ACDIV-2021-02
March 2021

Development of a Fast Parallel Tracking Code Based on the OpenCL Framework

Author: Manu Canals Codina

Supervisor: Michele Carla

Abstract

In this report the implementation and the details of a high performance GPU-based code for tracking ultrarelativistic particles in a storage ring is discussed.
The code was developed using mostly the Python programming language, only the computationally intensive part exploits the OpenCL framework [3]. The validity of the results has been tested against the well-established MAD-X/PTC code [2]. The code has been tested on the optics of the current ALBA ring and on a candidate for the future ALBA-II ring.
This report shows the implementation of the developed GPU-based tracking code, including a User Guide, as well as the results of the accuracy and benchmarking tests. A derivation of the theoretical fundamentals is also explained, following the scheme from Bryant et al. [1]

Accelerator Division
Alba Synchrotron Light Source
c/ de la Llum, 2-26
08290 Cerdanyola del Valles, Spain

ALBA Synchrotron
Accelerators Department

INTERNSHIP REPORT

Development of a Fast Parallel Tracking Code Based on the OpenCL Framework

Author
Manu Canals Codina

Supervisor
Michele Carla

Abstract

In this report the implementation and the details of a high performance GPU-based code for tracking ultrarelativistic particles in a storage ring is discussed.

The code was developed using mostly the Python programming language, only the computationally intensive part exploits the OpenCL framework [3]. The validity of the results has been tested against the well established MAD-X/PTC code [2]. The code has been tested on the optics of the current ALBA ring and on a candidate for the future ALBA-II ring.

This report shows the implementation of the developed GPU-based tracking code, including a User Guide, as well as the results of the accuracy and benchmarking tests. A derivation of the theoretical fundamentals is also explained, following the scheme from Bryant et al. [1].

April 15, 2021

Contents

1 Introduction

An accelerator or a storage ring, such as the ALBA synchrotron, can be broken down to a collection of magnets and regions of free space (drifts) where particles travel through, deflecting and bending their paths according to the magnetic fields they find along their trajectories. In order to attain proper functioning of such a complex structure, one needs, among other things, to reproduce the physical phenomena occurring throughout this collection of elements. Certain predictions and information are needed for designing and managing such a machine. As a concrete example, dynamical aperture (DA) calculations, toward which, this work is focused.

There are already many libraries, modules and programs capable of simulating a very wide spectra of accelerator scenarios. Such as Methodical Accelerator Design (MAD) and SixTrackLib. Despite the ready availability of these simulation codes, there are specific applications where an ad-hoc solution is better performing. Such is the case of this work, with the specific task in mind of obtaining fast DA calculations.

The strategy followed in this work to gain computational speed goes down the path of exploring graphics processing units (GPUs). To speed up further the calculations, several approximations were also introduced, the effect of which will be discussed. For the same purpose, 32 and 64 bit floating point representation of the data was investigated.

A complete discussion of storage ring beam dynamics can be found in [1]. Nevertheless, in Section 2 a summary of the fundamental concepts is provided for reference.

On Section 3, the first result produced by this work is presented: the implementation of the developed theory using a GPU. An overview on the type of programming used and the code itself is given, as well as a short practical user guide.

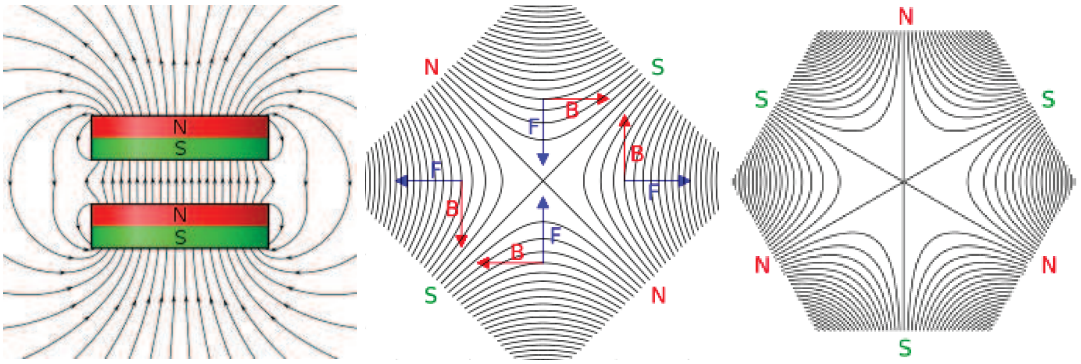
Finally in Section 4, the results of the comparison between the GPU code and MAD-X/PTC are provided together with some speed benchmarking of the GPU code.

2 Theory

A basic goal of this work is to simulate a particle's trajectory passing through a lattice, i.e. a sequence of elements (magnets and free space). This is actually no more than knowing how the particle passes through each individual element, and there are only 5 different kinds of them considered here:

- The drift (i.e. free space).
- The quadrupole, see Figure 1b.
- The sextupole, see Figure 1c.
- The sector bend (SBEND), see Figure 1a. A sector bend is a dipole with the exiting and entering faces perpendicular to the reference orbit, explained below. There are no focusing effects caused by the inclination of faces.
- The rectangular bend (RBEND), see Fig. 1a. A rectangular bend is a dipole with the exiting and entering faces parallel to each other. There will be some inclination with respect to the reference orbit, which will cause extra focusing. To account for this effect the RBEND is decomposed in an SBEND plus two wedge magnets.

The *reference orbit* is defined as the orbit such that a particle with a certain nominal momentum p_0 will move turn after turn on the same trajectory. It may also be called *circular orbit*.



(a) (b) (c)

Figure 1: Magnet fields lines and poles of (from left to right) a dipole, a quadrupole and a sextupole; in the plane transverse to the reference orbit.

In this section, the method for calculating how a particle *passes* through each element (the pass-method) is derived, motion with momentum deviation is considered and final results for each element are listed. Additionally, MAD-X/PTC syntax will be addressed, for such syntax has been adopted on the GPU's optics specifications.

2.1 Pass-method

Consider a Cartesian coordinate system (x, s, z) attached to a particle travelling onto the reference orbit with nominal momentum p_0. See Figure 2. A *mutable* variable representing either x or z will be used to ease the expressions. Namely, y. It is worth mentioning that the coordinate s is defined as

$$\frac{d}{dt} \equiv v\frac{d}{ds}, \tag{1}$$

where $v = p/m$ and p is the momentum of the particle to be studied.

Equation of Motion

In absence of non linear elements (that will be discussed later on) the motion of a particle with momentum p_0 through an accelerator element can be synthesized as (see [1])

$$\frac{d^2y}{ds^2} + K(s) \cdot y = 0. \tag{2}$$

The particle is considered to remain in the transverse plane (the $s = 0$ plane). The position dependent focusing parameter $K(s)$ has been introduced. The forms it takes, depending on the plane and element, are shown in Table 1. The s-dependency makes (2) valid for the whole ring, with K changing from element to element. Notice that the magnet field is then solely characterized with the *normalized gradient* k:

$$k \equiv \frac{e}{p_0}\frac{\partial B_z}{\partial x}\Big|_{x=0} = -\frac{1}{B_{z,0}r_0}\frac{\partial B_z}{\partial x}\Big|_{x=0}. \tag{3}$$

Solutions

Solving (2) will give the position and velocities of a particle after travelling through an specific element. Consider a particle entering a certain element at x_0, z_0 (generally y_0) with velocities x_0', z_0' (generally y_0')[1]. Its position and velocities when exiting the element – their *propagated*

[1]Notice that these velocities refer to the variable s and not t, with s defined in (1): $y' \equiv dy/ds$

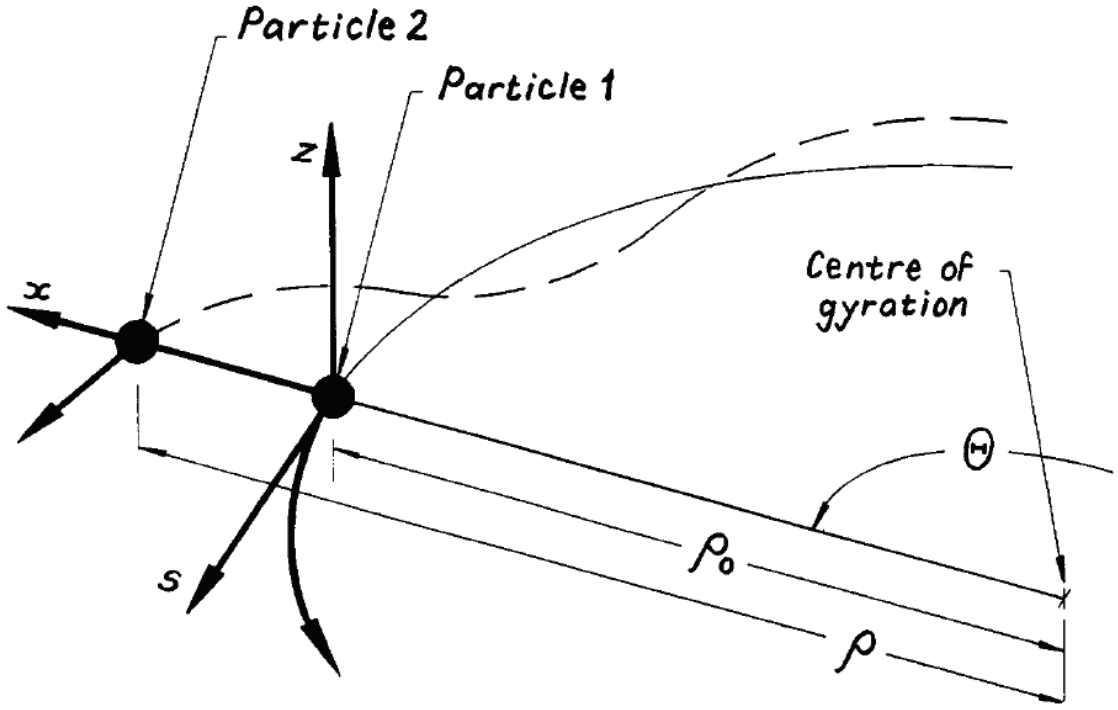


Figure 2: Coordinate system used to describe the motion of Particle 2 through the synchrotron. Particle 1 represents a particle travelling onto the reference orbit with nominal momentum p_0.

Element	Horizontal Plane	Vertical Plane
Drift	$K \to 0$	$K \to 0$
Quadrupole	$-k$	k
Pure Dipole	r_0^{-2}	$K \to 0$
Dipole with quadrupole component	$r_0^{-2} - k$	k

Table 1: Forms of K depending on the type of element, where k is defined as (3). "$\to$" means "take the limit". An SBEND is a single dipole (pure or with the extra component), and an RBEND is an SBEND with two *wedges* added. They will be further discussed later on.

coordinates – are

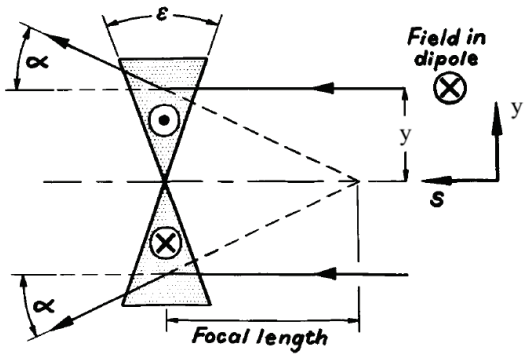
$$y = \cos(K^{\frac{1}{2}}L) \cdot y_0 + K^{-\frac{1}{2}} \sin(K^{\frac{1}{2}}L) \cdot y'_0$$
$$y' = -K^{\frac{1}{2}} \sin(K^{\frac{1}{2}}L) \cdot y_0 + \cos(K^{\frac{1}{2}}L) \cdot y'_0$$

(with K depending on the plane and kind of element, as shown on Table 1), which can be expressed in matrix form as

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} \cos(K^{\frac{1}{2}}L) & K^{-\frac{1}{2}} \sin(K^{\frac{1}{2}}L) \\ -K^{\frac{1}{2}} \sin(K^{\frac{1}{2}}L) & \cos(K^{\frac{1}{2}}L) \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}. \tag{4}$$

Notice that K may take negative values, in which case hyperbolic functions would appear. Additionally, a wedge's solution can be obtained from the above equation. A wedge with some aperture ϵ deflects a particle entering at y parallelly – i.e., with $y' = 0$ – a certain angle α given

by



$$\alpha \equiv y \cdot \frac{\tan(\epsilon)}{r_0},$$

where r_0 is the bending radius of the dipole it is added to. The resulting focusing effect produced by a wedge has a similar form to the thin lens approximation ($L \to 0$) of a quadrupole with focal length $-r_0/\tan(\epsilon)$. In matrix form,

$$\begin{pmatrix} 1 & 0 \\ \pm\frac{\tan(\epsilon)}{r_0} & 1 \end{pmatrix},$$

where $(+)$ is used for the horizontal plane and $(-)$ for the vertical plane – assuming the bending always happens on the horizontal plane – .

This propagation, matrix used to calculate the particle's coordinates after passing through one element is the so called *pass-method*. In order to consider multiple elements, the pass-methods relative to each element are applied in sequence, propagating the particle coordinate through the sequence of elements. This translates to a matrix multiplication when writing the pass-methods in matrix form.

With this composition rule, expression (4) and Table 1, a particle without momentum deviation could be tracked through the whole storage ring, if it wasn't for non linear elements: sextupoles.

Sextupoles

These elements, are solved using a thin lens approximation [4]. A thick sextupole is approximated with a certain amount of thin sextupoles interleaved with drifts. A thin sextupole can then be solved, and results in the following map:

$$\begin{aligned} x &= x_0, \qquad p_x = p_{x,0} - \frac{1}{2}k_2 L(x_0^2 - z_0^2), \\ z &= z_0, \qquad p_z = p_{z,0} + k_2 L x_0 z_0, \end{aligned} \tag{5}$$

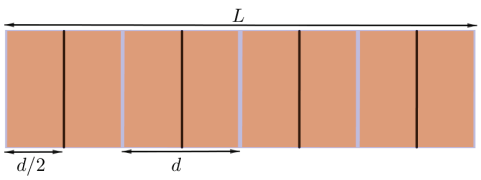
where k_2 is the *normalized sextupole gradient*, analogous to k, defined as[2]

$$k_2 \equiv -\frac{e}{p_0}\frac{\partial^2 B_z}{\partial x^2}\Big|_{x=0} = +\frac{1}{B_{z,0} r_0}\frac{\partial^2 B_z}{\partial x^2}\Big|_{x=0}. \tag{6}$$

Map (5) applies only the thin element kick. The full approximation works by combining these kicks with drifts. There are several methods to do so [5]. In this work only the so called Simple and Teapot method have been implemented to fully approximate a sextupole of length L and strength k_2 with n slices. As shown and explained in Figure 3, a thick sextupole is sliced into thin sextupoles (represented by black lines). Coloured areas represent the drifts introduced in between.

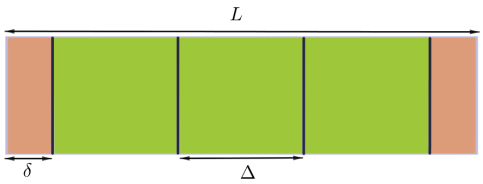
[2]Notice the sign!

SIMPLE



The thick sextupole is divided into n equal "boxes", each of length $d = L/n$. Each box is made up of: a drift of length $d/2$, followed by the sextupole kick – with the length parameter taking the value L/n and its strength k_2 –, and a closing drift of length $d/2$.

TEAPOT



There is a drift of length δ at the beginning and end of the approximated element. Then, the n slices are all equally spaced by drifts of lengths Δ. Each sextuple kick has a length value of L/n and strength k_2. The values of δ and Δ are given by

$$\delta \equiv L/\left[2(1+n)\right] \text{ and } \Delta \equiv L \cdot n/\left(n^2-1\right).$$

Figure 3: Notice that kicks take the strength value of the thick element, but not its length.

The same composition rule previously introduced for linear elements can be applied to sextupoles, although they can not be expressed in matrix form (thus, no matrix multiplication here).

Currently, every tool needed to track particles without momentum deviation has been presented. Expression (4) and Table 1 account for drifts, quadrupoles and dipoles (with wedges being a particular case), while (5) and Figure 3 account for sextupoles. Also, the composition rule of these pass-methods has been already explained so far. In the next section, momentum deviation is introduced, along with the few changes it comes with.

2.2 Momentum Deviation

The particle to be tracked can have now a different momentum than p_0: p. This is quantified with

$$\delta \equiv \Delta p \equiv \frac{p - p_0}{p_0}. \tag{7}$$

Definition of a General $K(s) : K'(s)$

The equation of motion with momentum deviation slightly differs from (2):

$$\frac{d^2y}{ds^2} + K'(s) \cdot y = \frac{\delta}{1+\delta}\frac{1}{r_0(s,y)} \quad \text{with} \quad K'(s) \equiv K(s)/(1+\delta), \tag{8}$$

where the bending r_0 may depend on the element and on the plane, and $K(s)$ is still given by Table 1. For the sextupole case, its corresponding k_2 is analogously changed, a $(1+\delta)$ factor must appear for similar reasons.

The right equivalence on (8) defines the general – with and without momentum deviation – position dependent focusing parameter, K'. This is the first difference that momentum deviation implies.

Solving the Equation of Motion: Extra Term

The extra term on (8) vanishes for straight elements. Therefore, their solutions still have the form of (4), although the above K' must be used. For dipoles, the new equation of motion must be solved. Since an RBEND is composed with two wedges and an SBEND, only the latter pass-method needs to be obtained. The SBEND pass-method results in

$$\begin{pmatrix} y \\ y' \\ \delta/(1+\delta) \end{pmatrix} = \begin{pmatrix} \cos(K'^{\frac{1}{2}}L) & K'^{-\frac{1}{2}} \cdot \sin(K'^{\frac{1}{2}}L) & \pm\mathbf{r_0^{-1}K'^{-1}}\left[\mathbf{1} - \cos(\mathbf{K'^{\frac{1}{2}}L})\right] \\ -K'^{\frac{1}{2}} \cdot \sin(K'^{\frac{1}{2}}L) & \cos(K'^{\frac{1}{2}}L) & \pm\mathbf{r_0^{-1}K'^{\frac{1}{2}}} \cdot \sin(\mathbf{K'^{\frac{1}{2}}L}) \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \\ \delta/(1+\delta) \end{pmatrix},$$

where r_0 is the bending on the plane that y refers to, and K' is defined in (8). Also in this case, negative values of K' requires to switch to hyperbolic functions. The upper sign (+) is used when the bending is in the horizontal plane, and (−) when the bending is in the vertical one. The appearing extra terms are highlighted on the pass-method above, and they make up the second difference on momentum deviation implementation.

Differences Between MAD-X/PTC and GPU

It's worth to note that MAD-X/PTC and the GPU code use a different momentum definition. The variable on MAD-X/PTC referring to the transverse momentum `py_MX` is defined as ([2], p. 241)

$$\texttt{py\_MX} \equiv \frac{p_y}{p_0} = \frac{m}{m_0} \cdot \frac{v_y}{v_0} \qquad \text{with the usual} \quad v_y \equiv \frac{dy}{dt},$$

where p_0 always refers to the reference orbit *without* momentum deviation (travelling at v_0 with relativistic mass m_0). The analogous variable in GPU is the velocity that has been used throughout this section, y'. It is defined – making use on its turn of the definition of s – as

$$\texttt{y'\_GPU} \equiv \frac{dy}{ds} \equiv \frac{1}{v}\frac{dy}{dt} = \frac{v_y}{v},$$

where in this case, v refers to the tangential velocity *with* momentum deviation. Thus, when there is no momentum deviation, the values of `py_MX` can be directly passed to `y'_GPU`, since $v = v_0$ and $m = m_0$. On the other hand, when there is momentum deviation,

$$p = p_0 \cdot (1+\delta) \quad \Longrightarrow \quad \texttt{py\_MX} \equiv \frac{p_y}{p_0} = \frac{p_y}{p}(1+\delta) = \frac{\cancel{m}v_y}{\cancel{m}v}(1+\delta) = \texttt{y'\_GPU} \cdot (1+\delta). \tag{9}$$

The input values used on both sides of the simulation comparison, on MAD-X/PTC and on GPU, don't refer to the same quantity! This change, (9), must be introduced to correctly compare both codes.

With this comparison remark, all differences that come with momentum deviation have been accounted for, namely:

1. $K'(s) \equiv K(s)/(1+\delta)$.
2. Extra terms on the bending solutions $\propto \frac{\delta}{1+\delta}\frac{1}{r_0}$.
3. `py_MX` $=$ `y'_GPU` $\cdot (1+\delta)$.

2.3 Accelerator Elements

In this section, the pass-method for each element is shown, together with its declaration in the optics file[3]. Even though MAD-X/PTC elements were more sophisticated and admitted

[3]There is no distinction between uppercase and lowercase letters in the declaration.

several parameters referring to several effects such as fringe fields, misalignments, inclination of faces... only the parameters shown here are supported in the GPU code. Moreover, their units follow the MAD-X/PTC convention shown in Table 2.

Magnitude	Unit
Length, `L`	m
Quadrupolar strength, `K1`	m^{-2}
Sextupolar strength, `K2`	m^{-3}
Angle, `ANGLE`	rad

Table 2: Units of the declared magnitudes on the MAD-X/PTC optics file.

A couple of general comments must be me made before the element listing.

- Firstly, there are 3 variables named "K" in this work: the gradient k defined in (3), the parameter K used in the solutions (defined in Table 1); and in this section will appear another one, $K1$, the gradient defined by MAD-X/PTC, which is the parameter given in the optics file. Since every pass-method is based on K, but the input given is $K1$[4], they ought to be related. From K1 MAD-X/PTC's definition:

$$K1 \leftrightarrow K \qquad \text{using} \qquad K1 \overset{\substack{MAD-X/PTC's \\ def.}}{\equiv} -k \quad \text{and} \quad k \overset{Table1}{\rightarrow} K.$$

  Additionally, the sextupole's gradient k_2 defined in (6) and the MAD-X/PTC related variable $K2$ are defined exactly the same. Thus, the input $K2$ can be directly passed to the variable used on the pass-method, k_2.

- Secondly, in the dipoles pass-methods the bending radius appears, while on the MAD-X/PTC input file, only the bending angle, A, is given. Recalling that the length of a bending element is in general defined as a curved distance following the reference orbit, a connection can be established:

$$L \equiv r_0 \cdot A \implies r_0 = \frac{L}{A}.$$

Drift

`MY_DRIFT: DRIFT, L =` L`;`

The drift `MY_DRIFT` declared as shown, which has a length of L, has the following pass-method:

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} 1 & L \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}. \tag{10}$$

Quadrupole

`MY_QUADRUPOLE: QUADRUPOLE, L =` L`, K1 =` $K1$`;`

The quadrupole `MY_QUADRUPOLE` declared as shown, which has a length of L and a strength of $K1$, has the following pass-method:

[4]The reader will find in the code that this naming of "K"'s isn't strictly followed, although each corresponding amount is correctly used at all times.

$K = K1/(1+\delta)$

`if K > 0:`

$$\begin{pmatrix} x \\ x' \end{pmatrix} = \begin{pmatrix} \cos(|K|^{\frac{1}{2}}L) & |K|^{-\frac{1}{2}}\sin(|K|^{\frac{1}{2}}L) \\ -|K|^{\frac{1}{2}}\sin(|K|^{\frac{1}{2}}L) & \cos(|K|^{\frac{1}{2}}L) \end{pmatrix} \begin{pmatrix} x_0 \\ x'_0 \end{pmatrix}$$

$$\begin{pmatrix} z \\ z' \end{pmatrix} = \begin{pmatrix} \cosh(|K|^{\frac{1}{2}}L) & |K|^{-\frac{1}{2}}\sinh(|K|^{\frac{1}{2}}L) \\ |K|^{\frac{1}{2}}\sinh(|K|^{\frac{1}{2}}L) & \cosh(|K|^{\frac{1}{2}}L) \end{pmatrix} \begin{pmatrix} z_0 \\ z'_0 \end{pmatrix}$$

`if K < 0:`

$$\begin{pmatrix} x \\ x' \end{pmatrix} = \begin{pmatrix} \cosh(|K|^{\frac{1}{2}}L) & |K|^{-\frac{1}{2}}\sinh(|K|^{\frac{1}{2}}L) \\ |K|^{\frac{1}{2}}\sinh(|K|^{\frac{1}{2}}L) & \cosh(|K|^{\frac{1}{2}}L) \end{pmatrix} \begin{pmatrix} x_0 \\ x'_0 \end{pmatrix}$$

$$\begin{pmatrix} z \\ z' \end{pmatrix} = \begin{pmatrix} \cos(|K|^{\frac{1}{2}}L) & |K|^{-\frac{1}{2}}\sin(|K|^{\frac{1}{2}}L) \\ -|K|^{\frac{1}{2}}\sin(|K|^{\frac{1}{2}}L) & \cos(|K|^{\frac{1}{2}}L) \end{pmatrix} \begin{pmatrix} z_0 \\ z'_0 \end{pmatrix}.$$

Sextupole

`MY_SEXTUPOLE: SEXTUPOLE, L =` L`, K2 =` $K2$`;`

The sextupole `MY_SEXTUPOLE` declared as shown, which has a length of L and a strength of $K2$, has the following pass-method when slicing it with n kicks:

$k_2 = K2/(1+\delta)$

`if SIMPLE:`

`for box in n:`

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} 1 & L/2n \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}$$

$$x' = x'_0 - \tfrac{1}{2}\cdot k_2 \cdot \tfrac{L}{n}\cdot(x_0^2 - z_0^2)$$
$$z' = z'_0 + k_2 \cdot \tfrac{L}{n}\cdot x_0 z_0$$

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} 1 & L/2n \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}$$

`if TEAPOT:`

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} 1 & L/\left[2(1+n)\right] \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}$$

`for slice in (n-1):`

$$x' = x'_0 - \tfrac{1}{2}\cdot k_2 \cdot \tfrac{L}{n}\cdot(x_0^2 - z_0^2)$$
$$z' = z'_0 + k_2 \cdot \tfrac{L}{n}\cdot x_0 z_0$$

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} 1 & L\cdot n/(n^2-1) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}$$

$$x' = x'_0 - \tfrac{1}{2}\cdot k_2 \cdot \tfrac{L}{n}\cdot(x_0^2 - z_0^2)$$
$$z' = z'_0 + k_2 \cdot \tfrac{L}{n}\cdot x_0 z_0$$

$$\begin{pmatrix} y \\ y' \end{pmatrix} = \begin{pmatrix} 1 & L/\left[2(1+n)\right] \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_0 \\ y'_0 \end{pmatrix}.$$

Notice how drifts and sextupoles kicks are being applied following the slicing methods on Figure 3.

Sector Bend

MY_SBEND: SBEND, L = L, ANGLE = A, K1 = $K1$;

The sector bend MY_SBEND declared as shown, which has a (curved) length of L, bends an angle A and includes a quadrupolar component of strength $K1$, has the following pass-method:

HORIZONTAL PLANE

$$K = \left[\frac{A^2}{L^2} + K1\right] \Big/ (1+\delta)$$

if K > 0:

$$\begin{pmatrix} x \\ x' \\ \delta/(1+\delta) \end{pmatrix} = \begin{pmatrix} \cos(|K|^{\frac{1}{2}}L) & \frac{1}{|K|^{\frac{1}{2}}} \cdot \sin(|K|^{\frac{1}{2}}L) & \frac{A}{L|K|}\left[1-\cos(|K|^{\frac{1}{2}}L)\right] \\ -|K|^{\frac{1}{2}} \cdot \sin(|K|^{\frac{1}{2}}L) & \cos(|K|^{\frac{1}{2}}L) & \frac{A}{L|K|^{\frac{1}{2}}} \cdot \sin(|K|^{\frac{1}{2}}L) \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x'_0 \\ \delta/(1+\delta) \end{pmatrix}$$

if K < 0:

$$\begin{pmatrix} x \\ x' \\ \delta/(1+\delta) \end{pmatrix} = \begin{pmatrix} \cosh(|K|^{\frac{1}{2}}L) & \frac{1}{|K|^{\frac{1}{2}}} \cdot \sinh(|K|^{\frac{1}{2}}L) & \frac{A}{L|K|}\left[\cosh(|K|^{\frac{1}{2}}L)-1\right] \\ |K|^{\frac{1}{2}} \cdot \sinh(|K|^{\frac{1}{2}}L) & \cosh(|K|^{\frac{1}{2}}L) & \frac{A}{L|K|^{\frac{1}{2}}} \cdot \sinh(|K|^{\frac{1}{2}}L) \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x'_0 \\ \delta/(1+\delta) \end{pmatrix}$$

VERTICAL PLANE

$$K = -K1/(1+\delta)$$

if K > 0:

$$\begin{pmatrix} z \\ z' \end{pmatrix} = \begin{pmatrix} \cos(|K|^{\frac{1}{2}}L) & |K|^{-\frac{1}{2}} \sin(|K|^{\frac{1}{2}}L) \\ -|K|^{\frac{1}{2}} \sin(|K|^{\frac{1}{2}}L) & \cos(|K|^{\frac{1}{2}}L) \end{pmatrix} \begin{pmatrix} z_0 \\ z'_0 \end{pmatrix}$$

if K < 0:

$$\begin{pmatrix} z \\ z' \end{pmatrix} = \begin{pmatrix} \cosh(|K|^{\frac{1}{2}}L) & |K|^{-\frac{1}{2}} \sinh(|K|^{\frac{1}{2}}L) \\ |K|^{\frac{1}{2}} \sinh(|K|^{\frac{1}{2}}L) & \cosh(|K|^{\frac{1}{2}}L) \end{pmatrix} \begin{pmatrix} z_0 \\ z'_0 \end{pmatrix}.$$

Rectangular Bend

```
Option, RBARC=false;
MY_RBEND: RBEND, L = L, ANGLE = A, K1 = K1;
```

Notice the extra specification line! It should be included only once, prior to any call to an RBEND. This extra command comes from the MAD-X/PTC syntax, and it alters the meaning of the declared L:

$$\texttt{RBARC=false} \longrightarrow \texttt{L} \equiv L_{curved}$$
$$\texttt{RBARC=true} \longrightarrow \texttt{L} \equiv L_{straight},$$

where $L_{straight}$ is the distance along a straight line through the magnet. Therefore, this extra command must be included since the pass-method (4) uses L_{curved}.

The rectangular bend MY_RBEND declared as shown, which has a (curved) length of L, bends an angle A and includes a quadrupolar component of strength $K1$, has the following pass-method:

$$
\begin{aligned}
x' &= x'_0 + x_0 \cdot \frac{1}{1+\delta}\frac{A}{L}\tan\left(\frac{A}{2}\right) \\
z' &= z'_0 - z_0 \cdot \frac{1}{1+\delta}\frac{A}{L}\tan\left(\frac{A}{2}\right) \\
&\left.\begin{array}{c}\vdots\\ \vdots\\ \vdots\end{array}\right\} \texttt{SBEND pass-method with: } L,\ A \texttt{ and } K1 \\
x' &= x'_0 + x_0 \cdot \frac{1}{1+\delta}\frac{A}{L}\tan\left(\frac{A}{2}\right) \\
z' &= z'_0 - z_0 \cdot \frac{1}{1+\delta}\frac{A}{L}\tan\left(\frac{A}{2}\right)
\end{aligned}
$$

Recall that faces are inclined with respect to the reference orbit on a RBEND. Concretely, when an RBEND is bending an angle A, each face has a $A/2$ inclination. And that's exactly the aperture of the wedges that implement the focusing effects, which can be seen in the above pass-method.

MAD-X/PTC Syntax for Declaring the Ring

In the optics file, after the desired elements have been declared, a line can be created gathering all of them (each one can appear multiple times) in a specified order with the following command:

```
MY_LINE: LINE = (MY_RBEND, MY_Q, MY_DRIFT, MY_Q, MY_S, MY_SBEND);
```

A line can be made up of other lines.

```
MY_RING: LINE = (MY_FIRST_LINE, MY_MID_LINE, MY_LAST_LINE);
```

An example of a complete optics specification file is shown below:

```
Option, RBARC=false;
D1:  DRIFT, L = 0.3;
D2:  DRIFT, L = 0.3;
QF: QUADRUPOLE, L = 0.3, K1 = 1.5;
QD: QUADRUPOLE, L = 0.31, K1 = -1.1;
RBF: RBEND, L = 0.7, ANGLE = 0.1, K1 = 0.6;
RBD: RBEND, L = 0.6, ANGLE = 0.08, K1 = -0.6;
SBF: SBEND, L = 0.7, ANGLE = 0.1, K1 = 0.6;
SBD: SBEND, L = 0.7, ANGLE = 0.1, K1 = -0.6;
SF: SEXTUPOLE, L = 0.17, K2 = 38;
SD: SEXTUPOLE, L = 0.16, K2 =-62;
RING: LINE = (QF, D1, SD, D2, QD, D1, SF, D2, D1, QD, D2, SF, D1, QF, D2, SD, D1,
RBD, D2);
```

3 Implementation

3.1 Data Parallelism

In this particular problem, the code to be parallelized is used for tracking a certain amount of *distinct* particles through the *same* accelerator. Let's specify what these two words mean in this context. Particles are *distinct* in the sense that, although they all share the same type (mass

and charge), they all can have different initial conditions: position and velocity on the horizontal and vertical plane, together with momentum deviation. On the other hand, the accelerator each particle gets tracked through consists of the *same* set of elements on every case. However, any parameter of any individual element (such as the length of a drift, for instance) can be set to vary from particle to particle. Let's name this parameters *variable parameters* for future references.

This is to say, this script will simulate multiple particles which can start with various initial conditions, and one can choose any parameters of the ring to change for each particle (to become variable parameters).

Moving on, the heart of parallelisation resides in the simultaneous execution of code, a feature that, depending on software and hardware, could be exploited to speed up computations. Gladly enough, one sort of parallelization consists of carrying out parallelly the *same* task on *distinct* sets of data, the so called *data parallelism*. One can already see how naturally this fits with the particular software for tracking multiple particles, explained above. The simulation code is going to be executed in parallel with different sets of data. Each set is going to comprise a particles' initial conditions and the values of the variable parameters.

Lastly, let us call each distinct computational unit able to execute a process a *core*. They can be provided by several hardware, such as GPU or multicore CPU. Nonetheless, the speed up of parallel computing depends on the hardware, and for the concrete case of data parallelism, a GPU's design is highly optimized.

The connection between software and hardware is accomplished through OpenCL – this is, choosing the GPU or CPU[5], moving the program and its variables to each core, managing memory, . . . – . Moreover, in this work, its Python's application programming interface has been used. Python scripts are only used for "housekeeping" tasks, while heavy duty computations are relegated to OpenCL.

3.2 Code Architecture

The tracking code is structured into three parts: importing (reading) the optics of the accelerator (using `madx_utils.py`), setting up the tracking kernel (using `strackino.py`) and running said kernel. All complexity and difficulties come from the second step. More specific details on each step will be given in the following section.

Firstly, the optics of the accelerator are read from a plain text file written in the MAD-X/PTC-like syntax, explained in the previous section.

The second part, the set up of the tracking, can be broken down to:

- **Initialization of the device.** OpenCL finds all available devices, i.e. GPU, CPU. . . The user can choose which one should be used for computation.

- **Assembling the kernel.** The *kernel* is the actual code passed to the selected hardware (GPU, CPU, . . .), the actual instructions to be executed by each core. Since data parallelism is going to be used, here's how the tracking is approached, i.e. the contents of the kernel.

  Recall that each track is characterized by a single particle initial conditions and the value of the variable parameters. All of this information, the "properties" of every track, is going to be stored in an array named `pool`.
  Initially, when the kernel starts to run, each core will look at `pool` and take the properties of one track. Each different core will take a different track. With these properties, each core will parallelly start a loop through the turns.

[5]Although OpenCL has been designed with GPU use in mind it can also be used on normal CPU without requiring any change to the simulation code (in fact, it is only the OpenCL back-end that is swapped).

On each turn, the same one-turn pass-method (due to the lattice periodicity) is going to be applied, propagating the particle's coordinates element-by-element through the lattice until the end of the turn.
After each turn, it is checked whether or not the particle is lost, and if that is the case, the loop will be interrupted. Otherwise, it will continue.
When the specified amount of turns is reached, or if the particle was lost, final coordinates are copied into the same variable `pool`[6]. After a core ends its turn-loop, it looks again at `pool`, takes another track's properties and repeats the one-particle tracking. When all tracks have been used, the program execution end and the data is copied back to the host CPU.
Conclusively, on each set up call, the kernel containing the above instructions is assembled. The one-turn pass-method is assembled in `generators.py` by combining together the pass-method of each individual element.

- **Building the kernel.** Once the kernel is assembled, it has to be processed and compiled by the OpenCL hardware specific back-end: it has to be "built". This is actually the most time consuming step. On the other hand, luckily, it needs to be carried out only once. Since the built kernel is cached, further execution of this step would not require additional time.

Lastly, each track's properties are given (filling `pool`) and the kernel is executed.

3.3 Practical User Guide

First of all, the (file-wise) set up to use this tracking code consists of:

- A plain text file with the optics of the accelerator, written in MAD-X/PTC-like syntax[7].
- The python script called `madx_utils.py`, with the tools to read said plain text file.
- The python script called `generators.py`, containing the physics of each magnet type.
- The python script called `strackino.py`, where parallelization and tracking happens.

One should import `madx_utils.py` and `strackino.py` (`generators.py` gets internally imported). Furthermore, NumPy is also required.

After the initial imports, the first thing to do is reading the optics of the accelerator with:

```
lattice = madx_utils.Lattice(path, 'RING')
```

where `path` is the location of the file containing the optics, and `'RING'` is the name of the desired line to be used for tracking (its name must appear in the optics file). The returned object, here called `lattice`, is a custom class instance which only has two attributes: `elements`, a dictionary with magnets' names, the name of its parameters and their corresponding values; and `line`, the lattice, an ordered list of labels referring to the magnets from the previous dictionary.

Once the accelerator optics are read, the tracking must be set up with the following call:

```
t = strackino.Track('Device Name', parameters, lattice, num_of_tracks, turns)
```

where:

[6] Tracks which have already been used and those who still haven't are properly distinguished.

[7] It refers to the concrete syntax explained in Section 2, not the complete MAD-X/PTC format. Some element's parameters are not supported here but they are in MAD-X/PTC.

- `'Device Name'` is the hardware to be used, for our purposes, the GPU. Its name depends on the computer configuration. Running it once with whatever name, a list of the available devices will be printed out, in case you don't know which one to use.

- `parameters` is a list of tuples specifying the element's parameters that could be changed from track to track. For example, if you only want the length (called `L` on the optics file) of the drift `myDrift` (custom name on the optics file) to possible vary depending on the track, then:

  ```
  parameters = list([('myDrift', 'L')]).
  ```

  This variable can be an empty list, in which case, no parameters will be allowed to vary from track to track.

- `lattice` is the returned object explained above.

- `num_of_tracks` is the amount of tracks to be done, i.e. number of particles to be tracked, each tracking characterized by some initial conditions and some values of the variable parameters.

- `turns` is the number of turns.

This call returns a custom class instance, here called `t`, which only has a few features of interest to the reader right now. An attribute called `pool`, containing the initial conditions and variable parameters of each track. It works as a dictionary, whose labels can be looked up in another attribute, `pool_t`[8]. Lastly, its method `run` execute the computations.

Before running the tracking, the initial condition of each particle must be initialized. This needs to be repeated for each coordinate and parameters. In the case of the x variable:

```
t.pool['x'] = numpy.array(custom_horizontal_positions).
```

Finally, the tracking is executed by calling the method run:

```
res = t.run(thread_count).
```

The input `thread_count` is a suggestion to OpenCL about the amount of cores to use. Its value for best performance depends on the specific device, and it should be tuned by hand. Empirically was found that setting it to a few times the amount of available cores gives the best performances (this condition in fact ensure to keep saturated the hardware). The output `res` is the result of the tracking. It has the structure of `pool`, but its type is instead `pyopencl.array.Array`. Results can be copied back with

```
res_np = res.get().
```

This variable contains the final values of each particle's position and velocity (as well as the values of the variable parameters used).

In the code, a lost particle is defined as having an absolute position greater than `0.1` (meters), or a position which is `inf` or `NaN`[9]. This gets checked after every turn, and the maximum position allowed can be manually changed inside `strackino.py`, in `init_kernel`'s statement

```
if ( !(fabs(x) < 0.1) || !(fabs(y) < 0.1)) .
```

[8] In case it helps, here "`t`" stands for "type".

[9] This is equivalent to check if the particle's position *is not* lower than `0.1` meters.

Moreover, the `dp` value is set to `1000` for lost particles, in order to recognize them in the output `res`.

Summing up, one could track `2000` particles in the accelerator `'RING'` for `4000` turns, with no variable parameters and various initial conditions, using a GPU with `1000` threads, writing the following lines:

```
# imports...
lattice = madx_utils.Lattice('../example.txt', 'RING')
t = strackino.Track('Intel(R) Iris(TM) Graphics 6100',
set(), lattice, 2000, 4000)
t.pool['dp'].fill(my_dp)
t.pool['x'] = numpy.array(custom_horizontal_positions)
t.pool['dx'] = numpy.array(custom_horizontal_velocities)
t.pool['y'] = numpy.array(custom_vetical_positions)
t.pool['dy'] = numpy.array(custom_vertical_velocities)
res = t.run(1000)
```

4 Results

In this section, the accuracy (with MAD-X/PTC as reference) is checked – through dynamical aperture calculations –, and the performance of the implementation is *measured*, benchmarking multiple cases on different devices.

4.1 Dynamical Aperture

The dynamical aperture (DA) calculation is carried out, both in MAD-X/PTC and in GPU, by tracking a number of particles with different starting positions (all with null initial transverse momentum) and checking whether or not they get lost after 1000 turns. These particles may have a certain momentum deviation as well. The region of initial positions where particles aren't lost makes up the DA, which is then compared across the different simulations.

Parametrization

In this calculation, the set of initial positions sits on a $n \times n$[10] grid of amplitude a centred at the origin of coordinates (positions range from $-a$ up to a). The amplitude is adjusted to fit the dynamical aperture, which depends on the simulated optics (here, either ALBA-I or ALBA-II). In order to attain a considerable density of the grid, $n = 61$ has been chosen.

The tracking goal is to simulate for 1000 turns on the real lattice, but since the ring defined in the lattice file is made of only one of the four identical quadrants that make up ALBA„ 4000 turns are needed there.

MAD-X/PTC tracking is governed by several parameters. First of all, `model` and `method` are set at 2 (Matrix-Kick-Matrix) and (integration order) 6, respectively. Secondly, increasing the number of integration steps, `nst`, improves MAD-X/PTC's results in exchange of computational speed. It is kept at 10. Lastly, the use of the exact relativistic Hamiltonian is controlled with `exact` (the exact form is used when `exact = true`). Since `exact = false` doesn't fully consider relativistic effects, it produces a simulation of lower quality.

Similarly, GPU also depends on some parameters. In this case, GPU is less flexible and only two of them affect the results (the other ones are performance related). Namely, `stl` and `nst`, the style and number of slices on the slicing of sextupoles. They are kept at TEAPOT and 5,

[10] Thus, this is the amount of particles on the grid.

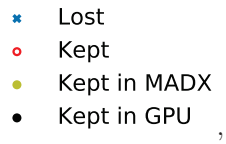
respectively.

For both rings, ALBA-I and ALBA-II, the DA has been computed for several momentum deviation values:

$$\texttt{dp} = +0.05, +0.03, +0.01, 0.00, -0.01, -0.03, -0.05.$$

The use of the exact Hamiltonian on MAD-X/PTC (i.e. the `exact` parameter) has been studied on every case.

On each figure that will be presented, the initial position of the particles, `x` and `y` respectively[11], are plotted, i.e. the initial grid. For each particle, it is indicated whether or not it was lost during the simulations. Markers follow the legend



where `Lost` stands for particles predicted lost on both simulations, similarly for kept ones denoted by `Kept`, both favourable results for the comparison. Last two markers imply that the other simulation predicted a lost particle there, hence an unmatching result. Accordingly, the difference between simulations is quantified and shown in the plots, named `diff.`. It is calculated counting the unmatching results and dividing them by the amount of kept particles on MAD-X/PTC.

ALBA-I

Results are given on Figure 4, where the amplitude of the grid has been set to $a = 0.03\ m$.

Preventing the use of the exact Hamiltonian on MAD-X/PTC with `exact = false` always resolves in GPU results matching better. The value of `diff.` can be seen to always decrease from `true` to `false`. This is, GPU results approach the MAD-X/PTC `exact = false` situation, and not the `exact = true` one. This is to be expected, since the tracking on the GPU code doesn't consider all relativistic effects, which is precisely what `exact = false` does.

When comparing the GPU results with the `exact=true` MAD-X/PTC situation, it is found that results are still acceptable for some cases, but diverge for high momentum deviations values.

ALBA-II

Results are given on Figure 5, where the amplitude of the grid has been set to $a = 0.01\ m$, a third of the ALBA-I case, to match the smaller DA.

In this case, avoiding the use of the exact Hamiltonian also improves the amount of matches between GPU and MAD-X/PTC. Despite that, GPU results are still comparable to those from MAD-X/PTC's `exact = true` case for all momentum deviation values.

Summing up, the differences between GPU and MAD-X/PTC's `exact = true` simulation can be mainly attributed to relativistic effects, since MAD-X/PTC's `exact = false` yields results in great agreement with GPU. These relativistic discrepancies are only significant on the ALBA-I case (and high momentum deviation values). On the ALBA-II ring, where the DA is smaller, relativistic effects aren't that significant.

[11] Notice that here `y` stands for vertical position.

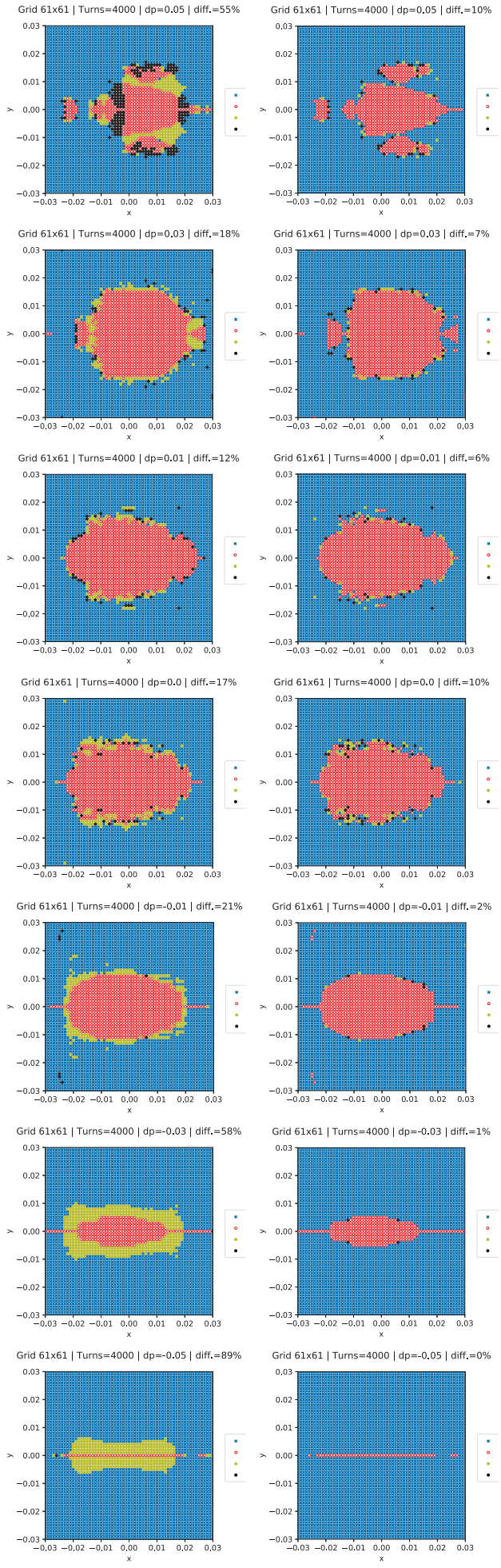


Figure 4: *Left*: `exact = true`. *Right*: `exact = false`.

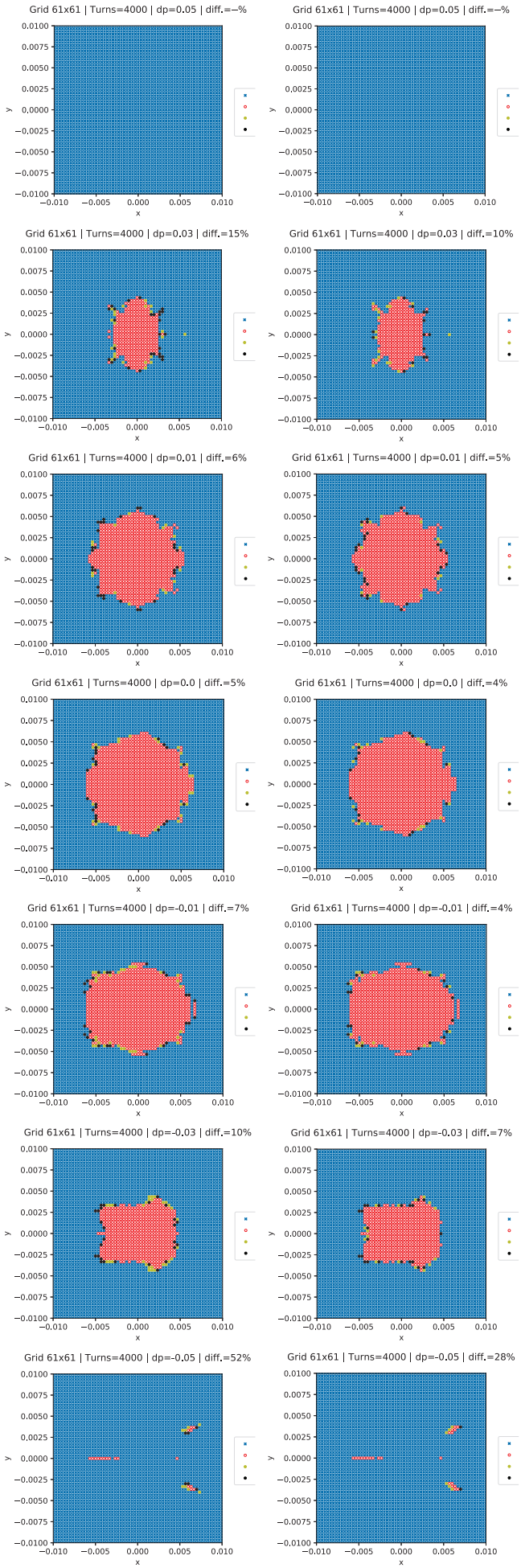


Figure 5: *Left*: `exact = true`. *Right*: `exact = false`.

4.2 Benchmarks

The performance of the tracking may depend on several factors, naming three of them: number of parameters of the simulation, hardware used and current state of the computer (applications running, memory available, power of the laptop...). The first two factors can be more or less controlled and maintained stable for different tests, but that's not the case for the last one, the state of the computer upon testing a tracking.

Due to that factor, run-times of the same test can change from hour to hour, or from day to day. Thus, a considerable error on the run-times is to be expected. Consequently, when testing a certain parameter and its influence on the performance, only major run-time differences that overcome those computer-state errors will be significant and reliable.

There are various parameters on the simulation that can be controlled: amount of turns, t, square grid size for computing the DA (with amplitude a), momentum deviation, dp, amount of slices on the sextupoles, nst, data representation accuracy, $bits$, amount of variable parameters, var, and of course, amount of particles on the grid, $n \times n$. DA tests are done with ALBA-I optics only.

Only $bits$, var and n are going to be studied. The rest of the parameters will be fixed with the values of a usual case. That is,

$$t = 4000 \qquad a = 0.04\ m \qquad dp = 0 \qquad nst = 5.$$

Hardware-wise, the devices available for testing are those from two computers – my personal laptop, here referred to as *Mine*, and a PC at ALBA that is been remotely used –, each having a GPU and a CPU device. For reference:

Mine, CPU : Intel(R) Core(TM) i5-5250U CPU @ 1.60GHz
Mine, GPU : Intel(R) Iris(TM) Graphics 6100
ALBA, CPU : pthread-Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz
ALBA, GPU : Quadro P600.

When considering different tests, results are going to be computed either for some or for all of the devices. Particularly, in some situations the laptop *Mine* takes about 40 minutes to compile a single test, making the test practical only on the ALBA's computer.

Recall as well that run-times for this tracking are fundamentally different from the first call to the second one (and any further call). On the first call, the code is both build (compiled) and executed. On further calls, no compilation is needed, speeding up the tracking process. Therefore, the difference between run-times could be considered as the compiling time, with the time of the second call being that of the execution.

Amount of Particles

Maintaining the values of *bits* and *var* at their defaults (32 and 0 respectively), the amount of tracked particles has been varied. Results are shown on Figure 6, where only the computer at ALBA has been used.

Looking at the execution times (run-times of the second calls), GPU is considerably faster than the CPU. Furthermore, the execution times grow steeper on the CPU than in the GPU. From $n \times n = 25$ to $n \times n = 3025$, CPU execution time increases by a factor of ~ 16, while GPU's only by ~ 2.5.

Considering now the compiling time (the difference between first and second call), GPU has a really hard time building the program. CPU takes about 1 s to compile, while GPU takes up to 16 s. This compiling time stays approximately constant with respect to n.

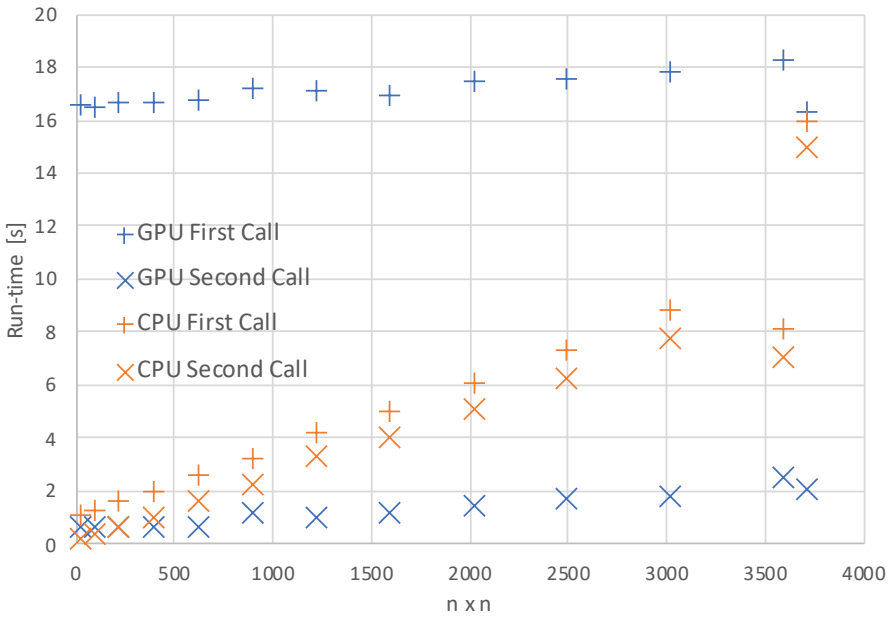


Figure 6: Run-times for first and second call of the tracking as a function of the total amount of particles $n \times n$. The computer at ALBA has been used for this plot. The difference between calls can be clearly seen, as well as the difference between GPU and CPU behaviours.

Consequently, when taking into account the full run-time (the whole first call run-time) GPU has been measured slower than the CPU. In spite of that, due to the fast increase on the CPU's execution time, the GPU can be expected to perform better for larger number of particles. It has to be kept in mind that the GPU's second call run-time performs remarkably better than the CPU.

Remember, the actual values of the run-times could vary according to that computer-state factor, as it has been empirically seen. This is, the actual values can vary from day to day: the CPU's $n = 61$ case lasted more than double the time shown on the figure the day before the figure data was gathered. Other examples of run-time values drastically changing could be presented too. Nonetheless, the time consumption distribution on CPU and GPU, as well as the steepness of the run-times, are facts independent from the state of the computer. They are always observed on all tested devices.

Giving values from the devices on the laptop *Mine*, the values' hard dependence on the specific hardware is evidently illustrated. See Table 3. As it was already pointed out, the computer-state factor can be seen here, noting that values of the table don't match those of the previous figure, which were registered on different days.

Amount of Variables

Fixing $n = 61$ and still keeping the precision at $bits = 32$, the amount of variable parameters is now studied. Results using the ALBA computer are shown on Figure 7.

Data on Figure 7 doesn't have a clear tendency, it could be considered constant at most. Moreover, even going to the extreme case were all available parameters were considered as variable, no significant change in behaviour is observed. As always, the computer-state factor plays a significant role on every benchmarking, but in this case the feature explained in section Amount of Variables Quantities upon Compiling mainly contributes to the results shown here.

$\boxed{n=5}$	GPU	CPU
ALBa's PC	(37) 1.1	(2.4) 0.35
Mine	(43) 2.3	(6.3) 0.04
$\boxed{n=61}$		
ALBa's PC	(39) 2.6	(34) 31.7
Mine	(594) 2.5	(10.6) 5.1

Table 3: Run-times of the (first) second call of the simulations for different devices and $n = 5, 61$ (*bits* and *var* have been kept at 32 and 0, respectively). Values given in seconds. Notice the huge time consumption on *Mine* for $n = 61$.

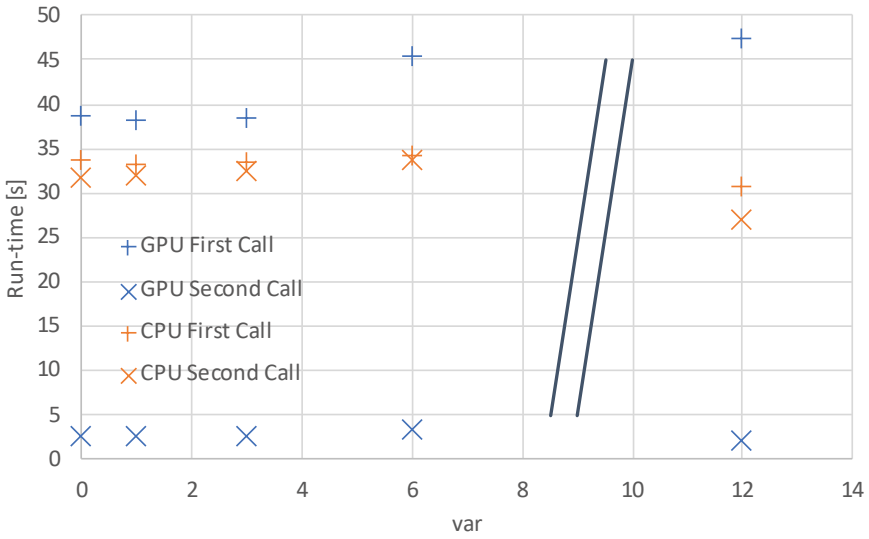


Figure 7: Run-times for first and second call of the tracking as a function of the amount of variable parameters, using the computer at ALBA. The two diagonal lines stand for an axis break. Then, **the value var = 12 actually refers to the value 86** (the case where all possible variables on the ALBA-I ring are being used).

Anyhow, the differences between GPU and CPU (compiling and execution time distribution) can be still clearly spotted, in the previous figure as well as in Table 4.

Precision

Lastly, considering the two possible situations concerning precision, *bits* = 32, 64, the results are the following for the devices on the computer at ALBA. Amount of particles and variables has been kept now at 61 and 0, respectively. Giving both first and second call run-times:

$$\underline{bits = 32}$$
GPU: run-time = (39) 2.6
CPU: run-time = (34) 32.
$$\underline{bits = 64}$$
GPU: run-time = (63) 4.4
CPU: run-time = (33) 31.

Therefore, a difference can be seen on the GPU results while CPU remains almost constant, but it can not be assured that it is the effect of the precision. Again, the state of the computer has

$\boxed{var = 0}$	GPU	CPU
ALBa's PC	(39) 2.6	(34) 31.7
Mine	(594) 2.5	(10.6) 5.1
$\boxed{var = 6}$		
ALBa's PC	(45) 3.3	(35) 33
Mine	(2445) 3.1	(12.1) 5.0
$\boxed{var = 86}$		
ALBa's PC	(47) 2	(31) 27
Mine	(3134) 5.1	(15) 4.6

Table 4: Run-times of the (first) second call of the simulations for different devices and $var = 0, 6, 86$ (*bits* and n have been kept at 32 and 61, respectively). Values given in seconds.

too much effect, varying the values of a single test from call to call. In order to trust and rely on these values, the computer-state factor should be properly isolated. Nonetheless, no drastic change on the values is observed, therefore, no concerns about the use of 64 bits should arose.

Amount of Variables Quantities upon Compiling

A special focus is made on the amount of *variable quantities* that the compiler considers. On the compiling process, a distinction is made between quantities that will remain constant throughout the different tracks and quantities that will vary from track to track. An example of a constant quantity could be the factor `ANGLE/L` of some bend whose parameters are the same for each particle (they are not set as variable parameters). An example of a variable quantity could be the square root

$$\sqrt{\frac{\texttt{K1}}{\texttt{1 + dp}}},$$

since even if `K1` is constant, `dp` is a quantity that could vary from track to track, making the whole square root a variable quantity.

When there are a lot of variable quantities in the kernel the compiler has a hard time managing the code. It has to keep track of every quantity that can change depending on the tracked particle. Constant quantities just get evaluated once while the code is compiled. Therefore, a low number of variable quantities is desired for a fast compiling process.

Consider now a ring with all parameters (`L`'s, `ANGLE`'s, `K1`'s and `K2`'s) fixed – there are no parameters set as variable parameters. This should give the lowest number of variable quantities. Then, consider the strengths of the magnets used on the implementation (see section 2.3 and the expressions below), remembering that momentum deviation implied a `(1 + dp)` factor everywhere.

`K = K1 / (1 + dp)` $\qquad$ k_2`= K2 / (1 + dp)`

$$\texttt{K} = \left[\frac{\texttt{A}^2}{\texttt{L}^2} + \texttt{K1}\right] \Big/ \texttt{(1 + dp)} \qquad \texttt{K = -K1 / (1 + dp)}$$

Since, `dp` is a variable quantity, all `K`'s will be considered as a variable quantity, regardless of the fact that strengths `K1` and `K2` are set as fixed parameters! This has a huge impact on the compiling process. It implies that the code will have a lot of variable quantities which could otherwise remain constant quantities if it wasn't for the `dp`. Remarkably, trigonometric functions such as

$$\cos\left(\sqrt{\frac{\texttt{K1}}{\texttt{1 + dp}}} \cdot L\right),$$

will be considered as variable quantities. Trigonometric functions are specially interesting to keep as constant quantities since they are expensive to evaluate, and keeping them as variable ones really slows down the compilation and execution process.

Furthermore, in the DA calculations, the same `dp` value is passed to each particle. This means that the actual value of the trigonometric functions doesn't change from track to track. In spite of this, the compiler still considers these "*constant*" trigonometric quantities as variable ones, since at the moment of compiling, no values have been passed to `dp` yet.

Summing up, just the implementation of momentum deviation alone implies that all strengths will be variable quantities, which will slow down the compiling process, even if the actual value of `dp` is the same for all particles.

In order to test this effect of the `dp`, the DA on the ALBA-I ring has been once more simulated, with $a = 0.01$, $n = 11$ and $bits = 32$. Three different situations have been considered:

1. **With `dp` implementation and $var = 0$.** In this case, the considered variable quantities are: the usual variables `x, dx, y, dy, dp`, and the strengths `K` (and all the subsequent quantities that this implies, such as trigonometric functions). Only `ANGLE`'s and `L`'s remain as constant quantities.

2. **Without `dp` implementation and $var = 0$.** Here, only the usual variables `x, dx, y, dy, dp` are variable quantities. `K`'s, `ANGLE`'s and `L`'s are truly constant quantities. Thus, all trigonometric functions and factors on the expressions will be considered as constant values. This has the lowest amount possible of variable quantities.

3. **Without `dp` implementation and $var = full$.** This means that all parameters are set as variable (in the ALBA-I ring there are 86 of them). This is the extreme case where besides simple numbers on the expressions, everything else will be a variable amount, including `ANGLE`'s and `L`'s. There are more variable quantities here than in case 1.

It has been tested on both computers, and results obtained are shown on Table 5.

1	GPU	CPU
ALBa's PC	(16.11) 0.78	(3.8) 2.87
Mine	(1067) 2.97	(7.62) 0.91
2		
ALBa's PC	(6.21) 0.56	(2.21) 1.46
Mine	(132) 2.42	(6.8) 0.83
3		
ALBa's PC	(19.08) 0.75	(3.35) 2.43
Mine	(3013) 3.5	(4.43) 0.81

Table 5: Run-times of the (first) second call of the simulations for different devices and the situations explained above. Values given in seconds.

Notice how compiling run-times generally drop in situation 2, where there are the fewest variable quantities. The effect can be best seen on GPU devices. Remarkably, run-times of the laptop *Mine* drop to an astonishing 132 s on situation 2, while peaking up to a 3031 s run-time on situation 3. And they are simulating the same ring!

5 Conclusions

As a general conclusive remark: a simple, yet fast, tracking code has been successfully built using a GPU as back-end device. Such implementation accomplishes both speed (in comparison with a CPU) and accuracy, with respect to the tracking with approximated Hamiltonian MAD-X/PTC's PTC tracking. Remarkably, in many situations, results are comparable to the full Hamiltonian ones (still provided by MAD-X/PTC's PTC). This conclusion is expanded on the following points, including some suggestions for improvements on future versions.

Code

The simplicity of the code comes from two sources: the reduced amount of magnetic effects considered and the initial supposition of avoiding some relativistic effects. Regarding the first source, the most complex effect considered here is the inclination of the faces, and it has only been considered on one element (the RBEND). Said effect, as well as many others such as fringe fields, misalignments or higher order components, could be considered on each element in future upgraded versions of the code. The second source, the approximation of some relativistic effects, greatly simplified the calculations, uncoupling the equations of motion and allowing for a thick treatment of the elements. A future version of the code could also include the relativistic tracking option.

Accuracy

The results this tracking provides are in agreement with the analogous ones computed by MAD-X/PTC, although some differences show up. Generally, the relativistic discrepancies appear most significantly for high values of momentum deviation ($|dp| \gtrsim 0.03$). More concretely, these relativistic effects depend on the amplitude of the trajectories: higher amplitudes show greater mismatches. That's the reason why the ALBA-II case doesn't present meaningful relativistic deviations, since its DA is $\sim 1\ cm$ while ALBA-I's is 4 times bigger ($\sim 4\ cm$).

Speed

In spite of the fluctuations on the measures of the run-times, a couple of reliable conclusions can be made. On the one hand, GPU devices take longer to compile the program that is passed to them, CPU outperforms them every time. On the other hand, the execution time of a GPU, the actual time where calculations are made, largely overcomes those of a CPU device. Again, this should also be expected since the parallelisation method fits remarkably well with the functioning of a GPU. A short last comment could be also added: a good benchmarking method should be developed on future versions, since the current one allowed for few to none conclusions on the use of variables, precision and amount of particles.

Compiling process

Taking care of the variable quantities on the compiling process can lead to a reduction on the run-times, as has been seen on Table 5. The effect shown there *sacrifices* the implementation of momentum deviation to gain compiling speed. Nonetheless, since on all DA calculations the `dp` value is kept constant for all particles, one could figure out a way to maintain said implementation while avoiding to consider `dp` as a variable, and therefore as a variable quantity, potentially reducing compiling run-times.

References

[1] P. J. Bryant and K. Johnsen, *The Principles of Circular Accelerators and Storage Rings*, *1*st ed. Cambridge, GB: CUP, 1993, ch. 2-3 and app. A.

[2] L. Deniau, H. Grote, G. Roy and F. Schmidt. (2020, Sep.). The MAD-X Program. CERN. Geneva, CH. [Online]. Available: `http://mad.web.cern.ch/mad/webguide/manual.html`

[3] *The OpenCL Specification*, 1.2 ver., Khronos Group, Beaverton, OR, 2021, pp. 244-318

[4] A. Wolski. *Nonlinear Single-Particle Dynamics in High Energy Accelerators. Part 4: Canonical Perturbation Theory* [Online]. Available: `https://www.cockcroft.ac.uk/archives/course/non-linear-dynamics`. pp. 23-25

[5] H. Burkhardt, R. De Maria, M. Giovannozzi, and T. Risselada. Improved TEAPOT method and tracking with thick quadrupoles for the LHC and its upgrade. In *Pro- ceedings of the 2013 IPAC Conference*, number MOPWO027 in International Particle Accelerator Conference, 2013. `http://accelconf.web.cern.ch/AccelConf/IPAC2013/papers/mopwo027.pdf`.