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, Princeton, NJ 08544, USA
[2]Harvard University, Cambridge, MA 02138, USA

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] package 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]. The additional performance gains are due to avoiding data copies between the two languages, as the data buffers are shared, as discussed in this publication [5]. In conclusion, we observe no performance penalties for crossing the language barrier, as no data copies are involved.

5 Executing Native Julia Code in Python

Python users can include and execute Julia functions within their Python scripts, as shown in figure 1. We recommend writing computationally heavy operations as Julia functions and invoking them from Python. To illustrate this, consider the following `main_looper` example. It demonstrates how to create an Awkward Array structure from a Julia ROOT tree, perform heavy computations, and return an Awkward Array that is not copied to Python, as discussed in section 4. The memory allocated by the array will be retained as long as both the Python and Julia objects remain accessible and are not deleted.

```
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 ROOT tree data.

As discussed above, using Julia code from Python can offer significant benefits. However, proper code organization is essential for both performance and maintainability. We recommend using the ‘include‘ function with the correct file path. For larger projects, structuring the code into modules and separate files improves maintainability. Additionally, using ‘export‘ statements helps manage namespace exposure and prevent function name collisions.

6 Efficiency of Calling Julia from Python

We measured the following for three different scenarios: the left group corresponds to calling Julia from Python, the middle group represents Julia before pre-compilation, and the right group corresponds to Julia precompiled:

- **Blue bar:** Opening a ROOT file with `UnROOT.ROOTFile` (94.32% compilation time in the worst case).

- **Green bar:** Loading a tree with `UnROOT.LazyTree` (99.26% compilation time in the worst case).
- **Red bar:** Executing the Julia function `main_looper` (45.32% garbage collector time and 6.97% compilation time in the worst case).
- **Yellow bar:** Executing the same Julia function `main_looper` as above, but in a different context with more memory available (5.16% garbage collector time in the worst case).

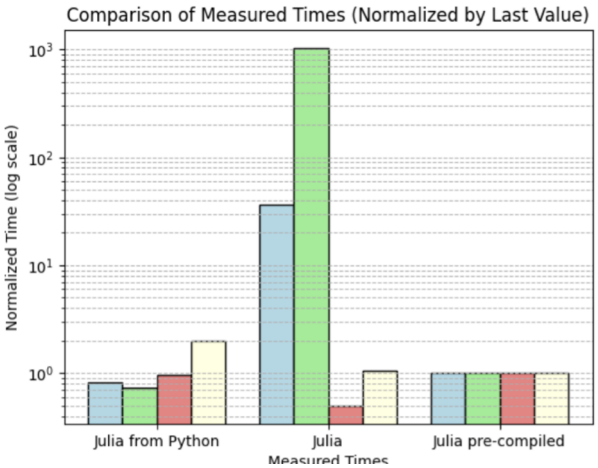

Figure 2. Comparison of measured times normalized by last value

In this measurement, we compare the execution time of a Julia function called from Python with 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 less noticeable in multi-processing and distributed computing, as discussed in Section 8.

With minimal measured overhead, we are confident that Julia can deliver significant performance gains—especially when type stability is maintained and unnecessary recompilation is avoided, as illustrated in Figure 3. These practices are crucial for unlocking Julia's full performance potential, and we will explore them in detail in the next section.

7 Overhead Analysis: AwkwardArray.jl vs. Typed Arrays

We conducted a comparative analysis between AwkwardArray.jl and Julia's native typed arrays (Vector), focusing on potential sources of performance overhead. After applying minor optimizations to Julia's *main_looper* code (Figure 1), we observed no significant performance difference—see Figure 3.

The leftmost bar in Figure 3 represents the performance of the original code, which was submitted to ChatGPT v3 for optimization suggestions. The second bar reflects performance after applying ChatGPT's recommendation to use native Julia TypedArray structures. The third bar, marked by the red arrow, shows the performance after manually reverting those changes and replacing the native structures with Awkward Arrays.

These results indicate that the primary performance gains stemmed from general code optimizations rather than the choice between Awkward Arrays and native arrays. Additional improvements and insights are discussed in Subsection 7.1.

7.1 Key Optimizations

The first key improvement, guided by ChatGPT, involved minor adjustments to destructuring and skip logic. These changes are reflected in the column immediately to the right of the one marked with the red arrow in Figure 3.

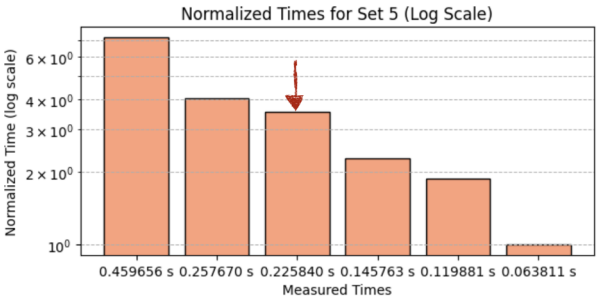

Figure 3. Code optimization and restructuring

The second major optimization focused on ensuring type stability to prevent unnecessary recompilation. Its impact is shown in the column two steps to the right of the red arrow.

Finally, the rightmost column illustrates the performance achieved when running the code with four threads, highlighting the benefits of parallel execution.

7.2 Performance Gains

The execution time was reduced by 88%, dropping from 0.5 seconds to 0.06 seconds—an impressive 8.3× speedup. At the same time, memory allocations decreased dramatically, from 398k to just 24k, significantly boosting efficiency.

Overall, these changes made the code not only faster but also more memory-efficient, demonstrating substantial improvements in both performance and resource usage.

8 Experimental Multithreading in JuliaCall

Although multithreading support in `JuliaCall` is still experimental, notable performance gains can already be observed. The rightmost bar in Figure 3 illustrates a significant speedup achieved simply by enabling execution with four threads, configured using the `JULIA_NUM_THREADS` environment variable.

While more robust and scalable improvements may be possible through multiprocessing or distributed computing, even basic multithreading can substantially enhance Julia's performance—especially when processing large numbers of events.

9 Performance Scaling with Python's Awkward Arrays

As expected, the execution time scales with the number of events, as shown in Figure 4. Additionally, we compare the performance of the same function implemented in both Python and Julia, as shown in Figure 5. This comparison highlights key areas for optimization.

While Julia does not always outperform NumPy, Awkward, or Numba in highly vectorized operations, it excels in tasks that are less suited for these traditional array libraries. Julia demonstrates clear advantages when dealing with flexible or dynamic data structures, which are often required in complex computations.

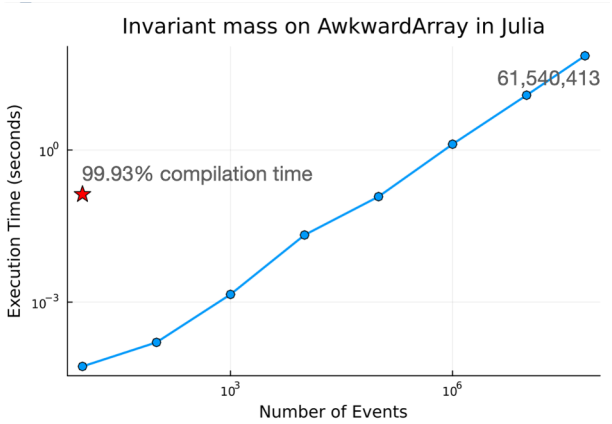

Figure 4. Execution time scaling with the number of events. As the number of events increases, the execution time also increases. The rate of scaling varies depending on the implementation and optimizations used. This figure illustrates how well the system handles larger datasets, highlighting potential bottlenecks in the processing pipeline.

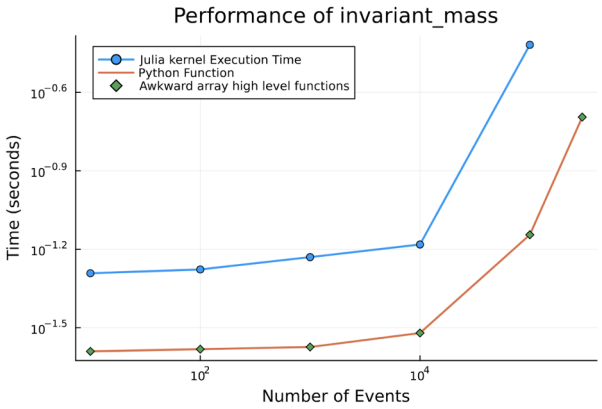

Figure 5. Comparison of the `invariant_mass` function implemented in Python and Julia. This figure highlights the performance differences between the two implementations, focusing on their efficiency in handling large datasets. Although the Julia implementation is not always faster in certain scenarios, it showcases advantages in flexibility and scalability, particularly in more complex computations that involve dynamic structures.

10 Summary and Conclusions

Our study demonstrates that Julia is particularly well-suited for developing custom computational kernels tailored to ragged arrays. While Python's ecosystem—with tools like NumPy, Awkward, and Numba—remains strong for highly vectorized operations, Julia offers notable advantages in flexibility, performance scaling, and low-level optimization opportunities.

Through targeted improvements such as type stability, reduced recompilation, and controlled memory allocations, we achieved significant speedups—up to 8.3x—and reduced memory usage by over 90%. Moreover, Julia's multithreading capabilities, even in their

current experimental state within JuliaCall, show promising potential for further speedup, especially in high-throughput environments.

Execution time scaled well with increasing event counts, and comparisons with equivalent Python implementations confirmed that Julia can match or exceed performance in scenarios that require custom processing beyond what vectorized libraries provide.

Although working within a multilingual runtime introduces challenges—such as limited thread support and more complex deployment—these are offset by Julia's growing maturity and continued ecosystem development. By integrating Julia's performance strengths with Python's mature tooling, physicists can construct efficient and scalable workflows that leverage the best of both worlds.

As adoption grows and interoperability improves, Julia is poised to become an increasingly powerful tool in the high-energy physics community, particularly for applications requiring both flexibility and performance.

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)