * x + b * y ,
```

where `x, y, a` and `b` are `GPUArray` objects. This setup is a simplification of the operations performed in the transverse map. A simple statement like this forces PyCUDA to launch three kernels, one for each operation, as it does not automatically detect the structure of the whole expression. [Each kernel performs an `axbpy` itself because PyCUDA does not have seperate kernels for `*,+`]. This has several drawbacks:

- The kernels perform only one operation per two array elements (8 Bytes each), meaning the arithmetic intensity[15] is very small. Such a low arithmetic intensity means the kernel is bound by the device memory bandwidth and the parallel computing capabilities of the GPU are not fully exploited.
- Launching a kernel produces some overhead compared to the kernel execution time, which is especially important when the amount of computation in the kernel is small.

Profiling the transverse map with the Nvidia profiler confirmed that $> 70\%$ is spent in kernels performing basic `axpby`[16] functionality: The kernel shows a high memory bandwidth utilisation of 115 GB/s (76% of the 150 GB/s theoretical bandwidth) and low utilisation of the arithmetic unit (15%). The longer the arrays, the more threads are spawned on the GPU and the more the memory latency can be hidden by scheduling other threads, which explains Figure 10. The performance measures of the `axpby` kernel are shown in Table 1.

To overcome this problem, the code could be rewritten to call the `axbpy` kernel directly, which would result in only one kernel invocation per such statement. Another possibility would be to rewrite the whole tracking method in CUDA C as one kernel. Both approaches conflict with the goal of rewriting as little code as possible in order to not have to maintain two code bases. Profiling has shown that the transverse tracking time share of a typical simulation is not dominating, therefore the possible gains of this approach are currently not worth the trade-off with the code simplicity.

kernel	**axpby**	
number of particles	10^7	10^3
achieved occupancy (theor.)	64% (66%)	5% (17%)
GPU utilisation	99%	5%
mem. bandwidth (150 Gb/s max)	115 GB/s	5 GB/s
arithmetic occupancy	15%	6%

Table 1: GPU performance metrics for the `axpby` kernel making up $> 70\%$ of the transverse map calls. The GPU utilisation refers to the fraction of GPU time of the total `map.track(bunch)` time, not only this kernel.

5.2.2 Wake Field

The dipolar wake kick shows a speedup of up to 6.5x compared to the CPU. The break-even point between the CPU and the GPU is at $2 \cdot 10^5$ particles. Generally, the wake field shows less speed up than the embarrassingly parallel transverse map.

The detailed analysis of the wake field computation is more involved than the analysis of the transverse map because no single kernel is clearly responsible for the runtime. A profiling of a kernel does not add additional insight and is not shown here. The two most time consuming

[15] The arithmetic intensity (operations count per memory traffic) is a performance measure which indicates whether an algorithm (step) is bound by memory (low) or compute (high) hardware limits.

[16] `axpby` in PyCUDA stands for `x = a*x + b*y`

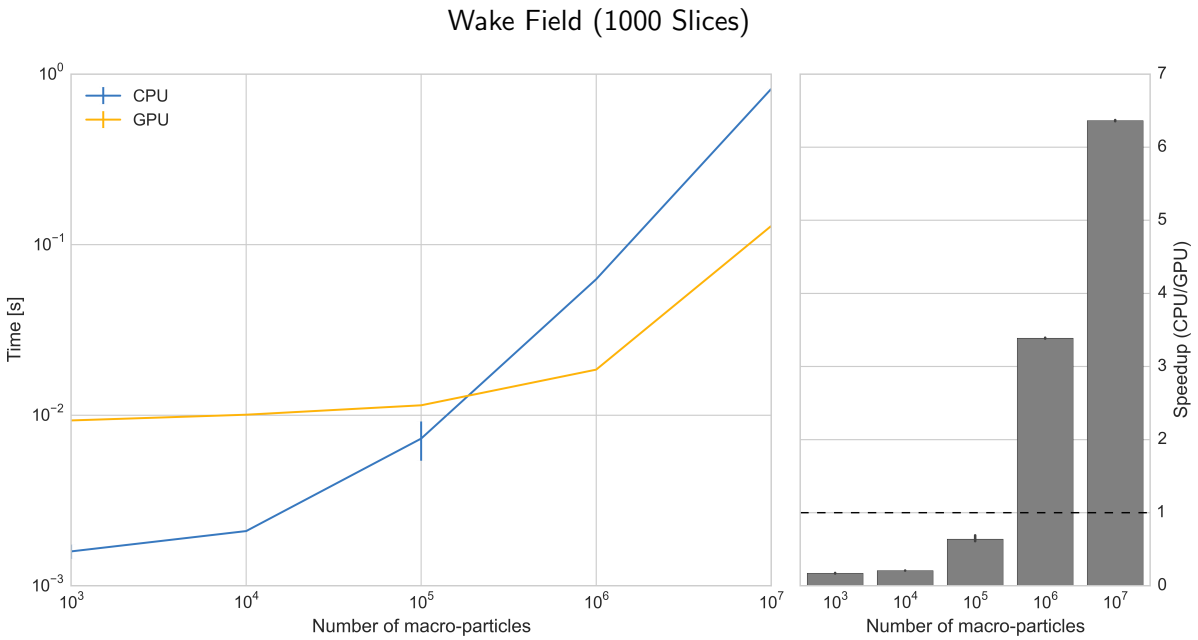

Figure 11: The timing and achieved speedup of the wake field using a wake table (`WakeTable` class). Only dipolar field components in x and y direction are applied.

kernels are the computation of the mean per slice and the sorting of the particles accounting together for 40% of the overall computing time. The GPU utilisation is - even though the convolution is performed on the CPU - as high as 90%.

Two challenges which are relevant to all kernels are

- The uneven work distribution per slice: The normal distribution of the particles in the bunch leads to a very high number of particles per slice in the center of the bunch and a very small number at the head and the tail.
- A lot of kernels have a low arithmetic intensity as the Python code is directly translated into PyCUDA kernels. This is the same problem which occurs in the transverse map and all the code based on `GPUArray` objects.

Performing the convolution on the GPU while leaving the other kernels untouched leads to a maximum speedup of 10% (the CPU time is 10% of the total time) and seems currently not worth the effort. The most promising optimisation is the `mean_per_slice` kernel, being self-written and not hidden inside a PyCUDA implementation.

5.3 Benchmark Study: LHC Instability at Injection

This section presents a benchmark study which was performed to validate the results of the GPU implementation by comparing them with previously performed studies on the CPU. A speedup of 5x was observed and the results agree. Furthermore the results were also compared to real measurements in the LHC. In order to run the script on the GPU, only 2 lines of code had to be added.

Study setup: The study simulates single beam instabilities in the LHC due to impedance effects at injection energy. Identifying the machine parameters (chromaticity, damper gain) which could mitigate these instabilities requires a large amount of simulations to scan the parameter space, each one taking about a day to finish on CPUs. $5 \cdot 10^5$ particles were tracked

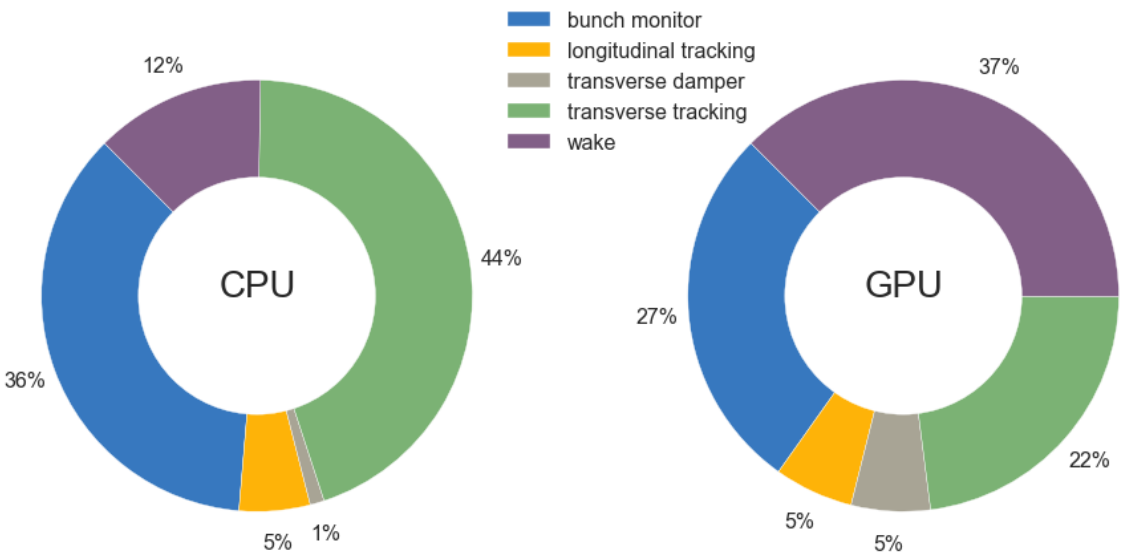

Figure 12: Relative time spent in different parts of the code for the LHC-instability study for both CPU and GPU. In absolute time the GPU is 5x faster than the CPU.

for $5 \cdot 10^5$ turns which corresponds to a 'real' time of 45s given a particles revolution frequency of 11'000 revolutions per second. The exact simulation parameters are shown in Table 2. The model consists of only one transverse tracking segment, a bunch monitor, and one interaction point applying a wake field and a transverse damper. The longitudinal motion is simulated by the linear map shown in Equation 18.

Simulation results: The simulation was run for several chromaticity values and the results agreed with the previously obtained results on the CPU. An example of an instability at a chromaticity of 6 is shown in Figures 13 and 14. The instability developed after $3{\cdot}10^5$ turns in the vertical plane with a rise-time of $\approx$5 seconds. The wake field excites a transverse motion of the particles which, when coupled with the longitudinal motion, leads to the observed oscillation. An instability showing similar oscillations was observed in the LHC in 2015 with the same machine and beam parameters, further validating the results obtained by this simulation (Figure 15).

Speedup: Figure 12 compares the relative time spent in the different parts of the simulation on CPU and GPU. While the CPU code spends most of the time in the transverse tracking, the GPU code spends most of its time in the wake field and bunch monitor. This is because the transverse tracking shows a very good speedup on the GPU (see Figure 10), while the monitor and wake-fields do not. The wake-field does barely show a speedup (compare Figure 11) at less than 1M macro-particles. Simulations consisting of more than one tracking segment would therefore show a better speedup, as would simulations with more macro-particles or longitudinal slices. The speedup of simulations which partition the synchrotron into more than one transverse tracking segment was measured: 2 segments lead to a speedup of 6.5x, 4 segments to a speedup of x8.5.

Conclusion The study confirmed the correctness of the GPU code and showed a good speedup. It therefore enables running more parameter scans and running simulations for a more turns to detect instabilities. However, it also shows that very big speedups can only be achieved by increasing the number of macro-particles, and that there is some optimisation potential in the computation of the wake-field interaction.

Machine parameters	
machine	LHC@injection
energy	450 GeV
chromaticity ξ_x, ξ_y	6
Beam parameters	
ϵ_x, ϵ_y	$3.5 \cdot 10^{-6}$ [m]
σ_z	$1.56 \cdot 10^{-9} \cdot c/4$ [m]
intensity	$1 \cdot 10^{11}$ particles per bunch
Numerical model	
macro-particles	500'000
turns	500'000
segments	1
Twiss parameters	β_x=65.9756 [m], β_y=71.5255 [m]
damping rate x, y	50 turns
wake table:	full machine LHC injection energy B1
wake fields	dipolar x, dipolar y
wake slices:	100

Table 2: Simulation parameters of the benchmark LHC@injection study

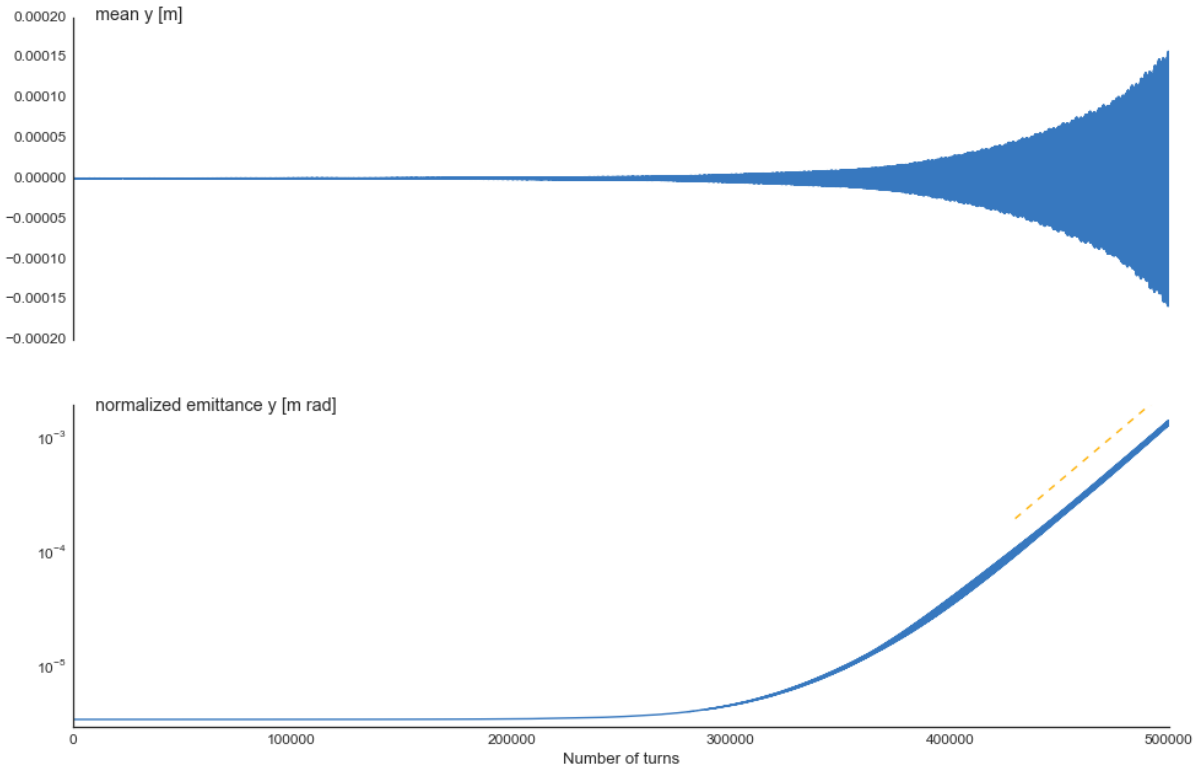

Figure 13: Simulation results for the LHC benchmark study: The growth of the mean y-position (top) and emittance blow up (bottom). The yellow slope is a linear fit to compute the rise time of the instability and suggests an exponential growth as expected. The corresponding intra-bunch motion is shown in Figure 14.

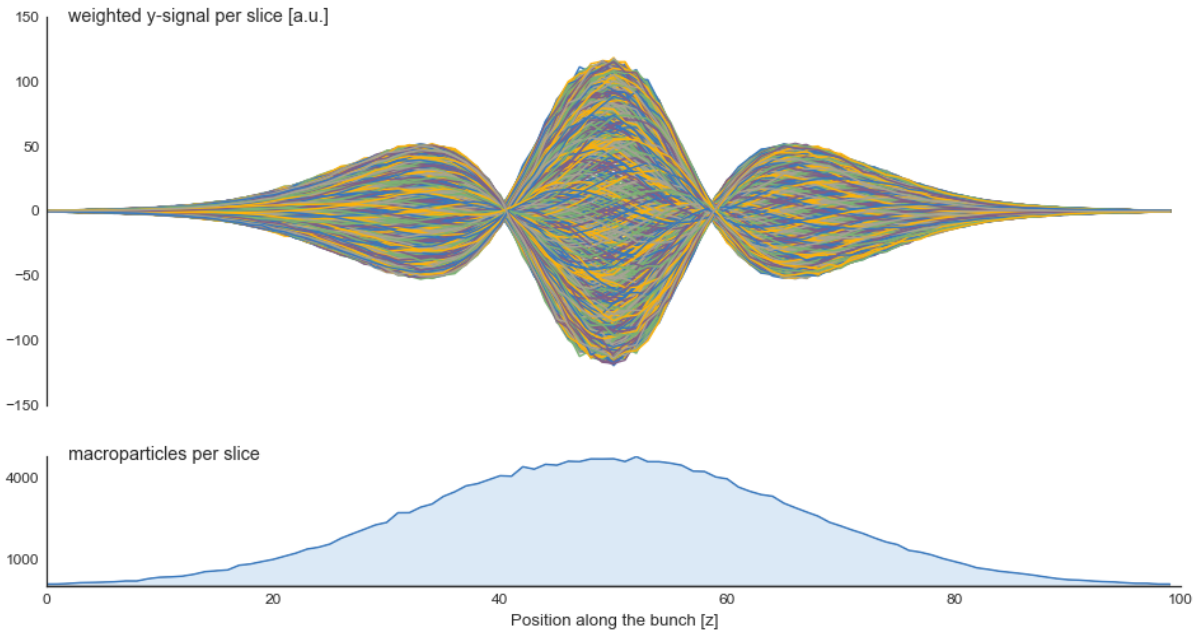

Figure 14: An instability, simulated on the GPU, occurring at a chromaticity of 6 after 500'000 turns. The upper part shows the charge weighted transverse offset per slice along the bunch over 1024 turns in arbitrary units, each line representing one turn. The lower part shows the number of macro-particles per slice.

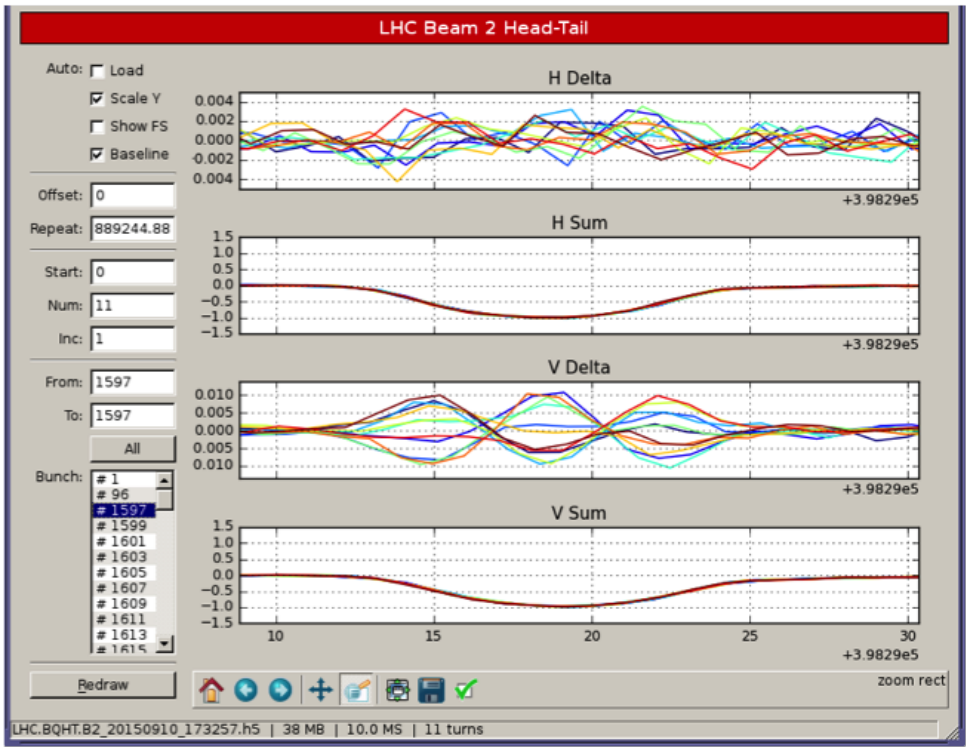

Figure 15: Measurement of an instability observed by the Head-Tail monitor in the LHC. The vertical signal 'V Delta' shows the same mode as predicted by the simulation. The absolute scales cannot be compared since they mainly depend on the calibration.

6 Conclusion

In this thesis, a GPU interface for PyHEADTAIL, a collective effects simulation software in the domain of accelerator physics, has been created and the most relevant tracking elements have been ported to GPUs using the proposed interface. PyHEADTAIL is used to simulate beam instabilities in synchrotrons such as the LHC at CERN, which are important to understand limitations arising due to high-intensity beams.

The implementation is based on PyCUDA which provides a layer between Python and Nvidia GPUs. The dynamic nature of Python and PyCUDA allow to write Python code which often automatically runs on GPUs. The GPU module consists of a context and a context manager which handle the memory transfer to and from the GPU and automatically dispatch the function calls to the correct implementation. The details of software design were given in Section 4.

The developed interface is intuitive for users and easily extensible for other developers. The resulting code has been benchmarked and speedups up to 27x compared to the CPU-version were observed and discussed in Section 5. Bottlenecks and restrictions were identified, the main restriction being the `GPUArray`-caused creation of temporary variables and resulting low arithmetic intensity of the CUDA-kernels. Various optimisation strategies, including the use of multiple streams and custom kernels, have been implemented and reviewed.

The trade-off between usability, readibility and the possible speedup proved to be challenging. While the goal was to write code which runs both on GPU and CPU, some implementations had to be written specifically for the GPU.

The GPU-implementation can be used to vastly reduce the runtime of simulation studies with PyHEADTAIL, allowing to increase the precision of simulations by increasing the number of macro-particles or running simulation for more turns.

Further steps are the implementation of more trackers and the rewriting of routines in CUDA C if a bigger speedup is required. The proposed interface is also compatible with the ongoing implementation of multi-bunch simulations using MPI.

References

[1] A. Adelmann, U. Locans, and A. Suter. The Dynamical Kernel Scheduler - Part 1. *eprint arXiv:1509.07685*, September 2015.

[2] B. A. Bryan. High-performance computing tools for the integrated assessment and modelling of social–ecological systems. *Environmental Modelling & Software*, 39:295–303, 2013.

[3] A. W. Chao. *Physics of Collective Beam Instabilities in High Energy Accelerators*. Wiley Series in Beam Physics and Accelerator Technology. Wiley, New York, 1993.

[4] C. D. Cooper, N. C. Clementi, and L. A. Barba. Probing protein orientation near charged nanosurfaces for simulation-assisted biosensor design. *The Journal of Chemical Physics*, 143(12):124709, 2015.

[5] G. Iadarola. *Electron Cloud Studies for CERN Particle Accelerators and Simulation Code Development*. PhD thesis, Università degli Studi di Napoli Federico II, 2014.

[6] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[7] A. Klöckner, T. Warburton, and J. S. Hesthaven. *GPU Solutions to Multi-scale Problems in Science and Engineering*, chapter High-Order, pages 353–374. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] S. Y. Lee. *Accelerator Physics*. World Scientific, Singapore, 3rd edition, 2012.

[9] K. Li. Beam Instabilities Simulations. Unpublished manuscript, 2015.

[10] K. Li. Numerical Methods I (CERN Accelerator School - Intensity Limitations in Particle Beams) [Lecture notes]. `https://cas.web.cern.ch/cas/Intensity-Limitations-2015/Lectures/Friday6/Li1.pdf`, 2015. [Online; accessed 04-February-2016].

[11] K. Li, H. Bartosik, A. Oeftiger, and G. Rumolo. PyHEADTAIL [Presentation]. `https://indico.cern.ch/event/320542/attachments/618347/850770/PyHEADTAIL-Meeting-ed.pdf`, 2014. [Online; accessed 04-February-2016].

[12] K. Li, A. Oeftiger, and G. Rumolo. cobra-HeadTail [Presentation]. `https://emetral.web.cern.ch/emetral/ICEsection/2013/2013-11-06/ICE-Meeting.pdf`, 2013. [Online; accessed 04-February-2016].

[13] K. Ng. *Physics of Intensity Dependent Beam Instabilities*. World Scientific, Singapore, 2005.

[14] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30:56–70, 2010.

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96:879–899, 2008.

[16] J. Qiang, S. Lidia, R. Ryne, and C. Limborg-Deprey. Erratum: Three-dimensional quasistatic model for high brightness beam dynamics simulation [Phys. Rev. ST Accel. Beams 9, 044204 (2006)]. *Physical Review Special Topics - Accelerators and Beams*, 10(12):2006–2007, 2007.

[17] G. Rumolo, H. Bartosik, and K. Li. Collective Effects in Beam Dynamics (US Particle Accelerator School) [Lecture Notes], 2015.

[18] G. Rumolo and F. Zimmermann. Practical User Guide for HEADTAIL, 2002.

[19] J. Sainio. PyCOOL — A Cosmological Object-Oriented Lattice code written in Python. *Journal of Cosmology and Astroparticle Physics*, 2012(04):038–038, 2012.

[20] W. C. Swope, H. Andersen, P. H. Berens, and K. R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637, 1982.

[21] C. Zannini. *Electromagnetic Simulation of CERN Accelerator Components and Experimental Applications*. PhD thesis, EPF Lausanne, 2013.

A Further profiling results

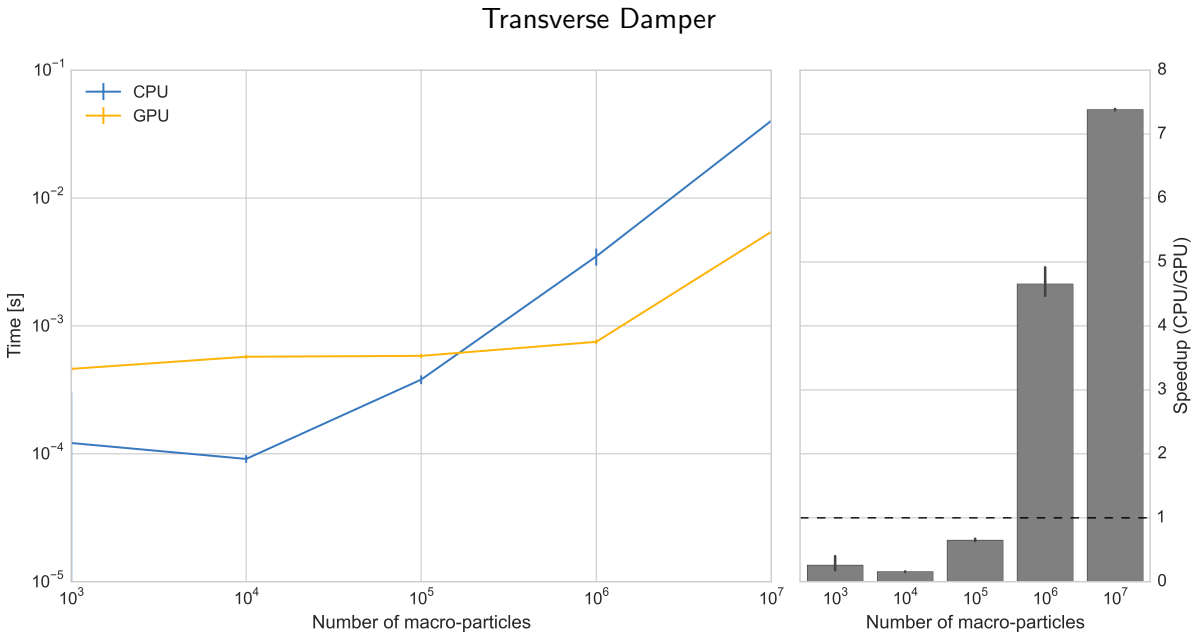

Figure 16: The timing and achieved speedup the transverse damper (`TransverseDamper` class)

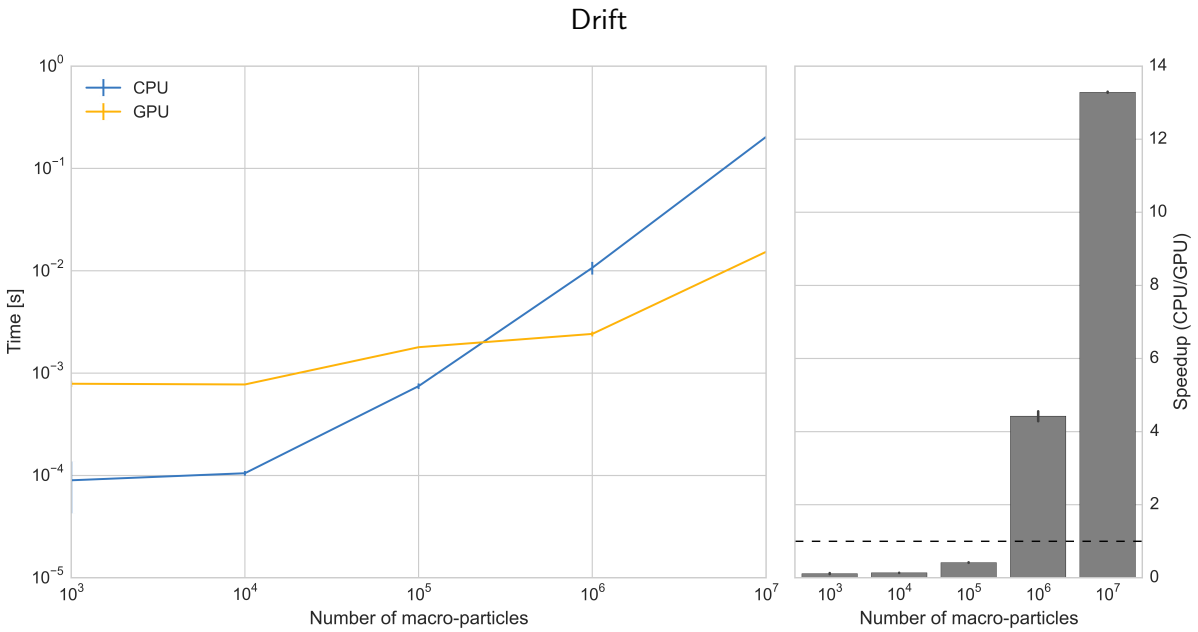

Figure 17: The timing and achieved speedup of the longitudinal drift (`Drift` class)

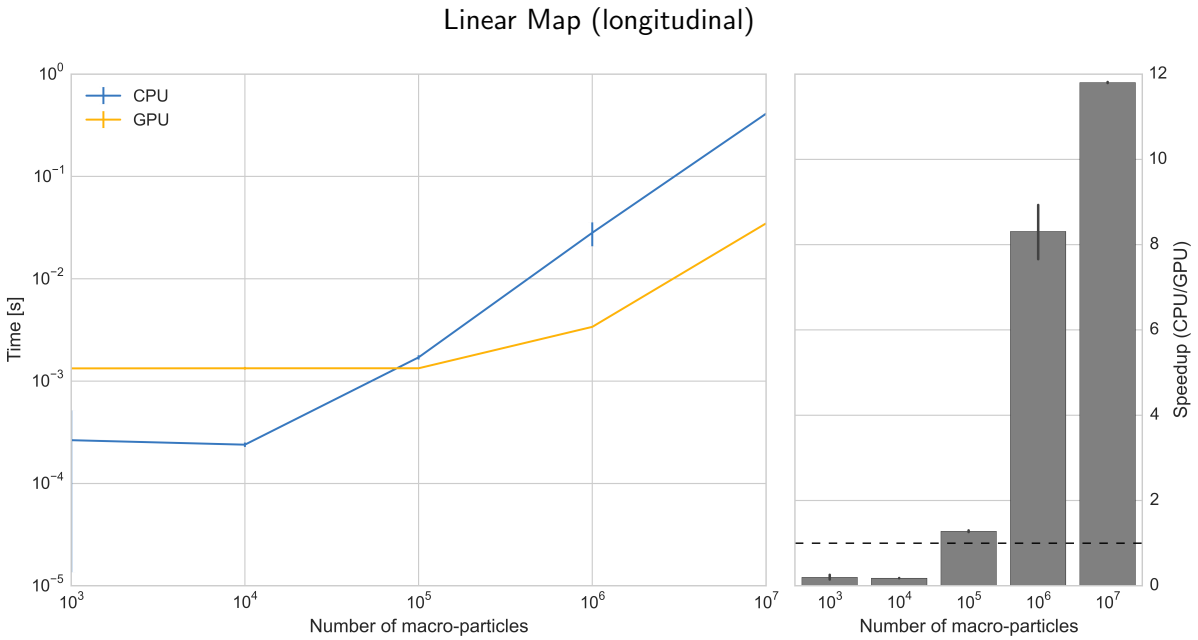

Figure 18: The timing and achieved speedup of the longitudinal linear map (`LinearMap` class)

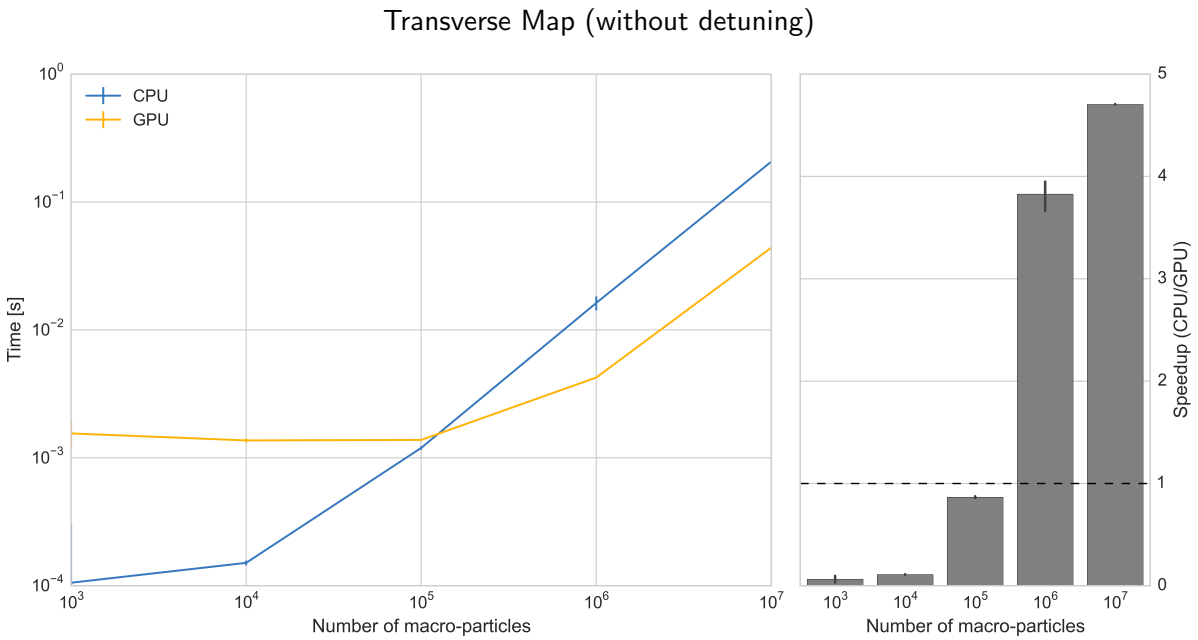

Figure 19: The timing and achieved speedup transverse map (`TransverseMap` class)

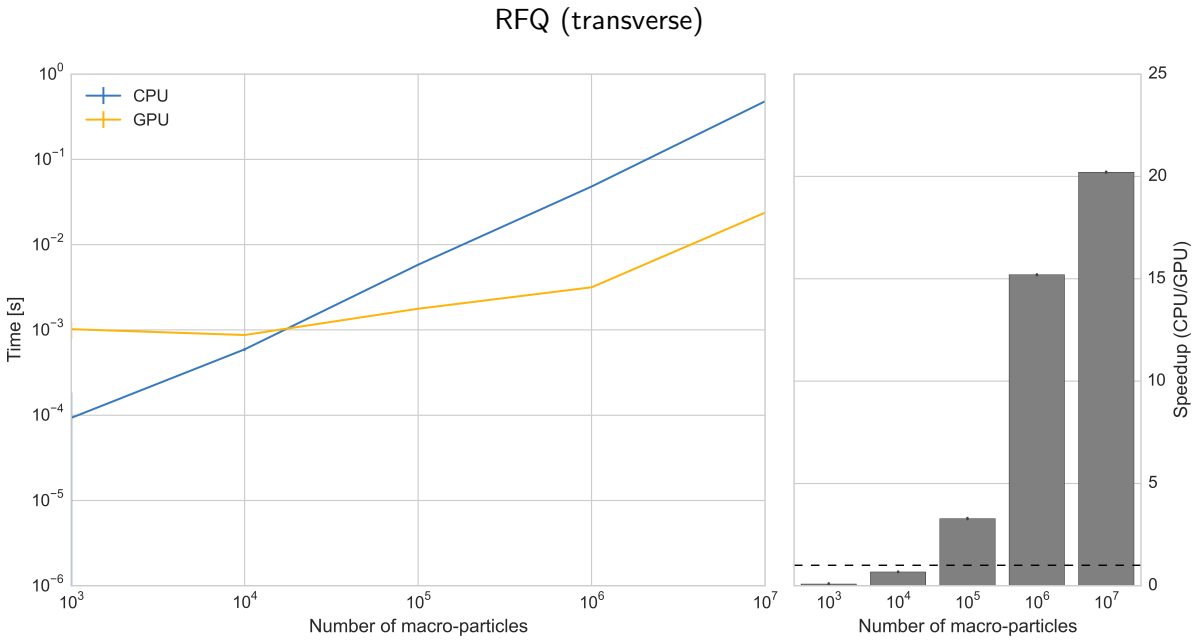

Figure 20: The timing and achieved speedup RFQ transverse tracking (`RFQTransverseKick` class)

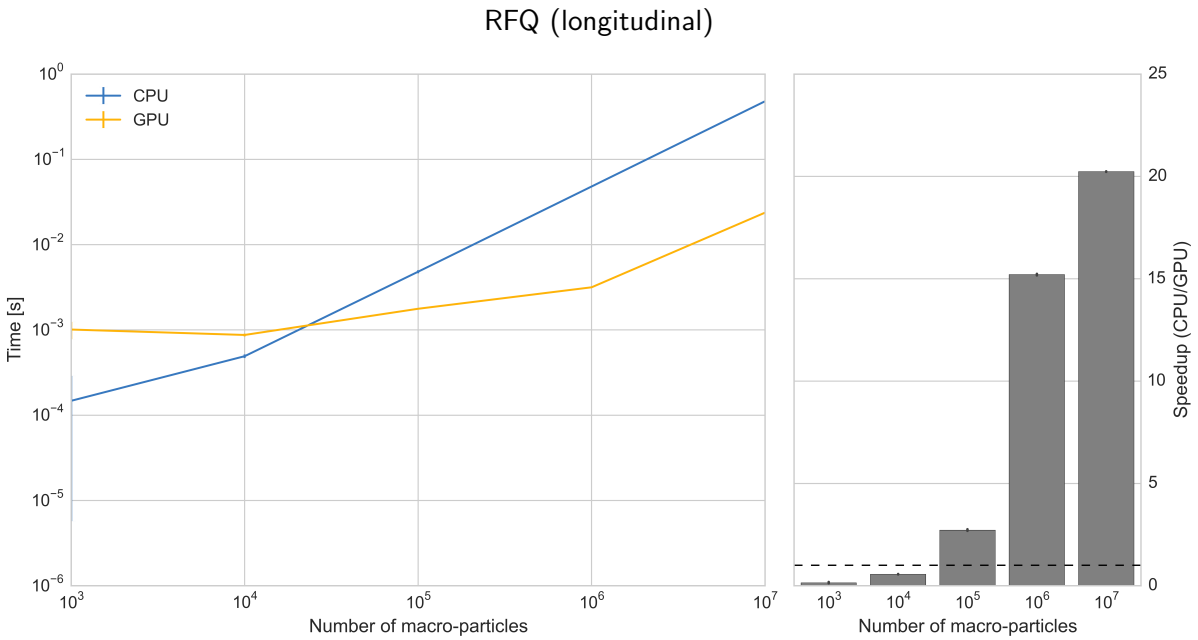

Figure 21: The timing and achieved speedup RFQ longitudinal tracking (`RFQLongitudinalKick` class)

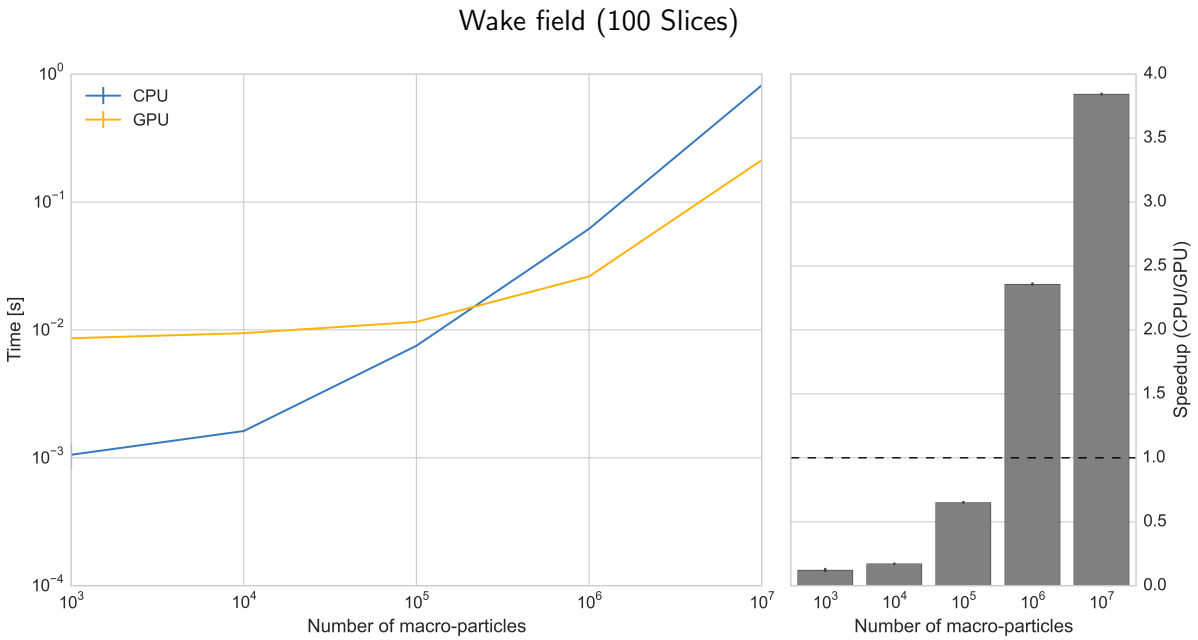

Figure 22: The timing and achieved speedup of the wake field using a wake table (`WakeTable` class). Only dipolar field components in x and y direction.

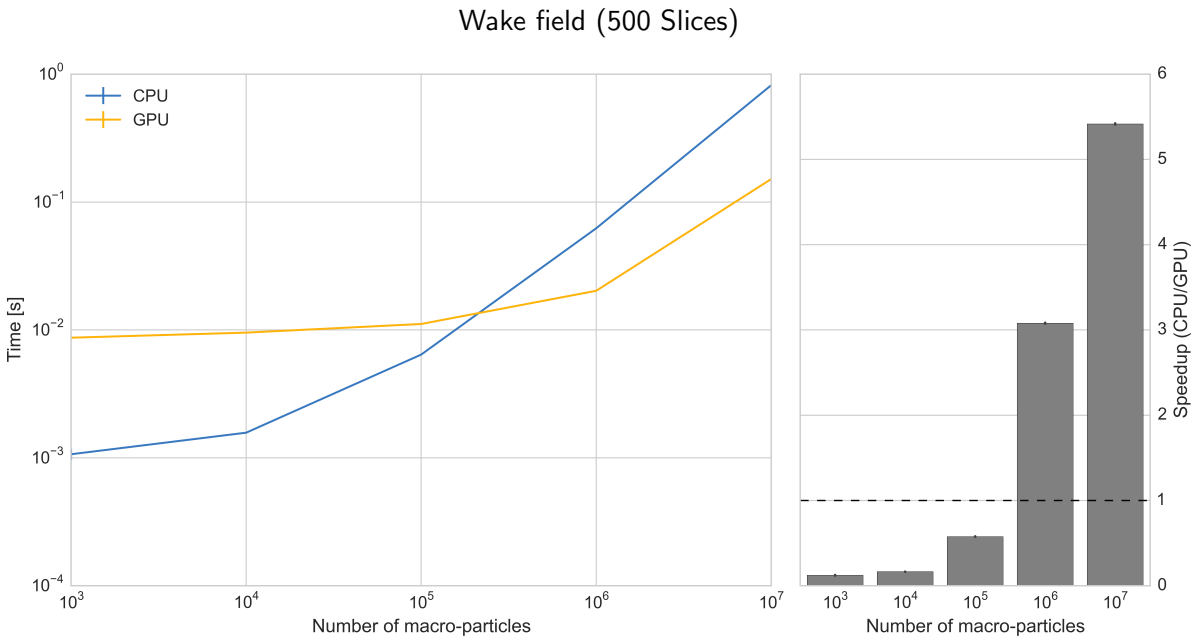

Figure 23: The timing and achieved speedup of the wake field using a wake table (`WakeTable` class). Only dipolar field components in x and y direction.