



SciPy 2025

July 7 - July 13, 2025

Proceedings of the 24th
Python in Science Conference
ISSN: 2575-9752

Advancing High Energy Physics Data Analysis with Julia -- A Case for JuliaHEP

Ianna Osborne¹  , and Jerry Ling²  

¹Princeton University, ²Harvard University

Abstract

For the past 25 years, the High-Energy Physics (HEP) community has steadily adopted Python as its primary language for data analysis, supported by compiled-backend libraries like [NumPy](#) and [Awkward Array](#), along with other Python tools like [Uproot](#) for I/O. However, the Julia programming language offers a compelling alternative, addressing the two-language problem with C++-comparable performance and Python-like ease of use. [JuliaHEP](#) is an informal organization that aims to unify effort in developing Julia projects related to HEP, outlining its advantages, ongoing developments, and integration with existing Python-based tools.

Keywords Awkward Array, Julia, High-Energy Physics, Data Analysis

1. INTRODUCTION

The High-Energy Physics (HEP) community has long relied on Python and C++ for data analysis. While Python provides ease of use and a rich scientific ecosystem, it struggles with performance for large-scale analyses. C++, on the other hand, offers speed but comes with increased complexity and slower development cycles. Julia, a modern language designed for scientific computing, promises the best of both worlds: high-level expressiveness with near-native execution speed.

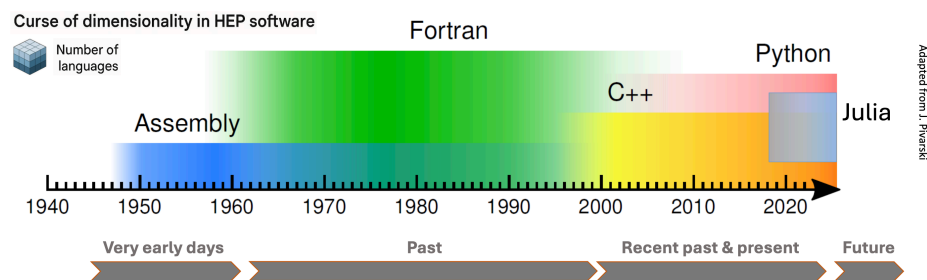



Figure 1. HEP Software: it all revolves around a language, or several.

JuliaHEP, with its emerging set of tools and libraries [1], aims to facilitate HEP data analysis in Julia. However, given the long timescales of HEP experiments and the substantial legacy software they depend on, introducing Julia alongside existing C++ and Python ecosystems inevitably creates a three-language problem — though this is intended as a transitional, rather than permanent, arrangement.

To navigate this, three main development strategies have emerged within the JuliaHEP community. The first involves wrapping mature C++ libraries to make them accessible from Julia — for example, [Geant4.jl](#) provides a Julia interface to the widely used Geant4 particle transportation toolkit, and [ROOT.jl](#) offers bindings to the ROOT data analysis framework. The second approach opts for complete reimplementations of existing tools in native Julia, as demonstrated by [BAT.jl](#) for Bayesian analysis and [UnROOT.jl](#) for reading ROOT files

Published Jul 10, 2025

Correspondence to
Ianna Osborne
ianna.osborne@cern.ch

Open Access 

Copyright © 2025 Osborne & Ling. This is an open-access article distributed under the terms of the [Creative Commons Attribution 4.0 International](#) license, which enables reusers to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the creator.

without relying on the C++ libraries. The third strategy focuses on integrating Julia with Python; we are following this strategy to expose the Python-based Awkward Array library via its dedicated Julia backend, `AwkwardArray.jl`, enabling efficient access to nested, irregular data structures commonly used in HEP.

This backend addresses one of the key challenges in multi-language environments: data sharing across language boundaries. By exposing the underlying data buffers directly to Julia, `AwkwardArray.jl` avoids data copying and offers a gentle, incremental path for physicists to explore Julia’s capabilities within their existing Python workflows.

2. AWKWARD ARRAY IN JULIA

Awkward Array is a key library for handling complex and jagged HEP data in the Python ecosystem. To access this data from Julia, it is integrated via a dedicated backend, `AwkwardArray.jl`. What sets this backend apart is its tight integration with Julia’s type system.

To bridge the Python and Julia ecosystems, we make extensive use of `PythonCall.jl` [2], a package that provides bi-directional interoperability between the two languages. This enables us to offload computationally heavy operations to Julia — following a pattern familiar from other JIT compilation tools in Python, such as Numba.

We evaluated this integration by comparing performance and usability using Julia-native types like `Vector of Vectors`, which provide similar capabilities for representing nested, irregular data. Our results show no significant overhead when combining the two ecosystems. Additionally, we explored Julia’s interoperability and native packages to further optimize HEP computations, offering a practical pathway for gradually introducing Julia to the Python-based HEP community.

3. PERFORMANCE BENEFITS

We present the benefits of using Julia for HEP workloads by benchmarking the same operation with JuliaHEP tools and comparing the results to Python and C++ implementations. As shown in [Figure 2](#), Julia offers a noticeable speedup with less boilerplate.

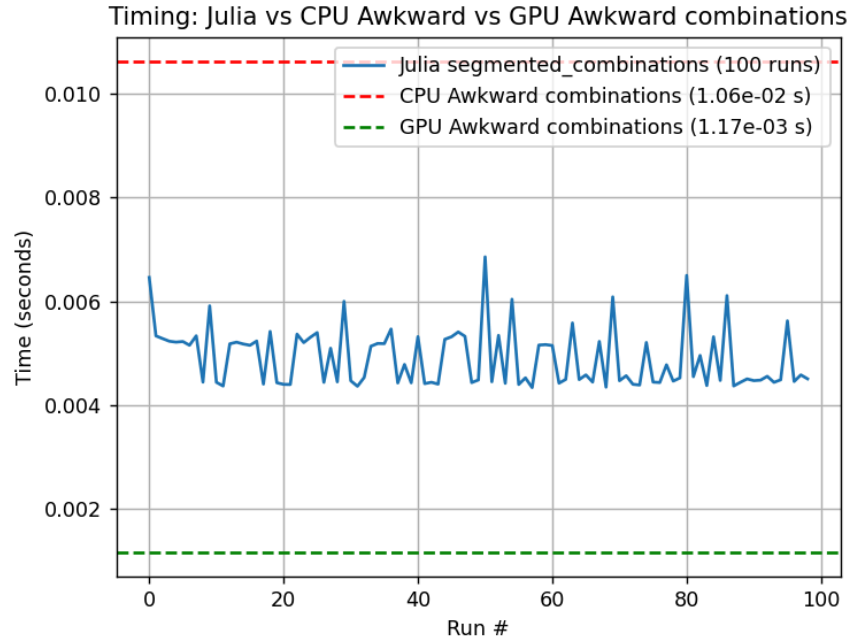


Figure 2. This is the combinations performance plot, comparing the `ak.combinations` function with a Julia kernel on an awkward array. The Julia kernel uses `JuliaMath/Combinatorics.jl` [3].

This is without yet exploiting Julia’s potential for parallel computing and GPU support, which opens the door to workflows where data loaded with `AwkwardArray.jl` is processed using Julia-native GPU kernels or multi-threaded code — offering scalable performance without disrupting existing pipelines.

4. SCALING IT UP

While performance gains matter, integrating Julia into established HEP workflows also involves practical challenges. `AwkwardArray.jl` enables physicists to explore Julia’s capabilities alongside existing Python tools by accessing shared data buffers directly, without copying. This allows heavy computations to be offloaded to Julia where appropriate, while maintaining full compatibility with Python’s data ecosystem.

Naturally, introducing a third language raises long-term questions. Could Julia eventually take on a larger role, perhaps replacing parts of the Python or C++ stack? It remains too early to tell. Legacy software must be maintained, Python remains dominant, and new contenders like Rust are emerging. For now, the focus is on enabling a flexible, incremental path to using Julia where it offers clear advantages — without disrupting current workflows.

The main technical hurdle is efficient data interoperability — a challenge addressed by the Awkward Array design, which uses contiguous data buffers and descriptive metadata. This model has also proven effective in related optimizations, such as its virtual array implementation.

Data access remains central. ROOT TTrees are still the dominant format, with `RNTuple` expected to grow in use. Both formats are accessible as Awkward Arrays via `uproot` (Python) and `UnROOT.jl` (Julia), ensuring consistency across languages.

Managing a shared Python–Julia environment remains challenging. Conda for Python, `juliaup`, and Julia’s package manager integrate well, though keeping package versions aligned requires care. A coordination tool like `pixi` is currently under evaluation to simplify this.

Overall, this approach offers a pragmatic, low-friction path for gradually introducing Julia into HEP workflows — letting physicists benefit from its strengths in JIT compilation, multiple dispatch, and GPU acceleration without leaving established ecosystems behind.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation under Cooperative Agreement PHY-2323298.

REFERENCES

- [1] Graeme Andrew Stewart, “Julia in HEP,” 2025, doi: <https://doi.org/10.48550/arXiv.2503.08184>.
- [2] C. Rowley, “PythonCall.jl: Python and Julia in harmony.” [Online]. Available: <https://github.com/JuliaPy/PythonCall.jl>
- [3] “Combinatorics.jl: A combinatorics library for Julia.” [Online]. Available: <https://github.com/JuliaMath/Combinatorics.jl>