Navigating the Multilingual Landscape of Scientific Computing: Python, Julia, and Awkward Array

Ianna Osborne[1,*], *Jim* Pivarski[1], and *Jerry* Ling[2]

[1]Princeton University
[2]Harvard University

Abstract. Scientific computing relies heavily on powerful tools like Julia and Python. While Python has long been the preferred choice in High Energy Physics (HEP) data analysis, there's a growing interest in migrating legacy software to Julia. We explore language interoperability, focusing on how Awkward Array data structures can connect Julia and Python. We discuss memory management, data buffer copies, and dependency handling, highlighting performance gains from invoking Julia from Python and vice versa. Particularly, we look into distributed array-oriented calculations involving large-scale HEP data and a unique role of Awkward Array in these workflows. We examine the advantages and challenges of achieving interoperability between Julia and Python in scientific computing.

1 Introduction

Both Python and Julia offer physicists the interactivity they need for effective data analysis. Python has long been established within our community, but Julia is making a strong entrance. At the this conference, there were four times more Julia-related contributions compared to the previous one, highlighting its growing influence. With Python's continued dominance and Julia's emergence, we explore how these two languages can be combined and what are the performance penalties.

2 Embedding Julia in Python

To integrate Python's ecosystem into Julia projects, we can use PythonCall [1], while JuliaCall allows embedding high-performance Julia code into Python scripts. This dual purpose package provides a bridge between the two languages.

This topic was explored in depth at the JuliaHEP 2024 workshop, particularly in this talk [2]. For configuration and runtime environment details, this presentation offers valuable insights.

3 Julia in a Python Jupyter Notebook

Jupyter Notebook allows for a Julia-Python workflow by loading the JuliaCall extension. This approach grants access to all Julia packages via Julia's package manager.

*e-mail: ianna.osborne@cern.ch

4 ROOT Trees as Julia Objects in Python

A notable performance improvement can be achieved by reading ROOT data (trees) as Julia objects in Python environments. Currently, the UnROOT.jl [3] provides a faster way to read data stored in ROOT trees. Additionally, the AwkwardArray.jl [4] bridge package enables presenting these datasets as Awkward Arrays in Julia and passing them to Python as native Python Awkward Arrays [5].

5 Executing Native Julia Code in Python

Python users can include and execute Julia functions such as shown on figure 1 within their Python scripts. This example shows how to create an Awkward Array structure from a Julia ROOT tree as discussed in section 4.

```
function main_looper(events)
    array = AwkwardArray.PrimitiveArray{Float64}()
    for evt in events

        (; Muon_charge) = evt
        if length(Muon_charge) != 4
            continue
        end
        sum(Muon_charge) != 0 && continue # shortcut if-else

        (; Muon_pt, Muon_eta, Muon_phi, Muon_mass) = evt
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        push!(array, higgs_mass)
    end

    return array
end
```

Figure 1. Example of a Julia function: *main_looper* that creates an Awkward Array structure from a ROOT tree data

However, proper code organization is crucial for both performance and maintenance. Using the ‘include‘ function with the correct file path is recommended. For larger projects, structuring the code into modules and separate files allows for better maintainability. Using ‘export‘ statements helps to manage namespace exposure and prevent function name collisions.

6 Efficiency of Calling Julia from Python

With minimal overhead, Julia offers substantial performance improvements, particularly when ensuring type stability and reducing unnecessary recompilation as shown on figure 2. These practices are key to leveraging Julia’s full efficiency.

Here, we measure the execution time of a Julia function called from Python and compare it to the same function executed directly in Julia. Note that the first call triggers compilation. We also observe that Julia’s garbage collector may negatively affect performance. However, the overhead caused by the garbage collector is much less noticeable in multi-processing and distributed computing (Section 8).

7 Overhead Analysis: AwkwardArray.jl vs. Typed Arrays

A comparative study between AwkwardArray.jl and Julia native typed arrays (Vector) revealed that, after small optimizations to Julia’s *main_looper* code on figure 3, no significant overhead was observed.

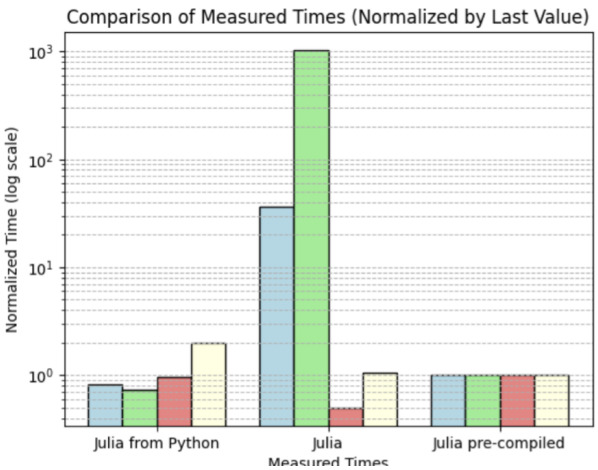

Figure 2. Comparison of measured times normalized by last value

The original code—shown as the left bar in Figure 3—was provided to ChatGPT v3 with a request for improvement. A summary of the improvements is discussed in subsection 7.1.

7.1 Key Optimizations

Firstly, there were small changes to destructure and skip logic, guided by ChatGPT. See the column marked with red arrow on figure 3. Secondly, ensuring type stability to prevent unnecessary recompilation was introduced.

7.2 Performance Gains

The execution time was reduced by 88 percent, from 0.5 seconds to 0.06 seconds that is an impressive 8.33x speedup. The memory allocations were cut from 398k to 24k, significantly improving efficiency.

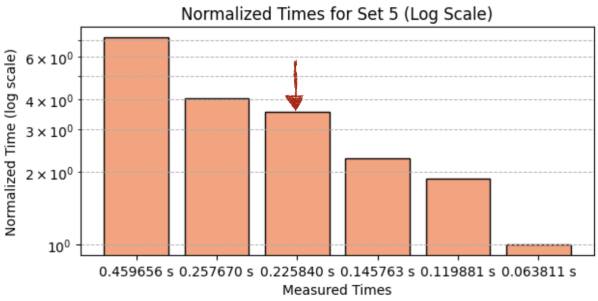

Figure 3. Code optimization and restructuring

Overall, the code became faster and leaner, demonstrating substantial gains in both speed and memory management.

8 Experimental Multithreading in JuliaCall

While multi-threading support in JuliaCall remains experimental, better results can be achieved through multi-processing and distributed computing. The right bar on the figure 3 shows significant speedup by simply enabling 4 thread execution. This approach can further enhance Julia's performance scaling when processing a large number of events.

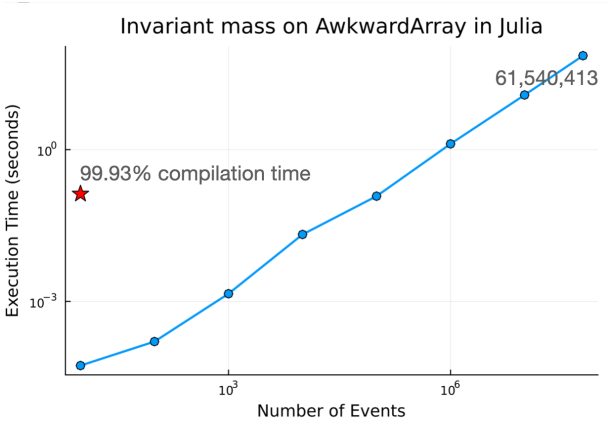

Figure 4. Scaling with number of events

9 Performance Scaling with Python's Awkward Arrays

Performance comparisons between Julia code and Python's Awkward Array functions reveal potential optimizations. While Julia may not always outperform NumPy, Awkward, or Numba in vectorized operations, it excels in tasks that do not fit neatly into these existing libraries.

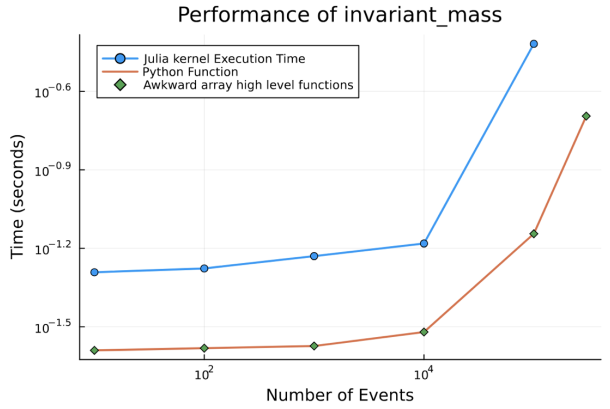

Figure 5. Python vs Julia *invariant_mass* function Comparison

10 Summary and Conclusions

Julia shines when developing custom kernels for specialized problems. While working in a multilingual runtime presents challenges—such as experimental thread support and environment complexities—the evolving nature of Julia offers exciting opportunities for performance enhancement.

By strategically leveraging Python's ecosystem alongside Julia's high-performance capabilities, physicists can achieve a powerful, efficient, and scalable computational workflow. As Julia's adoption grows, its role in advanced data analysis will continue to expand, making it a formidable tool in the physicist's arsenal.

11 Acknowledgment

This work is supported by NSF cooperative agreement OAC-1836650 (IRIS-HEP) and NSF cooperative agreement PHY-2121686 (US-CMS LHC Ops).

References

[1] Rowley, Christopher (2022). PythonCall.jl: Python and Julia in harmony, [Computer software]. https://github.com/JuliaPy/PythonCall.jl

[2] Osborne, I. (2024). Power of Python and Julia for Advanced Data Analysis, JuliaHEP 2024 workshop, CERN, https://indi.to/pWyfM

[3] Gál, T., Ling, J., Amin, N. (2021). UnROOT: an I/O library for the CERN ROOT file format written in Julia (Version v1) [Computer software]. https://doi.org/10.21105/joss.04452

[4] Pivarski, J., Osborne, I., Ling, J., AwkwardArray.jl, [Computer software]. https://github.com/JuliaHEP/AwkwardArray.jl

[5] Osborne, I., Ling, J., Pivarski, J., Bridging Worlds: Achieving Language Interoperability between Julia and Python in Scientific Computing, ACAT 2024 proceedings, (2024)