BNL-230328-2026-TECH

A Julia Implementation of the Faddeeva Function for Efficient GPU Evaluation with Application to Beam–Beam Simulations

Y. Kan

October 2026

Electron-Ion Collider Project

Brookhaven National Laboratory

U.S. Department of Energy

USDOE Office of Science (SC), Nuclear Physics (NP)

Notice: This technical note has been authored by employees of Brookhaven Science Associates, LLC under Contract No. with the U.S. Department of Energy. The publisher by accepting the technical note for publication acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this technical note, or allow others to do so, for United States Government purposes.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or any third party's use or the results of such use of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof or its contractors or subcontractors. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

A Julia Implementation of the Faddeeva Function for Efficient GPU Evaluation with Application to Beam–Beam Simulations

Yi-Kai Kan*, Derong Xu

Brookhaven National Laboratory

October 12, 2026

Abstract

The evaluation of the Faddeeva function can be a major computational bottleneck in weak–strong beam–beam simulations. In a representative Electron–Ion Collider simulation, about 85% of the total runtime is spent evaluating the Faddeeva function. We develop a native Julia implementation based on Weideman's approximation that is well suited for both CPU and GPU execution. On CPU, our implementation is approximately 1.5 times faster than the `faddeeva` function in `SpecialFunctions.jl` over the tested domain. When incorporated into our weak–strong beam–beam solver, the new implementation is approximately 4.7 times faster than the previous implementation on a GPU.

1. Introduction

The Faddeeva function plays an important role in physics and can be defined as

$$w(z) := e^{-z^2}\left(1 + \frac{2i}{\sqrt{\pi}}\int_0^z e^{t^2}\,\mathrm{d}t\right), \quad z \in \mathbb{C}.$$

In beam physics, it appears in the analytical expression for the electric field generated by a Gaussian-distributed charged-particle beam. For example, under the two-dimensional approximation [1], the transverse electric field generated by the beam can be expressed in closed form in terms of the Faddeeva function [2,3]

$$E_y(x,y) + iE_x(x,y) = \mathrm{Const}\cdot\left[w\left(\frac{x+iy}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right) - \exp\left(-\frac{x^2}{2\sigma_x^2}-\frac{y^2}{2\sigma_y^2}\right)w\left(\frac{\frac{\sigma_y}{\sigma_x}x + i\frac{\sigma_x}{\sigma_y}y}{\sqrt{2(\sigma_x^2-\sigma_y^2)}}\right)\right],$$

where x and y are the transverse coordinates relative to the beam center, and σ_x and σ_y are root-mean-square transverse beam sizes. This makes it particularly important for the numerical modeling of space-charge effects and beam–beam interactions. Therefore, efficient evaluation of the Faddeeva function is essential for beam-dynamics simulations involving these effects.

This work stems from the development of `WSTrack.jl` [4], a GPU-parallelized weak–strong beam–beam simulation code implemented entirely in the Julia programming language. Julia is well suited to scientific computing because it combines high performance with expressive programming features, such as multiple dispatch and metaprogramming. In particular, the package `CUDA.jl` for CUDA programming lets users write GPU kernels directly in Julia. During the development of `WSTrack.jl`, we encountered a limitation in evaluating the Faddeeva function. Although the Faddeeva function is available through `SpecialFunctions.jl`, its implementation relies on external C/C++ code and therefore cannot be called directly from a Julia GPU kernel. This limitation motivated the development of a native Julia implementation suitable for GPU execution. In our previous implementation, we used the algorithm proposed by Abrarov and Quine [5] to evaluate the Faddeeva

*ykan@bnl.gov

function. To assess its impact on the overall simulation performance, we profile a weak–strong tracking simulation of the hadron beam in the Electron–Ion Collider on a CPU. The hadron beam is represented by 1.024×10^6 macroparticles, and the beam–beam kick from the electron beam is evaluated using 7 longitudinal slices. The simulation model consists of linear transfer maps between the interaction point and the upstream and downstream crab cavities, crab-cavity kicks, the beam–beam interaction based on Hirata's synchro-beam map [6,7], and a linear one-turn map for the full hadron storage ring. The profiling results show that the beam–beam interaction takes about 99% of the total CPU time (Figure 1). Within the beam–beam interaction, approximately 86% of the CPU time is spent evaluating the Faddeeva function (Figure 2). Consequently, evaluating the Faddeeva function accounts for roughly 85% of the total simulation time and is the primary target for performance optimization.

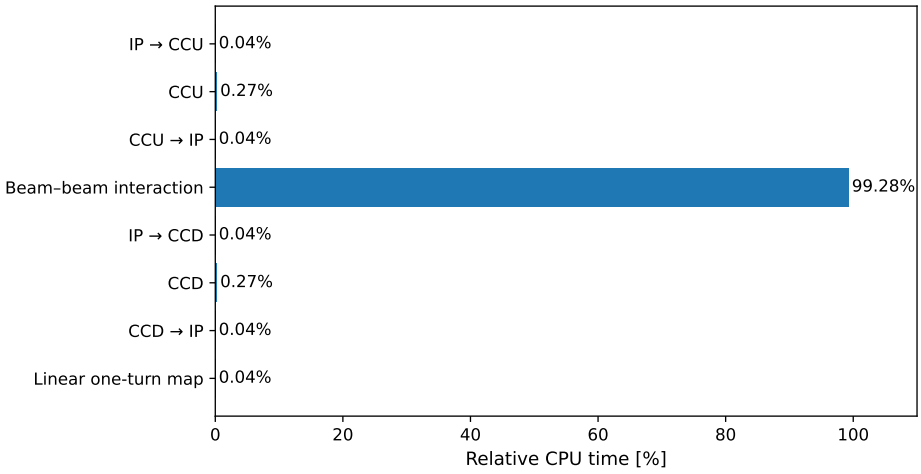


Figure 1: Profiling of the beam–beam tracking simulation. IP denotes the interaction point. CCU and CCD denote the upstream and downstream crab cavities, respectively. Labels of the form $A \to B$ denote linear transfer maps from A to B.

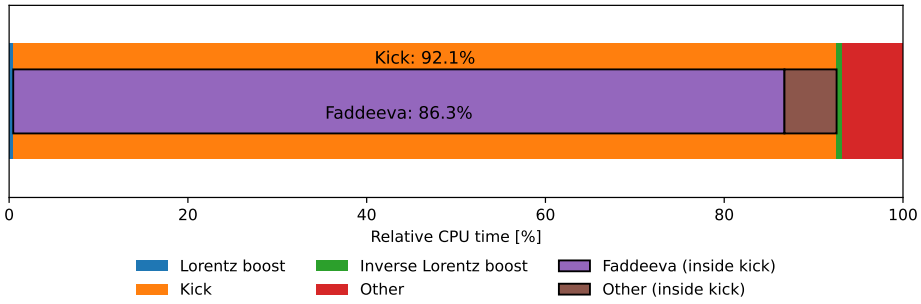


Figure 2: Profiling of the beam–beam interaction.

2. Algorithm Selection

General-purpose methods for evaluating the Faddeeva function often use different approximations in different regions of the complex plane to achieve fast and accurate results [8–10]. This branching strategy is well suited to independent scalar evaluation, where each function call can select an efficient approximation based on its input. On GPUs, threads within the same warp execute the same instruction at the same time. Warp divergence occurs when threads within the same warp follow different branches of the code. The warp may then need to execute these branches one after another, reducing the performance benefit of the region-dependent strategy. The Weideman approximation [11] provides an attractive alternative for GPU execution. It uses a single rational approximation over the computational domain without switching among several region-specific

approximations. For a fixed approximation order, every input is evaluated using the same simple polynomial calculation with precomputed coefficients. This simple and predictable computational structure makes the method attractive for GPU execution. For these reasons, we adopt the Weideman approximation in this work.

3. Implementation

The Weideman approximation of order N to the Faddeeva function for $z \in \mathbb{C}$ can be written in the form

$$w(z) \approx \left(\frac{2}{c(z)} \sum_{n=0}^{N-1} a_n b(z)^n + d \right) \frac{1}{c(z)},$$

where $c(z)$ and $b(z)$ are complex-valued functions, $\{a_n\}_{n=0}^{N-1}$ are real coefficients, and d is a real constant.

To evaluate this approximation efficiently, our implementation makes extensive use of Julia's metaprogramming capabilities. We use metaprogramming to generate a specialized implementation for an arbitrary order N. The coefficients $\{a_n\}_{n=0}^{N-1}$ are computed during code generation and embedded directly into the generated code. This eliminates the need to recompute or access the coefficients through an array during function evaluation. The polynomial term $\sum_{n=0}^{N-1} a_n b(z)^n$ is evaluated using Horner's rule to reduce the number of arithmetic operations, and the resulting expression is fully unrolled at compile time. As a result, runtime evaluation requires no loop control, bounds checking, or coefficient-array indexing. We also use `muladd` explicitly in the Horner expression so that fused multiply-add operations can be generated where supported. Together, these techniques produce code that executes a fixed sequence of arithmetic operations at runtime. This fixed execution pattern is well suited to GPU execution.

4. Accuracy and Performance Benchmarks

The accuracy of our implementation is benchmarked against the `faddeeva` function in `SpecialFunctions.jl` [12]. The relative error at each point $z \in \mathbb{C}$ is defined as $|w_{\text{test}}(z) - w_{\text{ref}}(z)| \;/\; |w_{\text{ref}}(z)|$, where $w_{\text{test}}(z)$ is evaluated using our implementation of the Weideman approximation with $N = 32$, and $w_{\text{ref}}(z)$ is the corresponding value returned by the `faddeeva` function in `SpecialFunctions.jl`. The relative error over a two-dimensional uniform grid of 401×401 points in the complex plane is shown in Figure 3. The maximum, mean, and median relative errors are 3.07×10^{-13}, 5.50×10^{-15}, and 2.82×10^{-16}, respectively.

To compare the performance, it is worth noting that the `faddeeva` function in `SpecialFunctions.jl` wraps the C++ implementation from The Faddeeva Package by Johnson [10]. Since it is not native Julia code, it cannot be called directly within a GPU kernel. Johnson's implementation uses different approximations in different regions of the complex plane and switches to a more efficient continued-fraction expansion for sufficiently large $|z|$. This region-dependent branching may not provide the same advantage on GPUs because of warp divergence. We therefore restrict the CPU benchmark to the region where the continued-fraction expansion is not used. The function is evaluated at 1×10^5 randomly sampled points within $-5 < \mathrm{Re}(z) < 5$ and $-5 < \mathrm{Im}(z) < 5$. The CPU execution times per evaluation are 111.5 ± 1.1 ns for our implementation and 165.8 ± 3.4 ns for the `faddeeva` function in `SpecialFunctions.jl`. This corresponds to a speedup of approximately 1.49 times within this domain.

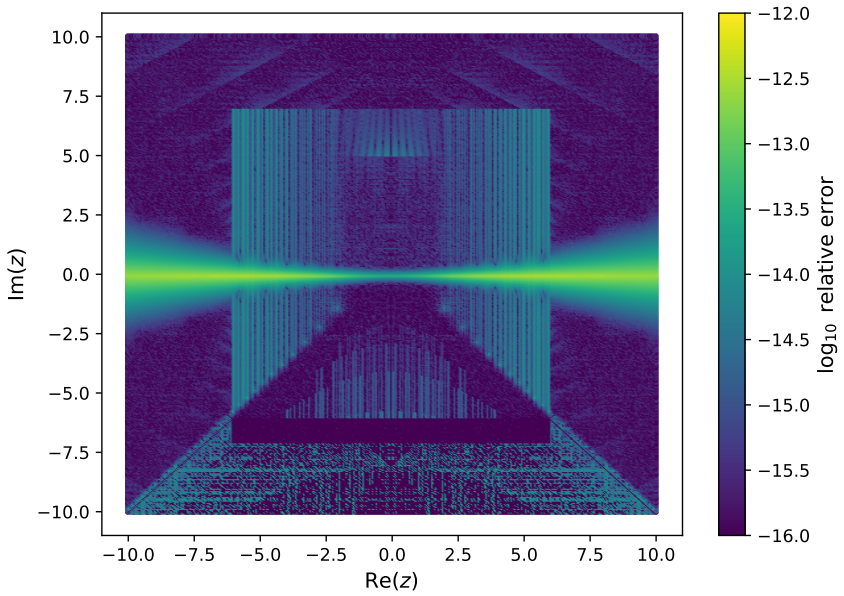


Figure 3: Relative error of our implementation of the Weideman approximation with $N = 32$.

5. Improvement in Beam–Beam Simulation Performance

We benchmark the total runtime of the tracking simulation for the hadron beam using the model and parameters described in Section 1 on a single NVIDIA A100 GPU. Replacing the previous Faddeeva implementation with the new implementation reduces the total simulation time from 16405 ± 32 s to 3461 ± 106 s. This gives a speedup of about 4.7 times. As shown in Figure 4, the turn-by-turn emittance and luminosity obtained with the two Faddeeva implementations agree well.

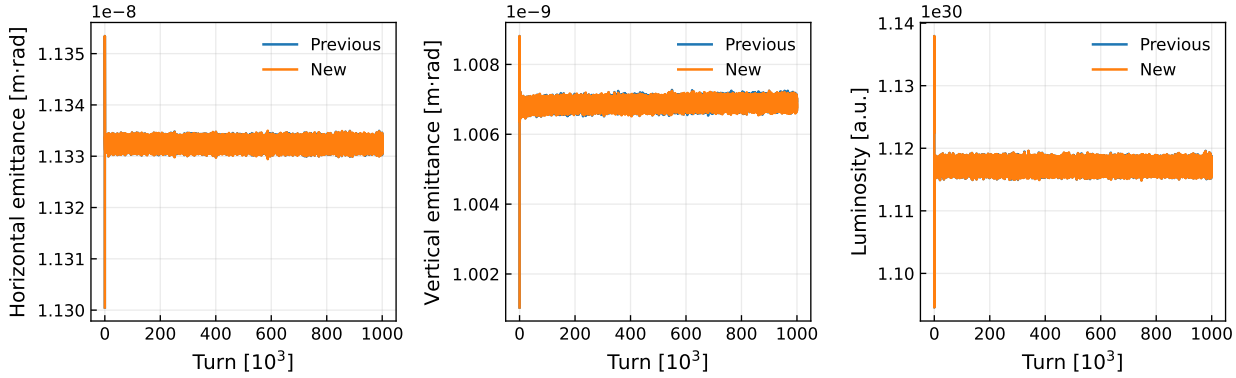


Figure 4: Comparison of turn-by-turn emittance and luminosity obtained from simulations using the previous and new Faddeeva implementations.

6. Conclusion

In this work, we present a native Julia implementation of the Faddeeva function with an emphasis on efficient GPU execution. Integrated into our weak–strong beam–beam tracking solver, it significantly reduces the simulation time. For the Electron–Ion Collider tracking case considered here, the simulation using the new Faddeeva implementation is approximately 4.7 times faster than that using the previous implementation when executed on a GPU. Our Faddeeva implementation is also available as the standalone Julia package `WeidemanFaddeeva.jl` [13].

Acknowledgement

This work was supported by Brookhaven Science Associates, LLC, under Contract No. DE-SC0012704 with the U.S. Department of Energy, and by the U.S. Department of Energy Early Career Award Field Work Proposal No. EC-002. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy User Facility, under NERSC award NP-ERCAP0036188.

References

[1] Y.-K. Kan and J. Qiang, On the Electromagnetic Field of a Focusing Charged Particle Beam and Its Two-Dimensional Approximation, technical report No. BNL-226325-2024-TECH, 2024.

[2] M. Bassetti and G. A. Erskine, Closed Expression for the Electrical Field of a Two-Dimensional Gaussian Charge, technical report No. CERN-ISR-TH-80-06, 1980.

[3] D. Xu, V. S. Morozov, D. Sagan, Y. Hao, and Y. Luo, Enhanced beam-beam modeling to include longitudinal variation during weak-strong simulation, Phys. Rev. Accel. Beams **27**, 61002 (2024).

[4] Y.-K. Kan, *WSTrack.jl: A GPU-Parallelized Weak–Strong Beam–Beam Solver for Beam Dynamics Simulations*, https://doi.org/10.5281/zenodo.23023601.

[5] S. M. Abrarov and B. M. Quine, Efficient algorithmic implementation of the Voigt/complex error function based on exponential series approximation, Applied Mathematics and Computation **218**, 1894 (2011).

[6] K. Hirata, H. W. Moshammer, and F. Ruggiero, A symplectic beam-beam interaction with energy change, Part. Accel. **40**, 205 (1993).

[7] K. Hirata, Analysis of beam-beam interactions with a large crossing angle, Phys. Rev. Lett. **74**, 2228 (1995).

[8] G. P. M. Poppe and C. M. J. Wijers, More efficient computation of the complex error function, ACM Trans. Math. Softw. **16**, 38 (1990).

[9] M. R. Zaghloul and A. N. Ali, Algorithm 916: computing the Faddeyeva and Voigt functions, ACM Transactions on Mathematical Software **38**, 1 (2011).

[10] S. G. Johnson, *The Faddeeva Package*, http://ab-initio.mit.edu/faddeeva/.

[11] J. A. C. Weideman, Computation of the complex error function, SIAM Journal on Numerical Analysis **31**, 1497 (1994).

[12] JuliaMath, *SpecialFunctions.jl*, https://github.com/JuliaMath/SpecialFunctions.jl.

[13] Y.-K. Kan, *WeidemanFaddeeva.jl*, https://doi.org/10.5281/zenodo.23171132.