# ACCELERATING THE PARTICLE-IN-CELL METHOD OF PLASMA AND PARTICLE BEAM SIMULATION USING CUDA TOOLS

**I.S. Kadochnikov**[1,2]

[1] *Joint Institute for Nuclear Research, 6 Joliot-Curie St, Dubna, Moscow Region, 141980, Russia*

[2] *Plekhanov Russian University of Economics, Stremyanny lane, 36, Moscow, 117997, Russia*

E-mail: kadivas@jinr.ru

Numerical simulation of charged particle dynamics is required for the development of electron beam ion sources (EBIS). Special attention should be paid to the formation and evolution of different instabilities that can manifest in such sources. Understanding these processes will allow the efficiency of ion sources to be improved. The simulation of two-stream instability emerging between slow ions and fast electrons is particularly important. To perform the required simulation, "ef" and "ef_python" applications are under development. In these applications simulation is performed using the particle-in-cell (PIC) method, which contains four main parts: a particle mover, a particle-to-grid scatter, a field solver, field-to-particle interpolation. Each part's performance depends on the number of particles, as well as on the granularity of the simulation spatial mesh. The field solver usually represents a major portion of computational difficulty. This report describes the efforts to improve the performance of the ef_python application using CuPy, AMGX and PyAMG libraries. With minimal changes in the source code, CuPy allowed us to port all simulation computations to CUDA. In addition, to accelerate the field solver, algebraic multigrid methods, implemented by the PyAMG library on CPU and the AMGX library on GPU, were utilized. Major speed-up was achieved, especially when running simulations with very high-resolution spatial grids on high-performance GPUs.

Keywords: plasma simulation, particle-in-cell, CUDA, GPU, CuPy

Ivan Kadochnikov

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2507/225-229-paper-39.pdf

# 1. Introduction

Numerical simulation of charged particle dynamics is required for the development of electron beam ion sources (EBIS) [1]. Special attention should be paid to the formation and evolution of different instabilities that can manifest in such sources. Understanding these processes can allow the efficiency of ion sources to be improved. The simulation of two-stream instability emerging between slow ions and fast electrons is particularly important.

To perform the required simulation, "ef" and "ef_python" applications are under development [2] using the particle-in-cell (PIC) method [3]. Ef is written in C++ for performance reasons, and there are several initiatives to use MPI, OpenMP and CUDA to use parallel and GPU computing in ef. Ef_python is written in python for fast and flexible development. It uses NumPy for simulation, and this report describes the efforts to improve its performance and adapt it to use graphics accelerators.

A previous study using a 2-dimensional dynamic particle model as a sample showed that both OpenCL and CUDA could dramatically improve particle simulation performance compared to NumPy [4]. For this project we chose to use CUDA as the acceleration backend because of the availability of high-level libraries, namely CuPy and AMGX.

# 2. Particle-in-cell

The idea of the particle-in-cell method is to simulate individual positions and velocities of particles over small time steps, while using a grid to approximate collective parameters, such as charge density and current density created by particles. The simulation domain is defined by a cuboid regular mesh and a time step. As part of the problem definition, Ef and Ef_pyhton allow defining conducting volumes with a fixed potential within the simulation domain. Only simple volume shapes, such as a cylinder or a sphere, are supported right now. The same shapes define particle sources that generate new particles with a normal velocity distribution within their volumes at each time step. It is possible to add externally defined electric and magnetic fields.

Each simulation step within Ef_python consists of the following operations: advance particle positions and momenta, generate new particles, calculate charge density, compute electric potential, calculate electric field.

### 2.1 Grid-to-particle: field interpolation

To compute the electric field acting on each particle, linear interpolation of the electric field from the 8 surrounding grid nodes is used. This is not the only possible interpolation kernel. Grid-to-particle and particle-to-grid kernels are an important area of possible algorithmic optimization to improve simulation accuracy and performance in this project.

### 2.2 Particle mover

A "particle mover" denotes the simulation component responsible for updating particle positions and velocities at each time step. The choice of the particle mover is important for assuring conservation of important invariants, such as energy. As the particle mover in ef_python, the well-established leapfrog second-order explicit method, known as the Boris scheme, is used [5].

### 2.3 Particle creation and destruction

Particles that leave the simulation volume or collide with inner conducting volumes are absorbed and removed from simulation. New particles generated by particle sources are added to the system and their momentum is simulated half a time step back as it is required by the "leapfrog" particle mover.

### 2.4 Particle-to-grid: particle weighting

The charge of each particle is linearly divided into the 8 surrounding grid nodes. This first-order scheme is sometimes called cloud-in-cell. The collective current is not used in ef or ef_python right now, as the magnetic field created by particles is considered negligible.

### 2.5 Field solver

To simulate the electromagnetic field created by particles, Maxwell's equations on the grid can be approximately solved by many algorithms based on 3 main approaches: finite difference methods (FDM), finite element methods (FEM) and spectral methods. In ef and ef_python, the FDM method is used to solve the Poisson's equation and compute the electric potential created by the simulated particles, while considering the effect of conducting volumes within the simulation domain. The electric field is then simply a numerically computed gradient of the potential on the grid.

## 3. Simulation performance improvements

The contribution of each step of simulation to the total execution time depends on different simulation parameters and implementation details. Profiling tools allowed us to find bottlenecks and improve performance using different simulation setups. The field solver is usually the major component of simulation in terms of resource usage.

### 3.1 Algorithm improvements

As a result of profiling, several unreasonably slow function implementations were fixed, including rewriting the FDM equation sparse matrix generator using diagonal SciPy matrix classes, caching which mesh nodes are inside conducting volumes, delegating field-to-particle interpolation to SciPy, and rewriting particle-to-field weighting for better NumPy parallelization.

In order to safely conduct the major code restructuring and reimplementation required for this project, over 200 unit tests were created, providing total code coverage of 91%.

### 3.2 Algebraic multi-grid solver

The point of major computation complexity of ef_python is the field solver. As such, it was the first component picked for acceleration. The SciPy conjugate gradient solver for sparse matrices was used in ef_python before these improvements.

Multigrid methods are methods of numerically solving differential equations on a hierarchy of grids with increasing coarseness, allowing one to faster reduce the large-wavelength error [6]. Algebraic multigrid methods construct the multigrid scale hierarchy directly from the matrix of linear equations, not explicitly relying on the geometry of the system or the partial differential equations being solved. Thus, algebraic multigrid methods are easy to apply without complex problem-dependent setup. The PyAMG library provides many AMG methods for Python, and the AMGX library implements AMG on CUDA. PyAMGX is a Python wrapper over AMGX, allowing one to run AMGX from Python, on one GPU only.

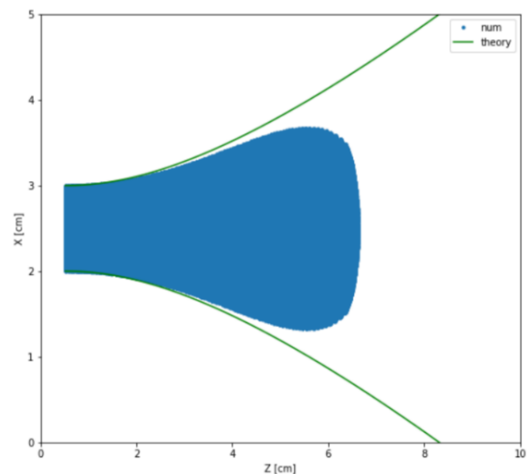


Figure 1 Comparison of the axially symmetric beam shape: an infinite beam predicted analytically and a beam starting from a cylinder computed numerically

### 3.3 Using CuPy for GPU acceleration

CuPy is a library aiming to help easy acceleration of the NumPy-based code on GPU by providing in-place replacements for most NumPy functions and operators. By using CuPy we accelerated most of the simulation operations in ef_python with minor code changes.

Grid interpolation was not available in CuPy and had to be rewritten as a custom CUDA kernel. The gradient function from Numpy that was used to compute the electric field from the potential was not implemented in CuPy. It had to be reimplemented for the case of regular grid steps using CuPy operators.

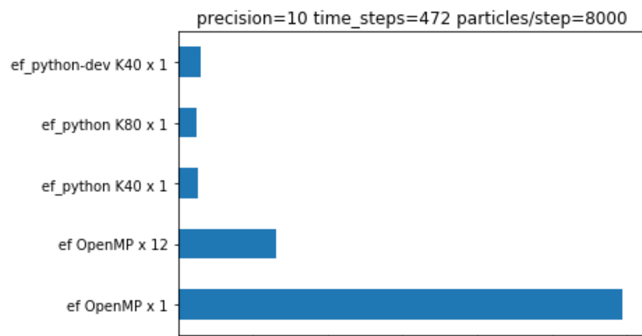


Figure 2 Comparison of the simulation time for the two-stream instability problem with higher precision of the field solver and a smaller number of particles

Simulation classes were rewritten to use class injection for the convenient runtime NumPy/CuPy and PyAMG/AMGX selection. This allowed using the selected simulation backend with minimal code duplication. Only the field solver and spatial mesh classes required explicit implementation-dependent subclasses.

## 4. Sample simulation

In addition to unit tests, sample simulation was regularly performed to detect errors during development. It was the simulation of a radially symmetric charged particle beam starting from a cylindrical emitter. The shape of the beam caused by electrostatic particle repulsion can be predicted analytically. The comparison between the theoretical shape and the numerical solution is shown in Figure 1.

The simulation of two-stream instability was performed with ef using OpenMP with 1 and 12 threads to provide a baseline of comparison. The performance was compared to ef_python using CuPy and PyAMGX running on Nvidia Tesla K40 and Nvidia Tesla K80 graphics cards. The results of this comparison are shown in Figure 2 and Figure 3.

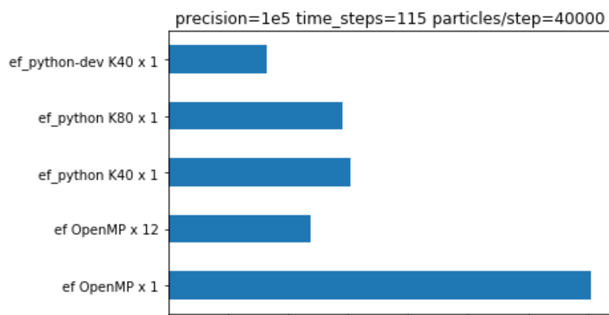


Figure 3 Comparison of the simulation time for the two-stream instability problem with higher precision of the field solver and a smaller number of particles

## 5. Results and acknowledgements

With minimal changes to the program, CuPy allowed accelerating most of the simulation operations on GPU through CUDA. In addition, to improve the field solver, algebraic multigrid methods, implemented by the PyAMG library on CPU and the AMGX library on GPU, were utilized. Major speed-up was achieved, especially when running simulations with very high-resolution spatial grids on high-performance GPUs.

Using multiple GPUs is an important future prospect for the ef_python application. This will require using MPI for process coordination and data exchange. AMGX already has MPI-based multi-

GPU support, but the PyAMGX interface library will have to be extended to add it in. Another promising avenue of future research is using OpenCL as an acceleration backend, as it can utilize non-Nvidia GPUs.

Profiling and testing were partly performed on the HybriLIT heterogeneous computing platform [7] in the Laboratory of Information Technologies at JINR.

The reported study was funded by RFBR according to the research project № 18-32-00239.

## References

[1] Donets, E.D., 1976. Review of the JINR Electron Beam Ion Sources. IEEE Transactions on Nuclear Science 23, 897. https://doi.org/10.1109/TNS.1976.4328375

[2] Ef [WWW Document], n.d. . GitHub. URL https://github.com/epicf (accessed 11.11.19).

[3] Grigor′ev, I.N., Vshivkov, V.A., Fedoruk, M.P., 2002. Numerical "particle-in-cell" methods: theory and applications. VSP, Utrecht; Boston.

[4] Boytsov, A., Kadochnikov, I., Zuev, M., Bulychev, A., Zolotuhin, Y., Getmanov, I., 2018. COMPARISON OF PYTHON 3 SINGLE-GPU PARALLELIZATION TECHNOLOGIES ON THE EXAMPLE OF A CHARGED PARTICLES DYNAMICS SIMULATION PROBLEM 5.

[5] Qin, H., Zhang, S., Xiao, J., Liu, J., Sun, Y., Tang, W.M., 2013. Why is Boris algorithm so good? Physics of Plasmas 20, 084503. https://doi.org/10.1063/1.4818428

[6] Shapira, Y., 2003. Matrix-Based Multigrid: Theory and Applications. Springer Science & Business Media.

[7] Gh. Adam, M. Bashashin, D. Belyakov, M. Kirakosyan, M. Matveev, D. Podgainy, T. Sapozhnikova, O. Streltsova, Sh. Torosyan, M. Vala, L. Valova, A. Vorontsov, T. Zaikina, E. Zemlyanaya, M. Zuev. IT-ecosystem of the HybriLIT heterogeneous platform for high-performance computing and training of IT-specialists. Selected Papers of the 8th International Conference "Distributed Computing and Grid-technologies in Science and Education" (GRID 2018), Dubna, Russia, September 10-14, 2018, CEUR-WS.org/Vol-2267″