FUNCTION TIMING AND OPTIMIZATION: NUMBA IMPLEMENTATIONS FOR THE BLOND CODE

Helena Perović, *University of Montenegro*
Supervisors: Konstantinos Iliakis, Danilo Quartullo

Abstract

This report delves into implementing Numba functions and examines their efficiency. The focus is on evaluating Numba implementations as potential replacements for C++ functions within the BLonD Code. By leveraging Numba's just-in-time compilation for Python, we analyze performance gains and consider the feasibility of transitioning from established C++ functions. Through meticulous function timing, profiling, and benchmarking, this project aims to inform decisions regarding performance optimization in scientific computing. The results offer insights into the interplay between ease of implementation and computational performance. Notably, the speedup factor of C++ varies from 0.9 (sometimes slower than Numba) to 1.6, demonstrating the versatility and competitive nature of Numba in optimizing code execution.

INTRODUCTION

The BLonD code

The Beam Longitudinal Dynamics code [1, 2], BLonD[1], has been developed at CERN since 2014 and has become a central tool for longitudinal beam dynamics simulations. BLonD is furthermore a well-tested and optimized simulation suite, which is demonstrated through examples, too. Some of the most important applications of the BLonD code include controlled longitudinal emittance blow-up during the ramp (Fig. 1), SPS impedance identification (Fig. 2), PS-SPS transfer (Fig. 3), and beam feedbacks (Fig. 4).

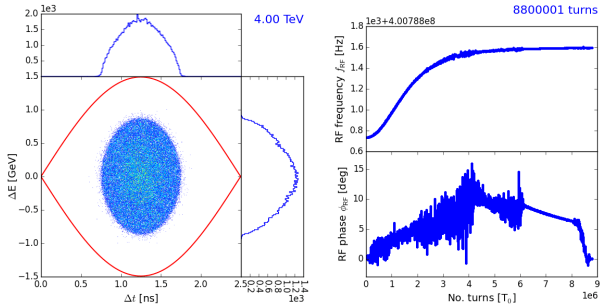

Figure 1: *Simulated longitudinal bunch distribution at LHC flat-top (left), RF frequency (top right) and phase (bottom right) during acceleration of a proton bunch in the LHC. The beam-phase loop cancels part of the injected RF phase noise through corrections on the RF frequency. The synchronisation loop keeps the RF frequency close to the programmed value.*

[1] The BLonD repository on GitLab

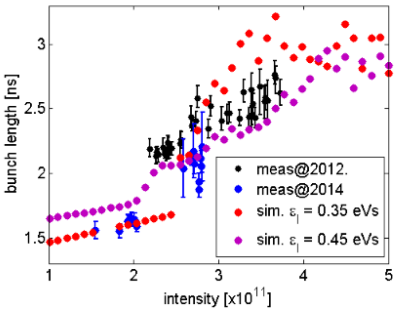

Figure 2: *Bunch lengthening due to microwave instability in the SPS. Comparison of measurement and simulation results in the Q20 optics.*

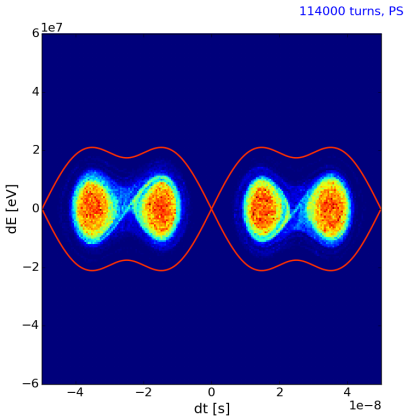

Figure 3: *Simulated bunch distributions during the second splitting in the bunch-to-bucket transfer between the PS and the SPS for LHC-type 25 ns beams. After two double splittings in the PS and a bunch rotation at PS flat-top, the bunch is injected and accelerated in the SPS.*

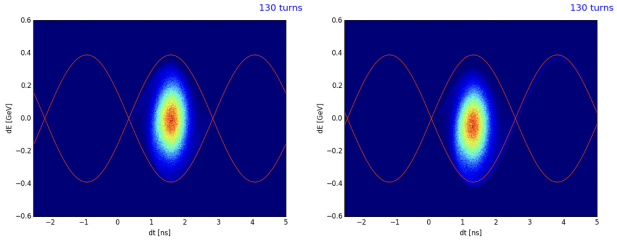

Figure 4: *Simulated bunch distributions during the correction of a phase injection error with only the phase loop (left) and with both phase and synchronisation loops (right).*

Motivation

The BLonD code is written using both C++ and Python, so the question that arises is *Could the BLonD code be rewritten by replacing C++ with Numba?*
The motivation behind doing so is profound. Foremost among these reasons is obtaining a uniform code, fostering collaborative efforts and simplifying code maintenance. This strategic move would streamline development processes, ensuring coherent understanding across diverse users of the code. Equally pivotal to our motivation is the fact that there would be no need for a different compiler. This addresses the persistent challenges associated with installation and compatibility arising from different compiler configurations. In addition, a downside to having code written in C++ is that it is harder to maintain on different platforms (Linux, Windows, MacOS). On the other hand, regarding the installation process, Numba is rather beneficial. It comes pre-installed with the Anaconda Python distribution and is also easy to install using the pip command. In conclusion, the primary motivation and impetus for undertaking this project are clearly evident.

NUMBA IMPLEMENTATIONS

To better understand the project, it is necessary to introduce Numba. Numba is an open-source just-in-time (JIT) compiler for Python that is used to accelerate the execution of numerical computations and data-intensive tasks. The many advantageous features of Numba are that it is best used with NumPy arrays and functions, it automatically translates some loops into vector instructions and it is great when using Jupyter notebooks for interactive computing. Furthermore, the aspect of Numba that is the most significant for the purpose of this project is that it can approach the speeds of C++.
As concerns the implementation process in the BLonD code, the first step of creating the Numba functions was simple as it solely consisted of adding the *@jit* decorator to already existing implementations in Python. The main functions that were implemented using Numba are:
- **RF kick**, which provides the energy kick to all the macro-particles in a bunch;
- **RF voltage computation**, which is performed only in correspondence of the bins centers of the slicing object;
- **linear interpolation kick**, which is used in combination with the RF voltage computation function to provide the energy kicks to all the macro-particles in a bunch;
- **drift equation of motion**, which is applied to all the macro-particles in a bunch;
- **slicing (projection) of the bunch distribution along the longitudinal coordinate**;
- **synchrotron radiation (with or without quantum excitation)**, which should be considered for instance when dealing with LHC beam-dynamics simulations;
- **(fast) beam phase**, which computes the beam-cavity phase, needed as input for the beam phase loop.

Scripts for profiling

After the Numba implementations have been made, the step that followed was to compare the runtime of funtions using C++, Python and Numba. This was done by creating simple scripts that would call all the functions in a loop, with the function parameters extracted from an external file. Over the course of the benchmarking, changes were bound to be made. Specifically, first testings were done locally, making the results dependent on the computer used for the benchmarks. By switching to testing on the cluster node, more objective timings were obtained. Next, to arrive at more accurate results, scripts with independent functions used for benchmarking were replaced by more complex scripts needed to perform realistic simulations. The transition to files containing real-world simulation parameters allowed for the derivation of timings that closely mirrored actual scenarios. Furthermore, the most computationally-expensive functions were optimized with Numba, and this allowed to have performances close to the ones obtained with C++. These functions were optimized by unrolling loops and modifying the algorithms, particularly in the functions kick, drift, RF voltage computation, linear interpolation kick, and slice beam. Lastly, the systematic testing of various input sizes not only led to a more extensive dataset for analysis but also resulted in heightened precision when evaluating performance outcomes.

PRELIMINARY BENCHMARKS

The first results of the benchmarks between Python, Numba and C++ are provided in Fig. 5. The most important parameters used for these tests are:
- number of macro-particles: 10^6, *used for all following simulations*
- number of bunches: 1
- number of slices: 512
- number of turns: 100
- number of RF stations: 2

In Fig. 5, the annotations with color have the following meaning: red signifies Numba's slower performance compared to C++, orange and yellow indicate comparable runtimes, and green highlights the cases where Numba outperforms C++. As one can see, Numba demonstrates significantly better performance than C++ in nearly half of the functions, reflecting a noteworthy achievement. Moreover, Numba's consistent out-performance over Python across all functions stands as a substantial accomplishment and it strongly indicates that the Numba optimized functions can replace the Python ones.

function	c++	numba	python
kick	0.895	1.443	1.492
simple drift	0.027	0.044	0.048
exact drift	0.151	0.155	0.603
rf voltage computation	0.002	0.002	0.003
linear interpolation kick	0.242	0.119	213.712
slice beam	0.131	0.119	0.55
slice smooth	0.415	0.432	84.539
synchrotron radiation	0.024	0.047	0.096
full synchrotron radiation	2.805	2.073	3.118
beam phase	0.004	0.002	0.013
fast beam phase	0.002	0.001	0.003

Figure 5: *Total execution times (in seconds) obtained from performing the first benchmark of the C++, Numba and Python functions.*

The bar plot depicted in Fig. 6 provides a clear visualization of the data reported in Fig. 5. The plot demonstrates the higher speed-up of the C++ and Numba implementations over the Python ones. From these observations, these noteworthy conclusions emerge: given the well-optimized nature of the linear interpolation kick and slice smooth functions in Numba, their performance converges closely to that of C++, while the kick function requires further optimization. It is worth mentioning that the codes for the kick function and the simple drift function have undergone changes since these simulations, resulting in a considerable enhancement in serial performance.

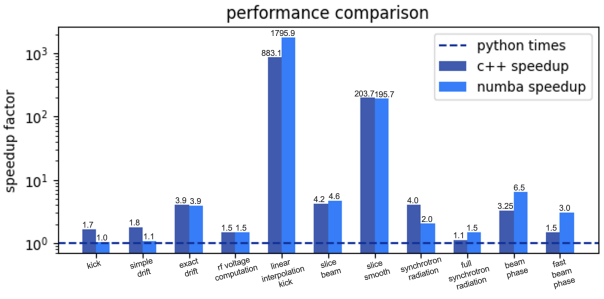

Figure 6: *The functions implemented in C++, Numba and Python, using as data the ones reported in Fig. 5. The timings are normalized to the the Python ones. For a better visualization, a logarithmic y-scale is used.*

REALISTIC BENCHMARKS

For the profiling of complete main-files used for realistic beam dynamics simulations, the SPS accelerator was considered as example. The simulations for the SPS, usually characterized by the utilization of many different functions and also by their high computational cost, presented an optimal test-bed for benchmarking. An initial imperative was to validate the consistency of output across C++, Python and Numba implementations, regardless of the language used. This verification is demonstrated in Fig. 7.

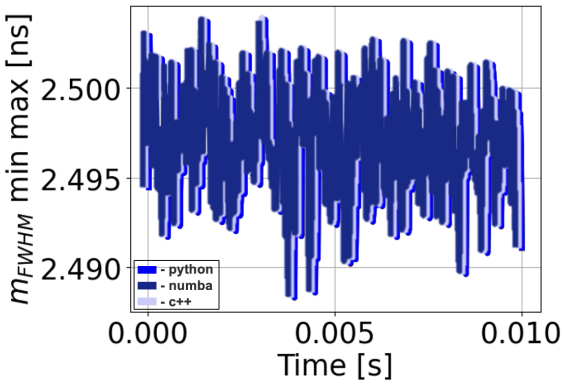

Figure 7: *Mean of the longitudinal bunch-profile as a function of the simulated time, obtained by performing a single-bunch simulation at SPS flat-bottom with the C++, Python and Numba implementations of the BLonD code.*

Initiating the testing process involved executing a main file while systematically modifying parameters. Among the large number of available parameters, six variables were singled out for manipulation. Three of these parameters were numerical, i.e. the number of bunches, the number of slices per bucket, and the number of turns to be considered for the multi-turn wake when collective effects were active. The other three parameters were Boolean, and they concerned the inclusion of collective effects, the activation of beam-based loops, and the use of the approximation of computing the RF kick only at the bins centers.
The number of bunches could be either 1, 72 (1 batch), or 288 (4 batches, typical of nominal LHC-type beams in the SPS). The number of slices per RF bucket varied from 128 to 512, since values in this range allow in general to sample large enough frequencies of the SPS impedance model, while keeping relatively low the numerical noise due to the finite number of macro-particles per bunch (one million). As concerns the number of turns considered for the multi-turn wake, values varied from 1 (i.e. no multi-turn wake, which is an approximation) to 50 (realistic value). Moreover, the number of protons per bunch could be either 0 (no collective effects) or 10^{11} (with collective effects). Within this extensive array of parameter combinations, three specific simulations are described below. These selected simulations cover a representative range of parameter values, thus they allow to assess the performance of the different code implementations across varied conditions. Firstly, detailed evaluations of the serial implementations of the functions are reported. Then, following a meticulous assessment and analysis of the achieved outcomes, the benchmarks performed with the parallel versions of the functions are also described.

Serial implementations of the functions

As just stated, three simulations are highlighted here, each characterized by distinct parameter sets. As concerns Python, however, only one simulation is taken into account, since the initial benchmark described above clearly revealed

Python's significantly slower performance compared to C++ and Numba.

First benchmark The first simulation, whose timings are shown in Fig. 8, uses this set of parameters:
- number of bunches: 1
- number of real particles: 10^{11}, *collective effects - on*
- slices per bucket: 128
- kick at slices: False
- beam phase: True
- multi-turn wake: 1

The first and most obvious conclusion is that Python can be more than 100 times slower than both C++ and Numba, which justifies the decision to omit testing it in the following simulations. Performance-wise, Numba proved to be faster than C++ in almost all the optimised functions mentioned in Section *Scripts for profiling*. The only function that did not surpass the Python execution times is *kick*, suggesting that this function needs further optimizations in Numba. Considering *slice beam*, Numba demonstrates remarkable efficiency, clocking in at twice the speed of its C++ counterpart, establishing its well-done optimization. The best result of Numba lies in the *linear interpolation kick*, delivering a staggering over 1000-fold acceleration in runtime compared to its Python counterpart. Overall, the conclusion of this first realistic benchmark is that the total runtime of Numba-implemented functions is only around 11% slower than that of their C++ implementations.

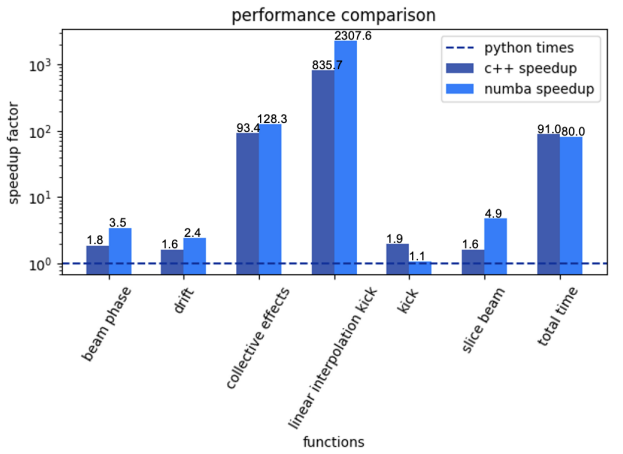

Figure 8: *First realistic benchmark of functions implemented in serial using C++, Python or Numba. The speed-ups are normalized to the Python timings. The collective effects include the linear interpolation kick when the parameter 'kick at slices' is false, otherwise, if this parameter is set to true, only the induced voltage sum is included.*

Figures 9, 10 and 11 show pie charts which highlight the functions that consume the majority of time in the context of the Python, Numba, and C++ implementations.

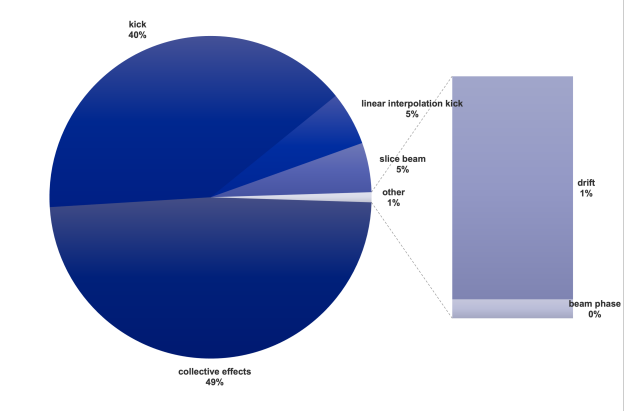

Figure 9: *Pie chart depicting the proportion of time spent by each function in C++ for the first realistic benchmark in serial.*

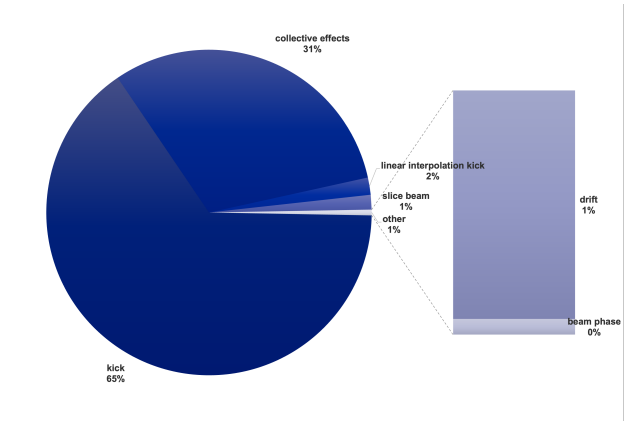

Figure 10: *Pie chart illustrating the proportion of time spent by each function in Numba for the first realistic benchmark in serial.*

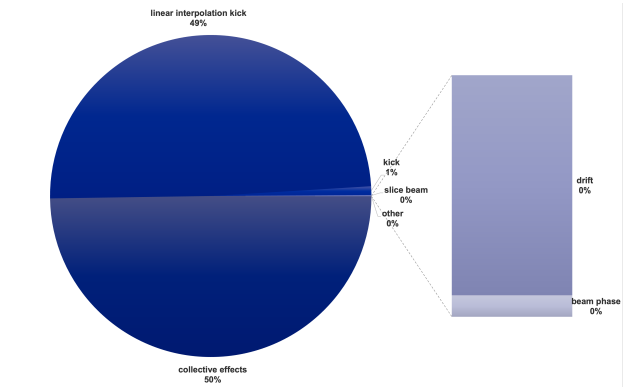

Figure 11: *Pie chart showing the proportion of time spent by each function in Python for the first realistic benchmark in serial.*

Examining the three pie charts, one can see that the *kick* and *collective effects* functions are the most time-expensive, indicating that these routines require particular attention for

further optimization. Notably, the faster execution of the *collective effects* in Numba compared to C++ is a significant achievement. Conversely, the Numba implementation of the *kick* function shows almost no improvement over the Python version, underlining that further optimization efforts are needed for this routine.

Second benchmark For the second profiling, whose results are shown in Fig. 12, the following set of parameters was used:
- number of bunches: 72
- number of real particles: 10^{11}, *collective effects - on*
- slices per bucket: 512
- kick at slices: True
- beam phase: False
- multi-turn wake: 1

The results obtained from this second benchmark clearly indicate the better performance of the C++ routines with respect to the Numba ones, since the C++ functions are in general almost factor-two faster. This finding is surprising and it needs to be studied further, as it shows that a more challenging simulation can lead to significant differences in performance of the C++ and Numba implementations. Another surprising observation is that the function to compute *collective effects* in Numba continues to outpace its C++ counterpart.

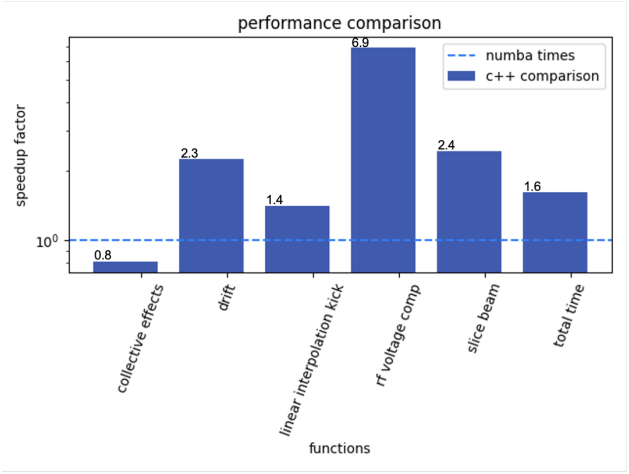

Figure 12: *Second realistic benchmark in serial, comparing C++ and Numba runtimes of functions, normalized to Numba timings.*

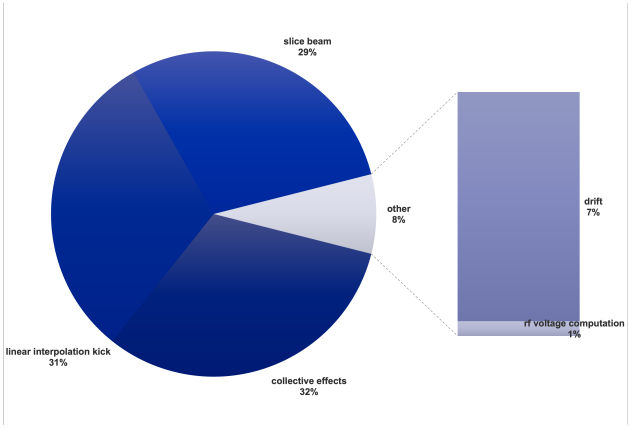

Figure 13: *Pie chart depicting the proportion of time consumed by each function in C++ for the second realistic benchmark in serial.*

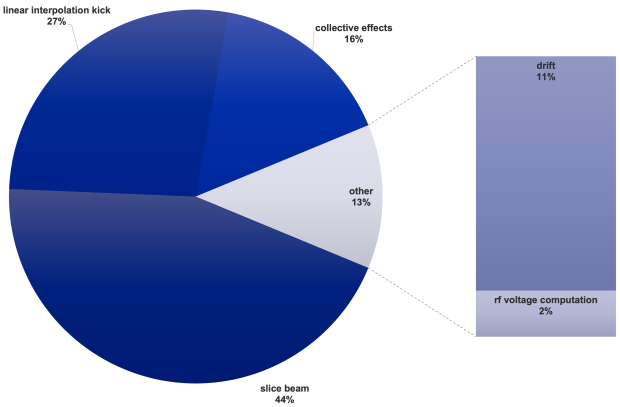

Figure 14: *Pie chart depicting the proportion of time consumed by each function in Numba for the second realistic benchmark in serial.*

The two pie charts in Figs. 13 and 14 highlight the functions which are the most computationally expensive, and which therefore require particular attention. These functions are the *collective effects*, *linear interpolation kick*, and *slice beam* ones. Despite having optimized the last two functions in Numba, their performance is lower than the one of the C++ implementations, signaling potential areas for further optimizations.

Third benchmark For the third realistic profiling, whose results are shown in Fig. 15, the following parameters were assumed:
- number of bunches: 288
- number of real particles: 0, *collective effects - off*
- slices per bucket: 512
- kick at slices: True
- beam phase: True

This third realistic benchmark yields more positive and expected timings for Numba, which align better with anticipated outcomes. Notably, this simulation excludes *collective*

effects, resulting in reduced time consumption. The total execution time for C++ is only slightly smaller than the one obtained with Numba. Among individual functions, the *linear interpolation kick* and *slice beam* functions are the most time-intensive ones, as the pie charts in Figs. 16 and 17 show. The comparison of the runtimes indicates that the *linear interpolation kick* function is faster in Numba than in C++, suggesting that this routine has been well optimized in Numba, as already mentioned multiple times. The run times for the *slice beam* function are similar in C++ and Numba. It is worth mentioning that the *beam phase* and *RF voltage computation* functions, which exhibit approximately threefold delay in Numba compared to C++, collectively contribute to only about 1% of the total time, indicating the negligible impact of this significant time disparity.

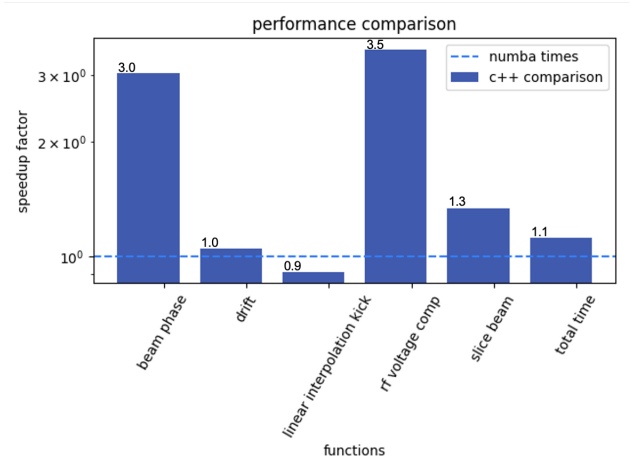

Figure 15: *Third realistic benchmark in serial, comparing C++ and Numba runtimes of functions, normalized to Numba timings.*

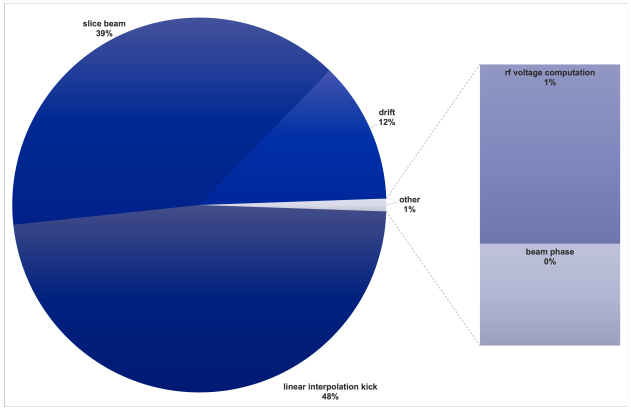

Figure 16: *Pie chart depicting the proportion of time consumed by each function in C++ for the third realistic benchmark in serial.*

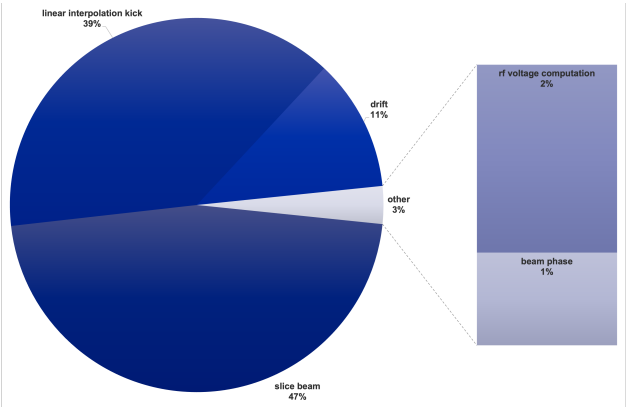

Figure 17: *Pie chart depicting the proportion of time consumed by each function in Numba for the third realistic benchmark in serial.*

Parallel implementations of the functions

Attention now turns towards the simulation outcomes obtained with the *Parallel configuration*. Everything stated in Section *Serial implementations of the functions* remains applicable also here, and the following three benchmarks mirror the ones performed with the *Serial setting*. One important difference as concerns the C++ implementations is that the BLonD library was compiled using the `-parallel` flag, while the Numba implementations were modified using the `parallel` attribute. Furthermore, certain routines underwent adjustments to ensure optimal parallel functionality.

First benchmark The first simulation (Fig. 18) confirms that the same functions consuming the most time in the *Serial simulations* also exhibit similar behavior in the *Parallel configuration*. Just as observed in the *Serial testing*, the *linear interpolation kick* and *collective effects* functions in Numba continue to perform better than their implementations in C++, indicating a consistent trend. This spotlight on Numba's performance leaves room for exploring enhancements in other functions. Notably, the *kick* function demands particular attention, as the parallel Numba version lags over four times behind the C++ version, even with parallel optimization efforts. Other functions collectively account for less than 4% of the total timing, indicating that the inferior performance of Numba does not have a significant impact. In summary, the conclusions drawn from the *Parallel simulations* align well with those obtained with the *Serial simulations*.

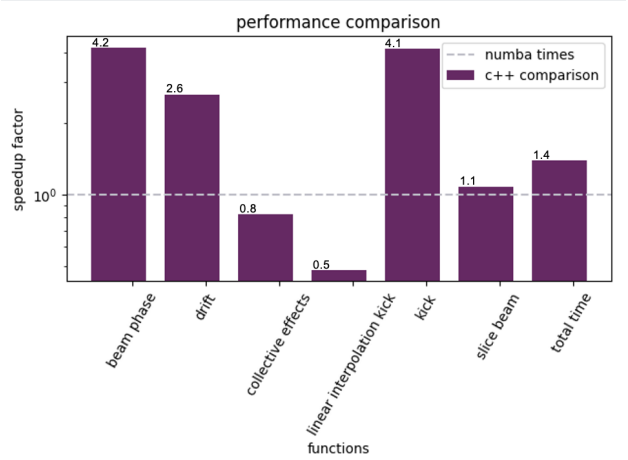

Figure 18: *First benchmark in parallel, comparing C++ and Numba runtimes of funcions, normalized to Numba timings.*

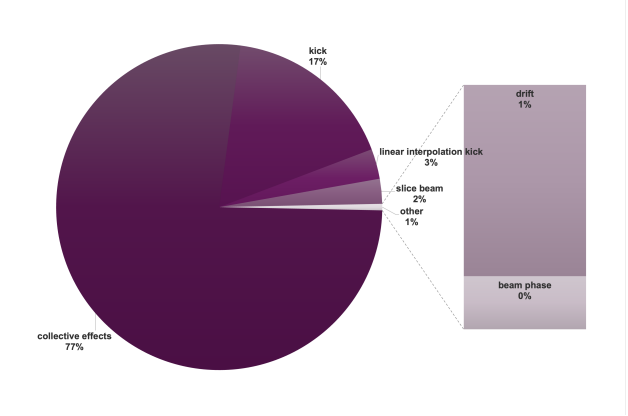

Figure 19: *Pie chart depicting the proportion of time consumed by each function in C++ for the first benchmark in parallel.*

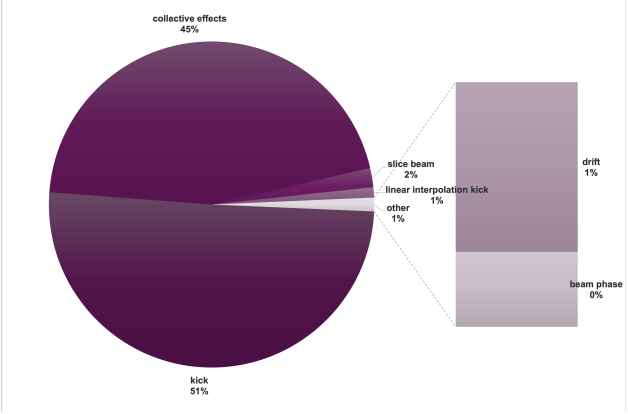

Figure 20: *Pie chart depicting the proportion of time consumed by each function in Numba for the first benchmark in parallel.*

Second benchmark The second simulation (Fig. 21) provides us with rather interesting results. Whereas in serial testing, almost every function in Numba performed worse than the C++ counterparts–with the only exception being *collective effects*–here the situation is quite the opposite, even resulting in Numba having the faster overall runtime.

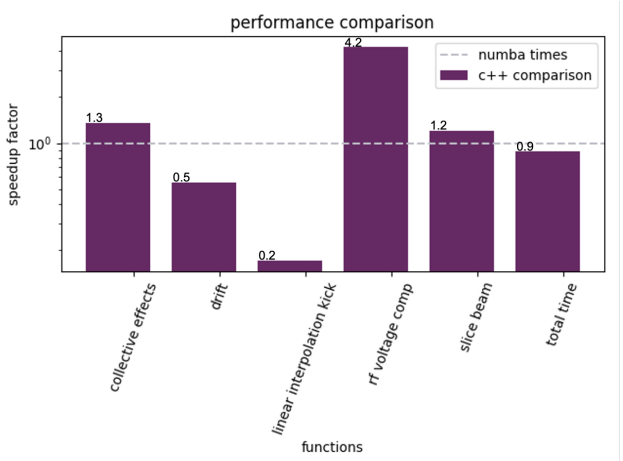

Figure 21: *Second benchmark in parallel, comparing C++ and Numba runtimes of funcions, normalized to Numba timings.*

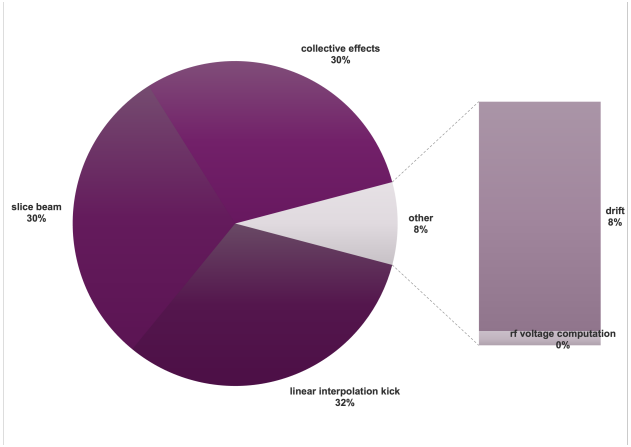

Figure 22: *Pie chart depicting the proportion of time consumed by each function in C++ for the second benchmark in parallel.*

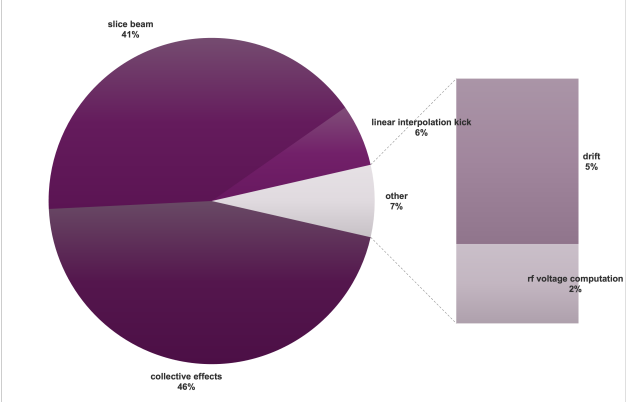

Figure 23: *Pie chart depicting the proportion of time consumed by each function in Numba for the second benchmark in parallel.*

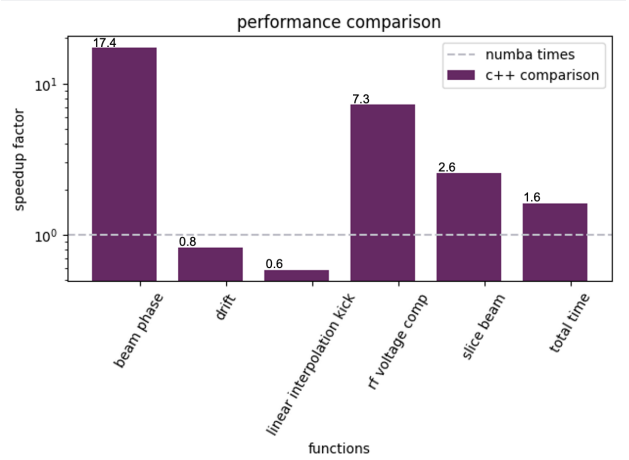

Figure 24: *Third benchmark in parallel, comparing C++ and Numba runtimes of funcions, normalized to Numba timings.*

The improvements applied to the *slice beam*, *drift*, and *linear interpolation kick* functions are evident, holding promising implications for Numba's future advancements. The positive aspect is that these functions, which yield favorable outcomes for Numba, are also the most time consuming, as the pie charts in Figs. 22 and 23 show. In summary, this simulation highlights the value of investing effort into parallel version optimization, hinting at the potential for the Numba functions to match, if not surpass, the performance of their C++ counterparts.

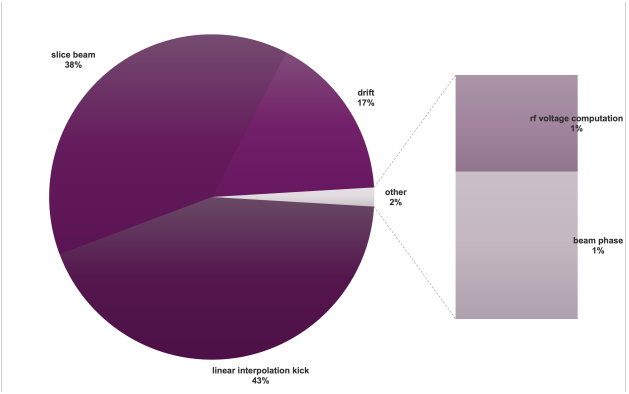

Figure 25: *Pie chart depicting the proportion of time consumed by each function in C++ for the third benchmark in parallel.*

Third benchmark The results obtained from the third simulation (Fig. 24) are encouraging. The performance comparison closely aligns with the serial version, though with a narrower timing gap between C++ and Numba–showcasing a linear improvement. Notably, all Numba functions demonstrate a relative improvement, with the *drift* function even outperforming its C++ counterpart. Surprisingly, this simulation stands out as one of the few instances where the distribution of total times between C++ and Numba is not entirely consistent (Figs. 25 and 26). A change in the functions consuming the most time adds a layer of complexity to the comparison.

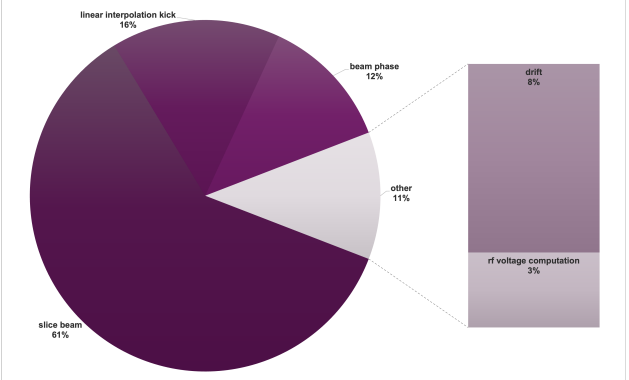

Figure 26: *Pie chart depicting the proportion of time consumed by each function in Numba for the third benchmark in parallel.*

Comparisons between the Serial and Parallel implementations

With the comprehensive analysis reported above, both the Serial and Parallel simulations have been thoroughly examined, providing in-depth insights. To better understand the substantial impact of transitioning from the *Serial* to the *Parallel configuration*, the bar plot shown in Fig. 27 highlights the extent of the performance improvement achieved.

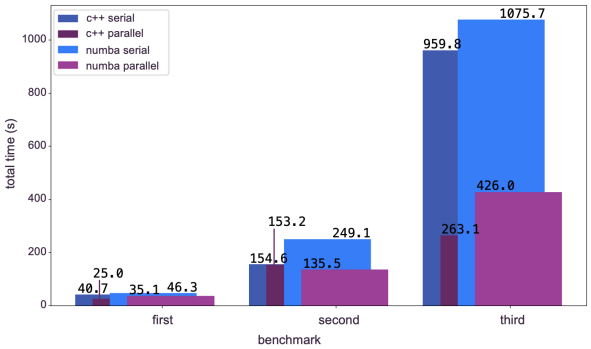

Figure 27: *Bar plot showing the total times (in seconds) of the simulations carried out with C++ or Numba, using either the serial or the parallel configurations. The simulations are grouped according to the three performed benchmarks.*

The bar plot shows that the simulations in parallel are always faster than the ones in serial. This is particularly evident for the Numba parallel implementations, which can approach the efficiency of the C++ ones. Notably, the most significant enhancement is observed in the second benchmark, where Numba even outpaces the performance of the C++ version. This achievement holds remarkable significance, as the second simulation is the most complex and challenging, in terms of parameters. In addition, this simulation closely resembles scenarios which the code will be used for, therefore Numba's higher performance is particularly important in this case.

CONCLUSIONS AND OUTLOOK

As a conclusion to this work, it is important to report here the most significant observations regarding Numba.

Loop Unrolling for Performance Maximization: It becomes evident that for optimal performance enhancement, the mere inclusion of the *@jit* decorator is not sufficient. Employing loop unrolling is essential to attain heightened efficiency.

Unveiling the Parallel Version Potential: While a parallel version has been initiated, it is evident that further refinements are necessary to fully harness its potential. Indeed, it is important to acknowledge that the potency of the parallel version depends significantly on the thorough modification of functions. If the routines are not appropriately tailored for parallelization, there is a risk of experiencing even slower performance. This underlines the critical importance of optimizing each function to fully realize the capabilities of the parallel version.

Comparable Performance to C++: Surprisingly, Numba's performance closely mirrors that of C++, especially in some functions. While Numba still needs some additional optimization to bridge the gap, its ability to potentially match the speed of C++ is evident. This suggests that with a bit more of fine-tuning, Numba holds the promise of achieving similar speeds as its C++ counterpart.

Potential Default Backend: Numba emerges as a viable contender for serving as the default backend for BLonD, especially with the parallel version, which is, ultimately, the version that will be used.

Promising results: The obtained results pave the way for a positive trajectory, laying the groundwork for promising future developments. Furthermore, in the context of comparing Numba with both C++ and Python, it is evident that Numba stands as a superior alternative to Python. It becomes clear that, at the very least, substituting the Python portion of the code with Numba could lead to substantial enhancements.
Furthermore, it is worth noting that, for the time being, C++ remains remarkably fast and remains the optimal choice. However, as Numba continues to evolve and improve, it is worth considering the potential for Numba to eventually match, or even surpass, C++ as a preferred option.

Looking ahead, the focus should shift to an ambitious agenda of advancing benchmarking endeavors. The strategy entails a meticulous, function-by-function optimization to achieve timings that surpass current standards, with the ambitious goal of surpassing the swiftness of C++ in certain cases. A crucial spotlight is cast on optimizing the parallel version. This choice is driven by its prevalent use and heightened importance. Encouragingly, initial results already showcase the parallel version's ability to outperform its serial counterparts. It is worth noting that even in its raw state, Numba's remarkable speed is evident. This enduring rapidity highlights the fact that Numba consistently delivers remarkable performance, regardless of optimization efforts. These forward-looking prospects chart a course where Numba continues to redefine performance standards, setting the bar higher for computational efficiency.

ACKNOWLEDGEMENTS

Konstantinos and Danilo: I want to express my sincere gratitude for being my supervisors. Your constant willingness to help, your patience and understanding, and your clear guidance have been truly exceptional. From walking me through each step to ensuring I was adapting well, your support has been a cornerstone of my journey here. Thank

you for your dedicated mentorship and for making this experience immensely valuable.

Helga: Amidst your demanding schedule, I am deeply grateful for your willingness to dedicate time to enhance my understanding of the BLonD code and delve into the underlying physics. I am incredibly thankful for the opportunity to learn from your expertise.

Bernardo: I want to extend my heartfelt thanks for your consistent support and guidance throughout my time here. Your constant availability for help and your genuine care for my experience have meant a lot to me. Your efforts in making sure I not only managed my work but also enjoyed my time here have truly made a difference, and I am incredibly grateful for your kindness.

Matis, Mariangela and Leonard: My sincere thanks go out to my colleagues whom I shared the office with. The ambiance in our workspace was nothing short of inspiring, striking a harmonious blend between focused work and moments of ease. This unique balance provided a genuine source of motivation for me to pour my best efforts into the project.

APPENDIX

In this appendix, you will find details about my BLonD repository and the work I have contributed to it during the project.

During the course of my project, I established my own personal copy of the BLonD repository on GitLab. This dedicated repository served as a collaborative workspace where I could commit, push, and maintain my project-related work. While the global BLonD repository exists, my personal repository allowed me to manage my contributions, track changes, and ensure that my work was accessible to fellow collaborators and the broader community.

I am grateful for the opportunity to have worked on the BLonD code and to have been part of its collaborative development efforts. My personal repository can be accessed at the following link, providing insight into my contributions and the progress made during the project: `https://gitlab.cern.ch/hperovic/BLonD`

REFERENCES

[1] *CERN Beam Longitudinal Dynamics code BLonD*. `http://blond.web.cern.ch`.

[2] Helga Timko et al. *Beam Longitudinal Dynamics Simulation Suite BLonD*. 2022. arXiv: `2206.08148 [physics.acc-ph]`.

[3] Danilo Quartullo. “Simulations of RF beam manipulations including intensity effects for CERN PSB and SPS upgrades”. PhD thesis. Istituto Nazionale di Fisica Nucleare (INFN) and "Sapienza" Universita di Roma, 2019. URL: `http://blond.web.cern.ch`.

[4] Konstantinos Iliakis et al. “Enabling Large Scale Simulations for Particle Accelerators”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.12 (2022), pp. 4425–4439. DOI: `10.1109/TPDS.2022.3192707`. URL: `https://ieeexplore.ieee.org/abstract/document/9834128`.

[5] *The Numba library: Notebook tutorial (with BLonD examples)*. `https://indico.cern.ch/event/1208416/`.