# SPACE CHARGE MODULES FOR PyHEADTAIL

A. Oeftiger*, CERN, Meyrin, Switzerland; S. Hegglin, ETH Zürich, Zürich, Switzerland

*Abstract*

PyHEADTAIL is a 6D tracking tool developed at CERN to simulate collective effects. We present recent developments of the direct space charge suite, which is available for both the CPU and GPU. A new 3D particle-in-cell solver with open boundary conditions has been implemented. For the transverse plane, there is a semi-analytical Bassetti-Erskine model as well as 2D self-consistent particle-in-cell solvers with both open and closed boundary conditions. For the longitudinal plane, PyHEADTAIL offers line density derivative models. Simulations with these models are benchmarked with experiments at the injection plateau of CERN's Super Proton Synchrotron.

## INTRODUCTION

The self-fields of particle beams superpose the electromagnetic fields applied by magnets and radio frequency (RF) cavities in synchrotrons. The corresponding space charge effects lead to defocusing in the transverse plane and focusing (defocusing) in the longitudinal plane for operation above (below) transition energy. For non-linear beam distributions, space charge results in a tune spread which is an important factor e.g. when investigating betatron resonances or the influence of Landau damping during instabilities. We present the implemented space charge models of the collective effects simulation software PyHEADTAIL [1] which is developed in Python. PyHEADTAIL models beam dynamics by transversely tracking macro-particles linearly between interaction points around the circular accelerator. Longitudinal particle motion is modelled either by linear tracking or non-linear (sinusoidal) drift-kick integration. The forces from collective effect sources such as electron clouds, wake fields from impedances or space charge are integrated over the respective distance and applied as a momentum kick at the following interaction point [2]. Recently, large parts of PyHEADTAIL have been parallelised for NVIDIA graphics processing units (GPU) architectures [3]. Our particle-in-cell library PyPIC used for the self-consistent space charge models in PyHEADTAIL especially benefited from these efforts – the corresponding speed-ups are reported here.

This paper is structured as follows: we first address the implemented space charge models, which is followed by our GPU parallelisation strategies and achieved improvements, and, finally, we compare simulation results with measurements at CERN's Super Proton Synchrotron (SPS). Our developed software and libraries are available online [4].

## SPACE CHARGE MODELS

A PyHEADTAIL macro-particle beam of intensity $N$, particle charge $q$ and particle mass $m_p$ is described by the 6D set of coordinates $(x, x', y, y', z, \delta)$, where $x$ denotes the horizontal offset from the reference orbit, $y$ the vertical offset, $x' = p_x/p_0$ and $y' = p_y/p_0$ the corresponding transverse normalised momenta for $p_0 = \gamma m_p \beta c$ the total beam momentum, $z$ denotes the longitudinal offset from the synchronous particle in the laboratory frame and $\delta = (p_z - p_0)/p_0$ the relative momentum deviation. Most of the space charge models are based on "beam slices" which represent longitudinally binned subsets of the beam distribution. The density per slice is determined by nearest grid point (NGP) interpolation (i.e. lowest order).

### *Longitudinal Space Charge*

For an emittance-dominated bunched beam, which is usually the case in a circular accelerator, the longitudinal electric field depends on the local line density $\lambda(z)$. Beams in CERN's circular accelerators typically have a very long bunch length in comparison to the vacuum tube diameter. Therefore, the non-linear image fields suppressing the longitudinal electric field have to be taken into account for the longitudinal space charge model. In PyHEADTAIL we provide such a so-called $\lambda'(z)$ model following the extensive analysis in [5, chapter 5]. The space charge forces are computed assuming a linear equivalent field

$$E_z^{\text{equiv}}(z) = -\frac{g}{4\pi\epsilon_0\gamma^2}\frac{d\lambda(z)}{dz} \quad , \tag{1}$$

where $g$ denotes the geometry factor and $\epsilon_0$ the vacuum permittivity. Conceptually, the real longitudinal profile is treated like a parabolic line density

$$\lambda(z) = \frac{3Nq}{4z_m}\left(1 - \frac{z^2}{z_m^2}\right) \tag{2}$$

with parabolic bunch half length $z_m$. In particular, this model identifies the mean value of the real electric field $\langle z E_z^{\text{real}}(z)\rangle_z$ (which includes the non-linear image effects) with the corresponding analytical expression for the parabolic distribution with the generalised geometry factor $g$, which then absorbs the image field contributions. Our implemented model is valid for bunches satisfying $z_m > 3r_p$ where $2r_p$ denotes the diameter of a perfectly conducting cylindrical vacuum tube. In this case, the geometry factor $g$ becomes independent of the bunch length and can be averaged over the whole distribution yielding [5, Eq. (5.365b)]

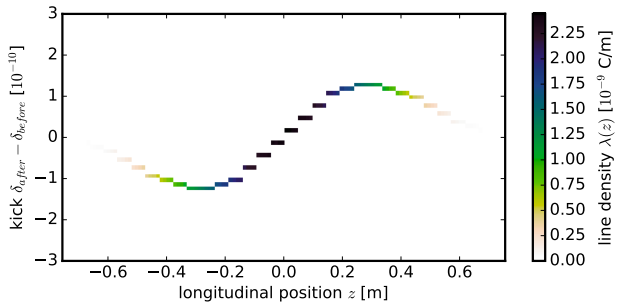
$$g = 0.67 + 2\ln\left(\frac{r_p}{r_b}\right) \tag{3}$$

where $r_b$ denotes the radial half width of a transversely round ellipsoidal beam with uniform charge distribution. As a further remark, [5, chapter 6] also discusses the case of two parallel conducting plates with distance $2r_p$, for which the

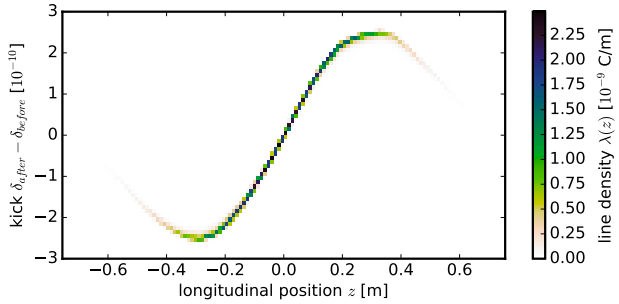
* adrian.oeftiger@cern.ch, also at EPFL, Lausanne, Switzerland

ISBN 978-3-95450-178-6

image term $2\ln(r_p/r_b)$ becomes $2\ln(4r_p/(\pi r_b))$. Figure 1 compares the longitudinal kicks for a round vacuum tube and the above model (for which $g = 5.25$, compare to $g_0 = 6.36$ in free space) with the real electric field kicks computed in free space by the particle-in-cell (PIC) algorithm described in subsection . Indeed, the $\lambda'(z)$ yields a lower $E_z$ due to the image effects. The beam has been divided into only 32 slices which is few for the NGP interpolation, correspondingly the $\lambda'(z)$ model kicks appear step-like (the PIC algorithm in contrast uses one order higher interpolation).




(a) $\lambda'(z)$ model (incl. an $r_p = 5$ cm round vacuum tube boundary).




(b) 3D PIC model (free space).

Figure 1: Longitudinal kicks vs. position for 32 slices.

## *Transverse Gaussian Space Charge*

A frequently employed 2D space charge model for the transverse plane has been established by M. Bassetti and G.A. Erskine in 1980 [6]. They derived a computationally optimised analytical expression (the "Bassetti-Erskine" or B.E. formula) for the electric field of a two-dimensional Gaussian charge density function

$$\rho(x,y) = \frac{Nq}{2\pi\sigma_x\sigma_y}\exp\left(-\left(\frac{x^2}{2\sigma_x^2}+\frac{y^2}{2\sigma_y^2}\right)\right) \qquad (4)$$

which generates the electric fields [7]

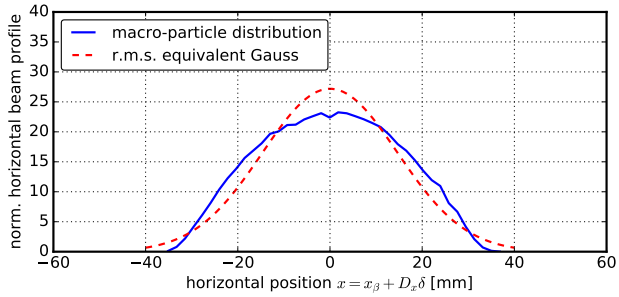
$$E_u = \frac{Q}{4\pi\epsilon_0}u\int_0^\infty dt\frac{\exp\left(-\frac{x^2}{2\sigma_x^2+t}-\frac{y^2}{2\sigma_y^2+t}\right)}{(\sigma_u^2+t)\sqrt{(\sigma_x^2+t)(\sigma_y^2+t)}} \qquad (5)$$

for $u = x, y$. For $\sigma_x > \sigma_y$, the B.E. formula reads

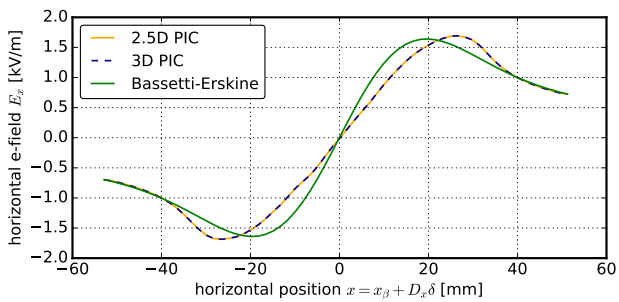
$$E_y + i\,E_x = \frac{Nq}{2\epsilon_0\sqrt{2\pi(\sigma_x^2-\sigma_y^2)}}\left[w\left(\frac{x+iy}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right)-\exp\left(-\frac{x^2}{2\sigma_x^2}-\frac{y^2}{2\sigma_y^2}\right)w\left(\frac{x\frac{\sigma_y}{\sigma_x}+iy\frac{\sigma_x}{\sigma_y}}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right)\right] \qquad (6)$$

involving the complex-valued Faddeeva function $w(x+iy)$. It belongs to the family of error functions and can be evaluated much faster than numerically solving the full integral (5) [8].

PyHEADTAIL applies this semi-analytic model slice by slice to the respective transverse distribution along the beam. If the actual distribution deviates from a Gaussian, the "Bassetti-Erskine" formula (6) only represents an approximation. This needs to be kept in mind especially in the presence of dispersion, when a non-Gaussian momentum distribution contributes to the transverse profile. Figure 2 depicts an $N = 2\times10^{11}$ SPS bunch with a very large longitudinal r.m.s. emittance of $\epsilon_z = 0.42$ eV s (at an RF bucket acceptance of 0.68 eV s) where the RF bucket non-linearities deform the matched momentum distribution. This reflects in the evidently non-Gaussian horizontal beam profile given a dispersion of $D_x = 7.96$ m and a horizontal normalised emittance of $\epsilon_x = 0.84$ mm mrad.




(a) Beam profile and Gaussian r.m.s. equivalent for central beam slice.




(b) Corresponding horizontal electric field from PIC and (Gaussian) Bassetti-Erskine models.

Figure 2: Large $\epsilon_z$ entail non-Gaussian momentum distributions which affect the horizontal distribution via dispersion.

## *Particle-in-cell Model*

PIC algorithms model space charge self-consistently [9]. In order to compute the kicks, the macro-particle distribution is first interpolated to nodes of a regular mesh (particle-to-mesh or "P2M" step), then the charge distribution on the mesh is solved for the potential and consequently the force (solve step), and finally the force is interpolated back to





ISBN 978-3-95450-178-6

the particles (mesh-to-particles or "M2P" step). A separate library has been developed named PyPIC to encapsulate the PIC algorithm.

In synchrotrons, the relative momenta between the particles are usually much smaller than $p_0$. This can be exploited when solving the Maxwell equations by Lorentz boosting to the beam rest frame, where we neglect the relative particle motion and correspondingly only have to deal with an electrostatic problem. Before the P2M step, we Lorentz boost from the co-moving laboratory frame $(x, y, z)_{\text{lab}}$ to the beam rest frame

$$(\tilde{x}, \tilde{y}, \tilde{z})_{\text{beam}} = (x, y, \gamma z)_{\text{lab}} \quad . \tag{7}$$

Correspondingly, the bunch becomes much longer, the mesh is then constructed in the beam frame. In PyPIC we have implemented a linear spatial Cloud-In-Cell (CIC) interpolation, i.e. the shape function of the macro-particles is a constant Heaviside step function over the diameter of a cell. With the resulting mesh charge density $\rho$, we are ready to solve the discrete version of the 2D or 3D Poisson equation,

$$\Delta\phi = -\frac{\rho}{\epsilon_0} \quad , \tag{8}$$

on the mesh for the mesh potential $\phi$.

To this end, we implemented several Poisson solvers in both 2.5D (i.e. slice-by-slice 2D transverse solving) and full 3D variants. The solvers cover finite difference (FD) approaches with direct matrix solving via QR or LU decomposition (with Dirichlet or arbitrary boundary conditions) and Green's function methods (free space or rectangular boundary conditions) exploiting the Fast Fourier Transform (FFT) algorithm.

The FD implementations construct a sparse Poisson matrix $A$ with a first-order nearest-neighbour stencil. In case of the LU decomposition $A = LU$, the sparse lower and upper triangle matrices $L, U$ are then precomputed at set-up. At each solve step, the linear matrix equation $LU\phi = -\rho/\epsilon_0$ is solved given the respective vector $\rho$ containing the mesh charge density of all nodes. This approach is extremely efficient on the CPU in conjunction with the KLU algorithm [10] if the Poisson matrix remains constant over many solve steps [11]. Therefore, e.g. matrix element indices for the boundary conditions should not change due to a differently shaped boundary, otherwise the LU decomposition needs to be recomputed which is computationally expensive. The QR decomposition is slower than the LU decomposition (cf. e.g. [12]) but numerically more stable than the LU decomposition, hence it serves as a reference. Finite difference equations always require boundary conditions which can be advantageous if indirect space charge effects from the vacuum tube need to be taken into account. If the transverse beam sizes are rather small compared to the vacuum tube, the mesh can become prohibitively large though. In this case, the following method may be more appropriate.

Another approach to solve Eq. (8) is to use the free space $D$=2 (for $\mathbf{x} = (\tilde{x}, \tilde{y})_{\text{beam}}$) resp. $D$=3 (for $\mathbf{x} = (\tilde{x}, \tilde{y}, \tilde{z})_{\text{beam}}$) Green's functions $G$,

$$\phi(\mathbf{x}) = \frac{1}{2^{D-1}\pi\epsilon_0} \int d^D\widehat{\mathbf{x}} \quad G\left(\widehat{\mathbf{x}} - \mathbf{x}\right) \rho\left(\widehat{\mathbf{x}}\right) \quad . \tag{9}$$

We apply Hockney's trick where the domain of the Green's function is doubled by cyclical expansion in each dimension [9]. The potential in this expanded region will be incorrect and discarded but the periodicity allows to make use of the computationally very effective FFT algorithm for the convolution. In principle, the Green's function needs to be evaluated on a square (cuboid) mesh with equal distances in all directions. In the transverse plane, aspect ratios may be large due to the betatron function ratio and additional dispersion effects. At least in the 3D case, the aspect ratio of the transverse with respect to the longitudinal plane will certainly be large and will thus require many mesh points to cover the whole distribution which may become computationally heavy. Therefore, we make use of the integrated Green's functions (IGF) $\widehat{G}$ ( [13, Eq. (56),(57)] for the 2D case and [14, Eq. (2)] for 3D), which conceptually include the aspect ratio into the discrete Green's function by integrating over each cell assuming $\rho$ to be constant across the cell. This approximation has to be kept in mind when choosing the mesh size. The Fourier transform of the IGF is computed at set-up and stored until the mesh is changed. At each solve step, it is multiplied by the Fourier transformed mesh charge density $\rho$ and the result inversely Fourier transformed to yield the potential,

$$\phi = \mathcal{F}^{-1}\left[\left(\mathcal{F}\widehat{G}\right) \cdot \left(\mathcal{F}\rho\right)\right] \quad . \tag{10}$$

The FFT convolution approach is much more efficient at a complexity of $O(n \log n)$ than computing the convolution integral in real space with $O(n^2)$, where $n$ the total number of mesh nodes. Using this method with the FFTW [15] implementation on the CPU is found to take less than twice as long as the KLU direct solving approach mentioned before [11].

After the potential on the mesh $\phi$ has been determined, the electric mesh fields $\tilde{\mathbf{E}}$ in the beam frame are calculated as

$$\tilde{\mathbf{E}} = -\nabla\phi \tag{11}$$

via a numerical first-order finite difference gradient implementation. Finally, the electric fields are interpolated back to the macro-particles (M2P) and Lorentz boosted back to the laboratory frame,

$$(E_x, E_y, E_z)_{\text{lab}} = \left(\gamma\tilde{E}_x, \gamma\tilde{E}_y, \tilde{E}_z\right)_{\text{beam}} \quad . \tag{12}$$

The Lorentz forces for each macro-particle include the magnetic fields arising when transforming to the laboratory frame,

$$(B_x, B_y, B_z)_{\text{lab}} = \left(-\beta E_y/c, \beta E_x/c, 0\right)_{\text{lab}} \tag{13}$$

$$\implies (F_x, F_y, F_z)_{\text{lab}} = q\left(\frac{\tilde{E}_x}{\gamma}, \frac{\tilde{E}_y}{\gamma}, \tilde{E}_z\right)_{\text{beam}} \quad . \tag{14}$$

ISBN 978-3-95450-178-6

To conserve the total charge, the interpolation functions and order at both the P2M as well as the M2P step necessarily need to match [9]. As a side note, the usual straight-forward interpolation functions can be the source of noise and grid heating effects in the traditional PIC approach. Recently, symplectic algorithms have been derived to solve these issues [16], which can be interesting for long-term simulations over many turns since the symplectic nature implies a finite bound on the energy error.

Figure 3 shows the PIC computed electric field of a coasting beam with a Kapchinsky-Vladimirsky (KV) distribution in the transverse plane for the SPS. For $\lambda = 5.1\,\mathrm{C/m}$ and beam edges at $r_x = 2.5\,\mathrm{mm}$ and $r_y = 1.6\,\mathrm{mm}$, the maximal electric field at the beam edge analytically gives

$$E_x = \frac{\lambda}{\pi\epsilon_0}\frac{1}{r_x + r_y} = 15.2\,\mathrm{kV/m} \quad , \tag{15}$$

which matches the result from the PIC algorithm.

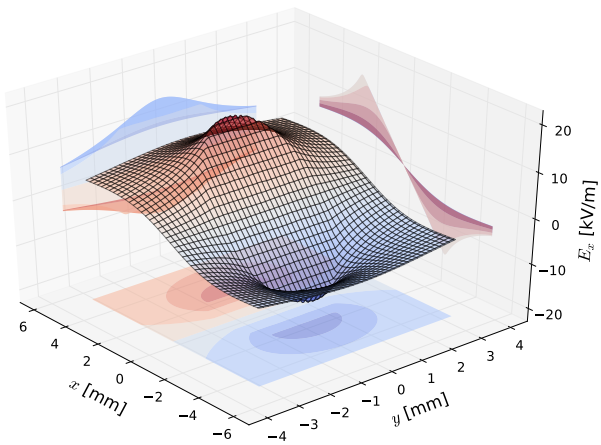



Figure 3: Horizontal electric field of a KV beam in the SPS.

## GPU HIGH-PERFORMANCE COMPUTING

PyHEADTAIL has been made available for graphics processor unit (GPU) high-performance computing. The library PyCUDA [17] provides an interface for GPU memory stored arrays that adopts the API of the standard Python library for scientific computing, NumPy, hence making large portions of the code easily applicable to both NumPy arrays and GPUArrays. To make the GPU usage in PyHEADTAIL as transparent and flexible as possible for new GPU users, a context management system to switch between GPU and CPU contexts has been developed [18]. Algorithms for the GPU need to make use of the pronounced parallel hardware structures and therefore often differ from serial CPU algorithms. The context managers switch between implemented algorithms e.g. for the bunch distribution statistics. For the GPU implementations it is therefore of crucial importance to have access to the underlying CUDA [3] API from Python which is fully provided by PyCUDA. Implementing and calling custom CUDA kernels is flexible and straight forward from the Python top layer. Besides the NumPy array API and the CUDA access, the third important ingredient to PyHEADTAIL on the GPU is the incorporation of powerful GPU computing libraries such as cuFFT [19], cuSOLVER [20], cuSPARSE [21] and Thrust [22]. We achieve this partly via the Python binding library scikit-cuda [23] and partly via self-implemented interfaces using ctypes.

For the PyHEADTAIL space charge suite we have developed a GPU version of PyPIC. The performance bottlenecks appear very differently during the aforementioned three particle-in-cell steps P2M, solve and M2P when comparing runtime profiles between the CPU and GPU versions. Figure 4 shows the fraction of time spent on both architectures during each step for quadratically increasing transverse mesh sizes given a fixed number of macro-particles. Comparing the cuFFT 3D Fourier transform performance to FFTW on a mesh of size $(16, 32, 64)$ gives a speed-up of up to $S = 35.8$, comparing to the standard NumPy FFT extension even reaches $S = 65.5$. This explains why the solve step with the free space FFT-based Green's function Poisson solver does not have such a significant impact on the overall timing during the particle-in-cell algorithm on the GPU, while it essentially marks the bottleneck on the CPU.

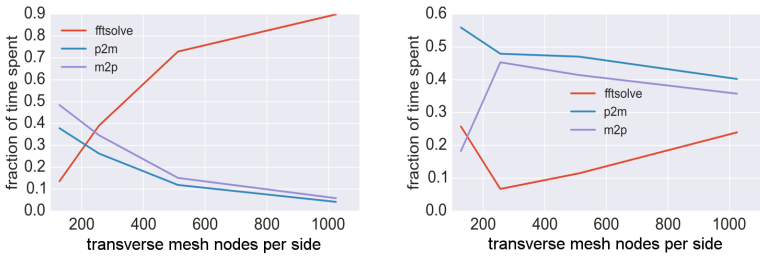
Effectively, the particle deposition on the mesh is the most performance critical part in the GPU PIC algorithm. We have implemented an atomic deposition algorithm, in which a CUDA thread for each particle is launched which locks the memory location of the respective mesh node charge from access by other threads, reads the memory value, adds to it and then stores the updated value. With this approach, we observed a rather slow performance as memory bank conflicts and thread stalls can happen for both the software-emulated 64-bit and the hardware-accelerated 32-bit `atomicAdd` variants. This finding is especially pronounced in the 3D case where each particle updates eight surrounding mesh nodes (instead of four in the 2D case). The problem decreased when using less macro-particles for a given mesh size – however, to achieve a good resolution for the electric fields, at least 10 macro-particles per cell are required [9].

To address this issue, we implemented a sorted deposition particle-in-cell algorithm described in [24]. In this approach, the macro-particle coordinate arrays are first sorted by their cell IDs, for which we used the Thrust library with its `sort_by_key` functions. Subsequently, a thread is launched for each cell which loops through the particles within this cell to construct guard cell charge densities. In a third step, a kernel merges the four (2D) resp. eight (3D) guard meshes to the final mesh charge density $\rho$ array. We achieved a speed-up of $S = 3.5$ for the mesh size $(64, 64, 32)$ and $1 \times 10^6$ macro-particles when comparing the sorted deposition to the double precision `atomicAdd` deposition. In addition, the M2P step profits from the sorted arrays since global GPU memory is accessed in a coalesced manner: the kernel call on unsorted arrays takes 25% longer than on sorted arrays. Further approaches to address the memory bank conflicts (such as using L1 caching) have been investigated e.g. for SYNERGIA in [25] and for ELEGANT in [26, 27].





ISBN 978-3-95450-178-6




(a) CPU implementation. (b) GPU implementation.

Figure 4: Timing proportions between the P2M, solve and M2P step for the FFT-based Poisson solver vs. number of mesh nodes per transverse side. The number of macro-particles is fixed to $5 \times 10^5$.

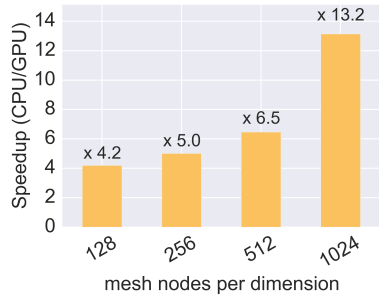



Figure 5: Overall 2.5D PIC speed-up achieved vs. number of mesh nodes per transverse side comparing a NVIDIA K40m GPU to a single 2.3GHz Intel Xeon E5-2630 (v1) CPU core.

For the 2.5D case, the transverse Poisson equations can be solved for all slices in parallel. Since the cuFFT calls for each slice work with small arrays compared to the GPU memory size, the cuFFT batch solving works very effectively. All in all, we achieved overall PIC speed-ups of up to $S = 13.2$ compared to the CPU. Figure 5 shows how the GPU usage becomes increasingly beneficial for larger mesh sizes at a fixed number of $5 \times 10^5$ macro-particles. Also increasing the number of macro-particles scales less than linearly for the relevant parameter range as opposed to the CPU. These results allow the GPU accelerated space charge simulations to access much higher resolutions and increase the validity of simulations (also over longer time scales).

## SPS BENCHMARK

Since strong space charge leads to transverse detuning, the resonance condition especially in the centre of a Gaussian bunch is shifted to higher tunes. Large-scale static tune scans with high-brightness single bunch beams at the injection plateau of the SPS revealed a significant influence of the $4Q_x = 81$ octupolar resonance [28]. Here, we deliberately drive this resonance with a single extraction octupole (`LOE.10402`) at $k_3 = 25\,\mathrm{m}^{-4}$ acting as a localised octupolar field error. The $N = (2.05 \pm 0.1) \times 10^{11}$ single bunches arrive from the upstream Proton Synchrotron with normalised transverse emittances $\epsilon_x = (0.84 \pm 0.05)$ mm mrad and $\epsilon_y = (1.06 \pm 0.04)$ mm mrad at an r.m.s. bunch length of $\sigma_\tau = (0.93 \pm 0.01)$ ns. The incoherent space charge tune spread of these bunches amounts to $\left(\Delta Q_x^{\mathrm{SC}}, \Delta Q_y^{\mathrm{SC}}\right) = (-0.09, -0.16)$. While fixing the coherent vertical tune $Q_y = 20.31$, we measure the transverse averaged emittance growth for horizontal tunes between $20.16 \leq Q_x \leq 20.30$ over a time span of 3 s in a set-up equivalent to [28]. For each working point, three consecutive shots per transverse plane are wire scanned to obtain the beam profiles and extract the respective normalised emittance via a Gaussian fit (subtracting the dispersion contribution in the horizontal plane). The orange curve in figure 6 shows the dependence of the averaged transverse emittance blow-up on $Q_x$. The $4Q_x = 81$ resonance causes a shifted significant emittance growth peak with its maximum at $Q_x = 20.28$.

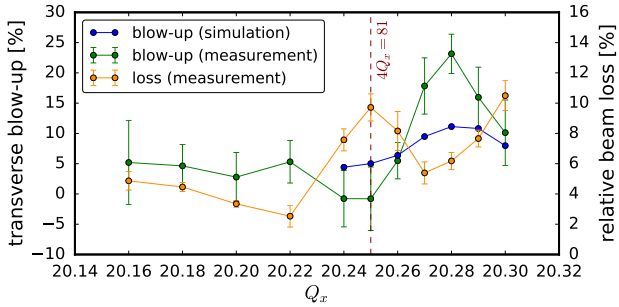



Figure 6: Transverse emittance growth vs. coherent horizontal tune for Bassetti-Erskine space charge simulations over $10 \times 10^3$ turns and measurements over $130 \times 10^3$ turns.

Corresponding simulations spanning 0.23 s cycle time with the PyHEADTAIL space charge models (including the non-linear model of the SPS [29]) have been set-up resolving the TWISS parameters and corresponding beam sizes around the SPS ring. The results plotted in blue also show the maximum blow-up at $Q_x = 20.28$. However, around $Q_x = 20.25$ itself we find emittance growth predicted by the simulations which is not observed in the measurements. Simulations for longer cycle times with a good loss model might recover this: the measurements show strong losses around $Q_x = 20.25$ as the beam halo is excited to large transverse amplitudes where the weaker beam self-fields lead to a resonance condition much closer to $Q_x = 20.25$. So far our simulations did not include the SPS impedance model which leads to significant vertical coherent detuning of $\Delta Q_y^{\mathrm{imped.}} = -0.03$ at $N = 2 \times 10^{11}$ [30].

## CONCLUSION

We have described the space charge models implemented in PyHEADTAIL. The GPU parallelisation strategies and achieved speed-ups of up to $S = 13.2$ for our self-consistent particle-in-cell space charge algorithms have been reported. These will play a major role in the on-going developments of the SPS model for high-brightness beams, for which a first benchmark has been presented in the last section.

ISBN 978-3-95450-178-6

## REFERENCES


[1] E. Metral *et al.*, "Beam Instabilities in Hadron Synchrotrons", in *IEEE Transactions on Nuclear Science*, vol. 63, no. 2, Apr. 2016, pp. 1001-1050.

[2] K.S.B. Li *et al.*, "Code development for Collective Effects", in *ICFA Advanced Beam Dynamics Workshop on High-Intensity and High-Brightness Hadron Beams (HB2016)*, Malmö, Sweden, July 2016, paper WEAM3X01, this conference.

[3] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable parallel programming with CUDA.", in *Queue*, vol. 6, no. 2, 2008, pp. 40-53.

[4] PyCOMPLETE, Python Collective Effects Library, Accelerator Beam Physics Group, CERN, Switzerland, 2016, `http://github.com/PyCOMPLETE/`.

[5] M. Reiser, "Theory and Design of Charged Particle Beams", John Wiley & Sons, Jun. 2008.

[6] M. Bassetti and G.A. Erskine, "Closed Expression for the Electrical Field of a Two-dimensional Gaussian Charge", in CERN-ISR-TH-80-06, CERN, Switzerland, 1980.

[7] H. Wiedemann, "Statistical and Collective Effects", in Particle Accelerator Physics, 3rd ed. New York: Springer, 2015, p. 644.

[8] A. Oeftiger *et al.*, "Review of CPU and GPU Faddeeva Implementations", in *Proc. 7th Int. Particle Accelerator Conf. (IPAC'16)*, Busan, Korea, May 2016, paper WEPOY044, pp. 3090-3093.

[9] R.W. Hockney and J.W. Eastwood, "Computer Simulation Using Particles", CRC Press, 1989.

[10] T.A. Davis and E. Palamadai Natarajan, "Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems", in *ACM Transactions on Mathematical Software*, vol. 37, no. 6, 2010, pp. 36:1-36:17.

[11] G. Iadarola, A. Axford, H. Bartosik, K. Li and G. Rumolo, "PyECLOUD for PyHEADTAIL: development work", presentation in Electron Cloud Meeting, May 14, 2015, `http://indico.cern.ch/event/394530/`.

[12] G.A. Geist and C.H. Romine, "LU Factorization Algorithms on Distributed-memory Multiprocessor Architectures", in *Siam. J. Scl. Stat. Comput.*, vol. 9, no. 4, 1988, pp. 639-649.

[13] J. Qiang, M.A. Furman and R.D. Ryne, "A Parallel Particle-in-cell Model for Beam–beam Interaction in High Energy Ring Colliders", in *Journal of Comp. Phys.*, vol. 198, no. 1, 2004, pp. 278-294.

[14] J. Qiang, S. Lidia, R.D. Ryne and C. Limborg-Deprey, "Erratum: Three-dimensional Quasistatic Model for High Brightness Beam Dynamics Simulation", in *Phys. Rev. ST Accel. Beams*, vol. 10, no. 12, Dec. 2007, p. 129901.

[15] M. Frigo and S.G. Johnson, "The Design and Implementation of FFTW3", in *Proceedings of the IEEE*, vol. 93, no. 2, 2005, pp. 216-231.

[16] H. Qin *et al.*, "Canonical Symplectic Particle-in-cell Method for Long-term Large-scale Simulations of the Vlasov-Maxwell System", arXiv:1503.08334v2 [physics.plasm-ph], 2015.

[17] A. Klöckner *et al.*, "PyCUDA and PyOpenCL: A Scripting-based Approach to GPU Run-time Code Generation", in *Parallel Computing*, vol. 38, no. 3, Mar. 2012, pp. 157-174. See also `http://documen.tician.de/pycuda/`.

[18] S. Hegglin, "Simulating Collective Effects on GPUs", MSc thesis, D-MATH/D-PHYS Dep., ETH Zürich, Zürich, Switzerland, 2016.

[19] cuFFT, Fast Fourier Transform library, CUDA Toolkit, NVIDIA, 2016, `https://developer.nvidia.com/cufft/`.

[20] cuSOLVER, collection of dense and sparse direct solvers, CUDA Toolkit, NVIDIA, 2016, `https://developer.nvidia.com/cusolver/`.

[21] cuSPARSE, Sparse Matrix library, CUDA Toolkit, NVIDIA, 2016, `https://developer.nvidia.com/cusparse/`.

[22] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA.", in *GPU computing gems Jade edition*, vol. 2, 2011, pp. 359-371.

[23] L. Givon, scikit-cuda, Python interface to CUDA device / runtime, cuBLAS, cuFFT and cuSOLVER, 2015, `http://scikit-cuda.readthedocs.io/`.

[24] K. Ahnert, D. Demidov and M. Mulansky, "Solving Ordinary Differential Equations on GPUs", in *Numerical Computations with GPUs*, Springer International Publishing, 2014, pp. 125-157.

[25] Q. Lu and J. Amundson, "Synergia CUDA: GPU-accelerated Accelerator Modeling Package", in Journal of Physics: Conference Series, vol. 513, no. 5, 2014, p. 052021.

[26] I.V. Pogorelov, K. Amyx, P. Messmer, "Accelerating Beam Dynamics Simulations with GPUs", in *Proc. Particle Accelerator Conf. (PAC 2011)*, New York, USA, May 2011, paper WEP164, pp. 1800-1802.

[27] K. Amyx *et al.*, "CUDA Kernel Design for GPU-based Beam Dynamics Simulations", in *Proc. 3th Int. Particle Accelerator Conf. (IPAC'12)*, New Orleans, USA, May 2012, paper MOPPC089, pp. 343-345.

[28] H. Bartosik, A. Oeftiger, F. Schmidt and M. Titze, "Space Charge Studies with High Intensity Single Bunch Beams in the CERN SPS", in *Proc. 7th Int. Particle Accelerator Conf. (IPAC'16)*, Busan, Korea, May 2016, paper MOPOR021, pp. 644-647.

[29] H. Bartosik, A. Oeftiger, M. Schenk, F. Schmidt and M. Titze, "Improved Methods for the Measurement and Simulation of the CERN SPS Non-linear Optics", in *Proc. 7th Int. Particle Accelerator Conf. (IPAC'16)*, Busan, Korea, May 2016, paper THPMR036, pp. 3464-3467.

[30] H. Bartosik *et al.*,"TMCI Thresholds for LHC Single Bunches in the CERN-SPS and Comparison with Simulations", in *Proc. 5th Int. Particle Accelerator Conf. (IPAC'14)*, Dresden, Germany, May 2014, paper TUPME026, pp. 1407-1409.






ISBN 978-3-95450-178-6