Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Informatik und
Wirtschaftsinformatik

HKA

Master Thesis

Name:	Bernardo Abreu Figueiredo
Topic:	Evaluation of Different Methods for Runtime Optimization of Longitudinal Phase Space Tomography
Place of work:	CERN, Meyrin
Supervisor:	Prof. Dr.-Ing. Vogelsang
Co-examiner:	Prof. Dr. Körner
Deadline:	11/03/2024

Karlsruhe, 12/09/2023

The Chairman of the examination committee

Prof. Dr. Heiko Körner

Eidesstattliche Erklärung
Statutory Declaration

(English version below)

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Masterthesis selbständig und ohne unerlaubte Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

I hereby declare on oath that I have written this Master's thesis independently and without unauthorized assistance, that I have not used any sources or aids other than those specified and that I have marked the passages taken from the sources used as such, either literally or in terms of content. This thesis has not been submitted to any other examination board in the same or a similar form and has not been published.

Ort, Datum	Unterschrift
Place, Date	*Signature*

Zusammenfassung

Um die Parameter und damit auch die Qualität des Teilchenstrahls in den CERN-Synchrotrons zu messen, werden verschiedene Methoden zur Strahldiagnostik eingesetzt. Eine davon ist die longitudinale Phasenraumtomographie, mit der aus Messungen von Partikelbündeln die Verteilung der Teilchen im Phasenraum rekonstruiert werden kann. Aufgrund der hohen Anzahl von Parametern und Berechnungen dauert der gesamte Prozess für eine einzige Rekonstruktion meist einige Sekunden. Für einige Anwendungen ist eine große Anzahl von Rekonstruktionen erforderlich, sodass die Größenordnung schnell von einigen Sekunden auf mehrere Minuten oder sogar Stunden ansteigen kann. Für andere sind schnellere Rekonstruktionen gewünscht, sodass einige Sekunden schon zu lang sind.

Diese Arbeit vergleicht verschiedene Ansätze zur Optimierung der Laufzeit der longitudinalen Phasenraumtomographie. Dabei werden hauptsächlich Ansätze mit GPUs sowie Frameworks zur Optimierung von Python-Code verwendet. Die Laufzeiten werden mit der aktuellen CPU-Implementierung verglichen und eine GPU-basierte Version wird auf Basis der Erkenntnisse implementiert. Am Ende steht eine schnellere Anwendung mit CuPy und CUDA C++ zur Verfügung, die je nach GPU-Modell Laufzeitverbesserungen mit einem Faktor zwischen sechs und 19 erreicht. Die Anwendung erreicht eine Laufzeit von unter 300 Millisekunden für die Rekonstruktion an sich, während die Gesamtlaufzeit der Anwendung weniger als zwei Sekunden dauert.

Abstract

In order to measure the parameters, and therefore the quality, of the particle beam in the CERN synchrotrons, various methods are used for beam diagnostics. One of them is longitudinal phase space tomography, with which the distribution of particles in the phase space can be reconstructed from measured bunch profiles. Due to the large number of parameters and the high number of calculations, the entire process for a single reconstruction usually takes a few seconds. For some applications, a large number of reconstructions are required, so that the order of magnitude can quickly increase from a few seconds to several minutes or even hours. For others, fast reconstructions are desired, in which case several seconds is too long.

This work compares different approaches to optimize the runtime of the longitudinal phase space tomography. Thereby, mainly approaches with GPUs are used as well as frameworks for optimizing Python code. The runtimes are compared with the current CPU implementation and a GPU based version is implemented on the basis of the findings. In the end, a faster application is created using CuPy and CUDA C++, reaching speedups between six and 19, depending on the GPU model, and reconstruction runtimes of below 300 milliseconds, while the whole application workflow takes less than two seconds.

Acknowledgements

First of all, I would like to thank my supervisor from Hochschule Karlsruhe, Prof. Dr. Holger Vogelsang, for agreeing to supervise my master's thesis. Without his support for my application at CERN, this would not have happened.

I am also deeply grateful to my supervisor from CERN, Dr. Simon Albright, for the invaluable guidance, support, and supervision throughout my project. Despite my lack of knowledge in accelerator physics, he was very patient in sharing his expertise and explaining me everything I needed to understand for my project.

Next, I want to say thank you to my friends and colleagues in SY-RF-BR, especially Dr. Konstantinos Iliakis, Mariangela Marchi, Oleksandr Naumenko, and Dr. Leandro Intelisano.
Konstantinos contributed a lot to my project by sharing his expertise in GPU acceleration and helped me to further advance in my project. Besides many hints for GPU programming, he explained me how to the HPC cluster at CERN works and how I can conduct benchmarks in it.
Mariangela supported me a lot during my journey at CERN as well, as a friend and colleague. Her patience in making me understand the basics of accelerator physics better was crucial and helped me to get more acquainted with the topic.
Oleksandr, with whom I shared the office, was a great help to me in many aspects. When it came to understanding physics, he was always available and helped me to improve.
Finally, Leandro's help enhanced my thesis aesthetically by showing me how to make my plots in PGF/TikZ.
Thank you to everyone of the section for your valuable advice, the nice chats and the good time I spent here.

A big thank you also goes to Nils Ruf, who proofread my writing and who was a great friend throughout our entire studies in Karlsruhe.

I also want to extend my deepest gratitude to my family. Without their caring and invaluable support throughout my entire life, I would not be where I am right now.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The following sections provide a brief overview of the present work and the context in which it was created. First, CERN as the institutional entity in which the work takes place will be introduced as well as the group and their research field. This is followed by a brief motivation of the problem and how this work aims to optimize the longitudinal tomography. Subsequently, the goals and the necessary requirements of the work will be outlined.

1.1 Work Environment

The present thesis takes place at CERN, the European Organization for Nuclear Research, the largest particle physics laboratory in the world, comprising 23 member states and located near Geneva, Switzerland. Here, a community of more than 17000 scientists from more than 80 different states collaborate together to study the foundational building blocks of the universe [1]. CERN hosts and provides a range of particle accelerators to enable research in fundamental physics.

The SY-RF-BR section (Accelerator Systems department, Radio Frequency group, Beams & RF Studies section) focuses on the particle beam and its environment, including through studies of longitudinal beam dynamics. This involves designing future colliders as well as upgrading and improving the performance of the existing accelerators with a focus on the design and production of new Radio Frequency (RF) systems. Furthermore, beam diagnostics in the longitudinal plane is another point of interest. In this context, Longitudinal Phase Space Tomography is applied as well as tools like beam quality monitors. In addition, the section is developing and maintaining the Beam Longitudinal Dynamics (BLonD) code [2] which is used for macroparticle simulations [3].

Particle accelerators are used to increase or decrease the energy of charged particles. The stream of particles inside an accelerator is referred to as a beam which may be grouped into one or more ensembles of particles called bunches. Depending on the experiment and the objective of the study, the particle beam will reach a certain energy with specified beam parameters. Figure 1 shows the CERN accelerator chain. The proton chain starts with LINAC4, a linear accelerator. The beam is then injected into the Proton Synchrotron Booster (PSB), in the figure also labeled as BOOSTER, the first circular accelerator in the proton chain. After increasing the energy of the particles, the chain continues to the Proton Synchrotron (PS), the Super Proton Synchrotron

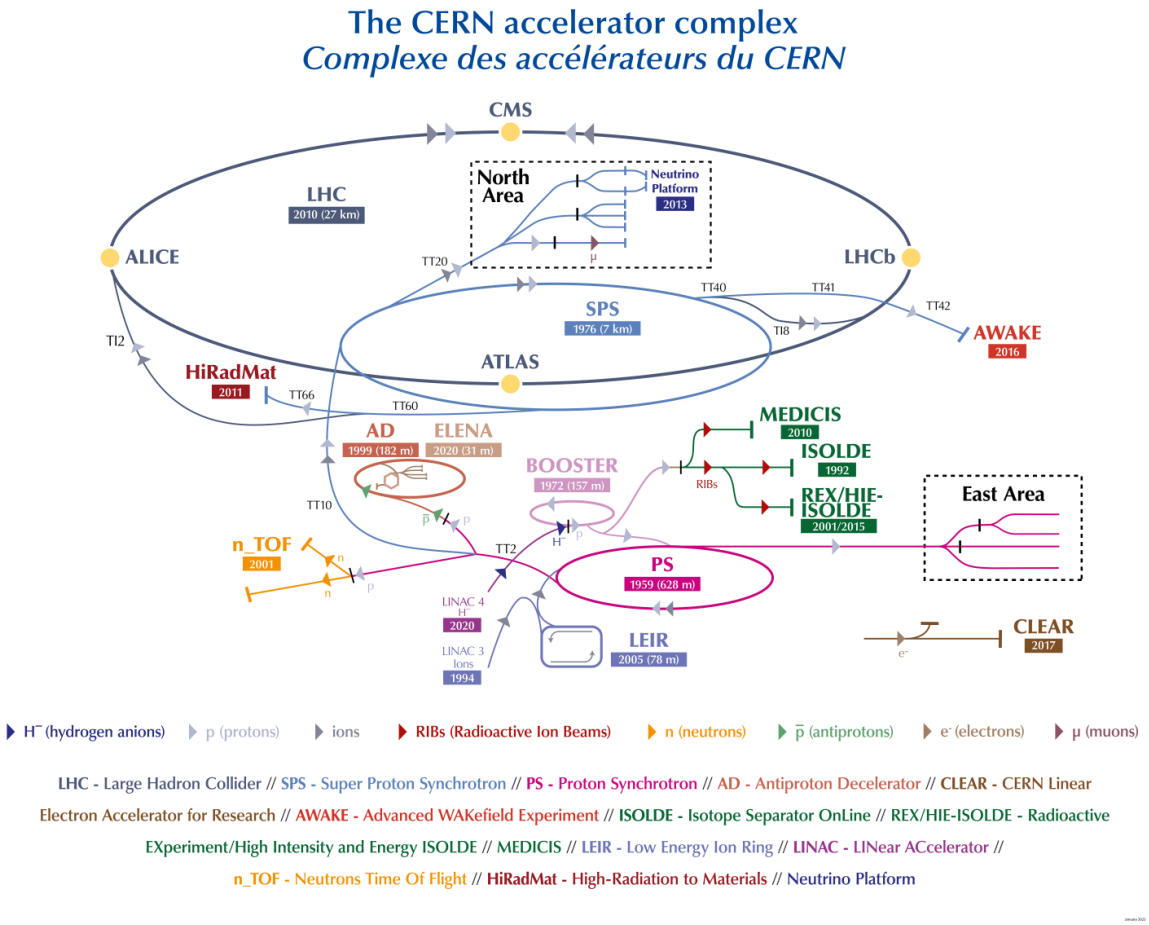


Figure 1: The CERN accelerator complex, as of 2022 [4].

(SPS), and finally, the Large Hadron Collider (LHC). The further the chain goes, the higher the circumference of the accelerators to cope with the increasing energy of the particles.

1.2 Motivation

The analysis of the beam's phase space distribution in an accelerator is a vital part of beam diagnostics. This distribution gives insights about beam stability and the time and energy distributions in the longitudinal plane. Furthermore, it provides a picture of how close the particles of the bunch are to each other and what the offset is with respect to a particle on the ideal trajectory. This particle is taken as a reference (the so called *synchronous particle*). The phase space distribution contains valuable information about the beam. However, measuring the phase space directly is not possible. The time projection can be measured for example using a Wall Current Monitor [5], whereas the energy projection cannot be directly measured. For this purpose, *Longitudinal Phase Space Tomography*, in this work mostly referred to as *Tomography*, is used to reconstruct

the phase space from inputs coming from machine parameters and raw measurements.

Tomography is the method of reconstructing an n-dimensional object from its (n-1)-dimensional projections [6]. The underlying principle is to merge the information of several pictures to reconstruct the full picture with the extra dimension. This technique is widely used for medical purposes, for example when many one-dimensional x-ray profiles are taken from different angles to reconstruct an image of a two-dimensional slice through a patient. A bunch moves in the longitudinal plane and its motion has a certain period and frequency. Such a motion is defined as synchrotron motion and can be interpreted analogously to a patient rotating in a stationary body scanner, making longitudinal phase space reconstruction a valid application of tomography [7]. Instead of rotating the patient, the natural motion of particles used, which is explained in the following chapter. The input used for longitudinal phase space tomography is a series of one-dimensional measurements of the bunch profiles.

The longitudinal tomography algorithm has initially been implemented in Fortran95 and optimized using the High Performance Fortran (HPF) extension [8]. It was originally designed for optimal speed and memory usage, which were significant considerations due to the computing power available at the time. However, introducing new features would require a significant effort due to the structure of the original implementation of the algorithm [9]. To account for this, the code was translated and refactored into a Python library which increases the flexibility of the code and facilitates future changes as well as the incorporation of new features. In order to maintain comparable runtimes, the computationally expensive parts were written in C++, which is directly accessed from the Python code [6].

In the context of analyzing the longitudinal phase space of the beam in the accelerator online, the runtime of the algorithm is a crucial factor since the cycle duration for the accelerators can be low, for example 1.2 seconds in the PSB. In the most demanding case, the time for a tomography reconstruction should be around 200 milliseconds in the context of automated tomography, setting a constraint if one wants to use tomography multiple times per cycle [10]. Another use case where runtime is an important factor is when performing voltage calibration studies. Since it is not possible to measure the voltage seen by the beam directly, one can use tomography to evaluate it with a certain accuracy [11, 12]. In this case, tomography is performed multiple times since the input dataset considers different voltages. These are used to calculate the difference between the applied and the actually detected voltage by the beam, as a consequence of the accuracy in the reconstruction of the phase space.

Considering these use cases, it is desirable to improve the runtime of the tomography process. Since there are multiple computationally intensive parts which are repeated for many different particles, there is the potential to parallelize the algorithm to perform the same calculations on many data points at the same time. One way to achieve this is to use the power of a Graphics Processing Unit (GPU) which besides its use for computer graphics is also specialized in massively parallel calculations. Since there are different GPUs available at CERN, these could be potentially used to enhance the performance of the tomography application.

1.3 Goals and Requirements

The main goal of the present work is to optimize the runtime of the longitudinal phase space tomography using different frameworks and optimization techniques, with a focus on developing the tomography algorithm for GPUs. For this purpose, a performance study was conducted to identify the bottlenecks of the current implementation and ways to optimize them. While the new implementation uses GPUs, the current CPU implementation should be kept as an available option. One requirement is the maintainability of the library, meaning that the optimized version and the current version should keep a unified interface. This implies that most parts of the workflow should be kept in Python as well.

Besides achieving a better performance and keeping the library maintainable, another requirement is that the quality of the tomographic reconstruction remains equivalent between the GPU and the CPU implementation. The runtime-optimized should not change the expected results, or only within a certain tolerance. In the course of the thesis, the GPU version will be tested to compare the results between the new and the current implementation.

2 Fundamentals in Accelerator Physics and Tomography

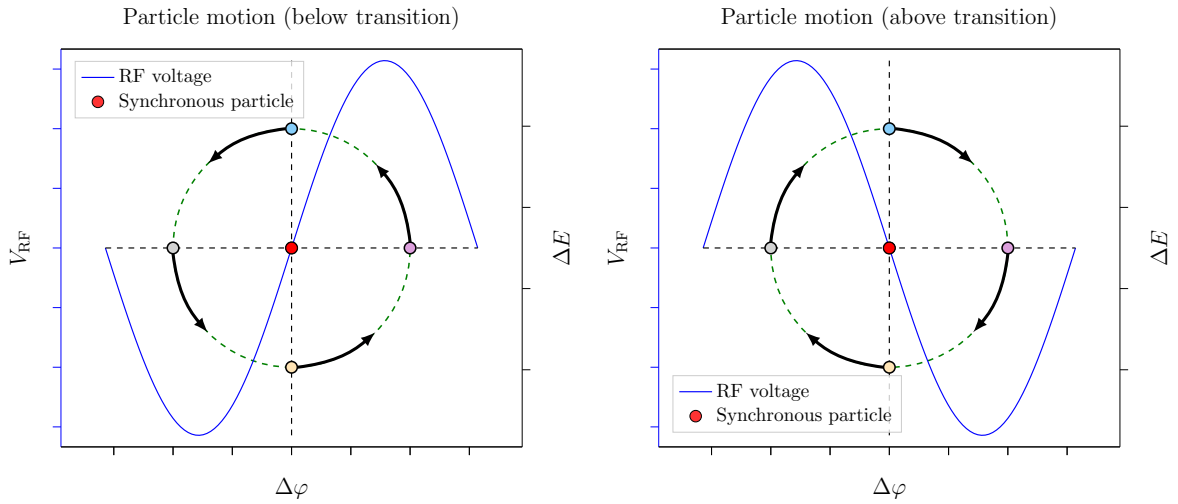
This chapter covers the fundamentals to understand the context in which this work is embedded. First, as part of the introduction to accelerator physics, synchrotron motion will be introduced as well as the concept of the phase space distributions to visualize particle bunches in the longitudinal plane. Subsequently, tracking will be introduced. Following this, the principle of longitudinal phase space tomography will be explained in detail.

2.1 A Brief Introduction to Accelerator Physics

The circular accelerators mentioned in the introduction are also known as synchrotrons. In this type of accelerator, the orbit around which particles are accelerated is kept constant and magnets are used to bend the trajectory of the particles to follow the circular path of the machine. Radio Frequency (RF) cavities are used to change the energy of the particles [13]. These are metallic chambers inside which oscillating electro-magnetic fields are established. Every time a particle passes through an RF cavity, it will be exposed to the electro-magnetic field inside the cavity and consequently experience a force which changes its energy. This energy kick increases or reduces the energy of the particle, depending on the phase it arrives at the cavity. Since the electric field changes in time, a synchronization is needed between the frequency of the electric field in the cavity and the motion of the particle [14, 15]. Additionally, when particles travel through the ring, their phase will drift. The longitudinal motion of a particle in a synchrotron can be described by alternating kick and drift.

The motion of a particle or a particle bunch can be described by phase space coordinates. For this purpose, the concept of a synchronous particle is introduced which is defined as having the ideal phase and energy and is used as a reference point for other particles' phase and energy coordinates. The relative energy gain or loss for a particle depends on the phase offset with respect to the synchronous particle and the amplitude of the RF voltage. The plots in Figure 2 show an example for the motion of particles below and above transition energy. The horizontal axis refers to the difference in phase with respect to the synchronous particle, here defined as $\Delta\varphi$, while the right vertical axis refers to the energy difference with respect to the synchronous particle, defined as ΔE. Additionally, the left vertical axis shows the voltage of the RF system, shown in the plot as a blue sine curve.

Depending on the design parameters of a synchrotron and the initial and final energies



(a) Synchrotron motion below transition.

(b) Synchrotron motion above transition.

Figure 2: An example of synchrotron motion with the synchronous particle in the origin [13].

of the particles to be accelerated, two regimes can be distinguished which describe the evolution of synchroron motion differently. For low energy, the machine is said to work below transition energy. The transition between the two regimes is determined by the machine configuration and occurs at a particular energy. When a particle's energy is increased, the velocity can increase as well as the orbit length which both impact the duration of a turn. When a machine works below transition energy, the change in velocity has a stronger impact than the change in path length, leading to a faster revolution. The opposite is the case for machines working above transition energy where the path length has a stronger impact and therefore increases the duration of a revolution.

In Figure 2a, the motion is shown for particles below transition energy. When a particle arrives earlier than the synchronous particle, i.e. $\Delta\varphi$ is negative, it will see a lower voltage than the synchronous particle which decreases the relative energy. The gray particle is an example where this happens. Since it loses energy it arrives relatively later than before, thus $\Delta\varphi_{n+1} > \Delta\varphi_n$ for turns $n+1$ and n, as well as $\Delta E_{n+1} < \Delta E_n$. For the pink particle, the opposite happens. It arrives later than the synchronous particle, i.e. $\Delta\varphi$ is positive, therefore it sees a higher voltage and the relative energy increases, thus $\Delta E_{n+1} > \Delta E_n$. As a consequence, the particle will arrive earlier at the next turn with respect to the synchronous particle, so $\Delta\varphi_{n+1} < \Delta\varphi_n$.

For the motion above transition energy, as shown in Figure 2b, the behavior is inverted. The gray particle with negative $\Delta\varphi$ arrives earlier and sees a higher voltage. Therefore the relative energy increases instead of decreasing, which increases the relative phase.

Overall, the particles move in the opposite direction in the phase space when comparing motion below and above transition. The motion of particles can be computed by means of the longitudinal equations of motion [16]. These are given by

$$\Delta E_{n+1} = \Delta E_n + eV(\sin\varphi_n - \sin\varphi_s), \tag{1}$$

$$\varphi_{n+1} = \varphi_n + \frac{2\pi h\eta}{\beta^2 E}\Delta E_{n+1}, \tag{2}$$

where ΔE_i is the relative energy of the particle in turn i, φ_i is the phase of the particle in turn i, e is the elementary charge, V is the RF voltage, φ_s is the phase of the synchronous particle, h is the harmonic number, η is the phase slip factor, E is the synchronous energy and β is the synchronous relativistic factor given by $\beta = \frac{v}{c}$ for velocity v and speed of light c.

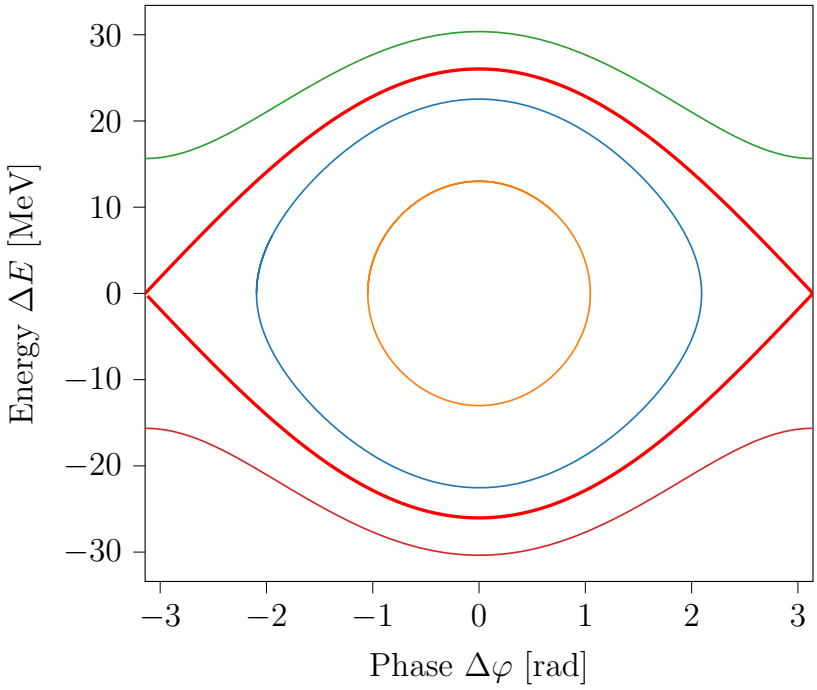


Figure 3: Example of a phase space containing bounded and unbounded particle trajectories.

By computing these equations for a sufficiently large number of turns, the trajectories of the particles in the phase space can be grasped. Figure 3 shows the phase space with the trajectories of four different particles, represented by the orange, blue, green and crimson lines. Depending on the accelerator and beam parameters, particles can move in a bounded area. This is considered as bounded motion and applies for the orange and blue trajectories. In the other case, the motion is considered unbounded, which in the figure applies to the green and crimson line. The bold red line in the figure is called the separatrix and separates bounded from unbounded motion. Its area and shape depends on various accelerator parameters, including if acceleration is happening

or not and if the motion is below or above transition. The area inside the separatrix is called the RF bucket and corresponds to the maximum acceptance for stable motion in the phase space [14]. Inside this RF bucket, particles have stable trajectories and orbit around the synchronous particle. When considering bounded motion, particles make a full turn around the bucket with different periods. The time in which a particle makes a full turn is called the synchrotron period and the reciprocal is called the synchrotron frequency.

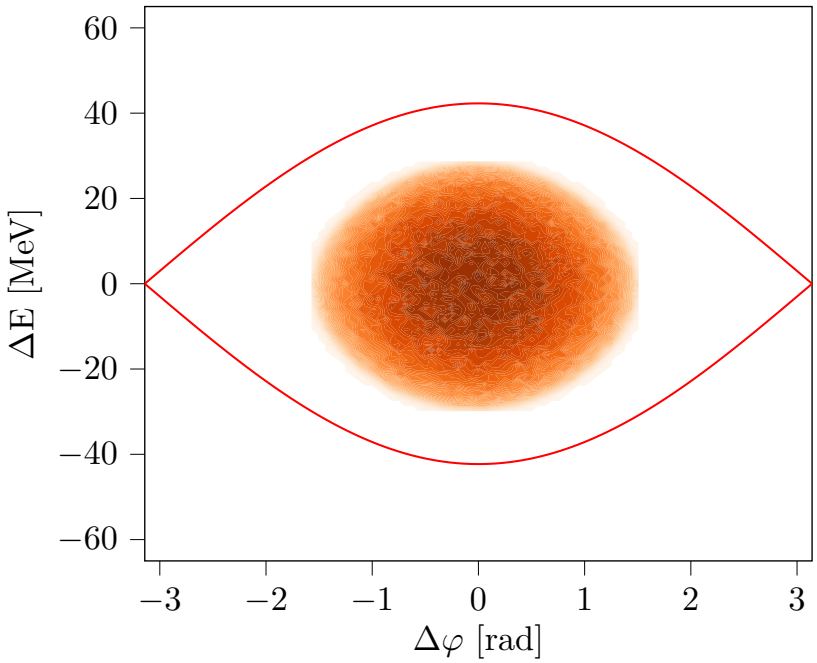


Figure 4: Example of a phase space with the bunch density displayed. The darker the color, the more particles are inside it.

A phase space distribution of a bunch can look like the one shown in Figure 4. It shows the particle density in the phase space with a simulated particle bunch consisting of one million particles. In this example the number of particles is quite limited and chosen only for demonstrating purposes, due to the high computing power needed to simulate real bunches. In fact, the number of particles in a real bunch, also called bunch intensity, is much higher and can reach values over 10^{13} particles per bunch. The simulated particles are then usually referred to as macroparticles since they represent multiple particles in a bunch. As in the previous plot, the x-axis shows the phase difference and the y-axis the energy difference with respect to the synchronous particle.

2.2 Longitudinal Phase Space Tomography

The following explanation about phase space tomography is strongly based on [6].

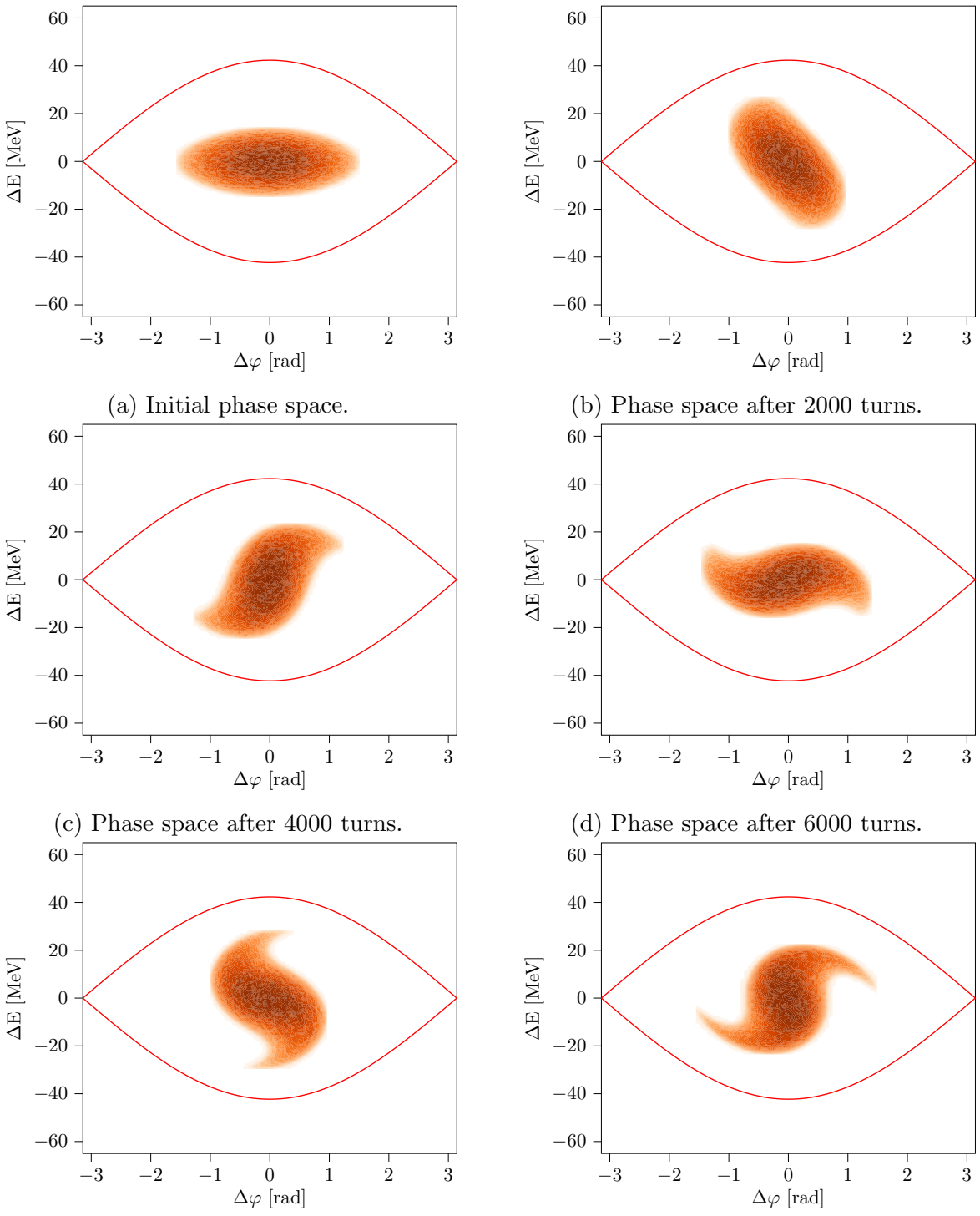
As introduced in the motivation, tomography refers to a methodology allowing the reconstruction of an n-dimensional object from multiple (n-1)-dimensional representations. In order to visualize synchrotron motion in the longitudinal plane, the two-dimensional phase space is reconstructed from multiple one-dimensional bunch profiles. Due to synchrotron motion, the beam distribution rotates in phase space. Therefore, a series of bunch profiles will view the bunch from different angles, and can be used to reconstruct the phase space distribution.

The tomography algorithm uses the measured bunch profiles to reconstruct the phase space distribution based on the Algebraic Reconstruction Technique (ART) [17]. Figure 5 shows the simulated phase space distributions for a particle bunch after a variable number of turns. The bunch moves changing its initial elliptic shape to an S-shaped form, due to a process called filamentation. Because of the initial bunch distribution, particles on the outer edges have a higher synchrotron period than particles in the middle, causing this bunch shape. This is an example showing that synchrotron motion is inherently non-linear, so the direct usage of ART, which is a linear technique, is not enough to reconstruct the phase space accurately. For that purpose, synchrotron motion is introduced into the process via particle tracking. Test particles are distributed uniformly in phase space and tracked for the number of turns corresponding to the time spent measuring the profiles. The particles get tracked for this specific number of turns and their phase and energy deviations with respect to the synchronous particle are extracted.

During the tracking, at every turn, the equations of longitudinal motion are applied to all the particles contained in the bunch. These equations are slightly modified from [16] because two RF harmonics are used instead of a single harmonic. Depending on the initial conditions, the tracking can be performed both in the forward and backward direction. In the forward direction, the change in phase (drift) and energy (kick) for a particle in turn i is given by

$$\begin{aligned} \Delta E_{i+1} &= \Delta E_i + V_1 \sin(\Delta\varphi_i + \varphi_0) + V_2 \sin[h_r(\Delta\varphi_i + \varphi_0 - \varphi_{1,2})] - \Delta E_{acc} \\ \Delta\varphi_{i+1} &= \Delta\varphi_i - \frac{2\pi h \eta_0}{\beta^2 E}\Delta E_i, \end{aligned} \tag{3}$$

where ΔE_i is the particle energy deviation from the energy of the synchronous particle on turn i, $\Delta\varphi_i$ is the particle phase deviation from the phase of the synchronous particle



(a) Initial phase space.

(b) Phase space after 2000 turns.

(c) Phase space after 4000 turns.

(d) Phase space after 6000 turns.

(e) Phase space after 8000 turns.

(f) Phase space after 10000 turns.

Figure 5: Simulated phase space distributions for different number of turns.

on turn i, V_1 is the voltage of the fundamental (first) harmonic, φ_0 is the phase offset of the fundamental harmonic, V_2 is the voltage of the higher harmonic, h_r is the ratio of the higher harmonic to the fundamental harmonic, $\varphi_{1,2}$ is the phase offset between the fundamental and higher harmonics, η_0 is the phase slip factor and β is the synchronous relativistic factor. For backwards tracking, the equations are modified to

$$\begin{aligned} \Delta E_{i+1} &= \Delta E_i - V_1 \sin(\Delta\varphi_i + \varphi_0) - V_2 \sin[h_r(\Delta\varphi_i + \varphi_0 - \varphi_{1,2})] - \Delta E_{acc} \\ \Delta\varphi_{i+1} &= \Delta\varphi_i + \frac{2\pi h \eta_0}{\beta^2 E} \Delta E_i. \end{aligned} \tag{4}$$

It can be observed that only the signs for the additions to ΔE_i and $\Delta\varphi_i$ are changed because ΔE_{acc} changes its sign for the backward tracking and thus it should still be subtracted [6].

In order to apply tomography, the tracked phase and energy coordinates are discretized. Figure 6 shows the time binning process where in the upper plot the line density of a bunch profile is shown, corresponding to the number of particles per nanosecond. The line density is then discretized, with a predefined number of bins, obtaining a histogram. In this way, after the tracking is complete, the number of particles present at each bin is known for each profile. For simplicity, only five bins are used in the figure. However, for a more precise histogram, more bins have to be considered. The bins are enumerated and the phase and energy coordinates are converted to these bin numbers. Another point worth mentioning is that these values are normalized in relation to the total number of particles, as seen in the lower plot. This example shows the binning procedure for the phase coordinates, the exact same principle can be used for the energy coordinates as well, creating a grid of phase and energy bins which will later be the dimensions for the reconstructed phase space output.

The tomography algorithm consists of three fundamental steps which are executed iteratively: back projection, projection and difference. While this explanation serves to illustrate how the tomography works for this use case, the current implementation of these steps as well as the auxiliary functions will be discussed in Section 4.1.2. Generally, the algorithm is based also on assigning weights to each particle which are used to compute the reconstructed time projections. Iteratively, these weights are changed every iteration to make improve the reconstructed projections and finally to use both the weights and reconstructed profiles to compute the phase space.

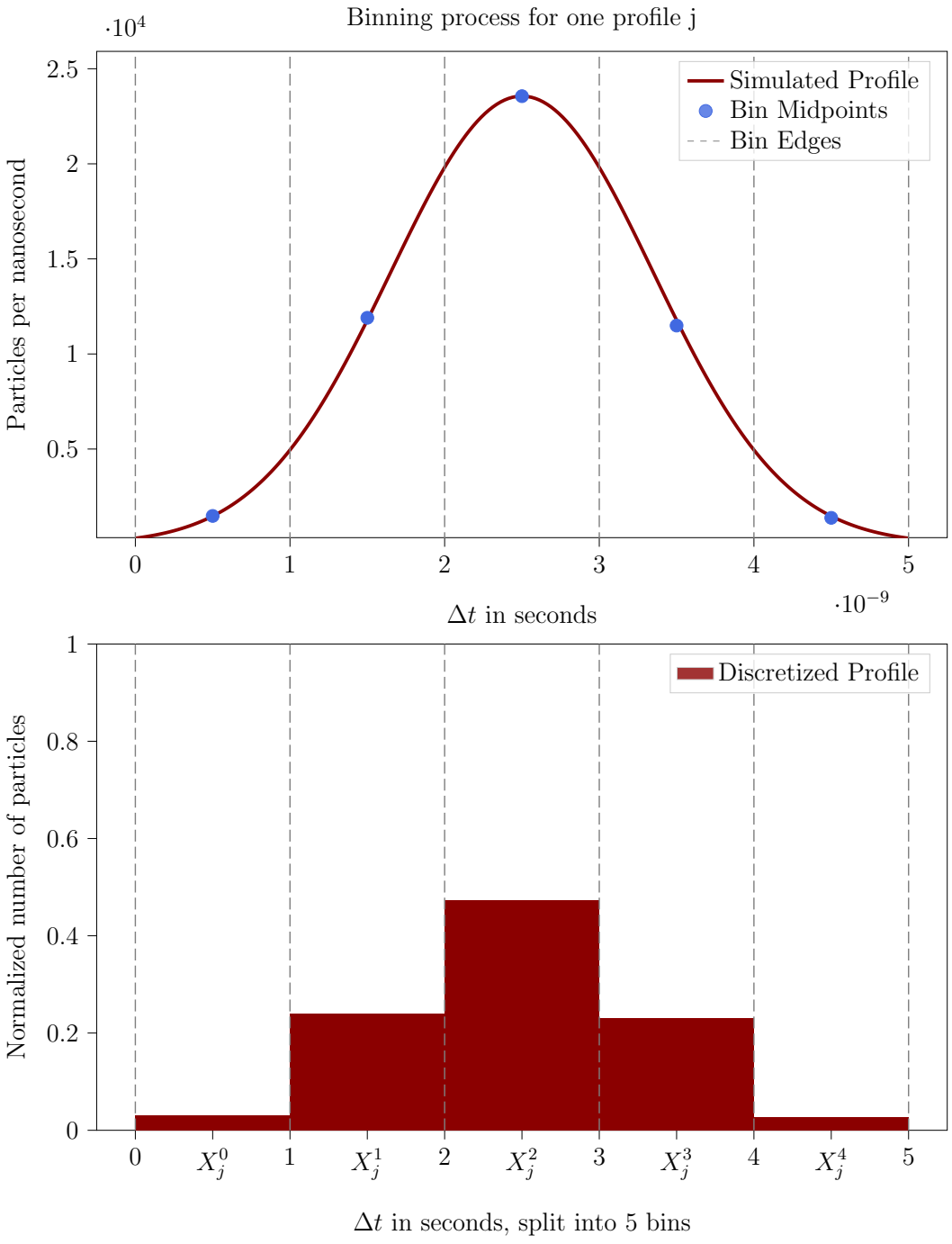


Figure 6: An example of the binning procedure for one profile using a bunch of 50000 particles [18].

2.2.1 Back Projection

The back projection step assigns and updates for every bunch profile the weight of each particle. This weight is set based on the line density of the profiles. In the first iteration, the weight W_i for particle i is initialized according to

$$W_i = \sum_{j=0}^{N} P_j(X_{i,j}) \tag{5}$$

where N is the number of measured profiles, P_j is the jth profile and $X_{i,j}$ is the phase bin index of the ith particle at the jth profile. This weight serves as the initial value to compute the first time projection.

Figure 7 shows how the particles' weights are initialized over five steps. For simplicity, the same profile is used for all the five steps, which is not necessarily true in real applications. The upper plot shows the histogram of the measured profile, with the phase bins and their values, while the lower plot displays the corresponding phase space, with phase and energy bins as the axes. The particles' weights are shown by the size of the symbol in the plot.

As the measured profile represents a histogram of values relative to the number of particles, it does not contain negative values. This condition also applies for weights because negative weights for particles would be unphysical and would distort the reconstruction, thus it is $W_i \geq 0$ for all particles. For the first iteration, the particles can only increase their weight but not decrease it. The weights for the first case are also reported in Table 1, where it can be observed that the initial weights in step one are equal to the amplitude of the phase bin from the measured profile. For every step, the weight is updated by adding the amplitude to it, corresponding to the phase bin for each particle.

Step	Red circle	Blue star	Green triangle
1	1	1	0
2	2	3	1
3	4	5	2
4	7	8	4
5	11	12	6

Table 1: Weights of the two particles over the five steps for the first example.

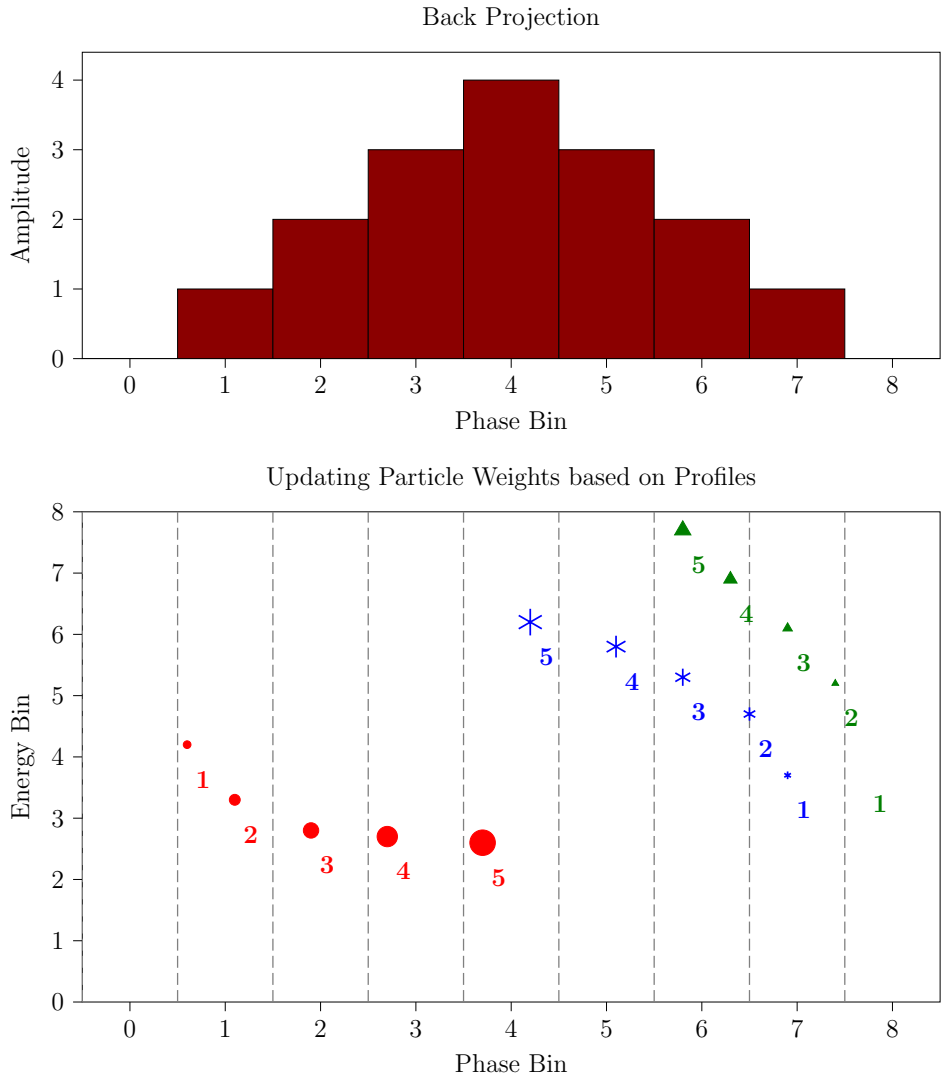


Figure 7: Weights of particles tracked for five steps [6].

To modify the weights for the following iterations, the equation changes to

$$W_i = W_i + \sum_{j=0}^{N} \Delta\widetilde{P}_j(X_{i,j}) \tag{6}$$

where the weight W_i is not initialized anymore but changed according to the results of the projection and difference. Instead of the measured profile P_j, the weighted difference between the measured and the reconstructed profile $\Delta\widetilde{P}_j$ is used, which can both increase and decrease the weights. However, the weights cannot be negative because the line density can also not be negative. If a particle would end up with a negative weight value, this is clipped to zero, so $W_i \geq 0$ is not only true for all particles but also for all iterations.

Figure 8 illustrates the second iteration of the back projection where the values used

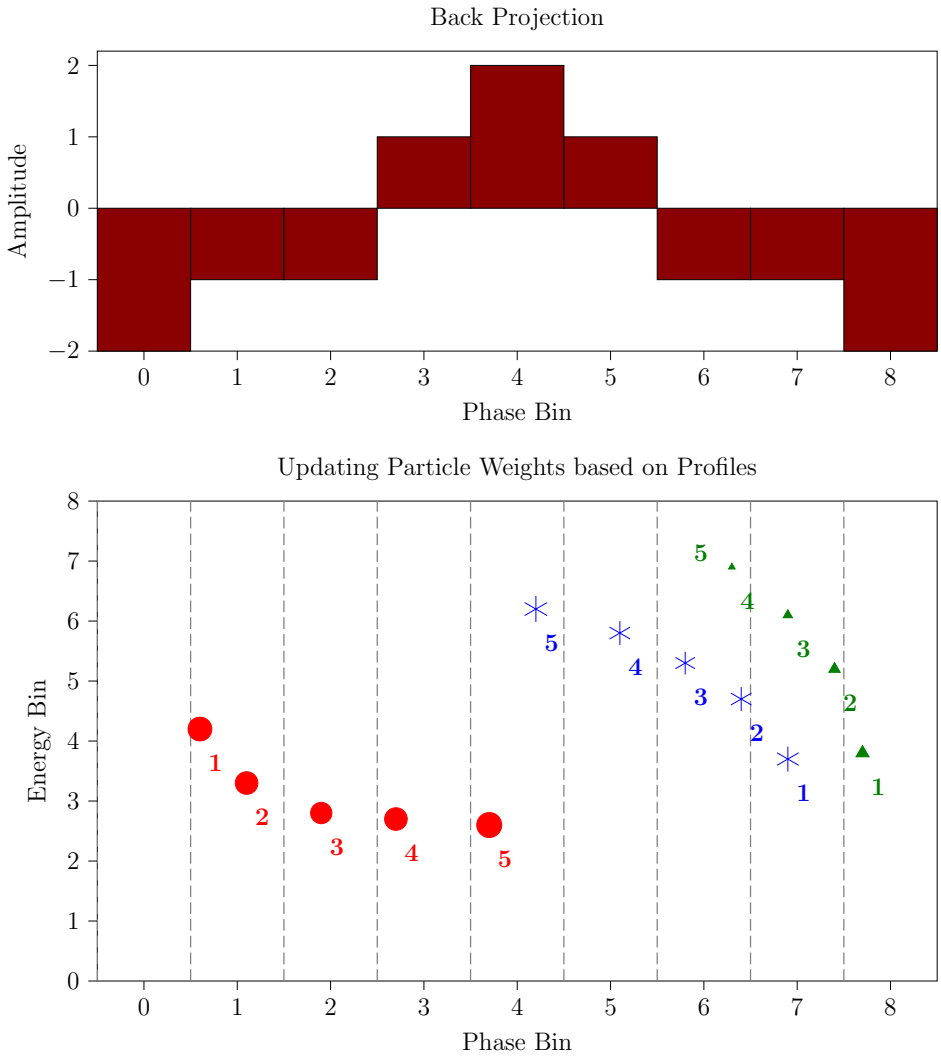


Figure 8: Second example of modifying weights for particles over five steps [6].

Step	Red circle	Blue star	Green triangle
1	10	11	4
2	9	10	3
3	8	9	2
4	9	10	1
5	11	12	0

Table 2: Weights of the two particles over the five steps for the second example.

to update the weights are not the measured profiles but the difference between the reconstructed profile and the measured profile. Besides this difference, the plot is structured in the same way as the previous one. The corresponding weights for this example are shown in Table 2. As an example, the green particle loses its weight completely and would get a negative value assigned if there were no clipping.

2.2.2 Projection

In the projection step, the calculated weights of the particles from the previous step are used to reconstruct a time projection of the phase space for each profile. For this purpose, the weights of the particles are summed in each bin. The measured profile is then compared with the reconstructed profile and the difference is used to update the weights in the next step. This difference profile was used in the example in Figure 8 to modify the weights for the back projection. The reconstructed time projection for each iteration is given by

$$\overline{P}_{j,n} = \sum_{i=0}^{M} w, w = \begin{cases} W_i, \text{ if } X_{i,j} = n \\ 0, \text{ otherwise} \end{cases} \tag{7}$$

where $\overline{P}_{j,n}$ is the nth bin of the jth reconstructed profile, M is the number of macro-particles which were tracked, W_i is the weight of particle i and $X_{i,j}$ is the bin number of particle i in profile j.

How the weights of the three particles from the previous example contribute to calculating the projection is shown in Figure 9. The upper plot shows the amplitude that is calculated by summing all the weights from one phase bin for each profile. Each color corresponds to a profile and since the projection is done for every profile and every bin, multiple bars in different colors are shown for the reconstructed projection in the corresponding phase bins.

After calculating the projection, the values need to be normalized in order to be compared with the measured profiles which are normalized as well. For that, the amplitudes of each profile and bin are divided by the sum of all amplitudes over the bins. The values are given by

$$\widehat{\overline{P}}_{j,n} = \frac{\overline{P}_{j,n}}{\sum_{b=0}^{B} \overline{P}_{j,b}} \tag{8}$$

where $\widehat{\overline{P}}_{j,n}$ is the normalized projection, B is the number of bins and $\overline{P}_{j,n}$ the nominal projection for profile j and bin n, as calculated with the projection equation.

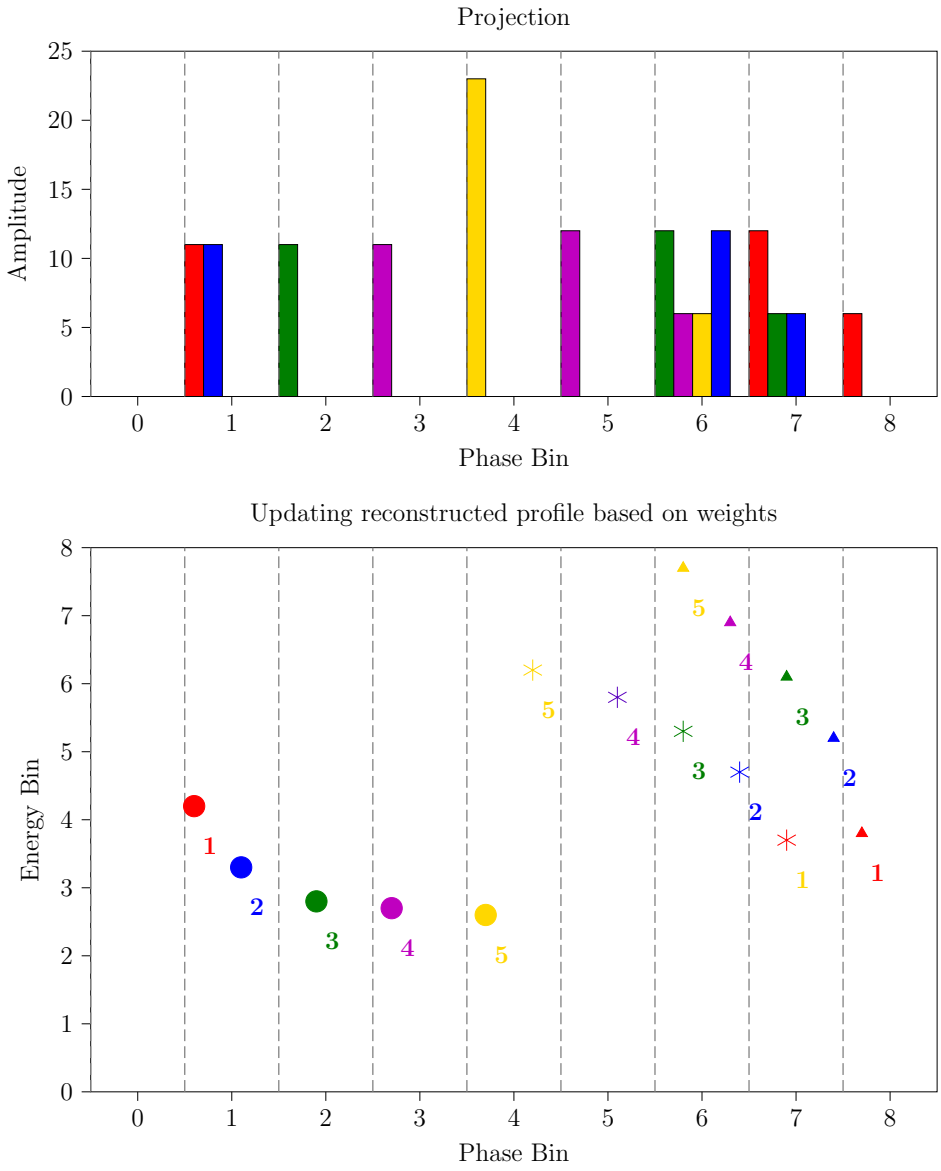


Figure 9: Contributions made by the three particles for the reconstructed phase space with the weights 11 (circle), 12 (star) and 6 (triangle) [6].

2.2.3 Difference

To compare the reconstructed profile with the measured profile, the difference profile $\Delta P_{j,n}$ for profile j and bin n is given by

$$\Delta P_{j,n} = P_{j,n} - \widehat{\overline{P}}_{j,n}. \tag{9}$$

The particles are not evenly distributed over the bucket, so changes in the projection of bins with a small number of particles implicate a higher error in weight. To tackle this, the weighted difference profile can be calculated by multiplying the difference profile with a bin-specific weight. This weight is the ratio of the maximum number of particles to the number of particles in the corresponding bin. The resulting weighted difference

profile $\Delta\widetilde{P}_{j,n}$ is given by

$$\Delta\widetilde{P}_{j,n} = \Delta P_{j,n} \cdot \frac{M_{\max}}{M_n} \tag{10}$$

where M_n is the number of macroparticles in bin n and M_{max} the maximum number of macroparticles over all bins. This value is used to calculate the next back projection to achieve a faster convergence of the algorithm. Therefore, this operation is not performed in the last iteration as the reconstruction output comes from the projection step.

2.2.4 Reconstruction Quality Metric

To estimate the quality of the reconstruction, the discrepancy of the difference profile is computed. It serves as a figure of merit and is calculated as the root mean squared error. The discrepancy D for an iteration is given by

$$D = \sqrt{\frac{\sum_{j=0}^{N}\sum_{n=0}^{B}\Delta P_{j,n}^2}{NB}} \tag{11}$$

where N is the number of profiles and B the number of bins. Care should be taken here as the unweighted difference profiles ΔP_j instead of the weighted difference profiles $\Delta\widetilde{P}_j$ are used. The latter are only used to amplify the change of the weights to achieve a faster convergence.

2.2.5 Creating the Reconstructed Phase Space

After a fixed number of iterations, a convergence should be achieved, which is reflected by the discrepancy values. The final step is to reconstruct the 2D phase space with the squared coordinate system, meaning that the number of bins used for phase in the x-axis and energy in the y-axis are the same. Thus, every bin in the $B \times B$ grid will contain the sum of the weights of the particles that are located inside it. The location of the particles is already determined by the tracking and, as a consequence, the weights of all the particles are added onto the corresponding bin in the grid. Finally, the phase space value $Z_{\varphi,E}$ for phase bin φ and energy bin E is given by

$$Z_{\varphi,E} = \sum_{i=0}^{M} W_i \cdot \delta_{i,\varphi,E} \tag{12}$$

for

$$\delta_{i,\varphi,E} = \begin{cases} 1, \text{ if } X_i = \varphi \text{ and } Y_i = E \\ 0, \text{ otherwise} \end{cases} \tag{13}$$

where X_i is the phase bin number, Y_i the energy bin number, W_i the weight of particle i and M the number of macroparticles.

3 Fundamentals on Runtime Acceleration

The focus of this chapter is to introduce the fundamentals of the computing-specific topics used in this work. In this context, Numba, a just-in-time compiler for accelerating Python functions will be explained. Furthermore, GPU development will be introduced as well as the corresponding application programming interface CUDA. Finally, the Python library CuPy will be explained to simplify GPU development in Python.

3.1 Numba

While Python excels in its simplicity and flexibility, making it a popular programming language in scientific applications, it is not designed for high-performance computing applications in particular. For this reason, several tools exist to speed up Python-based applications, especially with respect to the Python interpreter which can be a barrier when dealing with large data sets. Numba [19] and Codon [20] are examples of compilers that aim to improve the runtime of Python applications. The former is going to be introduced in this section.

Numba is a just-in-time (JIT) compiler based on Low Level Virtual Machine (LLVM) [21] focused on scientific and array-oriented computing. It is able to compile a subset of the language into efficient machine code that reaches a similar performance to traditional compiled languages like C [19]. Another notable point is the strong synergy between Numba and NumPy [22] which is one of the most important scientific libraries in Python for handling multidimensional data.

Integrating Numba to optimize functions in applications is straightforward. The function that should be compiled has to be annotated with the `@jit` annotation which tells Numba to compile the function. This happens when the function is called for the first time, however the compile procedure can be also called explicitly by calling the compile function and passing the function as a parameter with the compatible data types. The implicit version with the decorator is shown in Listing 1.

```
1 from numba import jit
2
3 @jit
4 def foo(a,b):
5     c = a + b
6     return c
```

Listing 1: A simple example of a function with the Numba JIT decorator.

When function `foo` is called with parameters, Numba compiles the function for the types it infers for the function. This function can for example be called with integers, floating-point numbers or NumPy arrays of different numeric data types. This implies that multiple bytecodes can exist for this function, depending on the data type of the parameters it is called with. Finally, for algorithms written in Python, Numba can be used to speed up the runtime. This has stronger effects for functions that are called very frequently or for functions that can be optimized by the compiler. The maintainability is not affected due to the small number of changes needed in the original code.

When compiling a function with Numba, there are two different modes which are considered, the `nopython` and the `object` mode. The `nopython` mode is the faster mode because the compiled machine code does not rely on the Python runtime for any operation. However, it requires that the native types of values can be inferred. Using more complex objects like instances of Python classes will force Numba to compile in `object` mode. Here, Numba will identify the loops and compile them into functions that run in machine code. However, the rest will run in the interpreter [19].

Additionally, the functionality running in machine code can run without the Global Interpreter Lock (GIL) and therefore on multiple threads [19]. This allows a further improvement of the runtime for these functions by using multithreading. By setting the parallel mode, multithreading can be enabled, for example in the decorator `@jit(parallel=True)`. Loops can be explicitly declared as parallel by using `prange` instead of the `range` function if there are no dependencies between the iterations.

```
1 from numba import jit
2
3 @jit(parallel=True)
4 def foo_2(X):
5     y = 0
6     for i in prange(len(X)):
7         y += X[i]
8     return y
```

Listing 2: Example of a function using parallel loops in Numba.

Listing 2 shows an example using explicit parallel loops. Although the code inside the loop looks like it can cause race conditions, this is not the case. Numba infers automatically if a variable is updated continuously and applies a reduction to merge the values from each thread together [19]. This simplifies programming with parallel threads considerably.

When benchmarking applications with Numba, the compile time should be considered as well. This can depend heavily on the functions that are compiled and often it is amortized by executing an optimized function various times. To demonstrate an example of this, the function in Listing 3 has been measured on three different optimization levels. One is using no optimization, the other two are compiled with Numba, one with the parallel option enabled, and the other without.

```
def f():
    x = np.arange(1e5)** 2
    for i in range(len(x)):
        x[i] -= np.sin(x[i]) + np.cos(x[i])
    return x
```

Listing 3: Example of a function performing arbitrary computation.

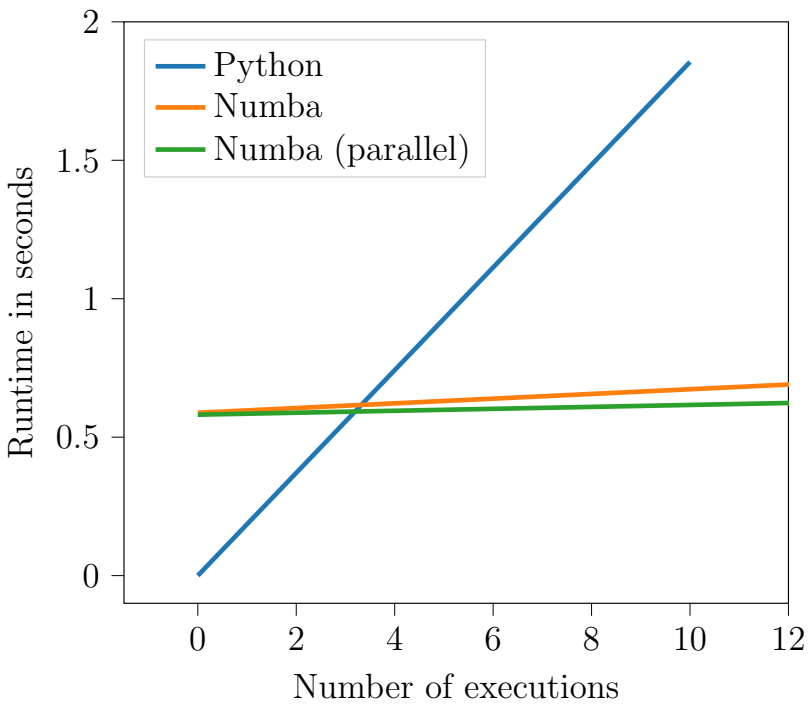


Figure 10: Benchmark taken for a Python function unoptimized and optimized with Numba.

In this measurement that has been conducted using an Intel® Core™ i7-6700 CPU with four cores and eight threads, the runtimes are shown in 10 after executing each function 10 times. The compilation time has been considered for the Numba versions as well, and is already amortized after four executions of the function. This shows that the speedup gained by optimizing with Numba can be very high, depending on the implementation of the function.

3.2 GPU Development with CUDA

Programming for the Graphics Processing Unit (GPU) emerged in the context of artificial intelligence and high-performance computing in order to perform computation on massive amounts of data. Originally developed for the graphics pipeline, it can also perform general purpose calculations.

Both CPUs and GPUs are designed for different purposes. The CPU is optimized to achieve low latency in the execution of instructions in a sequential manner, with a few tens of threads to enable parallel processing. For that purpose, a complex control unit is used as well as different layers of caching to minimize memory latency. Contrary to the CPU, the GPU is optimized for throughput by executing thousands of operations in parallel. It uses a less complex system of caches and a simpler control unit to allow a bigger set of cores to be established to execute the operations. Generally, the GPU has a smaller clock rate and the single-thread performance is worse than the CPU's, however due to the notion of executing instructions in a massively parallel manner, the slower single-thread performance is amortized by achieving a higher throughput [23]. The architectural difference is schematically shown in Figure 11.

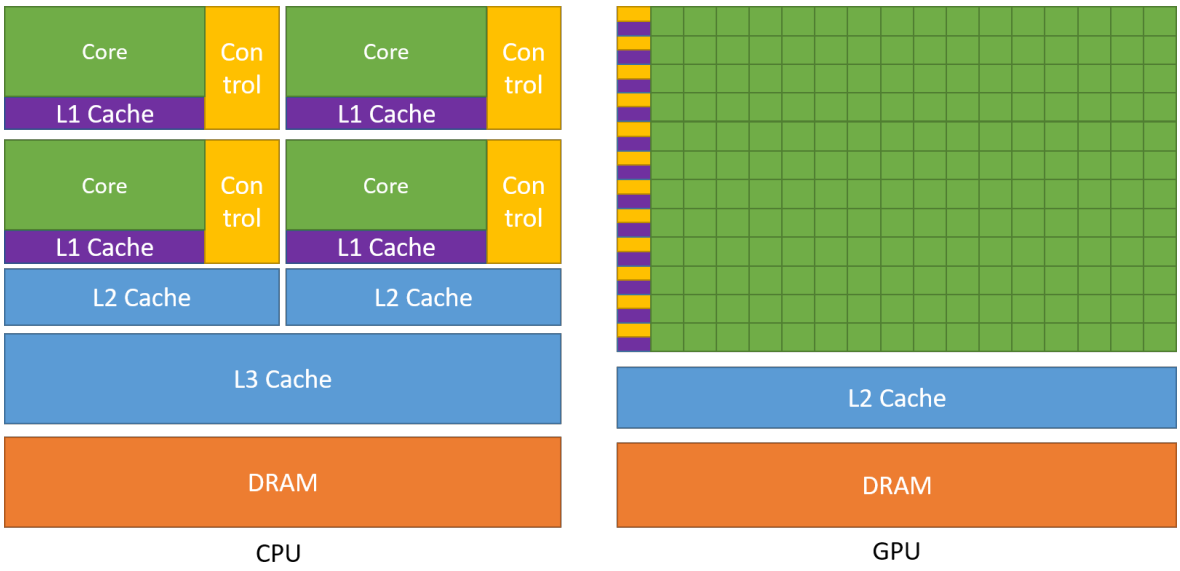


Figure 11: An example distribution of chip resources (cores, caches and memory) for a CPU and a GPU. The GPU dedicates more cores for data processing [23].

In the context of parallel computing, Single Instruction, Multiple Data (SIMD) processing is commonly used, referring to applying one instruction on multiple data points simultaneously [24], shown in Figure 12. CPUs often make use of this execution model through vectorization, i.e. using special registers to combine multiple numbers to an array in order perform one instruction on the whole array. Especially in the context of the GPU, the term Single Instruction, Multiple Threads (SIMT) is used, effectively

combining SIMD with multithreading. While the programming style is similar, threads are used instead of vectors, implying no need to read data into a vector register first. Additionally, synchronization is required on GPUs due to it having multiple processing elements [25].

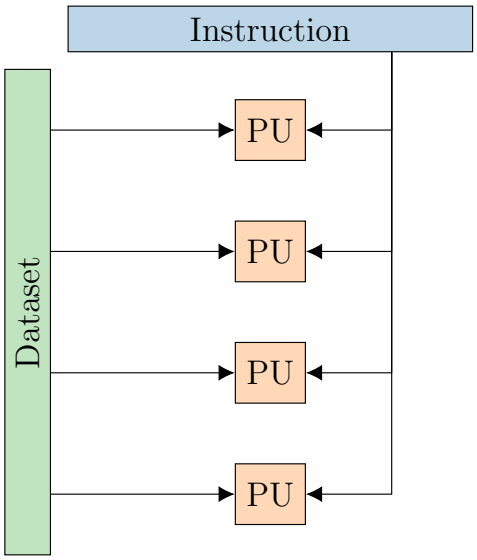


Figure 12: Illustration of SIMD instructions with PU as the processing unit.

Furthermore, an application consists of both sequential and parallelizable parts, therefore using a mix of CPU and GPU to harness the advantages of the respective units is essential to maximize performance. As of Amdahl's Law, there is a limit to the speedup one can achieve through parallelization [26], given by the equation

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}} \tag{14}$$

where p is the proportion of the program's runtime that is parallelizable and n is the number of threads by means of parallelizing instructions. As a consequence, for a program where p is 80%, the theoretical maximum speedup would be five for n tending to infinity. Reducing or optimizing the sequential part can be therefore vital to increase the impact of the improvements by parallelization.

Given the GPU resources available, this work consider only Compute Unified Device Architecture (CUDA), its software model and the architecture of the NVIDIA GPUs. CUDA is an application programming interface (API) created by NVIDIA to enable applications to use the computing power of NVIDIA GPUs for their calculations. In addition to the programming interface, CUDA provides GPU-accelerated libraries like CUB [27] for parallel algorithms. The following sections will briefly introduce some elements of GPU architecture, the CUDA software model and how it is mapped to the hardware and performance considerations regarding GPU development.

3.2.1 GPU Architecture

NVIDIA GPUs are built around an array of Streaming Multiprocessors (SMs) which can be partitioned into processing blocks. Figure 13 shows the architecture for one SM of the Tesla V100 GPU [28], which, among others, is used to benchmark the work. While other GPUs can have other architectures regarding partitioning and which computing elements they use, they are generally comprised of instruction caches and a data cache which is used as shared memory in the scope of the SM. They also include at least one dispatch unit, a warp scheduler, registers, load and store units and several execution units called cores.

Execution units are specialized on performing a certain type of operation. In the Figure, there are *INT*, *FP64*, *FP32* units and tensor cores. As the abbreviations imply, the *INT* cores perform arithmetical operation on integer numbers. The *FP64* and *FP32* are specialized on floating-point operations for 64 bit and 32 bit numbers respectively. It has to be noted that other GPUs can have other types of cores. For example, cores specialized on floating-point operations for 16 bit numbers. In addition, other GPUs may not contain FP64 cores and perform floating-point operations on 64 bit with multiple FP32 cores. Tensor cores are dedicated for training neural networks and offer mixed-precision calculations, for example when combining 16 bit and 32 bit arithmetic and matrix multiplications. These are not considered in the following work as these operations will not be used in this context.

The load and store units, in Figure 13 denoted as *LD/ST*, are used for loading data from the global memory into the registers and storing data into the global memory, if it is not obtained from the data cache. Generally, GPUs have multiple layers of memory with different scopes. The global memory resides in the device memory, also referred as Dynamic Random Access Memory (DRAM) and is accessible from every SM, consequentially having a lower bandwidth and a higher latency than other memory accesses, for example shared memory. In contrast to global memory, shared memory resides on the multiprocessor, thus memory accesses take less time.

Every SM is able to execute hundreds of threads concurrently. To account for the SIMT execution model and to manage such a high number of threads, the smallest execution unit in such a SM is a *warp*, consisting of 32 parallel threads. The instructions are pipelined in order to use the advantage of instruction-level parallelism and thread-level parallelism through hardware multithreading. For further reference, see [23]. A warp executes one instruction at a time, harnessing its full efficiency if every thread in the warp executes the same instruction. If a branching occurs, each branch is executed

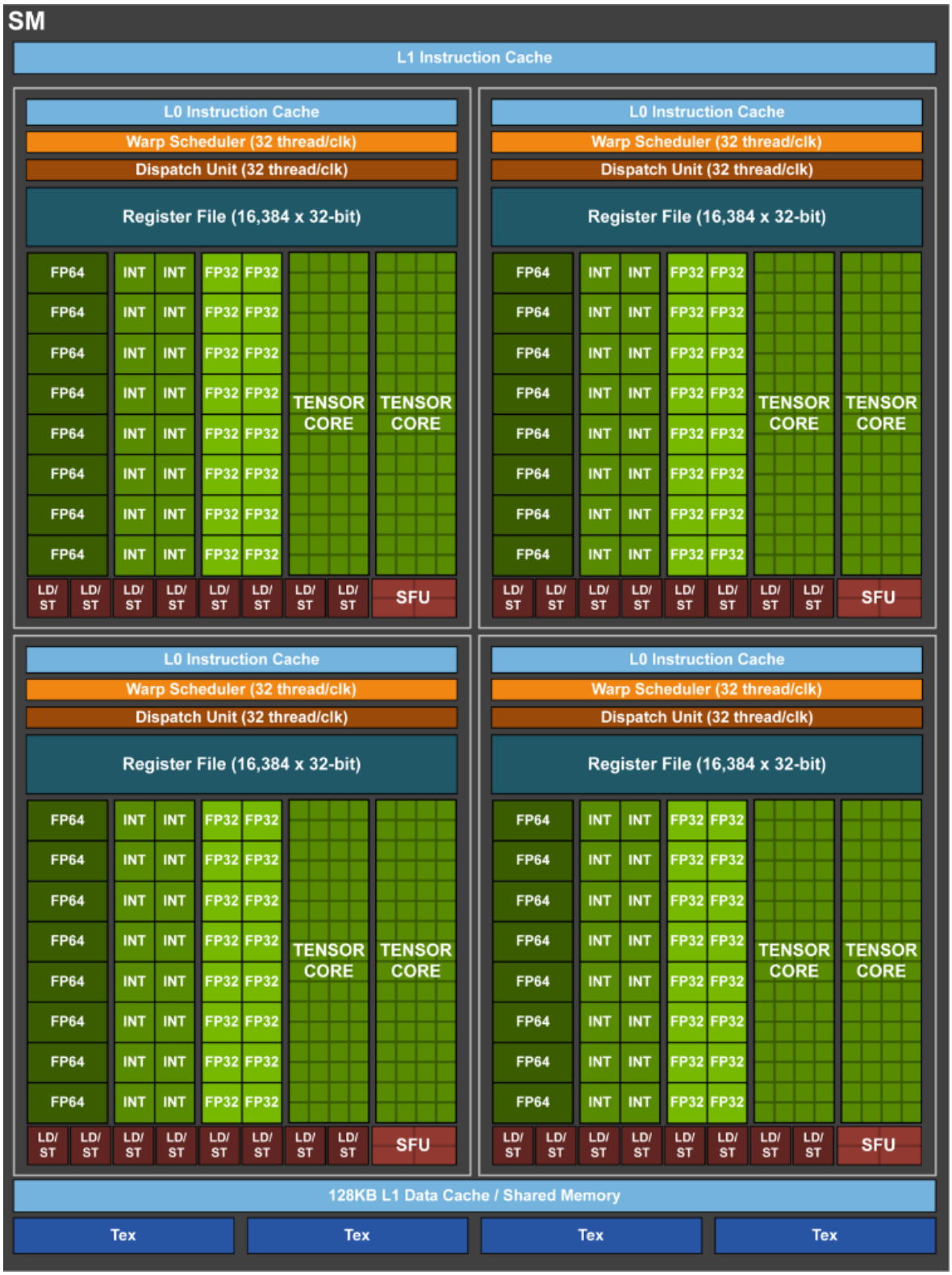


Figure 13: V100 Streaming Multiprocessor containing several different computing cores, load/store units and caches [28].

sequentially, meaning that the threads not in the respective path are disabled for the cycle. Since different warps can execute different instructions at the same time, there is no branching between warps.

3.2.2 Software Model

The CUDA toolkit comes with a range of components like a runtime environment with the CUDA runtime API, as well as a CUDA C/C++ compiler which is able to compile CUDA specific code in C++. CUDA extends C++ to allow the programmer to define functions to be executed in parallel on the GPU and it includes built-in variables to access the thread hierarchy and a smooth use of the CUDA libraries and the runtime API. In this context, to distinguish the execution of code, the terms *host* and *device* are used. Host refers to the main machine executing the code, i.e. the Central Processing Unit (CPU) while device refers to the GPU, acting as a coprocessor to the host.

The programming model used revolves around a thread hierarchy consisting of a *grid*, *thread blocks* and threads [23]. A grid contains an arbitrary number of thread blocks which consist of multiple threads. Grids and blocks can be defined to have one, two or three dimensions. The blocks and threads are enumerated when executing a function and can be accessed by the built-in variables `blockIdx` and `threadIdx` which have an x, y, and z property for the position in the respective dimension. The sizes of the grid and the blocks are saved in the `gridDim`, `blockDim` with the same dimensional properties mentioned before.

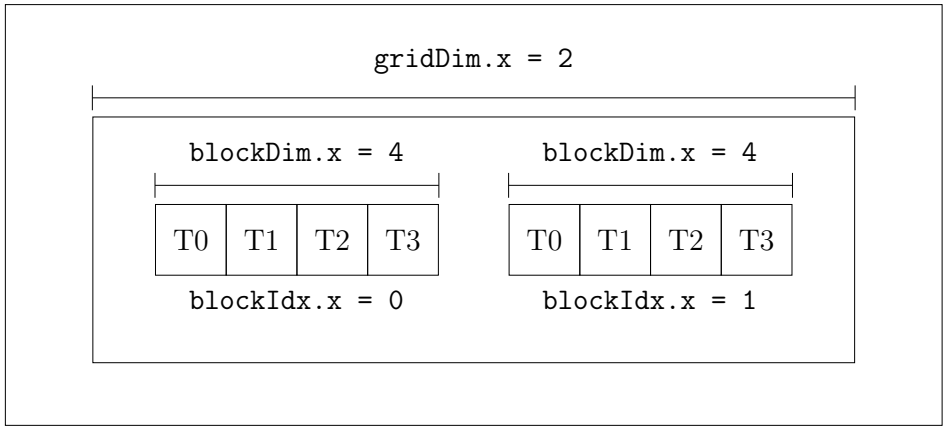


Figure 14: A diagram showing how the variables grid and block variables are assigned.

Figure 14 displays a diagram of a grid with two blocks, each one containing four threads. Throughout this work, only one dimension is used, therefore this example also uses one-dimensional grids and blocks. Since the grid has two blocks, the variable `gridDim.x` is two. The same applies for the blocks which contain four threads, therefore `blockDim.x` is four. The `threadIdx.x` for each thread corresponds to the number used in the diagram. Using this information, a global thread ID can be calculated with `blockIdx.x * blockDim.x + threadIdx.x`, giving numbers between 0 and 7 for the eight threads.

A thread block is a group of threads which are all executed inside one streaming multiprocessor. Each block has its own portion of the shared memory from the SM, effectively encapsulating the scope of shared memory to the context of a thread block. Each thread inside a thread block has its own private memory as well as registers.

A thread in the context of the software hierarchy corresponds to a hardware thread in an SM. As mentioned before, a warp consists of 32 threads, which are created based on the structure of the thread blocks. The first 32 threads of a thread block is treated as a warp, the next 32 threads as another warp, and so on. However, if a thread block consists of a number of threads which is not a multiple of 32, the last warp of a block will contain fewer threads. This cannot be compensated by the structure of the hardware, thus effectively leading to a lower occupancy of the cores due to the fact that the other threads are not activated for the warp. For this reason, the recommended block size is a multiple of 32. Ultimately, the number of warps $\mathcal{N}_{W,B}$ with warp size $\mathcal{S}_W$ generated by a thread block with size $\mathcal{S}_B$ is given by

$$\mathcal{N}_{W,B} = \left\lceil \frac{\mathcal{S}_B}{\mathcal{S}_W} \right\rceil = \left\lceil \frac{\mathcal{S}_B}{32} \right\rceil . \tag{15}$$

Lastly, the grid groups multiple thread blocks together to execute an operation. The number of threads N used to execute a function once per thread is therefore the product of the number of threads in a block and the number of blocks in a grid. While the size of the thread block is often fixed to the maximum limit of threads it can contain which in current GPUs is 1024 threads, the grid size orients on the size of the data that is being processed [23].

```
__global__ void vecAdd(double* A, double* B, double* C) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
```

Listing 4: A CUDA kernel to add two vectors for one-dimensional thread blocks.

One main element of CUDA programming are *kernels*, i.e. functions that are executed multiple times from different threads. A kernel is defined by using the `__global__` specifier in the function declaration. Listing 4 shows an example for such a kernel which takes two pointers `A` and `B` to input arrays, and a pointer for an output array `C`. The variable `idx` calculates a global index of the thread executing the function, depending on the index of the block and the thread inside the corresponding block. By ensuring that `idx` is different for every thread, every thread performs the addition once for the

corresponding index of the arrays in parallel to the other threads.

To synchronize threads inside a block, the intrinsics function `__syncthreads()` can be used. This is especially necessary when threads cooperate when using shared memory, so a coordination mechanism is required to obtain deterministic results. The function acts as a barrier in which every thread of the block waits until all threads arrived to that point, before resuming execution.

3.2.3 Performance Considerations and Metrics

When it comes to optimizing the performance of algorithms on the GPU, there are a few strategies in order to use the available resources in an efficient way. In general, it boils down to four basic strategies as in [23]:

- Maximize parallel execution to achieve maximum utilization of resources
- Optimize memory usage to achieve maximum memory throughput
- Optimize instruction usage to achieve maximum instruction throughput
- Minimize memory thrashing

However, it is very dependent on the problem and the structure of the application as well as the bottlenecks which limit the current performance.

Maximize Parallel Execution

The first strategy refers to making use of parallelism as much as possible in multiple layers. On a higher level, a better runtime can be achieved through execution of code on host and device simultaneously. While the host is optimized to perform serial workloads, the device should perform parallel workloads to achieve the best results. On a lower level, maximizing the number of warps currently executed in a streaming multiprocessor is an important performance factor. The ratio of active warps to the maximum executable warps in an SM is called *Occupancy* and is a metric that is often considered to evaluate how well the resources are used. As it depends on multiple factors if a warp can get scheduled, it is not trivially optimizable. However, the warp scheduler aims to find a balance to execute warps of different thread blocks to avoid having too many warps in the SM waiting for each other due to block-level synchronization, and to distribute the use of registers and shared memory evenly across different SMs.

Optimize Memory Usage

Next, optimizing memory usage is a concern too. To achieve a better performance, low-bandwidth memory transfers like data transfer from host to device and vice versa should be kept to a minimum. In many cases, it is advisable to keep data in the device until it is not going to be manipulated anymore, in which case it can be transferred back to the host. This can also lead to the decision to transfer more code to the device despite not exposing enough parallelism to reduce the latency from memory transfers. Depending on the situation, asynchronous memory operations can be used to mitigate the effect of waiting for memory transfers. Apart from host to device transfers, the memory access patterns for the global memory are relevant as well. When a warp executes a memory instruction, the memory accesses of the threads are coalesced into one or more memory transactions. The number of transactions depends on the size of the desired data and how distributed it is. Therefore, to minimize memory transactions, data that is accessed together should also be aligned in the memory, thus having close memory addresses.

Optimize Instruction Usage

In order to optimize instruction usage, there are multiple options, one of them being using instructions with a higher throughput, for example using lower precision floating-point operations if the effect is sufficiently small. Another point is to reduce divergence inside of warps since this leads to sequential execution of the branches. Lastly, one can also use restricted pointers with the `__restrict__` keyword, allowing the compiler to optimize more generously by assuring that the data of the pointers do not overlap. Incorporating this strategy for the `vecAdd` kernel, an optimized version can look like 5, under the constraint that the memory addresses of the parameters do not overlap.

```
__global__ void vecAdd(const float* __restrict__ A,
                       const float* __restrict__ B,
                       float* __restrict__ C) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    C[idx] = A[idx] + B[idx];
}
```

Listing 5: The vector add kernel optimized with restricted pointers and lower precision.

Minimize Memory Thrashing

The last strategy deals with memory allocations, which can impact the runtime. Allocating and releasing memory often tends to slow down the calls due to the release of

memory back to the operating system. Therefore, memory allocations should be sized accordingly to the data that will be used. A too generous allocation of memory can cause problems due to preventing other applications from using that memory, pressuring operating system schedulers [23]. In addition, allocations should be done early in the application when the application does not need to use the data right away. Besides that, limiting the number of allocations helps to improve the performance as well.

To conclude, there are several metrics that can be considered when optimizing the performance. The occupancy shows how well the available resources are used in a streaming multiprocessor. The throughput for both computing and memory is also very important to consider when analyzing potential bottlenecks. Finally, considering the memory access patterns, the hit rate of the caches are an important metric, although more intricate to optimize since the caches are not user-managed. Maintaining a high hit rate for the caches reduces memory transactions to higher layers which naturally have a lower throughput.

3.2.4 Profiling

There are a variety of profiling tools offered by NVIDIA to analyze and visualize the workloads in the GPU. This offers important feedback on where the bottlenecks are that can be optimized. In this work, two of the tools have been used: *NVIDIA Nsight Systems* [29] and *NVIDIA Nsight Compute* [30], both part of the NVIDIA Nsight Developer Tools suite.

NVIDIA Nsight Systems allows the performance of the whole algorithms or application to be visualized. One key feature of the GUI application of Nsight Systems is the timeline view which displays the chronological sequence of CPU and GPU activities as well as metrics during the execution of the application. These metrics include the number of instructions issued per SM, the SM warp occupancy, the DRAM bandwidth for memory accesses inside the GPU and the PCIe bandwidth for data transfers between host and device. It shows the execution time of the kernels, when the memory transfers happen and how much time they consume.

Additionally, Nsight Systems can be used by the command-line interface (CLI) where the application can be profiled using

```
nsys profile --stats=true -t cuda [application]
```

CUDA Metrics			
Time (%)	Total Time (ns)	Number of Calls	Name
73.2	344,960,319	4	`cudaMalloc`
23.8	112,134,082	1	`cudaDeviceReset`
2.6	12,357,716	4	`cudaMemcpy`
0.4	1,681,500	3	`cudaFree`
0.1	271,543	1	`cudaDeviceSynchronize`
0.0	69,003	1	`cudaLaunchKernel`
0.0	15,365	1	`cuCtxSynchronize`
0.0	5,587	1	`cuModuleGetLoadingMode`
Kernel executions			
Time (%)	Total Time (ns)	Instances	Name
100.0	250.846	1	`_Z13sum_array_gpuPiS_S_S_i`
Memory times			
Time (%)	Total Time (ns)	Count	Operation
72.8	8,370,357	3	`[CUDA memcpy HtoD]`
27.2	3,120,548	1	`[CUDA memcpy DtoH]`
Memory sizes			
Total (MB)		Count	Operation
50.332		3	`[CUDA memcpy HtoD]`
16.777		1	`[CUDA memcpy DtoH]`

Table 3: Example summary output of Nsight Systems CLI, showing CUDA metrics from the APIs.

This creates a report which can be used in the GUI as well as in the CLI version of Nsight Systems. Additionally, through setting the variable `stats` to `true`, a summary of the gathered metrics will be displayed after profiling. Setting `-t cuda` tells the profiler to trace CUDA-related metrics, coming from the CUDA runtime API and CUDA driver API. Table 3 shows an example output from executing the `nsys profile` command on an application that executes one kernel computing the sum of three arrays.

In the first section of Table 3, the CUDA API calls are shown as well as how often they are called and what the time they consumed. In this example, the memory allocation takes over 73% of the time API calls. Next, all the kernels executed are shown in the second section. This also includes the number of invocations and the runtime of the

Section: GPU Speed of Light Throughput		
Metric	Unit	Value
DRAM Frequency	cycle/ns	4.86
SM Frequency	cycle/µs	569.58
Elapsed Cycles	cycle	144,906
Memory [%]	%	91.22
DRAM Throughput	%	91.22
Duration	µs	254.40
L1/TEX Cache Throughput	%	27.95
L2 Cache Throughput	%	31.04
SM Active Cycles	cycle	140,681.08
Compute (SM) [%]	%	22.61
Section: Occupancy		
Metric	Unit	Value
Block Limit SM	block	16
Block Limit Registers	block	8
Block Limit Shared Mem	block	16
Block Limit Warps	block	2
Theoretical Active Warps per SM	warp	32
Theoretical Occupancy	%	100
Achieved Occupancy	%	85.15
Achieved Active Warps per SM	%	27.25

Table 4: Example metrics of a kernel profiling result with Nsight Compute CLI, showing statistics about the throughput and the occupancy.

block, as well as the grid and block structure they have been invoked with, which is omitted in this table. In this case, there is only one kernel which was executed once. The last two sections show the memory transfers from host to device (*HtoD*) and vice versa (*DtoH*), with the amount of data that was transferred and how much time the transfer took. One can see that three memory transfers took place from host to device, because the function took three input arrays as parameters, and computed the sum in one output array.

While Nsight Systems gives an overview over the whole application, it does not provide a detailed overview over the kernel execution itself and how well the resources are

utilized. To perform a profiling on the level of CUDA kernels, Nsight Compute can be used. As Nsight Systems, it is usable by a GUI as well as a CLI version. To execute a profiling, one can run

```
ncu -k [kernel-name] [application]
```

with the function name of the kernel to profile.

An example output for the same kernel as before is shown in Table 4. The first section shows the *GPU Speed of Light Throughput*, where speed of light refers to the maximum achievable throughput. Therefore, the throughput metrics are given in percent. Depending on the values of the metrics, some additional information or hints can be displayed from the profiler. In this case, the kernel utilizes a very high amount of the memory performance, implying that the kernel is well utilized and there are not many threads waiting. This is also shown in the *Occupancy* section where the achieved occupancy is shown which is around 85%. Achieving a high occupancy and a high number of active warps per SM close to the theoretical level are important criteria to consider when optimizing the performance of GPU workloads.

3.2.5 CuPy

Even though most developments for CUDA are done in C++ which has the advantage of optimizing code during the compile processes, many scientific applications are written in Python. CuPy is a Python library which enables the programmer to perform computations on GPUs, especially NVIDIA GPUs for which it uses CUDA and CUDA-related libraries to accelerate computations written in Python [31]. CuPy is highly compatible with NumPy and uses the same syntax, enabling the library to be used as a drop-in replacement for many cases.

The core of CuPy is the `cupy.ndarray` class which represents the multidimensional array analogously to `numpy.ndarray`. When declaring such an array, the memory is allocated on the current GPU device. To transfer memory from host to device and vice versa, CuPy offers the functions `cupy.asnumpy()` and `cupy.asarray()` which can convert `cupy.ndarray` to `numpy.ndarray` and respectively move the data from device to host and vice versa. Listing 6 shows the declaration of a CuPy array `a` on the GPU and performs calculations on it like reshaping the array to a matrix, transposing it and multiplying with a scalar. After that, the CuPy array is converted to a NumPy array `b`.

```
import cupy as cp
a = cp.arange(10**7).reshape(1000, -1).T * 2
b = cp.asnumpy(a)
```

Listing 6: CuPy array used in the same way as NumPy arrays.

When executing these functions, CUDA kernels are executed internally to perform the computation of the functions. By providing this interface, the programmer does not have to define the grid and block structure independently since this is inferred automatically by CuPy depending on the data used. Therefore, implementing computational-heavy functions in CuPy helps to improve the runtime by using the advantages of GPU-based processing while not having to manage the thread blocks and the threads inside of it.

Besides using CuPy as a replacement for NumPy, and therefore performing all calculations in Python, CuPy allows various types of kernels to be defined, one of which are raw kernels [32]. Raw kernels allow CUDA C++ code to be executed from Python, for example by providing compiled binary files. These can be loaded by CuPy and the kernels can be retrieved inside the application and called with the respective grid and block structure. Other types of kernels include the definition of element-wise or reduction kernels based on the function it performs which can be used to define kernels in a compact way without having to define the grid. The last way to define a kernel is experimental and allows to create raw CUDA kernels just-in-time with Python code. In that context, CuPy exposes the internal variables `threadIdx` and `blockIdx` to index the thread hierarchy. However, the scope is limited and only elementary scalar operations are supported, while functions for CuPy arrays are not supported.

Furthermore, through interfacing CUDA features, CuPy allows the runtime for the GPU parts to be measured by using CUDA Events, in addition to the host runtime, thus making benchmarking possible when using CUDA together with CuPy as well as using CuPy standalone.

To conclude, CuPy enables Python applications to perform calculations on the GPU for example by substituting NumPy arrays with CuPy arrays and performing the equivalent operations. Additionally, it also allows to the programmer to add his own kernels through CUDA source code or binaries which allows the fine-grained managing of the grid and the thread blocks. Both approaches have their advantages and disadvantages regarding the performance and maintainability of the code. Modifying a present Python code by using CuPy arrays is more easily maintainable since there is no other programming language involved and the syntax is the same. However, with more difficult algorithms

or access patterns for the kernels, the functions are not as optimized as they are when writing CUDA code.

4 Analysis of the Current Implementation

This chapter focuses on the current state of the application and explains the different parts of the workflow. Moreover, an explanation about the time complexity of the functions is given, and a discussion which parameters influence the runtime. Finally, a performance benchmark will be conducted to analyze the bottlenecks and where the functions can be optimized.

In terms of performance, the interplay of Python and C++ plays a substantial role because the computationally heavy parts have been migrated to C++ in the past to achieve a better performance. To enable a smooth communication between the Python workflow and the C++ functions, pybind11 [33] is used. It allows to call C++ functions by wrapping them as a Python module and make them callable from the Python application after importing the module.

4.1 Workflow

Figure 15 shows the workflow of the longitudinal phase space tomography application. The blue rectangles with rounded corners refer to functionality implemented in Python while the green rectangles with sharp corners show the parts which are written in C++. The orange circles with outgoing arrows represent inputs that are given before executing the program and the red circles with incoming arrows refer to the output, which, for the whole application, is the two dimensional phase space distribution.

Block 1 of the flowchart represents the work that is needed to get from the machine parameters to the tracked particles. For this purpose, the machine parameters are needed for the longitudinal tracking step, as detailed in Section 2.1. These parameters can be read from a file or passed from memory. The creation of this machine object and setting the parameters is done in the `Input to machine` procedure.

The next procedure in this tree is `Values at turns`, which calculates the values for each turn based on the initial machine parameters. Thus, arrays are created for the energy, energy gain, phase, magnetic field, time, phase slippage factor, the synchronous relativistic velocity, the revolution frequency and the voltages of the two harmonics. The size of the arrays, and therefore the runtime of the value calculations, depend on the number of turns that have to be calculated, given by the number of bunch profiles that have been taken and the interval in turns between each bunch profile. After calculating the values, a homogeneous distribution of macroparticles is created and the phase and energy coordinates are calculated for each turn. More details are given in Section 4.1.1.

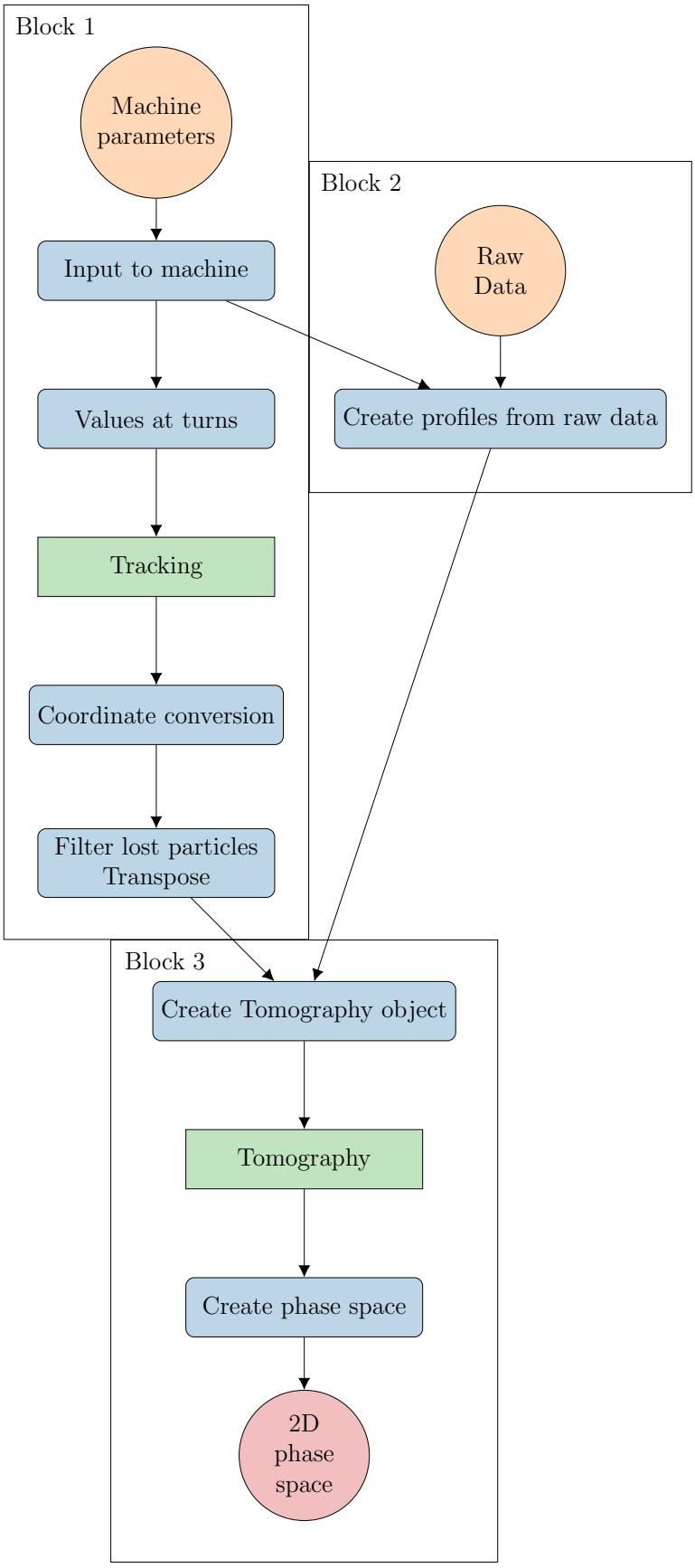


Figure 15: Flowchart of the whole application, divided in three functional blocks.

The tracking calculates the coordinates in phase and energy units (rad, eV). To make the coordinates usable for the phase space reconstruction, they are converted into fractional bin numbers.

In the last step before the coordinates can be used for the tomography, the lost particles have to be filtered. Particles outside the image width are identified and removed from the particle arrays. Additionally, the coordinate arrays are transposed for more convenient usage later and the fractional bin numbers are cast to integers by truncating the decimal part of the number. Consequently, negative integers are rounded up while positive integers are rounded down, in both cases towards zero.

In block 2 of the workflow, the process of creating the profiles from the raw data is shown. The data is put into a Profiles object, which holds the information about the measured data as well as the data itself. If a rebinning factor is given, the data is rebinned along each profile.

The final step is in block 3 where the tracked particles are combined with the profile to create the phase space distribution. Before tomography can be performed, a Tomography object containing the necessary parameters and arrays to perform the reconstruction is created. This step includes several assertions to verify that the coordinate arrays have the right shape, that there are no particles outside the image width, and that the particles have been tracked for the correct number of profiles. Additionally, the particle arrays are stored contiguously in memory.

Having prepared the coordinates and profiles, the tomography algorithm can be performed. Further details are given in Section 4.1.2. After performing the tomography, the phase space can be calculated from the macroparticle weights using the calculation in Section 2.2.5, which concludes the procedure.

4.1.1 Tracking

Figure 16 provides a more detailed view into the workflow for the tracking. As in the previous workflow, the blue rectangles with rounded corners denote procedures executed in Python and green rectangles with sharp corners are parts written in C++. The tracking workflow starts with the generation of a homogeneous particle distribution. For this, the reconstruction area is calculated using the machine parameters and filled with a uniform distribution.

The main part of the tracking routine is the `Kick and Drift` function, written in C++

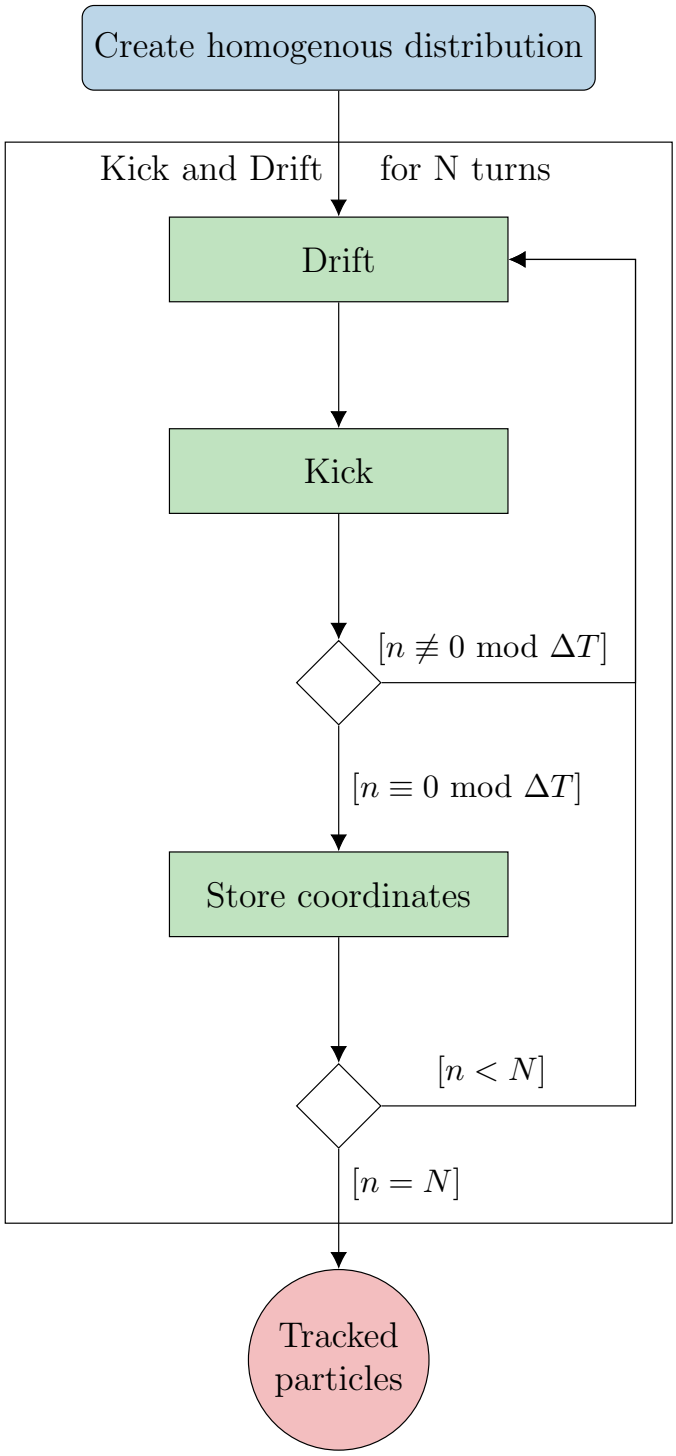


Figure 16: Flowchart of the tracking procedure.

and accessible via pybind11 [33] which is an interface to execute C++ code from Python. The function is called from Python through a wrapper that prepares all the input parameter arrays.

The particle coordinates are stored in two two-dimensional arrays. One array for phase and one for energy coordinates with shape (N, M). N is the number of profiles and M

is the number of macroparticles. Additionally, there are two one-dimensional arrays containing the coordinates for the current turn because not every turn is stored. The main function iterates through all the turns and calls the separate helper functions `kick_up`, `drift_up` for the forward direction and `kick_down`, `kick_down` for the backward direction to apply the longitudinal equations of motion to the arrays as shown in Equations 3 and 4. Since the measured profiles are not necessarily taken consecutively but in a fixed interval of turns d, only every d-th turn has to be stored.

Finally, the results stored in the output arrays are transferred to the Python application where they can be used for further computation.

4.1.2 Reconstruction

The tomography reconstruction is the heart of the application and consists of a more complex set of parameters and operations, all written in C++. As with tracking, the main function is called through pybind11 using a wrapper that creates arrays for the weights, the discrepancies and for the calculated time projections, which are the outputs of the reconstruct function. The function takes as inputs the binned phases from the tracking and the bunch profiles. The profiles are flattened to a one-dimensional array with $N \cdot B$ elements, N being the number of profiles and B the number of bins.

Figure 17 shows a flowchart of the tomography in more detail. The first procedure `Calculate reciprocal particles` is to calculate the weight factor which is used to calculate the weighted difference profile $\Delta\widetilde{P}_j$ from the unweighted difference profile ΔP_j, as shown in Equation 10. First, the particles inside the bins are all counted to identify the maximum number of macroparticles over all bins M_{max} to calculate the weighting factor $\frac{M_{\max}}{M_n}$ for each bin according to 2.2.3 and stored in an array.

The next step is to create a (flattened) array of the point coordinates `flat_points` for each profile, done in `Create flat points`. It has $M \cdot N$ elements and combines the information of particle coordinates and profiles to create indices pointing at flattened versions of the profiles. It is used to index a particle in the reconstructed profile where the amplitude is modified in the projection step. The array is structured in a way that it can be iterated easily. This implies that the indices for particle in different profiles are next to each other in this array. Since the unflattened shape of the bunch profiles is (N, B) and the one for the `flat_points` is (M, N), the initialization of the array is done as in Listing 7. In this example, `xp` is the array of phase bins for the coordinates gained from tracking, `npart` is the number of particles, `nprof` the number of profiles and `nbins` the number of bins.

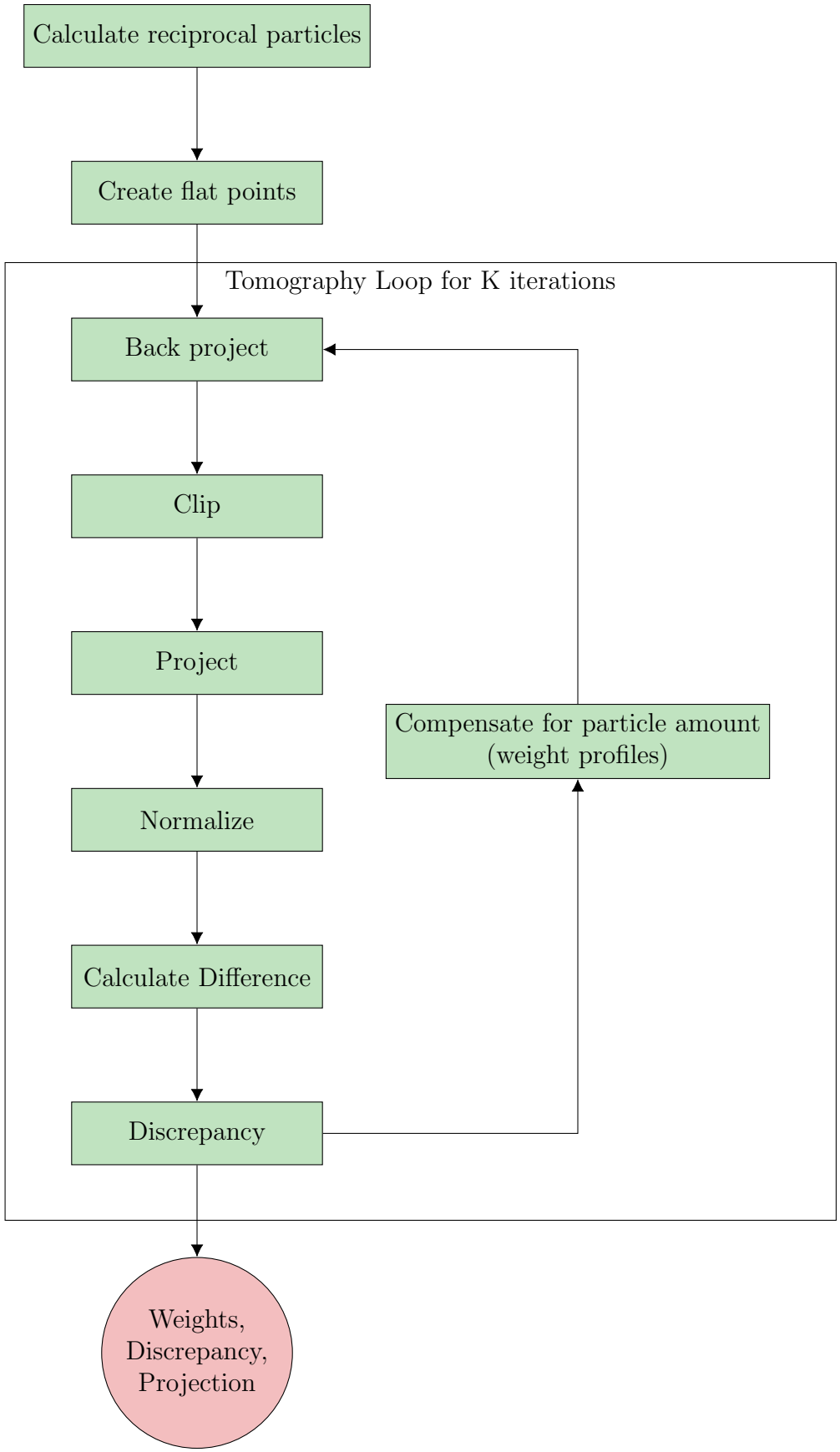


Figure 17: Flowchart of the tomography reconstruction.

```
std::memcpy(flat_points, xp, npart * nprof * sizeof(int));
for (int i = 0; i < npart; i++)
    for (int j = 0; j < nprof; j++)
        flat_points[i * nprof + j] += nbins * j;
```

Listing 7: Creation of the `flat_points` array.

In the middle of Figure 17, a rectangle encloses the functions that are executed for multiple times. Since the reconstruction algorithm is iterative, the procedures inside it are executed for a fixed number K of iterations to continuously improve the weights and the result of the reconstructed time projection. The first step in the tomography loop is the back projection, done in the `Back project` procedure. As explained in the fundamentals chapter, the weights are adjusted here based on the bunch profiles or the difference profile.

```
for (int i = 0; i < npart; i++)
    for (int j = 0; j < nprof; j++)
        weights[i] += flat_profiles[flat_points[i * nprof + j]];
```

Listing 8: C++ implementation of the `Back project` function.

In the implementation in Listing 8, the aforementioned `flat_points` array is used to index the value of the phase bin from the difference profile. This happens in a loop iterating over the particles and profiles. The correctness of the function is ensured for the first execution as the weights are pre-initialized to zero. Therefore, when being with the bunch profiles, the weights will reflect the sum of the values of the respective phase bins. Since this implementation does not take the fact into account that negative values should not be assigned to the weights, the helper function `clip` then iterates through the array and sets all negative values to zero.

```
for (int i = 0; i < npart; i++)
    for (int j = 0; j < nprof; j++)
        flat_rec[flat_points[i * nprof + j]] += weights[i];
```

Listing 9: C++ implementation of the `Project` function.

The next step is to modify the time projection based on the weights of the particles, done in the `Project` function. This function's implementation is shown in Listing 9 where the `flat_rec` array contains the time projection, therefore with the same dimensional structure as the bunch profiles (`flat_profiles`). The method's implementation

resembles the one from the back projection, but the weights are now used to modify the time projection.

Next, the reconstructed profile is normalized by dividing each bin of each profile by the sum of all amplitudes over the bins for the corresponding profile.This yields the normalized time projection, which is used in the `Calculate Difference` step to calculate the difference profile. In this function the normalized reconstructed profile is subtracted from the normalized bunch profiles, yielding the unweighted difference profile.

In the following step, the discrepancy is calculated using the unweighted difference profile. It serves as the figure of merit and can therefore give insights about the convergence of the algorithm. The root mean squared error is calculated in this function by summing the squares of all the values of the difference profile, divide them by the number of elements, which is equal to the product of the number of bins and the number of profiles, and computing the square root.

To use the difference profile for the back projection in the next iteration, it is modified to account for different particle densities in the bins. Therefore, if the loop is not in the last iteration, the weighted difference profile is calculated in the `Compensate for particle amount` step. Here, the array with the weight factor `rparts` from `Calculate reciprocal particles` is multiplied by the unweighted difference array. This weighted difference array is then used for the next back projection. Finally, in the last iteration, this function is not called and the tomography algorithm concludes after the discrepancy calculation. The outputs are the weights of the particles, the discrepancy calculated for each iteration and the last calculated time projection. The first and third output are used to reconstruct the phase space, while the second shows how well the phase space has been reconstructed.

4.2 Performance

To give a performance overview for the functions, several parameters have to be taken into account. Some functions scale differently to others depending on the initial parameters. The fundamental parameters are:

- ΔT: the number of turns between two profiles
- N: the number of profiles
- B: the number of bins

- $\hat{M}$: the square root of the initial number of particles per cell
- K: the number of iterations for the tomography reconstruction

From these parameters, other derived parameters can be calculated as follows:

- $T = (N-1) \cdot \Delta T$: the total number of turns
- $M \leq \hat{M} \cdot B^2$: the number of macroparticles

While the number of macroparticles scales with the squared number of bins, it is usually smaller than $\hat{M} \cdot B^2$ due to the fact that the bucket is not rectangular. Therefore, particles are filtered out of this stage.

Function	Relevant parameters
`Values at turns`	T
`Create profiles from raw data`	B, N
`Homogeneous Distribution`	B, M
`Kick and Drift`	T, M
`Coordinate conversion`	M, N
`Filter lost particles & Transpose`	M, N
`Create Tomo object`	M, N
`Tomography`	M, N, B, K
`Create phase space`	M, B

Table 5: The procedures of the application and their relevant parameters.

Table 5 shows the high-level procedures of the application workflow and the relevant parameters that can influence their runtime. However, it has to be noted that the effect of the parameters can vary strongly as it depends on the operations being executed in a function. `Kick and Drift` is a computationally heavy function where the number of arithmetic operations scales linearly with the product of turns and particles. Other functions, like the `Homogeneous Distribution` function, do not loop over these values but create bigger arrays and are better optimized through the way NumPy performs array operations [22].

In the C++ code where the computationally heavy parts reside, a few optimizations are incorporated. The first one is using parallelization with OpenMP, an API that supports parallel programming in C, C++ and Fortran [34]. All the loops used in the `Kick and Drift` function are optimized using OpenMP directives, as well as most of the functions

in the Tomography. One exception is the projection which cannot be parallelized this way due to race conditions happening through having multiple particles in the same bin on the same profile. This leads to adding both weights to the one memory address concurrently, and therefore to potential data inconsistencies.

A second optimization concerns the `kick` function, where the sine has to be calculated twice per execution. The mathematical library VDT (**v**ectorize**d** ma**t**h) is used which contains a collection of highly optimized mathematical functions for single and double precision floating-point numbers with an adequate accuracy, written in C [35].

To analyze the current runtime, a benchmark is conducted using a script that creates a bunch distribution using BLonD [2] using all the necessary machine and beam parameters. After that, a simulated bunch distribution is used which can be used as input along with the machine parameters to start the workflow of the tomography application. This starts with the creation of the machine object and concludes with to the calculation of the phase space after the reconstruction. Since a rebinning is not performed here, the creation of the profiles with the profiles object is omitted because it serves as a wrapper for the bunch profiles. As the Tomography object does not require a Profile object but just the bunch profiles array, the array can be used directly from BLonD instead.

The runtime measurements are taken with the time package in Python. This allows the runtime for all functions that are called inside Python to be measured. However, measuring the runtime of the functions called inside the C++ code is not possible with this method and is generally complex when using C++ code called from Python. One method could be to wrap all the C++ functions and transfer the workflow to Python, calling every C++ function separately. However, the total runtime of the application will be distorted and the runtimes of the subfunctions can only be used as an indicator of which functions take longer than others.

Another way to tackle this is to use a statistical profiling method instead of measuring the elapsed time with a timer. In a sampling-based approach, instead of calculating the difference between two timestamps, samples are taken at regular intervals to derive how much time the program spent in which part and several performance metrics. Samples can be taken for example of hardware performance counters like cache misses or branch mispredictions to find potential bottlenecks. With a high sampling rate, this can give insights about the runtimes of every part of the code and therefore conclusions can be drawn about how the runtime of the tracking and the reconstruction break down to its subfunctions. Here, Intel® VTune™ [36] is used to collect samples and to report the

runtimes spent on each element of the application.

The script is executed within the HPC cluster hosted at CERN. The node used is equipped with an AMD EPYC 7302 CPU [37], which has 16 cores and 32 logical threads within a CentOS7 operating system. For the baseline runtime measurements, a generated distribution of 500 profiles is used with a ΔT of nine, therefore the number of turns is $(500 - 1) \cdot 9 = 4491$. The number of bins is set to 200 without rebinning and the number of threads used by OpenMP is set to 8 by setting the environment variable `OMP_NUM_THREADS`. A higher value, like 16 or 32, would be possible but, due to the distribution of resources and the scarce availability of resources with high cores in the cluster, the CPU examples will use 8 cores except when comparing different numbers of threads. In order to get more accurate and reliable results, the application is executed multiple times and the means and standard deviations are taken into account. Additionally, only the forward direction tracking has been considered. Since both tracking functions are equivalent in complexity, with the only difference being an addition instead of a subtraction, and vice versa, one can expect the same runtime for both.

Function	Runtime in seconds
`Values at turns`	$0.079 \pm 5 \cdot 10^{-3}$
`Homogeneous Distribution`	$0.025 \pm 3 \cdot 10^{-4}$
`Kick and Drift`	3.93 ± 0.052
`Coordinate conversion`	1.58 ± 0.278
`Filter lost particles & Transpose`	0.826 ± 0.037
`Create Tomo object`	0.993 ± 0.034
`Tomography reconstruction`	3.91 ± 0.037
`Create phase space`	$0{,}038 \pm 0.029$

Table 6: Mean runtime of the main procedures for $B = 200$ and $N = 500$ (Baseline).

The values in Table 6 show the runtimes taken with the timing package in Python for the high-level workflow procedures. What can be seen is that the two procedures `Kick and Drift` and `Tomography reconstruction` take up most of the application's runtime despite being optimized by using C++ with OpenMP. Additionally, the conversion of the coordinates takes up a considerable amount of time, which scales with the number of profiles used. As mentioned before, the breakdown of `Kick and Drift` and the reconstruction algorithm cannot be measured with the timing package and a notion of the runtime of the subfunctions can be provided by VTune, which is shown in Table 7. Since VTune displays the runtime for each thread running, the maximum runtime for

each function of each thread has been taken and divided by three to account for three executions. Unfortunately, this does not give any information about the variance in runtime of the functions. However, it can be used as an indicator of which function the program spends most of the time in.

Function	Number of Calls	Runtime in seconds
`kick_up`	4491	2.88
`drift_up`	4491	0.12
`Calculate reciprocal particles`	1	0.179
`Create flat points`	1	0.08
`Back project`	21	0.542
`Clip`	21	< 0.001
`Project`	21	3.136
`Normalize`	21	< 0.001
`Calculate Difference`	21	0.002
`Discrepancy`	21	0.003
`Compensate for particle amount`	20	< 0.001

Table 7: Mean runtime of the subfunctions for $B = 200$ and $N = 500$ (Baseline).

The subfunctions of `Kick and Drift` and `Tomography reconstruction` do not necessarily add up to the runtime of the upper function. There are two reasons for this, the first is that there are memory write operations executed inside the `Kick and Drift` function to store the particle coordinates which do not happen in a subfunction and takes up a part of the runtime as well. In addition, the different profiling tool does not take the overhead into account caused by calling C++ code from pybind11 where NumPy arrays have to be converted to C++ arrays and vice versa.

Considering the `Kick and Drift` function, most of the runtime comes from the `kick_up`/ `kick_down` functions. Evidently, these functions are more complex than the corresponding `drift` functions since they have to compute two sines, which, despite using the fast version of the VDT library, take up more time than addition/subtraction operations in the drift functions. For the `Tomography` part, the `Project` function takes up most of the time, which can be explained by the fact that it is not parallelized due to the reasons explained earlier.

Figure 18 shows the runtime of the two main C++ functions with a different number of threads used. To enable this, the environment variable `OMP_NUM_THREADS` is set to the

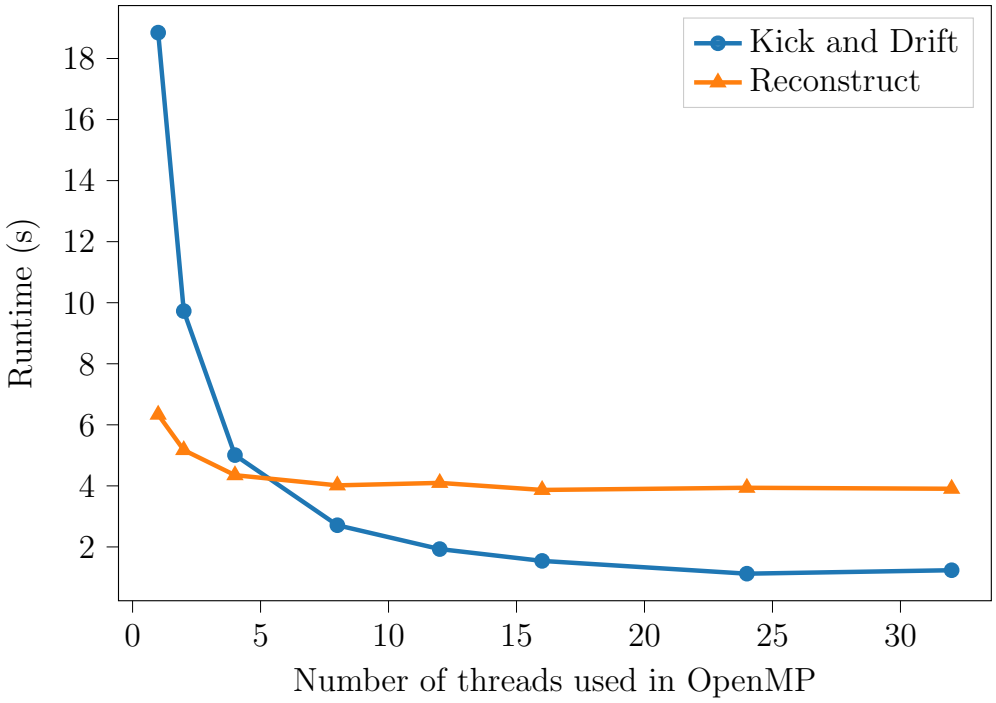


Figure 18: Runtime of `Kick and Drift` and `Tomography reconstruction` under different numbers of threads.

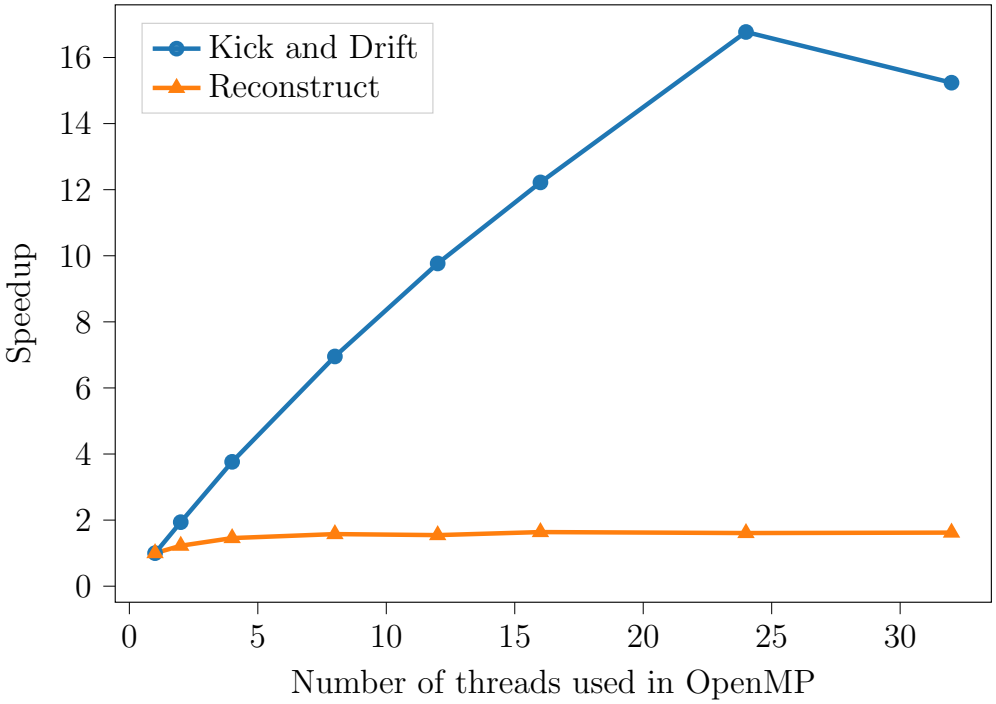


Figure 19: Speedup of `Kick and Drift` and `Tomography reconstruction` under different numbers of threads.

number of threads that should be used from OpenMP. A node with 32 cores was used to perform these measurements. The application was executed three times again and the mean runtime has been used for this. Due to the standard deviation being between

below 5% and in most cases even below 2%, the deviation cannot be seen in the plot.

However, it can be seen how `Kick and Drift` scale with the number of threads. This becomes even more visible in Figure 19 where the speedup of the two functions with respect to the runtime with one thread is shown. The speedup gained from using more threads decreases after reaching its peak at 24 threads with a speedup of almost 17.

Considering the reconstruct function, the speedup is only noticeable when using a low number of threads. Still, it does not reach a speedup of two, so the benefit of using multiple threads for the reconstruction is marginal. This is due to the fact that the `Project` function is not parallelized with OpenMP and takes up most of the runtime, as indicated by Table 7. One can conclude that `Kick and Drift` is a computing-heavy function where the speedup scales well with the number of threads since phase and energy coordinates of every particle can be changed independently. The reconstruction is limited by the `Project` function, which, due to the memory access patterns, would need a more fine-grained optimization.

Furthermore, the runtime can also be observed using varying numbers of bins and numbers of profiles to see how the functions scale. The plot in Figure 20 shows the runtime together with the standard deviations of several functions. In this comparison, different numbers of bins are used while fixing the number of profiles and turns as before. While it has no effect on the `Values at turns` function, because that one scales only with the number of turns and implicitly with the number of profiles, it shows a clear effect on all the other functions. This is due to the macroparticles scaling proportionally with the number of bins squared. The quadratic runtime complexity is shown clearly on the line for the reconstruction, but it is also present for the functions, although it is harder to see in the plot that these functions have a lower runtime at 50 bins than the others. The runtime of all functions at 500 bins is more than 100 times higher than at 50 bins, showing the quadratically growing behavior.

Furthermore, a benchmark was conducted the same way considering a different number of profiles and a different number of turns, while keeping 200 bins and an interval between profiles ΔT of nine turns. For the following comparison, 100, 200, 500, 1000 and 2000 profiles are used, and respectively a number of turns from ranging from 891 to 17991.

Figure 21 shows the runtime of this benchmarks together with the standard deviations. In contrast to the previous example, all the functions scale linearly with an increase of the bins. This observation is confirmed by the fact that the runtime at 2000 profiles is

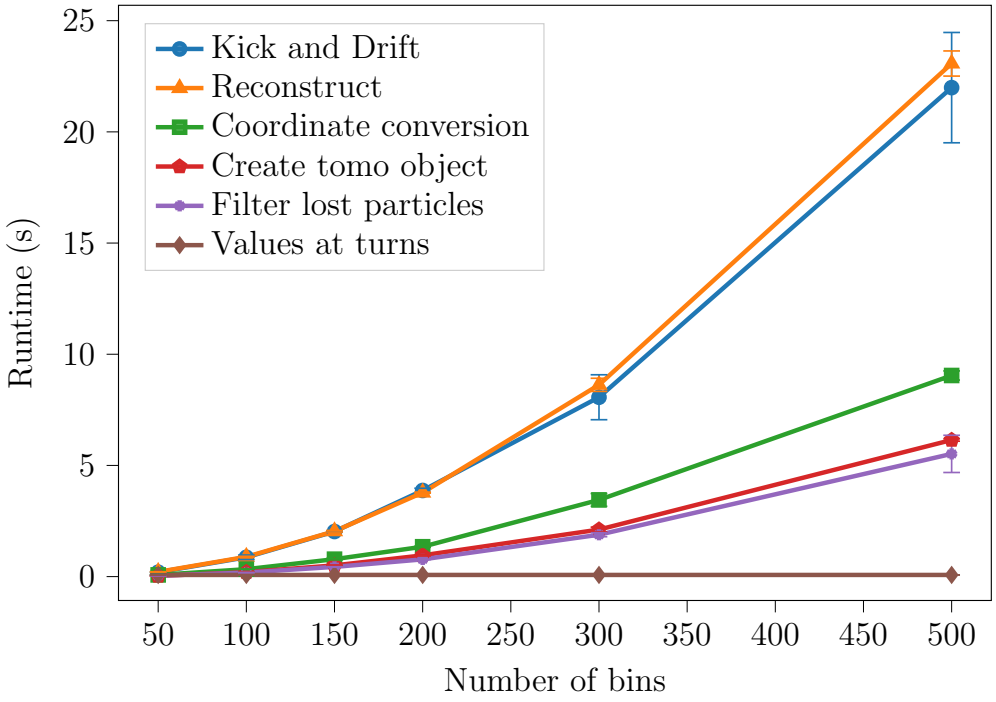


Figure 20: Runtime of several functions of the Tomography workflow under different numbers of bins.

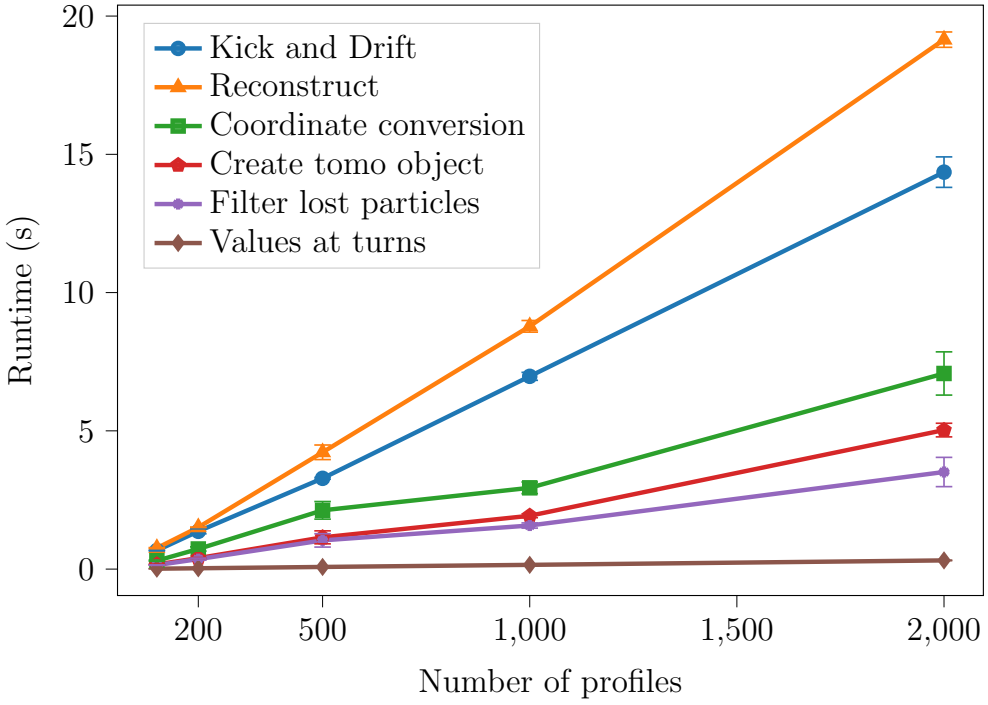


Figure 21: Runtime of several functions of the Tomography workflow under different number of profiles and turns.

between 20 to 30 times higher than at 100 profiles.

While this provides insights about the scaling of the functions when both of these

parameters are changed, the behavior is not necessarily the same when fixing one parameter and changing the other one. Figure 22 shows a benchmark in which the number of profiles was fixed to 500, the number of bins to 200, and only ΔT was varied between one and 32, resulting in a number of turns between 499 and 15968 turns.

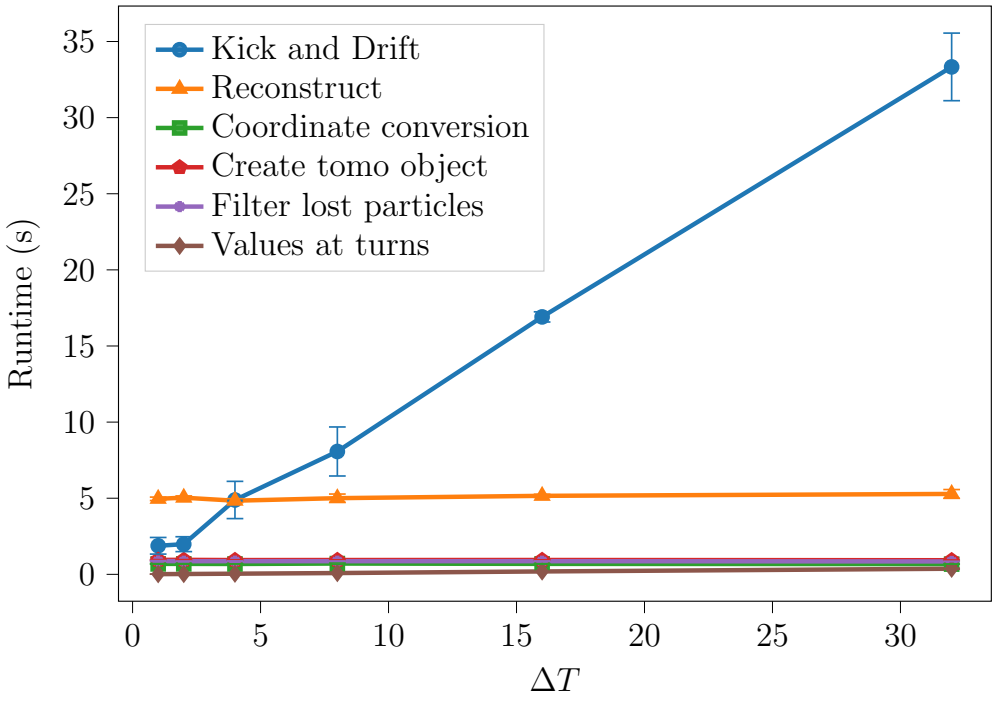


Figure 22: Runtime of several functions of the Tomography workflow under different number of profiles and bins.

This Figure validates the parameters shown in Table 5 since only `Kick and Drift` and `Values at turns` increase their runtimes in this plot, although the latter cannot be clearly seen due to the very low base runtime of the function. In this context, it can be seen that the change in number of turns has no effect to the reconstruction or the other functions that are executed after the calculation of the tracked coordinates. This is evident because it is only used to track the particles for the number of turns needed and the array of tracked particles only contains a number of coordinates corresponding to the number of profiles for each particle.

Finally, a last benchmark was conducted in which the number of turns was fixed to 4500 and the number of profiles and ΔT was varied. Table 8 shows the used values for the number of profiles and the interval between two profiles.

Since Figure 21 showed an increasing runtime for all functions when increasing the number of profiles and turns, it is expected that the other functions scale with the number of profiles, regardless of the number of turns. This expectation is validated by the last benchmark for which the runtimes are shown in Figure 23. Here, `Kick and`

Number of profiles N	Interval between two profiles ΔT
101	45
226	20
501	9
751	6
1126	4
1501	3

Table 8: Number of profiles and interval between turns for the benchmark using 4500 turns.

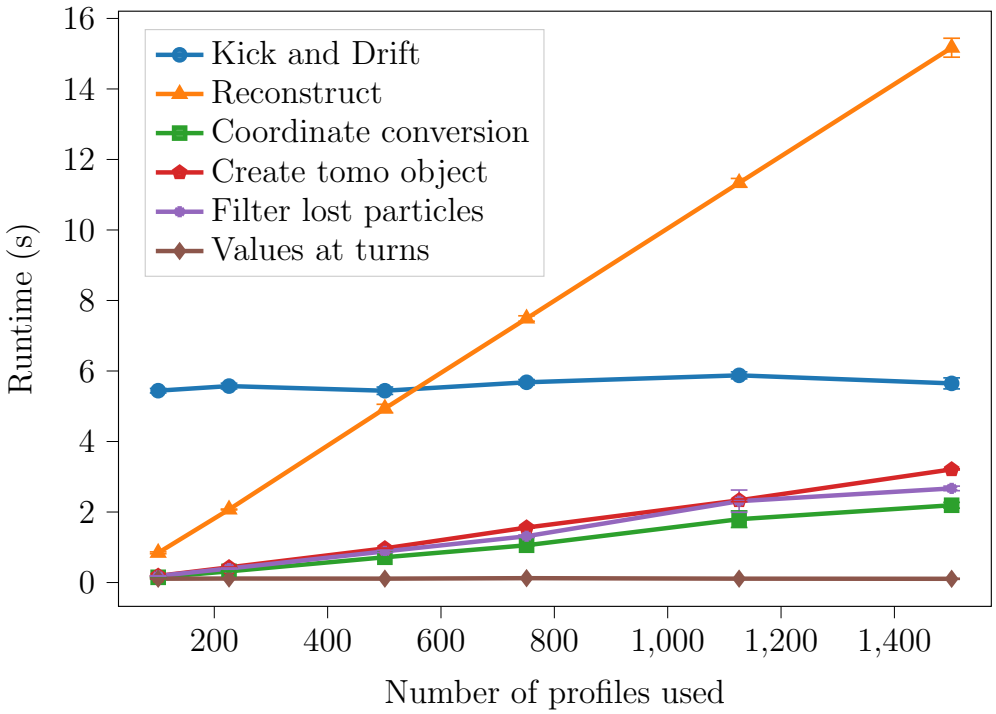


Figure 23: Runtime of several functions of the Tomography workflow under different numbers of profiles while fixing the number of turns to 4500.

`Drift` and `Values at turns` are the only functions which do not have a significant change in runtime when increasing the number of profiles while fixing the number of turns. This is expected as well according to the scaling parameters given in Table 5. All the other functions scale linearly with an increasing number of profiles.

To conclude, all the benchmarks show that the two functions `Kick and Drift` and `Tomography` account for most of the runtime of the application, despite being written in C++ and using OpenMP. Taking into account that this is the reason why the effort has been conducted to implement these functions in C++, it is an expected result, underlining the computational effort needed for these functions. This observation does

not change when scaling up the number of bins or profiles. Additionally, inside these functions, the bottlenecks appear to be in `kick_up` and `Project`, for different reasons. While `kick_up` is computationally heavy and computes two sines for each particle, the `Project` functions uses simple arithmetic instructions but performs multiple memory operations. Due to the memory access pattern, the function is not trivially parallelizable which could explain the large difference in runtime between this function and `Back project`.

5 Optimization Approaches

In this chapter, several possible optimization approaches will be discussed to optimize the application, with a focus on Kick and Drift and the tomography reconstruction. It breaks down to Python implementations in Numba and CuPy as well as an implementation in CUDA C++. As Numba did not show promising performance results, it is only considered once when comparing with the other approaches, but not when implementing other optimizations. CuPy will be used to deal with allocating and transferring arrays from host to device memory and vice versa, as well as for some functions which are not crucial for the runtime and therefore a more simple implementation can be profitable. The implementation in CUDA will be the main part where most of the performance gains are achieved.

5.1 Numba

Despite the fundamental performance disadvantages of using the interpreted language Python instead of the compiled language C++, benchmarks in BLonD have shown that some functions achieve a better performance when using Python with Numba instead of C++ [38]. BLonD shares similarities with the tomography application in the sense of using Python for the main workflow and C++ for computationally intensive functions, providing an incentive to explore using Numba for the longitudinal tomography.

In order to use Numba, the C++ functions were transcribed to Python. Since Numba has a good synergy with NumPy arrays, there is no need for additional conversion to any data types. For the functions in `Kick and Drift`, this step is simple due to the fact that basic mathematical operations work on a NumPy ndarray level with implicit looping. However, using NumPy functions to implement the algorithms is not always obvious in order to avoid loops entirely. Additionally, when using multiple threads, there can be constraints using only NumPy functions, for example, due to the more complicated indexing used in the projection and back projection. In this case, memory collisions can happen when executing the function in parallel. For this purpose, some functions retain the loops which are optimized when compiled through Numba.

To make the functions compiled by Numba, the decorator `@njit` is used which differs from `@jit` by not allowing a fallback from `nopython` to `object mode`, as explained in Section 3.1. To use parallel threads, the `parallel` option has to be set in the decorator as follows: `@njit(parallel=True)`. Listing 10 shows the `Back project` function implemented with Numba. Note that the function `prange` is used for the loops instead of `range`. It specifies that the loop can be executed in multiple threads when

setting the `parallel` option.

```
@njit(parallel=True)
def back_project(weights, flat_points, flat_profs, n_parts, n_profs):
    for i in prange(n_parts):
        for j in prange(n_profs):
            weights[i] += flat_profs[flat_points[i * n_profs + j]]
    return weights
```

Listing 10: `Back project` function in numba.

The `Kick and Drift` and `Tomography reconstruction` functions have been optimized this way. The other parts remain unchanged as the effect on the performance would not be significant. For many other functions, more complex Python objects like the Machine object are used. Therefore, it is not feasible to rewrite them using Numba.

Considering the runtime optimization of each function is not sufficient in this case. The compilation time has to be taken into account as well since the compilation happens during runtime. Depending on the number of functions and their complexity, this can take a few seconds.

5.2 CuPy

To optimize the functions using GPUs, CuPy is used to interface the GPU with Python code. This requires the functions for the tracking and reconstruction to be rewritten to perform the computation on CuPy arrays instead of NumPy arrays. Additionally, the required data has to be transferred to the device with CuPy, ideally as early as possible to profit from the faster computation. Since the `Values at turns` function does not impose a considerable burden on the runtime and due to its complexity in calculating the values, the data transfer can happen afterwards before starting the tracking.

CuPy functions are mostly compatible with NumPy. Therefore, many functions do not need to specify if they receive NumPy or CuPy arrays if they call the functions from the array objects. Since CuPy can only be installed and imported to the Python application if a CUDA environment is installed, which requires a GPU, it can cause problems when calling functions directly from the package. This requires changes in certain classes or adding new classes that will only be used when CuPy is enabled. Specifically, the package should only imported if a GPU is available to prevent the application crashing.

After preprocessing the bunch profile data, it can directly be transferred to the GPU. If

the profiles require preprocessing, this is done on the CPU because the operations are simple and it normally reduces the size of the input array. Therefore, this reduces the amount of memory copied to the GPU. When generating simulated data with BLonD, one has to convert it explicitly using `cupy.asarray`. Otherwise, when, for example by collecting the required data from a file, the data array can be created directly on the GPU without initializing it in the CPU first, if no preprocessing is required. Besides the bunch profiles, the coordinates of the tracked particles also have to be on the GPU to accelerate the tracking functions. Therefore, after creating the homogeneous distribution, CuPy arrays are created. Consequently, the tracking, the tomography functions, including the creation of the Tomography class, the coordinate conversion and the filtering of the particles can be performed on the GPU. This is desirable as these functions account for most of the runtime.

The `Kick and Drift` function requires only slight changes starting from the Numba version. The necessary parameter arrays for the equations have to be transferred to the GPU as well. These arrays scale with the number of turns, therefore they are not very large arrays and should not cause a long waiting time. Next, the `kick` function needs a slight modification to use the sin function from CuPy instead of the one from NumPy as in the Numba implementation. The `kick` function is demonstrated in Listing 11, performing the energy kick in the forward direction.

```
def kick_up(dphi, denergy, rfv1, rfv2, phi0, phi12,
            h_ratio, n_particles, acc_kick):
    denergy += rfv1 * cp.sin(dphi + phi0) \
        + rfv2 * cp.sin(h_ratio * (dphi + phi0 - phi12)) - acc_kick
    return denergy
```

Listing 11: Forward energy `kick` method in CuPy.

Additionally, some potential improvements in the transition from NumPy to CuPy are possible when checking the constraints if the lost particles have been filtered correctly. This is checked when creating the Tomography object. For that, the check can be performed in different ways, effectively achieving the same result. For example, to find the particles outside of the image width, the following expression is used:

```
invalid_pts = np.argwhere(np.logical_or(xp >= img_width, xp < 0)).
```

That provides a boolean mask for all the particles. Here, `np` is the abbreviation for NumPy, `img_width` is the number of bins in the profiles and `xp` is the array of the x-coordinates of all particles. When checking the constraint, building this mask is not

necessary as it can be checked if at least one particles does not fulfill the requirement. For that purpose, the `any` method can be used instead of `logical_or` for the condition check, as follows demonstrated with CuPy:

```
cp.any(xp >= img_width) or cp.any(xp < 0)
```

Note that `cp` is used to abbreviate CuPy. This statement works in NumPy as well, the runtimes for both approaches can however differ for both libraries. Given that these steps are executed between the tracking and the reconstruction, moving the arrays from the GPU to the CPU and then moving them back is unnecessary. Additionally, due to the fact that the particle arrays can be very large, using CuPy here instead of NumPy can make a substantial difference.

For the reconstruction, the functions implemented for Numba can be modified. However, using loops with CuPy is very slow and will lead to strong runtime penalties. This means that all the functions should be implemented using only the highly optimized CuPy operations as much as possible. For the `Back project` function shown in Listing 12, the weights can be adjusted by using the `sum` method for one axis and using an array of indices, which is very common in NumPy, instead of iterating over all indices explicitly.

```
1 def back_project(weights, flat_points, flat_profs, n_parts, n_profs):
2     return cp.sum(flat_profs[flat_points], axis=1) + weights
```

Listing 12: `Back project` function in CuPy.

The `flat_profs` array contains the measured profiles or profile difference, with shape $N \cdot B$, N being the number of profiles and B the number of bins, while `flat_points` contains the indices to take for the weights, given by shape (M, N), M being the number of macroparticles. Through the way of using the `flat_points` array for indexing, the slice of `flat_profs` array has the shape of `flat_points` and the sum is then computed over the profiles. The result is an array of size N which contains the difference to add to the weights of the particles.

```
1 def project(flat_rec, flat_points, weights, n_profs):
2     return cp.bincount(flat_points,
3     weights.repeat(n_profs), minlength=n_profs * n_bins) + flat_rec
```

Listing 13: Project function in CuPy.

The `Project` function is more intricate to implement without loops due to the indexing

used (see Listing 9). Here, the `bincount` function is used on the `flat_points` array to count the number of particles, multiplied with their weights, in each profile to add onto the projection. There are two things to be noted here. First, the `weights` array is repeated N times to adjust to the shape of `flat_points`. Second, a minimum length of the array is given because `bincount` infers the length based on the highest bin number occurring in the array. Since there can be bin numbers on the borders without particles, the resulting array could be smaller and mismatched with the projection area.

To count the particles in each bin, which is a subfunction of the `Calculate reciprocal particles` function, a loop was used to compute the histograms for each profile. Figure 14 shows the implementation of the function using the `histogram` function which computes the number of particles for each bin. This can result in a smaller improvement in terms of speedup compared to a fully parallelized solution, for example using CUDA.

```
1 def count_particles_in_bin(xp, n_profs, n_bins):
2     bins = cp.arange(n_bins + 1)
3     rparts = cp.empty((n_profs, n_bins), dtype=cp.int32)
4     for j in range(n_profs):
5         rparts[j], _ = cp.histogram(xp[:, j], bins=bins)
6     return rparts
```

Listing 14: Histogram of particles per profile in CuPy.

Finally, all the other functions were implemented only using CuPy operations. Since their runtimes were already very low in comparison to the other functions, and the fact that the functions are rather simple, they do not cause performance issues when executed with CuPy instead of C++.

5.3 CUDA

The main part of optimizing the longitudinal tomography is an implementation in CUDA C++ in combination with CuPy as a tool to interface between the Python code and the CUDA kernels. In this section, the general structure will be explained as well as how CuPy is used in this context. The other parts of the section will deal with specific runtime optimizations of the kernels.

5.3.1 General Implementation

To facilitate the option of choosing the GPU to execute the application, a convenience class `GPUDev` has been created which takes care of all the information needed. It is

implemented as a singleton which stores the information about the GPU device that is used, like the compute capability, the number of streaming multiprocessors and the maximum number of threads per block. This approach uses already compiled CUDA binary files, however it is also possible to compile at runtime. However, it is desirable to separate the compilation process from the execution since multiple executions would compile the code every time. Listing 15 shows an excerpt of the `GPUDev` class where information about the GPU is gathered and the module for `Kick and Drift` is imported based on the compute capability is has been compiled for.

```
dev = cupy.cuda.Device(0)
dev.use()
attributes = dev.attributes
comp_capability = dev.compute_capability
properties = cupy.cuda.runtime.getDeviceProperties(dev)
name = properties['name']
max_threads = attributes['MaxThreadsPerBlock']

kd_mod = cp.RawModule(path=f'path/to/kick_and_drift_sm{comp_capability}.cubin'))
rec_mod = cp.RawModule(path=f'path/to/reconstruct_sm{comp_capability}.cubin'))
```

Listing 15: Accessing GPU information with CuPy.

CuPy encapsulates the CUDA runtime API and makes it accessible through a wrapper `cupy.cuda.Device`, which allows the attributes and the compute capability of the GPU to be queried. The compute capability is important to know when making the application agnostic to which GPU is used since different models come with different compute capabilities. The source files have to be compiled separately for each of the compute capabilities used, therefore the class searches for a specific binary file when importing the module. Creating the `RawModule` allows the kernels to be called by querying the kernel function name and executing it with the corresponding arguments. Listing 16 demonstrates this for the `Project` function.

First, the kernel is retrieved by calling `get_function` on the raw module. The kernel execution takes three parameters, the first one being a tuple of all the required arguments needed by the kernel. The other two parameters determine how the grid and the thread blocks are structured with 3-tuples. Since throughout this implementation, only one-dimensional grids and thread blocks are used, the second and the third component are set to one. The size of the grid is calculated at runtime and is based on the parameters the function takes. Since the `Project` function iterates over particles and profiles and

one thread can be used for each particle and profile, the grid size is calculated to have a number of blocks that can fit at least $M \cdot N$ threads.

```
proj_kernel = rec_mod.get_function("project")
proj_kernel(args=(flat_rec, flat_points, weights, n_parts, n_profs),
            block=(max_threads,1,1),
            grid=(int((n_parts * n_profs) / max_threads + 1), 1, 1))
```

Listing 16: Executing the `Project` kernel through CuPy.

To showcase how a kernel is implemented and which optimizations are taken into account, Listing 17 displays the CUDA kernel that calculates the difference profile based on the measured profile `flat_profiles` and the reconstructed profile `flat_rec`. CUDA kernels are always void by definition, they do not return anything, so CuPy takes care of the arrays so the result arrays are visible to the Python application. To make the kernels visible to CuPy, they have to be declared with the `extern "C"` directive which prevents the compiler from mangling the function name. Although NVIDIA's NVCC compiler supports mangling, CuPy is unable to detect the function kernels from their name due to not supporting overloaded functions. This has the consequence that it is harder to work with templates, for example to differentiate between different data types like `double` and `float` for the precision. To enable this possibility, the function can be implemented with templates and has to be wrapped by one function for each possible type. This increases unnecessary code and worsenes the maintainability. In order to avoid unnecessary code, a compile time variable `real_t` was introduced which is already set when loading the module into CuPy. This comes with the disadvantage that the code has to be compiled for each precision, but, since there are only two options, this is not an issue. Further details will be discussed in Section 5.3.2.

```
extern "C" __global__ void find_difference_profile(
            real_t * __restrict__ diff_prof,            // out
            const real_t * __restrict__ flat_rec,       // in
            const real_t * __restrict__ flat_profiles,  // in
            const int all_bins) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    for (int i = tid; i < all_bins; i += blockDim.x * gridDim.x)
        if (i < all_bins)
            diff_prof[i] = flat_profiles[i] - flat_rec[i];
}
```

Listing 17: CUDA Kernel to calculate to difference profile after the projection.

Another point of interest is the usage of `__restrict__` pointers, which allow the compiler

to optimize more generously as this keyword promises the compiler that the memory of the pointers do not overlap with each other. Evidently, this is only useful if there are at least two arrays which could interfere with each other. If there is only one array as an argument, it does not provide additional benefit to use the keyword.

Next, the grid and block sizes are determined when calling the kernel from CuPy. Usually, the maximum possible block size is used, which is 1024 for all the models in this work. However, the kernels can have specific requirements how many blocks are needed, which will be relevant in Section 5.3.4 where the a deeper look into the `Back project` kernel is taken. In every case, the grid size will be set to fit enough blocks to ensure the smallest possible workload per thread. The kernel shown in the example subtracts one array from another, each with $N \cdot B$ elements, N being the number of profiles and B the number of bins. Therefore, an optimal grid would encompass enough blocks to provide $N \cdot B$ threads.

Always using an approach like this, the `for` loop as it is written in line seven of Listing 17 would be unnecessary since the possible number of threads outscale every array sizes used. The global thread id `tid` is calculated in line six, using the introduced equation for one-dimensional grids and blocks, which is used to index the corresponding elements inside the array. It is also necessary to check if the global thread id exceeds the array length, which is likely to happen as it is not a multiple of the thread block size.

For the `Project` function, atomic operations must be used due to the race conditions that can happen when adding the weights on the projection. Therefore, `atomicAdd` is used, which is provided by the CUDA API. The implementation is shown in Listing 18.

```
1 if (tid < npart * nprof)
2 {
3     int idx = flat_points[tid];
4     atomicAdd(&flat_rec[idx], weights[tid / nprof]);
5 }
```

Listing 18: Excerpt of the `Project` kernel where atomic additions are used to add the weights.

In this example, every thread gets one index given by the `flat_points` array which is used to index the projection. Since there can be collisions between the profiles, it is not possible to arrange the threads in a way that avoids using atomic operations. However, due to the structure of the `flat_points` array, indices next to each other are never the same due to the addition of the number of bins (see Listing 7), not many threads will

have the same index, even more unlikely inside the same warp. As a consequence, it can be one of the slower functions due to the large amount of memory indexing, which scales with the number of particles and profiles, but the runtime penalty should be acceptable.

Since reduction algorithms are more intricate to implement and understand, the functions `Normalize` and `Discrepancy` are kept in CuPy as their runtimes are very low and it would not make a significant difference when implemented in CUDA. In this case, the simplicity of the implementation is preferred over a slightly faster function in absolute terms.

5.3.2 Single Precision Floating-Point Numbers

One change which can have a strong impact on the runtime is using single precision floating-point numbers instead of double precision. This can lead to substantial speedups, depending on the GPU architecture used. GPUs like the V100 have several dedicated FP64 computing cores, which handle double precision floating-point numbers [28]. However, GPUs like the Tesla T4 [39] incorporate only very few FP64 to compute the calculations and where the peak performance is 32 times faster when using single instead of double precision. Especially because the Tesla T4 is the most accessible GPU model at CERN, this is a point that has to be taken into account.

As already implied in the previous section, this is implemented via a variable which is set at compile time to define if `double` or `float` is used as a data type. The kernels must be called with arrays that have this exact precision, otherwise the results will not be correct or the program might even crash due to illegal memory accesses. Therefore, the workflow in Python has to be aware which type is being used in multiple stages. These are when importing the raw CUDA modules and when declaring or preparing the CuPy arrays to call the kernels. Using this compile time variable reduces boilerplate code because it is not necessary to wrap each function for each data type.

To simplify the interface for the user, another convenience class `TomoConfig` has been established to keep a state about if the GPU or CPU implementation and which precision is used. While the default configuration uses double precision floating-point numbers and the CPU implementation, one can change this explicitly by calling `set_single_precision()` or `use_gpu()`. The latter serves as a facade which encapsulates the implementation by mapping the function names to the corresponding implementations. This also applies to the NumPy and CuPy functions which in most parts are smoothly exchangeable. Creating the arrays through this class liberates the

user from having to explicitly declare if they are allocated on the GPU or CPU, and if they contain double or single precision values. Incorporating these changes, a precision agnostic CPU version can be created, which can also perform the tomography with single precision values to analyze the performance advantages. This mapping is done by a function dictionary which is kept by the class and is updated every time a function is called that changes the precision or the device used. Depending on which device, it adds all the NumPy or CuPy functions into this dictionary, making it not visible for the user if NumPy or CuPy arrays are being handled.

Since the tracking part is computationally intensive and is more constrained by the floating-point operations the GPU can perform per second, this change can affect the tracking in particular. For the reconstruction, a large improvement is not expected due to the fact that the runtime is more constrained by the memory latency due to suboptimal access patterns than by the computations performed. As for the `Project` function, the memory access pattern for the `flat_points` is not coalesced and is likely to have a stronger effect on the runtime than performing the addition.

5.3.3 Increase Workload per Kernel to Reduce Invocation Overhead

Especially when the tracking is concerned, the number of kernel invocations can be very large as it scales with the number of turns needed, which scales linearly with the number of profiles N and the number of turns between two profiles ΔT. Therefore, it should be seen if it is better to keep the `Kick and Drift` workflow inside Python or if the whole `Kick and Drift` loop should be transferred to CUDA. The reason for this is that there is a overhead when invoking a CUDA kernel and this might cause runtime penalties if the called kernels are very short, which could be the case for the rather simple drift function. To analyze that, the profiling tool NVIDIA Nsight Systems can be used. As already described in Section 3.2.4, it tracks the number and duration of CUDA API calls from which the invocation overhead can be derived.

One step to increase the workload per kernel is to fuse the kick and the drift operation into one kernel. The fused kernel for `Kick and Drift` in forward direction is shown in Listing 19. The parallelization for `Kick and Drift` is simple since there are no dependencies between particles, so M threads can be used without further modification of the function. The only constraint is that the drift has to be performed before the kick in forward direction and after in backward direction so the old energy/phase value does not have to be saved additionally to be used in the equation. That is, if `denergy[tid]` would be modified before `dphi[tid]` in this kernel, it would lead to incorrect results. As the execution inside a single thread is sequential, this is not an issue. Fusing these

functions together to one kernel can already halve the number of kernel invocations for `Kick and Drift`, effectively removing 50% of the invocation overhead.

```
extern "C" __global__ void kick_drift_up(
        real_t * __restrict__ dphi, real_t * __restrict__ denergy,
        const real_t drift_coef, const real_t rfv1, const real_t rfv2,
        const real_t phi0, const real_t phi12, const real_t hratio,
        const int nr_particles, const real_t acc_kick) {
    int tid = threadIdx.x + blockDim.x * blockIdx.x;
    if(tid < nr_particles) {
        dphi[tid] -= drift_coef * denergy[tid];
        denergy[tid] += rfv1 * sin(dphi[tid] + phi0)
            + rfv2 * sin(hratio * (dphi[tid] + phi0 - phi12))
            - acc_kick;
    }
}
```

Listing 19: Fusing `Kick and Drift` in forward direction to one kernel.

Despite this approach already leading to fewer kernel invocations, the workload per thread is still low and the runtime for a kernel execution should be very low, ignoring the invocation overhead. To further improve this, the entire `kick_and_drift` function can be transferred from Python to a CUDA kernel. This will move the loop iterating over the turns inside the kernel, which therefore has to store the particle coordinates in the arrays as well. The effect of this is expected to be strong since it reduces the number of kernel calls to one or two, depending on if only forward or also backward direction is executed, effectively removing almost the entire invocation overhead.

The resulting runtimes, and the metrics given by the profiling tools will be presented in the next chapter where a conclusion can be made which approach leads to the best results.

5.3.4 Parallel Scan for Back Project

The next optimization concerns the implementation of the `Back project` kernel, in particular with respect to the memory access. Since the weight of one particle is the sum of the projections for every profile at the corresponding index, the calculation of a weight consists of N values that have to be added to the current weight, N being the number of profiles. Using a naive approach with $M \cdot N$ threads requires the use of atomic operations that can lead to substantial performance penalties. In contrast to the projection function, the consequences of using atomic operations here are more

severe due to the fact of always having N store operations per weight. Additionally, the number of collisions happen more frequently because the thread IDs are grouped for each particle; the first N threads add the values to the first particle, the second N threads to the next, and so on.

Another naive approach would be to use only M threads and do the N additions sequentially in one thread. This avoids the use of atomic operations, but increases the number of memory operations per thread, since N memory load operations are needed to get the indices for the projection from the `flat_points` array, another N memory load operations to access the values of the projection in the `flat_profiles` array, making $2N$ load operations and N additions per thread. Therefore, this approach is even worse than the first one where only one load and store operation happens per thread, but not necessarily immediately due to the atomic operations.

The alternative approach, which has been implemented is a parallel scan operation which aims to optimize the utilization of the GPU resources by using smaller blocks. Each block calculates the weight for one particle and thus, M blocks are used with smaller thread sizes, typically 32 threads. Although this increases the number of blocks from $\lceil \frac{M+N}{1024} \rceil$ to $\lceil \frac{M}{32} \rceil$, more blocks can be scheduled and executed on a streaming multiprocessor at the same time.

The parallel scan algorithm is visualized in Figure 24 in a simplified manner and consists of different stages. The values to be summed are distributed on different threads and the operations are spread over as many threads as possible. This is a typical pattern for reduction operations in parallel environments. In this example, one block is used with sixteen threads that are divided into four warps, denoted by four different colors. 64 values are being added in this example, so every thread controls four values, which are saved in registers. In the first step, these values are sequentially reduced to one value per thread between the timestamps $t = 0$ and $t = 3$. After that, every pair of threads adds their values together. Then, the result is stored in the thread with the smaller index. This happens twice, after timestamps $t = 3$ and $t = 4$, until every warp has been reduced to one value. Then, a synchronization is needed since different warps are not necessarily scheduled at the same time, in order to prevent using partial sums from the warps. From this point, every sum operation over different warps requires a barrier to ensure that the correct values are used.

In the implementation, a block size of 32 threads is used, therefore every thread blocks consists of one warp, meaning that no warp-wide additions happen. Therefore, only the thread-internal sequential register reduction and the reduction inside the warp is

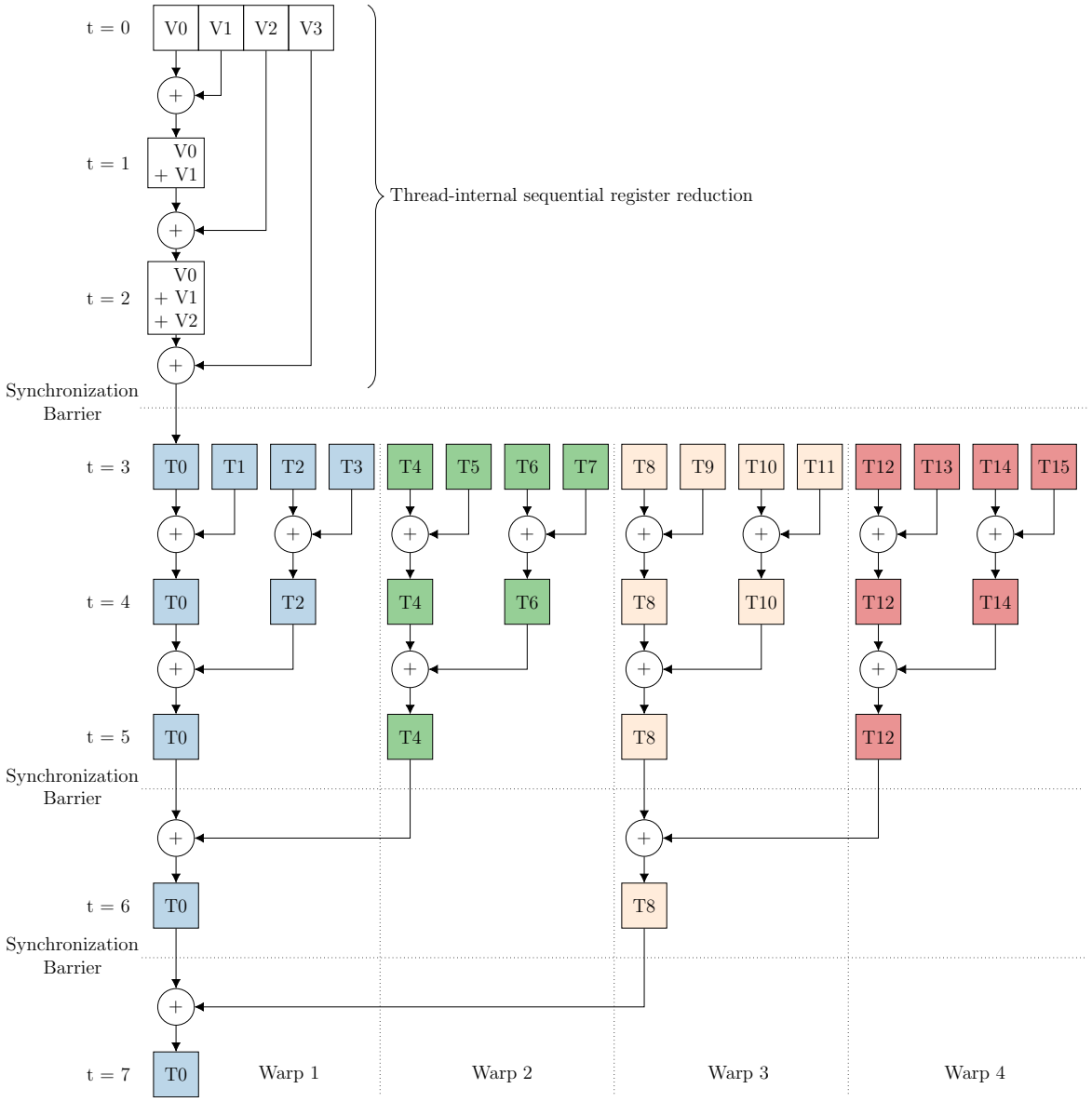


Figure 24: Flowchart of a parallel scan operation. For simplification, warps only contain four instead of 32 threads.

performed in order to aggregate the values. For 512 profiles, a block size of 32 and 16 values per thread for the register reduction, 15 sequential operations are needed for the register reduction, and another $\log_2 32 = 5$ for the reduction from 32 to one thread. This is smaller than 512 atomic additions to one memory address, since the waiting times are significantly lower because there is no memory latency involved in performing the reduction.

To implement this, the GPU-accelerated library CUB [27] was used. CUB provides primitives for block-wide operations that can be used to perform reduction operations and other types of parallel algorithms. In this case, the `BlockReduce` class is used which performs the algorithm described in Figure 24.

```
1  extern "C" __global__ void back_project(real_t * __restrict__ weights,
2          int * __restrict__ flat_points,
3          const real_t * __restrict__ flat_profiles,
4          const int npart, const int nprof) {
5      const int BLOCK_SIZE = 32;
6      const int ITEMS_PER_ARRAY = 16;
7      const int ITEMS_PER_IT = BLOCK_SIZE * ITEMS_PER_ARRAY;
8      int iterations = (nprof + ITEMS_PER_IT - 1) / ITEMS_PER_IT;
9
10     real_t aggregate = 0.0;
11     for(int i = 0; i < iterations; i++) {
12         typedef cub::BlockReduce<real_t, BLOCK_SIZE> BlockReduce;
13         // allocate shared memory for BlockReduce
14         __shared__ typename BlockReduce::TempStorage temp_storage;
15         real_t weight_prof[ITEMS_PER_ARRAY];
16         for (int j = 0; j < ITEMS_PER_ARRAY; j++) {
17             int index = i * ITEMS_PER_IT + j * blockDim.x
18                         + threadIdx.x;
19             if (index < nprof)
20                 weight_prof[j] = flat_profiles[
21                     flat_points[blockIdx.x * nprof + index]];
22             else
23                 weight_prof[j] = 0.0;
24         }
25         __syncthreads();
26         aggregate += BlockReduce(temp_storage).Sum(weight_prof);
27     }
28     if (threadIdx.x == 0)
29         weights[blockIdx.x] += aggregate;
30 }
```

Listing 20: The `Back project` kernel using CUB's `BlockReduce` class.

The kernel code implemented this way is shown in Listing 20. In the first part, some constants are set which have to be known at compile time in order to initialize the `BlockReduce` class. Cases in which the number of profiles exceeds 512 are taken into account by performing the reduction multiple times. Therefore, the number of iterations is calculated in `iterations` with $\lceil \frac{N}{512} \rceil$. The `BlockReduce` is then defined in line 12 for the data type and block size needed. To perform the computation, it requires a temporary storage, which is created in line 14. Providing this storage and an array of values, which is declared individually for each thread, the reduction is performed in line 26 by calling `Sum` on `BlockReduce`. The `weight_prof` array is declared for every thread and contains the values to sum in the corresponding thread. In the `for` loop between line 16 and 24, the `weight_prof` is filled with the values from the projections.

If there are fewer profiles than values for the iteration, the rest of the fixed size array is filled with zeros as this does not change the result in a sum. Finally, after filling this array and synchronizing the threads, the reduction is executed and stored in the `aggregate` variable. This aggregated value is added to the weights and in every case only one memory store operation is needed for each block. That is also why the check in line 28 is needed to perform this addition and store operation only once.

6 Results of the Performance Study

In this chapter, the results of the benchmarks will be discussed. First, the setup will be presented, showcasing the hardware used, with a focus on the different GPU models. Then, how the benchmarks are conducted and which parameters are used will be explained. Throughout the rest of the chapter, the runtime results will be examined, starting with a comparison between the C++, Numba, CuPy and CUDA implementations. After that, the focus will be only on the CUDA implementation and how the different changes and optimization approaches from the preceding section affect the runtime of the tomography application.

6.1 Benchmark Configuration and Hardware Used

The benchmark script is the same as the one explained in Section 4.2. In the first step, the parameters are set and a bunch distribution is created and tracked with BLonD. From this, bunch profiles can be taken that are equivalent to measurements. The creation of the bunch profiles in this way is deterministic, and the phase space distribution is perfectly known. These profiles are used as an input for the tomography application, along with the necessary input parameters. For the baseline measurement, as for the CPU version in Section 4.2, the number of bins is set to 200, the number of profiles to 500 and the number of turns between profiles to nine. Therefore, the particles will be tracked for 4491 turns. The reconstruction was run for 20 iterations, additionally to the initial back projection and the final projection. Before the tomography algorithm is started, it is chosen if single precision or double precision floats are used, as well as which implementation (C++, Numba, CuPy, CUDA). In the baseline case, double precision will be used to match the CPU version. After that, the tomography is executed.

The GPU workload is executed asynchronously with respect to CPU workloads. Therefore, the timing package has no knowledge about the runtime of the functions executed on the GPU and cannot be used to measure the runtime. Instead, to measure GPU runtime in a simple way in Python, CuPy is used. Since CuPy has access to the CUDA API, it can create two events and record them, yielding the elapsed time [40]. In the same way, GPU events can also be used to measure the runtime for CPU functions. To get a more reliable view on the measurement, the procedure is executed three times, yielding a mean runtime and a standard deviation for every function.

It is not possible to calculate the runtimes for the functions called inside the C++ code, therefore only the total runtime of the `Kick and Drift` and `Reconstruct` procedures

are given. Since the Numba and the GPU implementations are able to measure the runtime for each function, the runtimes and standard deviations have to be aggregated to compare the performance of the whole procedures. Therefore, if not calculated directly, the runtime R of a function is calculated by summing the runtimes of the subfunctions.

For the standard deviation σ, the standard deviations $\sigma_1, \sigma_2, \ldots, \sigma_n$ of the subfunctions are summed in quadrature, as shown in the following equation:

$$\sigma = \sqrt{\sum_{i=0}^{F} \sigma_i^2} \tag{16}$$

As with the benchmark of the CPU implementation in Section 4.2, each benchmark will consist of multiple executions of the tomography workflow, including the creation of the bunch distribution with BLonD. The initial CPU benchmark, which was created before, was executed for three times, the Numba and GPU benchmarks for ten times. Using this approach, the mean runtimes will be taken as a reference together with the standard deviation given from the ten or three executions. Only the runtimes of the tomography workflow are considered here. The creation of the bunch distribution with BLonD is not included in the runtimes.

The benchmark script is executed within the HPC cluster hosted at CERN. For the comparison among the GPUs, three different models are used: NVIDIA Tesla T4, NVIDIA Tesla V100 and NVIDIA A100 [39, 28, 41]. In this chapter, the GPU model names will be referred to as T4, V100 and A100. Due to the heterogeneous nature of the cluster, the nodes use different setups depending on the benchmarks. The nodes executing the CPU implementations (C++ and Numba) use an AMD EPYC 7302 CPU [37] with 8 cores and 16 threads and a CentOS7 operating system. GPU Nodes with a V100 or A100 GPU use an Alma Linux 9 operating system and an Intel Xeon Silver 4216, Cascade Lake [42] processor with 16 cores. Additionally, there are also machines with an Intel Xeon Silver 4114, Skylake [43] processor with 10 cores available, which are often equipped in nodes with T4 GPUs. Since it cannot be chosen which CPU of these two is selected, the runtimes for functions on the CPU may deviate when using the CUDA or CuPy implementation. For machines with an A100 GPU, an AMD EPYC 7313 CPU with 16 threads [44] is used within an Alma Linux 9 operating system.

Table 9 gives the specifications of the GPUs used [39, 28, 41]. The V100 and the A100 are high-end GPUs and outperform the T4 in many aspects, such as memory and

Model	NVIDIA Tesla T4	NVIDIA Tesla V100	NVIDIA A100
Generation	Turing (2018)	Volta (2017)	Ampere (2020)
Memory Size	16 GB GDDR6	32 GB HBM2	40 GB HBM2
Memory Bandwidth	320 GB/s	900 GB/s	1555 GB/s
SMs	40	80	108
FP32 Cores per SM	64	64	64
FP32 Cores per GPU	2560	5120	6912
FP64 Cores per SM	2	32	32
FP64 Cores per GPU	80	2560	3456
Peak FP32 TFLOPS	8.1	15.7	19.5
Peak FP64 TFLOPS	0.25 (estimated)	7.8	9.7

Table 9: Specifications of the GPUs [39, 28, 41] used for benchmarking.

computational capabilities. Considering the performance for floating-point operations, the NVIDIA A100 has the highest performance with $19.5 \cdot 10^{12}$ FLOPS (floating-point operations per second), respectively 19.5 TFLOPS, while the Tesla T4 comes to slightly more than 40% of the A100 peak value for single precision numbers. While the peak double precision performance for both the V100 and the A100 is approximately half of the single precision value, it is around 32 times smaller for the T4. The documentation does not specify the peak FP64 TFLOPS of the Tesla T4, however it states that the FP64 TFLOP rate is 1/32nd the TFLOP rate of FP32 operations for a similar GPU from which the T4 is derived [39]. There are only two FP64 units to ensure that programs relying on double precision floating-point numbers operate correctly on the T4. This implies that computational-heavy functions like `Kick and Drift` perform substantially better on a V100 or A100, especially if double precision floats are used. Furthermore, the number of SMs is larger on the two high-end GPUs, implying that more concurrent executions are possible. Although memory has not been a strong limitation, it allows reconstructions with a higher number of profiles, bins and particles, if needed.

6.2 Comparison of the Different Approaches

In the first comparison, a benchmark will be created to compare the four different implementations: C++, Numba, CuPy and CUDA. In all cases, double precision floating-point operations were used. The GPU model shown for CuPy and CUDA is the Tesla T4, because it was the most accessible. The next section will give a more

Function	Runtime in seconds			
	C++	Numba	CuPy (T4)	CUDA (T4)
`Values at turns`	0.079 ± 0.005	0.085 ± 0.013	0.066 ± 0.001	0.08 ± 0.004
`Homogeneous Distribution`	0.025 ± 0.0003	0.028 ± 0.003	0.029 ± 0.001	0.033 ± 0.01
`Kick and Drift`	3.93 ± 0.052	10.89 ± 4.99 CT: 3.589	1.22 ± 0.32	0.578 ± 0.011
`Coordinate conversion`	1.58 ± 0.278	1.64 ± 0.13	0.005 ± 0.012	0.005 ± 0.014
`Filter lost pts & transpose`	0.826 ± 0.037	1.27 ± 0.075	0.053 ± 0.009	0.053 ± 0.006
`Create Tomo object`	0.993 ± 0.034	1.86 ± 0.09	0.255 ± 0.02	0.279 ± 0.106
`Tomography reconstruction`	3.91 ± 0.037	5.93 ± 0.51 CT: 4.25	2.556 ± 0.017	0.265 ± 0.023
`Create phase space`	$0{,}038 \pm 0.029$	0.097 ± 0.096	< 0.001	0.002 ± 0.003
Total	11.38 ± 0.29	29.639 ± 5	4.6 ± 0.44	1.295 ± 0.11

Table 10: Benchmark of the different implementations for the higher level functions of the workflow, using double precision floating-point operations.

detailed view on the performance of different GPU models.

The measurements for `Kick and Drift` and `Tomography Reconstruction` for Numba, CuPy and CUDA are aggregated based on the runtime of the subfunctions and the initialization of the arrays and copying to the GPU, if applicable. The runtimes of the benchmark are shown in Table 10, ordered by the first execution of the function. In the context of the Numba implementation, *CT* refers to the compilation time of the subfunctions of `Kick and Drift` and `Tomography reconstruction`. Between the C++ and Numba, the values should be very similar, except for the two functions in which the implementations differ. In this context, the substantial difference in the runtime of the `Create Tomo object` function is surprising, since the implementation and the configuration for the execution in the cluster is the same for both. However, the measurements have been taken two months apart, there may be other aspects influencing the runtime, such as operating system changes in the nodes of the cluster.

The table clearly shows no improvement from using Numba. In fact, the `Kick and Drift` function takes almost three times as long and the reconstruction takes around 50% longer, without including the compilation time. Therefore, even when the functions are executed multiple times to compensate for just-in-time compilation, the initial C++ implementation outperforms Numba. Consequently, the conclusion can be made that Numba does not provide any benefit for runtime optimization of the longitudinal tomography.

Function	Runtime in seconds			
	C++	Numba	CuPy (T4)	CUDA (T4)
Initialize arrays	—	0.779 ± 0.03	0.415 ± 0.304	0.573 ± 0.648
`kick_up`	2.88	9.43 ± 3.88	0.97 ± 0.257	—
`drift_up`	0.12	0.68 ± 2.99	0.21 ± 0.192	—
Store coordinates	—	0.775 ± 0.96	0.05 ± 0.007	—
Whole Tracking	3.93 ± 0.052	11.664 ± 4.99	1.645 ± 0.44	1.15 ± 0.64
`Calc reciprocal particles`	0.179	0.22 ± 0.07	0.126 ± 0.005	< 0.001
`Create flat points`	0.08	0.234 ± 0.53	0.04 ± 0.001	0.008 ± 10^{-5}
`Back project`	0.542	1.01 ± 0.198	0.267 ± 0.002	0.09 ± 0.004
`Project`	3.136	4.04 ± 0.44	2.14 ± 0.014	0.147 ± 0.006
`Normalize`	< 0.001	0.041 ± 0.073	0.005 ± 0.001	0.005 ± 0.001*
`Calculate Difference`	0.002	0.02 ± 0.07	0.001 ± 10^{-4}	$0.001 \pm 2 \cdot 10^{-4}$
`Discrepancy`	0.003	0.016 ± 0.072	0.003 ± 0.003	0.005 ± 0.019*
`Compensate for pts amount`	< 0.001	0.01 ± 0.04	0.001 ± 0.006	$0.001 \pm 2 \cdot 10^{-4}$
Whole Reconstruction	3.91 ± 0.037	5.93 ± 0.51	2.556 ± 0.017	0.265 ± 0.023

Table 11: Runtime of the different implementations for the subfunctions of `Kick and Drift` and `Tomography reconstruction`, using double precision floating-point operations.

However, the table shows a positive trend for the two GPU versions. Using the T4 GPU, which is the least powerful of the available models, a speedup is not only achieved in the main bottlenecks `Kick and Drift` and `Tomography reconstruction`, but it is very notable in the steps between the tracking and the reconstruction. Therefore, even in the weakest setup, using the T4 with double precision floating-point operations leads to a notable performance improvement, both with the CuPy and with the CUDA implementation. Since the two implementations only differ for `Kick and Drift` and `Tomography reconstruction`, the other values are similar within a tolerance of measurement.

Considering `Kick and Drift`, the CUDA implementation reaches a speedup of almost seven, while it is almost 15 for `Tomography reconstruction`. Despite already reaching a speedup of seven for the former function, there is more potential when using other GPUs or single precision floating-point operations. This is due to the T4 not being designed for higher precision operations, as previously mentioned.

Table 11 shows a more detailed view on the subfunctions of `Kick and Drift` and `Tomography reconstruction`, with additional parts for the initialization of the arrays before and store the coordinates during the tracking. Since the initialization happens

always in Python, also for the C++ version, it is not considered for the aggregated runtime of the `Kick and Drift` function. The values used for the C++ are the ones from the benchmark with the Intel® VTune™ profiler, equivalent to the ones in Table 7. Therefore, no standard deviations are provided here, as well as a measurement for the initialization of the arrays. Additionally, the step of initializing the arrays has not been measured in the C++ benchmark. Since the procedure is equivalent to the one executed in Numba which has been added in a later stage, one can expect a comparable runtime to the Numba implementation. Furthermore, the values of the subfunctions of `Kick and Drift` cannot be measured since they have been executed in one kernel, as described in the optimization in Section 5.3.3. As a consequence, the whole runtime of the function is not aggregated but directly measured.

In all of the subfunctions, the same trend can be observed. The performance of the Numba implementation is worse than the C++ implementation, while the CuPy and CUDA measurements underline the effectiveness for most of the functions. The values with an asterisk * in the CUDA column refer to functions not executed directly with CUDA C++, but with CuPy, as discussed in Section 5.3.1. For the reconstruction, the `Project` function remains the bottleneck in both the CuPy and CUDA implementations. However, while the speedup for CuPy is only 1.5 compared to the C++ version, it is over 21 for the CUDA version, which shows the strong improvement by using atomic operations in CUDA compared to the CuPy implementation. Another point of interest is the `Back project` function which also shows a substantial higher speedup for the CUDA version than the CuPy version. This comes from the optimization described in 5.3.4. More details about the impact of this optimization will be elaborated in 6.6.

6.3 Differences Between Different GPU Models

While the preceding benchmarks gave an overall comparison over the four different implementations, the focus in the following sections will be on the CUDA implementation as it showed the best results for the runtime. Since the difference here is only the GPU, the function `Values at turns` and `Homogeneous Distribution` are not considered since they do not make use of the GPU. As in the previous benchmarks, only double precision floating-point operations are used.

Table 12 shows the results of the benchmark for the higher level routines. All functions are faster on the V100 and A100 than on the T4. The `Kick and Drift` function is particularly noteworthy, with speedups above 20 when going from T4 to V100/A100. Given the specifications of the GPUs, which were elaborated in Section 6.1, this result

Function	Runtime in seconds		
	T4	V100	A100
`Kick and Drift`	0.578 ± 0.011	0.021 ± 0.002	0.03 ± 10^{-5}
`Coordinate conversion`	0.005 ± 0.014	0.003 ± 0.006	0.001 ± 0.002
`Filter lost pts & transpose`	0.053 ± 0.006	0.015 ± 0.002	0.012 ± 0.001
`Create Tomo object`	0.279 ± 0.106	0.256 ± 0.107	0.285 ± 0.023
`Tomography reconstruction`	0.265 ± 0.023	0.102 ± 0.016	0.091 ± 0.004
`Create phase space`	0.002 ± 0.003	0.001 ± 0.003	< 0.001
Total	1.18 ± 0.11	0.398 ± 0.108	0.42 ± 0.023

Table 12: Benchmark of the CUDA implementation on the three GPUs T4, V100 and A100 for the higher level functions.

Function	Runtime in seconds		
	T4	V100	A100
`Calc reciprocal particles`	< 0.001	< 0.001	< 0.001
`Create flat points`	0.008 ± 10^{-5}	$0.002 \pm 8 \cdot 10^{-6}$	$0.002 \pm 2 \cdot 10^{-6}$
`Back project`	0.09 ± 0.004	$0.034 \pm 6 \cdot 10^{-5}$	0.026 ± 0.001
`Project`	0.147 ± 0.006	0.05 ± 10^{-4}	0.053 ± 0.001
`Normalize`	0.005 ± 0.001	$0.004 \pm 4 \cdot 10^{-4}$	$0.003 \pm 5 \cdot 10^{-4}$
`Calculate Difference`	$0.001 \pm 2 \cdot 10^{-4}$	0.001 ± 10^{-4}	0.001 ± 10^{-4}
`Discrepancy`	0.005 ± 0.019	0.003 ± 0.001	0.002 ± 0.002
`Compensate for pts amount`	$0.001 \pm 2 \cdot 10^{-4}$	0.0004 ± 10^{-4}	$0.0004 \pm 3 \cdot 10^{-4}$
Total	0.265 ± 0.023	0.102 ± 0.016	0.091 ± 0.004

Table 13: Benchmark of the CUDA implementation on the three GPUs T4, V100 and A100 for the subfunctions of the `Tomography reconstruction`.

is explainable due to the FP64 performance of the three GPUs, which is much higher for the V100 and the A100 than for the T4. However, contrary to the expectation, the runtime on the A100 was worse than the one on the V100. Since this function does not involve the CPU at all, a different setup due to the heterogeneous nature of the HPC cluster should not be the reason of this result.

Even though the `Tomography reconstruction` is not limited to the computational throughput, but to the memory latency and patterns, the differences in performance are substantial as well. The V100 and A100 performances show a speedup of 2.6 and

2.9 compared to the T4 performance, most likely from the larger caches and the faster memory accesses. This is also underlined by the measurements for the subfunctions of the reconstruction, which are shown in Table 13, where the significant differences appear for the two most complex functions: `Back project` and `Project`. Both functions are almost three times faster for the faster GPU models. This can be explained by a higher level of parallelism due to the presence of more streaming multiprocessors in the corresponding GPU models, as well as a faster memory access.

One interesting aspect of this benchmark, which can be seen from Table 12, is how the bottleneck shifted when changing from the T4 to the V100 or A100 GPU. While `Kick and Drift` takes by far the most runtime for the T4, it does not impose a big constraint on the V100 or A100. Additionally, the `Tomography reconstruction` also made improvements and needs around 100 milliseconds. Therefore, the bottleneck for the higher end GPUs lies on the `Create Tomo object` procedure, which contains several steps: Putting the arrays on the GPU, initializing arrays for the results, as well as asserting that the arrays are suitable to use.

6.4 Floating-Point Precision

The next aspect, which especially for the tracking can have a substantial impact, is the use of single precision floating-point operations. The option to use single or double precision was enabled in both the GPU and CPU implementations. The C++ implementation was modified with templates to achieve this, as briefly discussed in Section 5.3.2. Therefore, the difference for both the GPU and the CPU version can be observed.

The runtimes for the `Kick and Drift` function for both double and single precision are shown in Figure 25 for all GPU models and for the C++ implementation. It can be seen that using single precision improves the runtime in all cases, but in different orders of magnitude. While the single precision approach achieves a speedup of 20% on CPU, it is substantially higher for the GPU. The V100 and the A100 achieve a speedup of more than four, while the speedup for the T4 stands out with more than 20. This can be explained again with the specifications of the different GPU models. The T4 contains a small number of FP64 cores per SM compared to the number of FP32 cores per SM, therefore the performance difference is particularly strong here. The other two GPUs have more FP64 cores and are not particularly focused on low precision operations, therefore the performance difference is not as crucial as for the T4. Furthermore, considering that the peak FP32 performance is around the double of

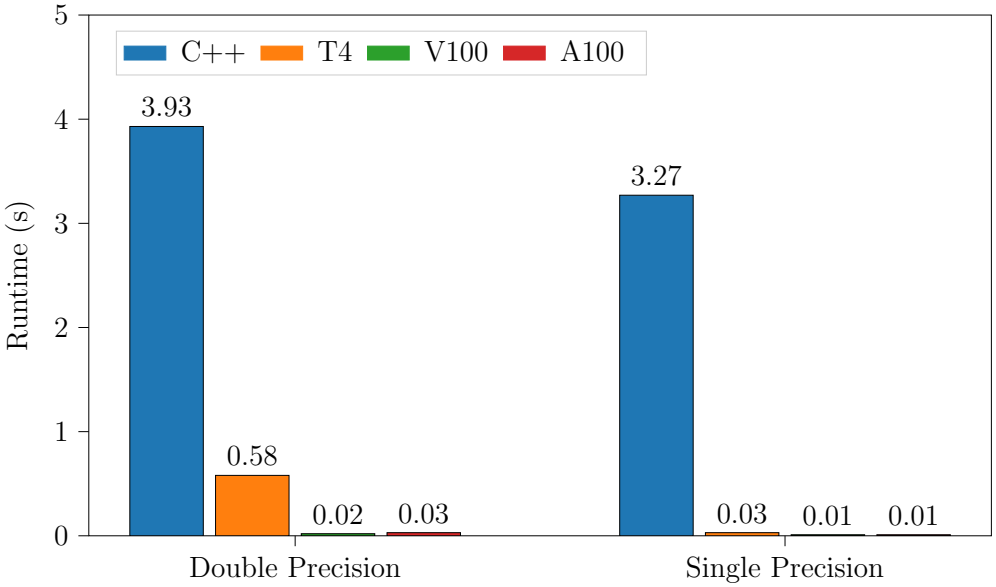


Figure 25: Runtime of `Kick and Drift` under different precisions and GPU models, as well as for the C++ implementation.

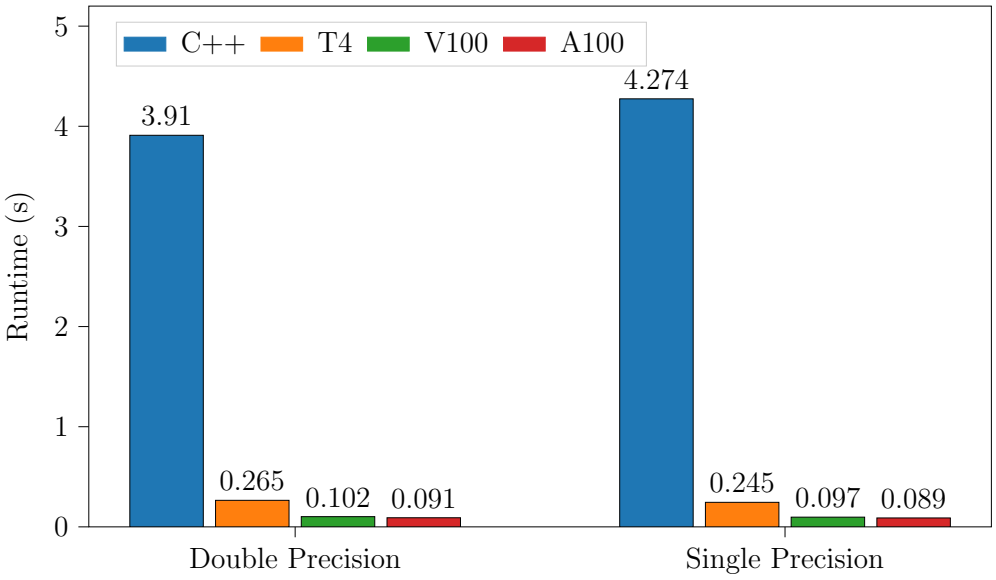


Figure 26: Runtime of `Tomography reconstruction` under different precisions and GPU models, as well as for the C++ implementation.

the FP64 performance for the V100 and A100, it is surprising that the speedup here is more than 4 when using single precision operations. One influence could be that the sine calculation is implemented differently for double and float precisions in the CUDA library. Since this is the most intensive computational calculation in the `Kick and Drift` function, a change here can have a strong impact on the overall runtime of the kernel.

The runtimes for the `Tomography reconstruction` procedure for the same setup is

Function	Precision	Runtime in seconds		
		T4	V100	A100
`Back project`	double	0.09 ± 0.004	$0.034 \pm 6 \cdot 10^{-5}$	0.026 ± 0.001
	single	0.07 ± 0.002	$0.028 \pm 3 \cdot 10^{-4}$	0.018 ± 0.001
`Project`	double	0.147 ± 0.006	0.05 ± 10^{-4}	0.053 ± 0.001
	single	0.147 ± 0.005	$0.05 \pm 8 \cdot 10^{-4}$	0.056 ± 0.003

Table 14: Runtime of `Back project` and `Project` under different precisions and GPUs.

shown in 26. One unexpected observation is that the runtime for the C++ version is worse when using single precision operations. While for the tracking the computational intensity of the operations is crucial for the runtime, it is not as important for the reconstruction. However, one would expect a slightly better performance for single precision operations. For the GPU implementations, the difference is barely visible, showing that computational intensity is not the main bottleneck for the reconstruction. Although not shown in the figures, the runtimes for `Project` are almost unchanged with the precision. This is due to the problem of multiple threads accessing the same memory chunk. For the `Back project` function, which uses the scan operation, an improvement in the runtime can be seen when using single precision, which can explain the slight improvements in Figure 26. The runtimes for the two functions are given in Table 14 for each configuration, which shows a speedup of 20% to 40% from double to single precision, subject to tolerance from the uncertainty of measurements.

Based on the benchmark, using single precision floating-point operations does not come with a great benefit for the C++ implementation. When using a T4 or a GPU with a small number of FP64 processing cores, using single precision can result in a substantially better performance. However, if a V100 or A100 is available, the change is minor. Therefore, if the loss of accuracy when using single precision floats is not an issue, and other GPU models are not available, using the T4 also yields a similar performance as the other models.

6.5 Kernel Workload and Invocation Overhead

This section focuses only on the `Kick and Drift` function and on the different approaches of computing it in the GPU, as described in Section 5.3.3. To recall, three approaches were presented. The first and naive approach is to define two GPU kernels for `Kick` and for `Drift`, one for the forward and one for the backward, direction. The

API Metrics							
Metric	GPU	Separated		Fused		Whole loop	
		Calls	Runtime	Calls	Runtime	Calls	Runtime
`cuLaunchKernel`	T4	102,540	1.8 s	57,630	1.065 s	2,750	0.061 s
	A100	102,540	1.369 s	57,630	0.747 s	2,750	0.039 s
Kernel executions							
Kernel	GPU	Separated		Fused		Whole loop	
		Calls	Runtime	Calls	Runtime	Calls	Runtime
`Kick`/`Drift` Kernels	T4	89,820	10.96 s	44,910	7.763 s	10	5.781 s
	A100	89,820	0.895 s	44,910	0.503 s	10	0.288 s
`cupy_copy_f64_f64`	T4	10,020	173 ms	10,020	154 ms	40	0.66 ms
	A100	10,020	81 ms	10,020	73.2 ms	40	0.34 ms

Table 15: Selected metrics for the three approaches for the A100 and T4 GPU from NVIDIA Nsight Systems.

workflow, including the loop and the store operations, are written in Python and one kernel is invoked for each energy kick, and one kernel for each phase drift. The second approach is to fuse both operations together to one kernel, effectively halving the number of kernel invocations for the routine, to reduce the overhead and increase the workload per kernel invocation. In the third approach, the whole loop and storing the coordinates are transferred to a single kernel. Therefore, only one kernel is called for the entire `Kick and Drift` routine, effectively reducing invocation overheads.

To analyze the impact of the different invocations, a new benchmark was executed with NVIDIA Nsight Systems on the three GPUs, using double precision floating-point operations. The aggregated results are shown in Table 15 and refer to the execution of the whole benchmark, repeating the workflow ten times. First, the metric `cuLaunchKernel` refers to the number and duration of kernel invocations of the application. This is not linked to the `Kick and Drift` routine only, but to the whole application. The three approaches show substantial differences here, with an overhead of 1.8 s in the naive approach for the T4, which improves to 61 ms in the approach with the whole loop inside the kernel. For the A100, the same trend can be observed, reaching from 1.369 s to 39 ms. If using a large number of turns, the first two approaches are not feasible due to the number of kernel invocations scaling with it, causing overhead.

The impact is also visible in the kernel runtimes which in the table were aggregated for the first approach, adding the calls and runtimes for the separate `Kick` and `Drift` kernels. The speedup from the separated to the fused approach is around 1.41 for the T4 and 1.78 for the A100. For the optimized approach, it gets to a speedup of

1.9 and 3.1 respectively, already including the store operations inside the kernel. The `cupy_copy_f64_f64` is the kernel executed when copying the coordinate arrays, which applies mostly to the first two approaches. In the third approach, only the initial copy is done in CuPy, therefore the number of calls is low, but not zero.

From these metrics, it can be concluded that the approach with the loop inside the kernel leads to the best performance. To get a closer look to the metrics during the kernel execution, the benchmark was also executed using NVIDIA Nsight Compute, to gather metrics explicitly for the `Kick`/`Drift` kernels. However, this tool was not used in the same way as the other benchmarks have been done. An interactive node, which can be shared with other users, with a T4 was used to take the samples. Here, the application is executed and the second kernel execution is profiled while the first one is skipped to avoid potential side effects from executing a kernel for the first time.

Metric	`Drift` (sep.)	`Kick` (sep.)	Fused	Whole loop
Occupancy	76.68%	92.08%	92.06%	98.45%
L1 hit rate	33.33%	65.25%	68.94%	93.39%
L2 hit rate	33.87%	50.53%	60.07%	99.91%

Table 16: Selected metrics for the three approaches for the T4 GPU from NVIDIA Nsight Compute.

A subset of the profiled metrics are shown in Table 16. Although, due to the differences of the kernels, the speedup cannot be determined from analyzing one kernel. However, a few metrics like the Occupancy and the Cache hit rates can be analyzed in order to compare the different approaches. For the separate approach, the `Drift` kernel reaches an occupancy of 76.68% and is substantially lower than the `Kick` kernel and the kernels from the other two approaches. This is likely due to the low complexity, since it consists of one multiplication and one addition per thread. Due to possible warp scheduling overheads, the occupancy can be lower. The separate `Kick` kernel reaches over 92%, as well as the kernel that fuses both operations. Here, the operations are more complex as sines are computed and multiple arithmetic operations. Finally, for the approach with the entire loop inside the kernel, the occupancy is the highest with 98.45%, almost reaching the theoretical occupancy.

In terms of the both the L1 and the L2 cache hit rates, the `Drift` also shows the lowest rates with about over 33% for both caches. This increases for the `Kick` kernel and the fused kernel. The last approach performs the best with over 93% L1 hit rate and more than 99.9% L2 hit rate. Having a low L2 hit rate implies that an access to

the device memory has to be performed, which has a higher latency than the caches. Therefore, the separate and the fused solution spend relatively more time on memory load operations. In that sense, the single kernel approach has a very low number of memory transactions that go to the device memory.

6.6 Atomic Operations and Reductions

The last section shows the results for a benchmark which was conducted for the `Back project` kernel to compare the impact of the parallel scan on the performance compared to atomic operations. This benchmark was conducted the same way as the initial ones, by executing the application ten times and taking the mean runtimes. The first one is the simple implementation of the kernel with atomic operations due to the collisions happening when accessing the same weights array. The disadvantages of this approach have been discussed in 5.3.4, together with the second approach, which uses a parallel scan to sum the values together over multiple threads to reduce the number of memory accesses.

For the second approach, the block size is set to a smaller value than the maximum and performs the projection for one weight. Therefore, the grid contains blocks equal to the number of particles instead of fewer blocks as in the naive implementation with a maximum block size. As well as the block size, the number of elements per thread to sum in the register reduction step is also relevant. A loop ensures that all the elements are processed. However, a smaller number of elements per thread leads to multiple executions of the reduction algorithm.

Type	Block size	Values per thread	Runtime in ms		
			T4	V100	A100
Atomic Add	1024	—	998.3 ± 12.08	420.6 ± 0.72	366.5 ± 0.59
Reduction	32	8	90.8 ± 3.11	40 ± 0.25	26.2 ± 0.68
	32	16	89.8 ± 2.85	39 ± 0.19	26 ± 0.87
	64	4	99.3 ± 4.43	39.6 ± 0.23	27.8 ± 1.31
	64	8	94.7 ± 3.28	39.4 ± 0.31	27.9 ± 1.32
	128	4	106.8 ± 3.51	41.4 ± 0.23	21.4 ± 1.14
	256	2	158.2 ± 2.61	42.3 ± 0.14	22.5 ± 0.36

Table 17: Runtime of `Back project` for different configurations on the three GPUs.

Table 17 shows the resulting runtimes for the three GPU models. The naive implementation with the atomic operations is clearly outperformed by the one with the parallel

scan. In most cases, the speedup is more than 10 when changing from atomic operations to the reduction, with slight deviations given by the selected parameters.

While the reduction approach is clearly more performant, the selection of the block size and values per thread only slightly changes the runtimes. However, in all of the three GPUs, the selection of blocks with 32 thread and 16 values per thread, leads to the best results, although very close to the ones with 64 blocks. In the previous benchmarks, this configuration was used as it provided the best results. What can be observed as well is that the performance gets worse the higher the block size is.

To conclude, implementing the parallel scan leads to a significant performance gain. Fine-tuning the parameters is secondary due to the low impact on performance. Since block size and the elements per thread have to be defined at compile time, these cannot be changed during the execution of the application.

6.7 Summary

While the performance study was not able to confirm an improvement from using Numba, it clearly demonstrated significant runtime improvements from using GPUs for the longitudinal tomography. In particular, for the two most computationally expensive parts, the tracking and the tomography, different levels of speedup were achieved. Compared to the baseline case, which was the C++ implementation with eight cores, Numba achieved substantially worse performances in all considered functions. CuPy, in its slowest case with the T4 and the double precision variant, showed clear improvements for all the functions, providing a perspective on further exploring the performance improvements when using GPUs.

Figure 27 shows the speedup for the `Kick and Drift` function for the CUDA implementation and the C++ implementation with eight cores. The speedups have been calculated with respect to the runtime of the single core execution of the C++ implementation. For the higher end GPU models, the speedups reach multiple hundreds for the double precision case, and even over 3600 in the single precision case for the V100, underlining the potential for massive parallelization which was harnessed here. The T4 also shows substantial improvements of 33.7 for the double precision case, and almost 700 for the single precision case, which, given its availability, makes it a valid choice to perform the tracking.

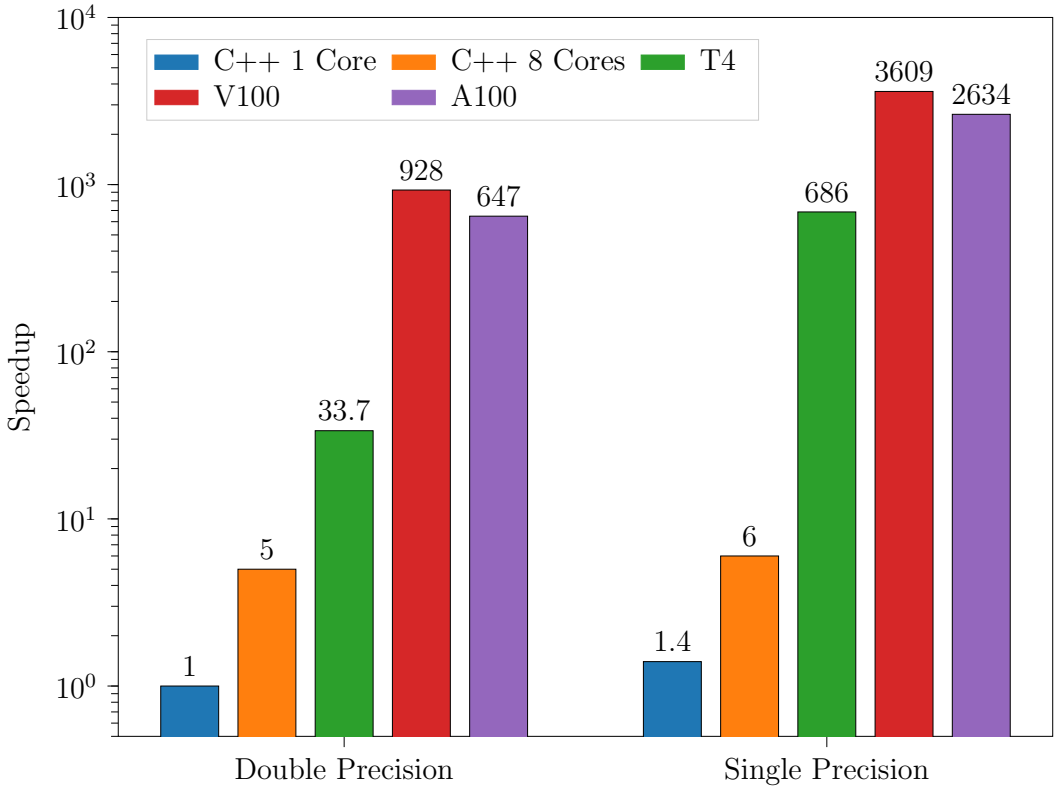


Figure 27: Speedup of `Kick and Drift` under different precisions and GPU models, normalized to the C++ single core double precision runtime.

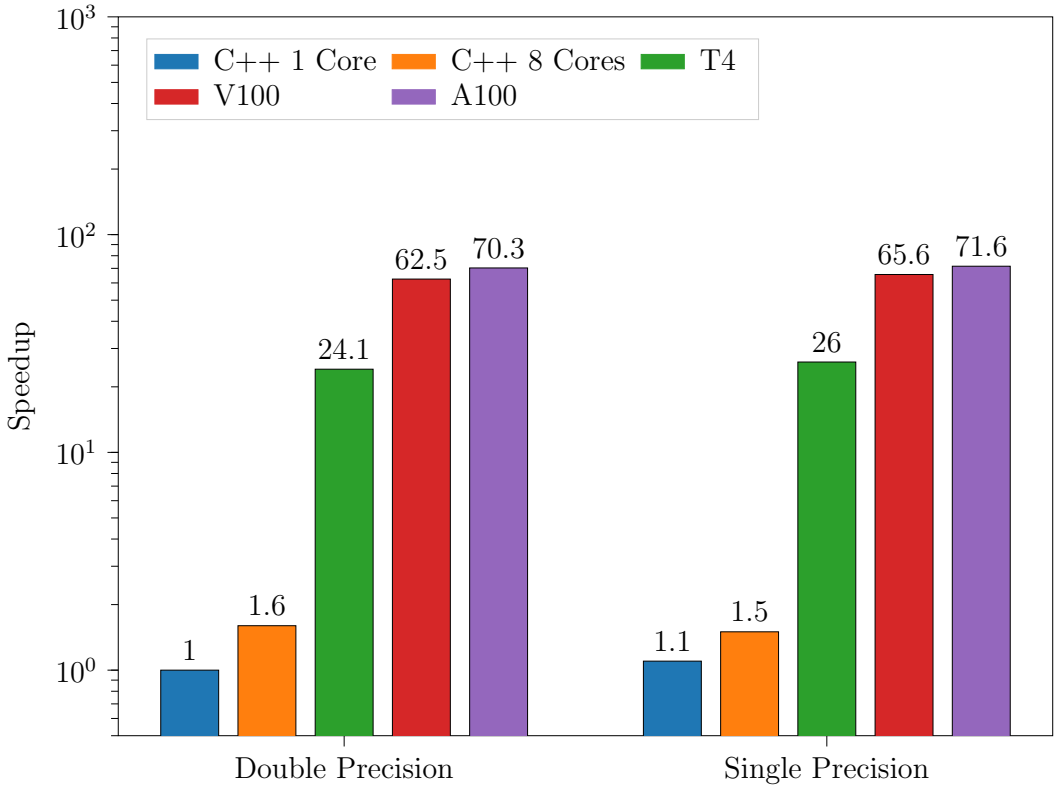


Figure 28: Speedup of `Tomography reconstruction` under different precisions and GPU models, normalized to the C++ single core double precision runtime.

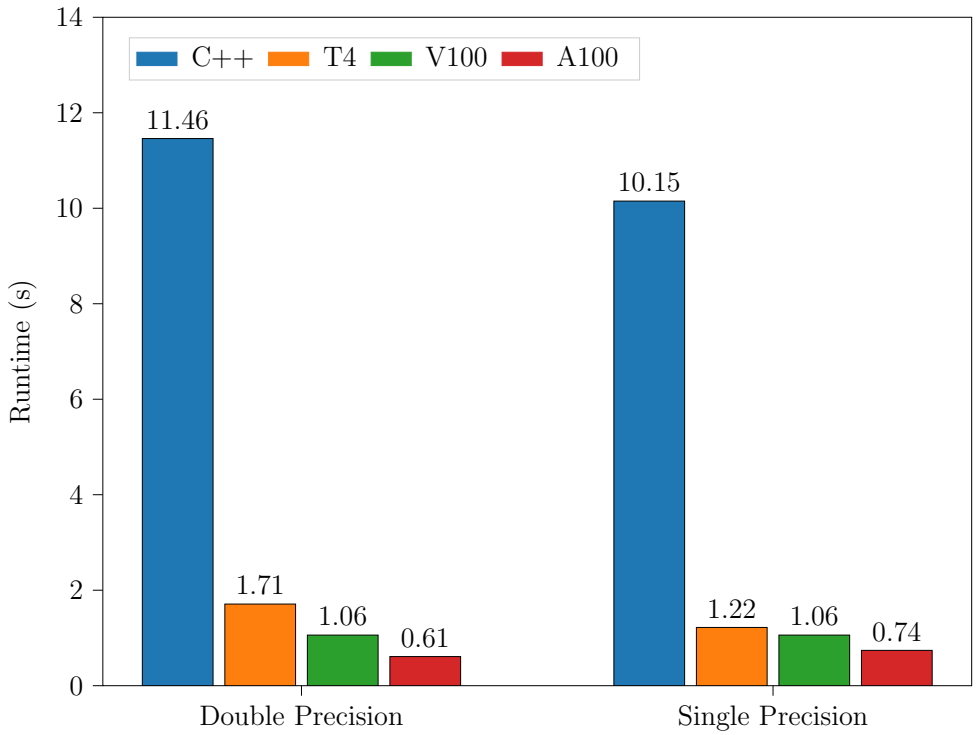


Figure 29: Runtime of the application for 8-core C++ and CUDA, excluding the setup of the GPU device.

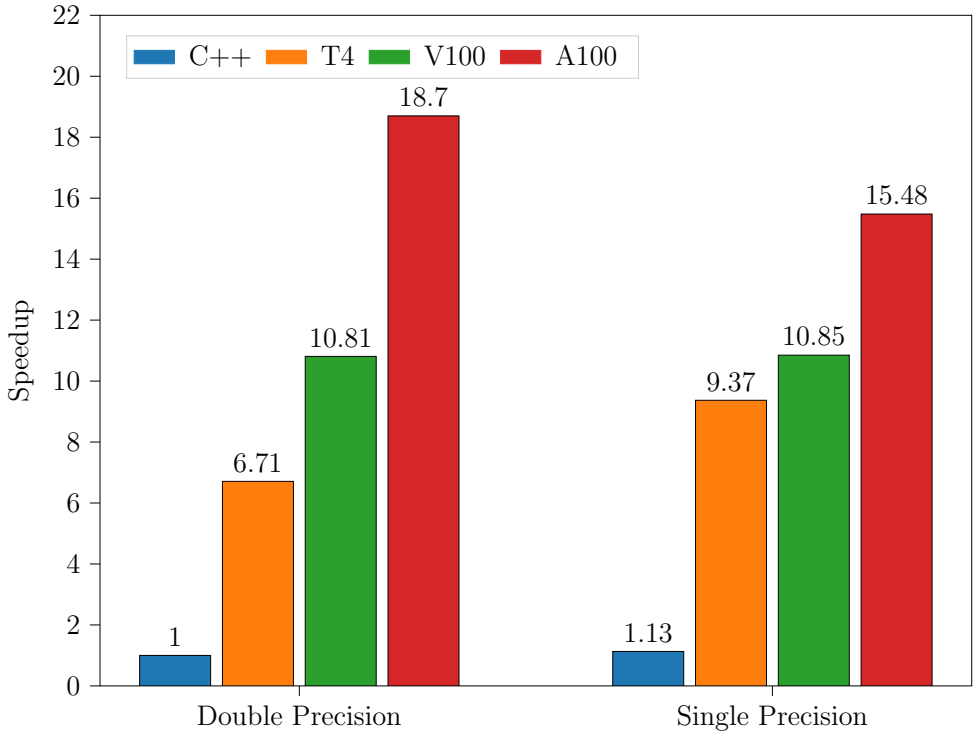


Figure 30: Speedup of the application for C++ and CUDA, excluding the setup of the GPU device, normalized to the 8-core double precision execution of the C++ version.

For the `Tomography reconstruction` function, the results are shown in Figure 28. Here, the speedups do not reach the same order of magnitude for the high-end GPUs as for the tracking. This is due to the nature of the reconstruction, where memory accesses play a more complex role than for the tracking. The T4 reaches a speedup of around 25 for both cases, while the V100 and A100 reach speedups between 60 and 70.

To finalize the view on the whole application, the runtime of the application was calculated based on the benchmarks made. For that, the runtimes of the parts where the bunch profiles were generated with BLonD were truncated. Additionally, the time to import packages and to setup the GPU device were also excluded. These parts are only executed once, regardless of how many tracking and reconstructions procedures are executed, therefore only the runtime without these factors is considered.

Figure 29 shows the runtime of the aggregated runtime of the application when executing the workflow once. For the setup with 200 bins and 500 profiles, the C++ runtime took almost 11.5 ± 0.29 seconds from processing the inputs to calculating the reconstructed phase space for double precision floating-point values. The GPU versions all execute the workflow below two seconds, ranging from 1.71 ± 0.66 s for the T4 to 0.61 ± 0.31 s for the A100. For the single precision versions, the runtimes for the GPU range from 1.22 ± 0.16 s to 0.74 ± 0.1 s, paradoxically implying that they run slower for the high-end GPU models. This is likely to be due to uncertainties of the measurements, because the double and the single precision benchmarks were taken on different machines, which can be equipped with different CPUs. As shown at the start of this chapter, the GPU functions performed significantly better for single precision workloads. In this particular benchmark, the `Values at turns` function had a substantially longer runtime for the single precision cases, which is purely a CPU function. Consequently, due to the uncertainties from the benchmarks, the runtimes can deviate in different setups, especially with respect to the CPU functions.

Besides benchmark uncertainties, which especially affect the runtime of CPU functions here, the runtime differences in GPUs are more systematic. The most important difference from the benchmarks is the difference in FP64/FP32 performance between the three GPUs, which affects the runtime of the tracking in particular. Another important difference is the number of streaming multiprocessors, which are different for the three GPUs and which determines how many threads can be executed concurrently. This is also a factor which influences the FP64/FP32 performance. Another point are the memory latencies and the cache sizes, which can explain runtime differences in the reconstruction.

While the runtimes have to be considered within a certain tolerance, the conclusion that the GPU runtimes are considerably smaller can still be made. Besides the absolute runtime, the speedup normalized to the 8-core execution of the C++ version, is shown in Figure 30. The functions that were optimized with CuPy and CUDA have a strong effect on the overall speedup. For the double precision case, the speedup is between 6.7 and 18.7, depending on the GPU model, for the single precision case between 9.37 to 15.48, with respect to the double-precision variant of the C++ implementation. Finally, the speedup can be lower when using a less complex parameter set, and if the workflow is only executed once, due to overheads coming from importing packages and initializing the CUDA context, which was not considered for these examples.

7 Conclusion

Longitudinal tomography is widely used in beam diagnostics of synchrotrons, providing insights about the beam by reconstructing the phase space from measured bunch profiles and machine parameters. In its first version, it was written in Fortran95, designed for optimal speed and memory usage. Later, to simplify the structure of the code and to facilitate adding new features, it was translated and refactored into a Python library [13]. The computationally expensive parts were written in C++ to maintain suitable runtimes. The desire for a better performance is still relevant, which resulted in the exploration of GPU accelerated methods to improve the longitudinal tomography further. In this context, different approaches were implemented and compared to the current application to evaluate the performance speedups.

The new version of the tomography was implemented iteratively, starting with implementing a Python-based version of the C++ parts to explore potential performance improvements with Numba, without using the GPU. The approach did not give any performance benefits. In fact, the performance of all the functions was notably worse compared to the current implementation. Considering that the compilation happens during the runtime as well, it is unlikely that there will be a case where Numba would be useful for the longitudinal tomography. Therefore, this approach was not pursued any further.

Starting the usage of GPUs by implementing a CuPy version led to substantial runtime improvements, not only for the computationally intensive functions, but also by incorporating parts of the Python workflow to handle arrays in the GPU. This reduces the data transfer between CPU and GPU and improves the runtime for these functions as well. Even though CuPy already showed performance speedups, an approach with CUDA C++ kernels for the computationally intensive parts was analyzed, which allows a lower level implementation and more control of multithreading. An example of this is the parallel scan, which was used for the back projection to minimize the memory accesses, which were subject to race conditions. The CUDA approach led to a substantial improvement for the tracking and the reconstruction, which was even stronger when using single precision instead of double precision floating-point operations.

In the spirit of keeping both the CPU and the GPU implementation, the interface was rewritten to allow the application to handle both arrays in the CPU and in the GPU. The user can choose, by calling a configuration function, if the CPU or the GPU implementation should be used, and if double or single precision floating-point operations should be used. This allows a transparent and simple interface for the user,

who is not necessarily familiar with the implementation. At the same time, it makes the application more maintainable by separating the different implementations and having a configuration class which takes care of the mapping for the functions and the precision for the floating-point numbers used.

7.1 Performance Improvements and Current Bottlenecks

The GPU implementation, which was developed during this work, showed significant runtime improvements for the longitudinal tomography. In the aspect of the whole workflow, speedups between 6 and 19 were reached, depending on the GPU model used. For the benchmarks, the models NVIDIA Tesla T4, NVIDIA Tesla V100 and NVIDIA A100 were tested, each with different levels of performance, especially for lower precision workloads. Furthermore, the benchmarks were conducted using an input of 500 measured profiles and 200 bins for the discretization.

While the T4 is the most accessible GPU from the three analyzed ones, it reached speedups between six and ten and ran the whole application in under two seconds, from initially over 10 seconds in the C++ version. Since it is not designed for higher precision workloads, it does not reach the performance speedups which the other GPU models reach. However, for lower precision workloads, the performance is comparable, making it a valid choice due to its higher availability. The V100 and the A100, as the higher end GPU models, reach faster speedups for both double and single precision cases. The runtime difference between double and single precision is not highly significant anymore for these models. Especially the A100 reaches very low runtimes and can perform the whole application workflow in less than 750 milliseconds.

Depending on which GPU model and which precision is considered, different bottlenecks are of relevance. The creation of the tomography object has a similarly high runtime throughout all the three GPU models, and therefore presents a bottleneck which should be analyzed further. This is due to the validation checks performed when creating the object. For the double precision case when using the T4, the tracking is the bottleneck, due to the low computational throughput for double precision floating-point operations from the GPU. Consequently, when high performance is required while keeping the same standard of accuracy, a GPU with a high double precision throughput, like the V100 or the A100, should be used. However, if a slight loss of accuracy is tolerable, using the single precision variant is a very relevant option and combines well with the T4 due to its high single precision throughput.

7.2 Future Work

While this benchmark showed clear performance improvements with the new GPU implementation, many aspects have not been considered during its development. Further improvements on the GPU implementation can be analyzed. This can include loading the values from the measured profiles directly into the GPU, and calculating the machine and beam parameters for each turn on the GPU too, instead of the CPU. Implementing this could mitigate the overhead of memory transfers even further, and boost the runtime by using the GPU for more calculations. As mentioned in the previous section, another look at the bottleneck from the creation of the tomography object should be given, since it presents an overhead, independently of which GPU model is used.

Another point which has not been covered from the GPU benchmarks is the scaling for bins and profiles. Due to the heavy multithreading involved, it would be interesting to see if the GPU reaches similar speedups for very low settings, like using a low number of profiles or bins. Here, overheads of calling kernels and memory transfers can be more significant due to the lower computational work performed. For a higher number of profiles and bins, the performance is expected to scale in a similar complexity as the CPU. However, the available memory can be a serious bottleneck. The T4 has a memory size of 16 GB, which is used up quickly when increasing the number of profiles or bins, and causes the application to crash when the memory is full. For the V100 and the A100, which have 32 and 40 GB respectively, the limit is higher, but not unreachable due to the size of the arrays used, which partly scale quadratically with the number of bins.

Since this benchmark focused only on the improvements in performance, the accuracy and quality of the results were not thoroughly analyzed. Unit tests have been created to compare the accuracy of the precision variants of the GPU implementation to the CPU implementation, and they showed very similar values, with a deviation of at most $10^{-4}\%$. Therefore, the GPU implementation has a similar accuracy in results for the respective variant on the CPU, and it can be concluded that the runtime improvements do not come at a cost of reduced accuracy of the results. However, it has to be further analyzed how much impact the change from double to single precision floating-point numbers has, because it leads to a loss of accuracy due to the lower number of bits used for the representation. Since the accuracy of the reconstructed phase space depends on the use case as well, the performance-accuracy trade-off of the single precision variant could be confirmed in a further study.

Finally, another future topic will be longitudinal tomography for multiple bunches.

The current implementation only works for a single particle bunch in the synchrotron. However, it is common to have multiple bunches moving in the synchrotron at the same time. This increases the complexity for the tomographic reconstruction and the GPU implementation will have to be adjusted for this use case.

References

[1] CERN. *Annual Report 2022*. 2023. DOI: 10.17181/AnnualReport2022.

[2] H. Timko et al. 'Beam longitudinal dynamics simulation studies'. In: *Phys. Rev. Accel. Beams* 26 (11 Nov. 2023), p. 114602. DOI: 10.1103/PhysRevAccelBeams.26.114602.

[3] CERN. *Beams & RF Studies (BR)*. URL: https://sy-dep-rf.web.cern.ch/content/br (visited on 06/02/2024).

[4] CERN. *The CERN accelerator complex, layout in 2022*. URL: https://cds.cern.ch/images/CERN-GRAPHICS-2022-001-1 (visited on 06/02/2024).

[5] CERN. *Wall Current Monitors*. URL: https://psring.web.cern.ch/psring/misc/wcm.html (visited on 10/10/2023).

[6] C. H. Grindheim and S. Albright. *Longitudinal Phase Space Tomography Version 3*. Tech. rep. CERN-ACC-Note-2021-0004. 2021. URL: https://cds.cern.ch/record/2750116 (visited on 06/02/2024).

[7] S. Hancock. *A simple algorithm for longitudinal phase space tomography*. Tech. rep. Geneva: CERN, 1997. URL: http://cds.cern.ch/record/1174559.

[8] S. Hancock, S. R. Koscielniak and M. Lindroos. 'Longitudinal Phase Space Tomography with Space Charge; rev. version'. In: *Phys. Rev. Spec. Top. Accel. Beams* 3.12 (2000), p. 124202. DOI: 10.1103/PhysRevSTAB.3.124202.

[9] S.C.P. Albright et al. 'Recent Developments in Longitudinal Phase Space Tomography'. In: *Proc. IPAC'22* (Bangkok, Thailand). July 2022, pp. 347–350. ISBN: 978-3-95450-227-1. DOI: 10.18429/JACoW-IPAC2022-MOPOPT043.

[10] S. Albright. *A Proposal for an Automatic Longitudinal Tomography Demonstration Project in the Proton Synchrotron Booster*. Tech. rep. CERN-ACC-Note-2020-0014. 2020. URL: http://cds.cern.ch/record/2710497 (visited on 12/12/2023).

[11] D. Quartullo, S.C.P. Albright and H. Damerau. 'Frequency-Dependent RF Voltage Calibration Using Longitudinal Tomography in the CERN PSB'. In: *Proc. IPAC'22* (Bangkok, Thailand). July 2022, pp. 845–848. ISBN: 978-3-95450-227-1. DOI: 10.18429/JACoW-IPAC2022-TUPOST006.

[12] D. Quartullo et al. 'RF Voltage Calibration Using Phase Space Tomography in the CERN SPS'. In: *Proc. IPAC'22* (Bangkok, Thailand). July 2022, pp. 841–844. ISBN: 978-3-95450-227-1. DOI: 10.18429/JACoW-IPAC2022-TUPOST005.

[13] C. H. Grindheim. 'A Python Library for Longitudinal Beam Tomography'. Master thesis. HVL, Norway, 2020. URL: https://cds.cern.ch/record/2717822 (visited on 11/07/2023).

[14] F. Lauro. 'General purpose tools for longitudinal beam dynamics studies '. Presented 2023. Universita Degli Studi di Milano, 2023. URL: http://cds.cern.ch/record/2856324 (visited on 12/02/2024).

[15] F. Tecker. 'Longitudinal Beam Dynamics in Circular Accelerators'. In: 2020. arXiv: 2011.02932 [physics.acc-ph].

[16] S. Y. Lee. *Accelerator physics*. Nov. 2011, pp. 229–239. ISBN: 978-981-4374-94-1. DOI: 10.1142/8335.

[17] R. Gordon and G. Herman. 'Three-dimensional reconstruction from projections: A review of algorithms'. In: *International review of cytology* 38 (1974), pp. 111–151.

[18] O. Naumenko. 'Simulations and measurements to improve the longitudinal quality of fixed target beams at the CERN PS and SPS'. [Unpublished Master's thesis] (accessed on 23/10/2023). Gottfried Wilhelm Leibniz Universität Hannover, 2023.

[19] S. K. Lam, A. Pitrou and S. Seibert. 'Numba: A LLVM-Based Python JIT Compiler'. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162.

[20] A. Shajii et al. 'Codon: A Compiler for High-Performance Pythonic Applications and DSLs'. In: *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*. CC 2023. Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 191–202. ISBN: 9798400700880. DOI: 10.1145/3578360.3580275.

[21] C. Lattner and V. Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0769521029.

[22] C. R. Harris et al. 'Array programming with NumPy'. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[23] NVIDIA Corporation. *CUDA C++ Programming Guide*. Version 12.3. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 30/10/2023).

[24] M. J. Flynn. 'Some Computer Organizations and Their Effectiveness'. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[25] D. von Bruch. *Programming for GPUs*. Lecture in Thematic School of Computing 2023.

[26] G. M. Amdahl. 'Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities'. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560.

[27] NVIDIA Corporation. *CUB*. URL: https://nvlabs.github.io/cub (visited on 31/10/2023).

[28] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. 2018. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 31/10/2023).

[29] NVIDIA Corporation. *NVIDIA Nsight Systems*. URL: https://developer.nvidia.com/nsight-systems (visited on 27/12/2023).

[30] NVIDIA Corporation. *NVIDIA Nsight Compute*. URL: https://developer.nvidia.com/nsight-compute (visited on 27/12/2023).

[31] Ryosuke Okuta et al. 'CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations'. In: *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*. 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf (visited on 12/02/2024).

[32] Preferred Networks, Inc. and Preferred Infrastructure, Inc. *User-Defined Kernels*. CuPy Documentation. URL: https://docs.cupy.dev/en/stable/user_guide/kernel.html (visited on 03/11/2023).

[33] J. Wenzel. *pybind11*. URL: https://github.com/pybind/pybind11 (visited on 15/11/2023).

[34] L. Dagum and R. Menon. 'OpenMP: an industry standard API for shared-memory programming'. In: *IEEE Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[35] D. Piparo, V. Innocente and T. Hauth. 'Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions'. In: *J. Phys.: Conf. Ser.* 513 (2014). DOI: 10.1088/1742-6596/513/5/052027.

[36] Intel Corporation. *Intel® VTune™ Profiler User Guide*. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/vtune-profiler-user-guide.pdf (visited on 22/11/2023).

[37] Advanced Micro Devices, Inc. *AMD EPYC™ 7002 Series Processors: A New Standard for the Modern Data Center*. 2020. URL: https://www.amd.com/system/files/documents/AMD-EPYC-7002-Series-Datasheet.pdf (visited on 22/11/2023).

[38] H. Perović. *Function Timing and Optimization: Numba Implementations for the BLonD Code*. 2023. URL: https://cds.cern.ch/record/2871506 (visited on 01/12/2023).

[39] NVIDIA Corporation. *NVIDIA Turing GPU Architecture*. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 31/10/2023).

[40] Preferred Networks, Inc. and Preferred Infrastructure, Inc. *Performance Best Practices*. CuPy Documentation. URL: https://docs.cupy.dev/en/stable/user_guide/performance.html (visited on 15/01/2024).

[41] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (visited on 31/10/2023).

[42] Mohamed Arafa et al. 'Cascade Lake: Next Generation Intel Xeon Scalable Processor'. In: *IEEE Micro* 39.2 (2019), pp. 29–36. DOI: 10.1109/MM.2019.2899330.

[43] J. Doweck et al. 'Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake'. In: *IEEE Micro* 37.2 (2017), pp. 52–62. DOI: 10.1109/MM.2017.38.

[44] Advanced Micro Devices, Inc. *AMD EPYC™ 7003 Series Processors: High performance and Efficiency for Mainstream Computing Needs*. 2022. URL: https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf (visited on 16/01/2024).