Navigating the Multilingual Landscape of Scientific Computing:

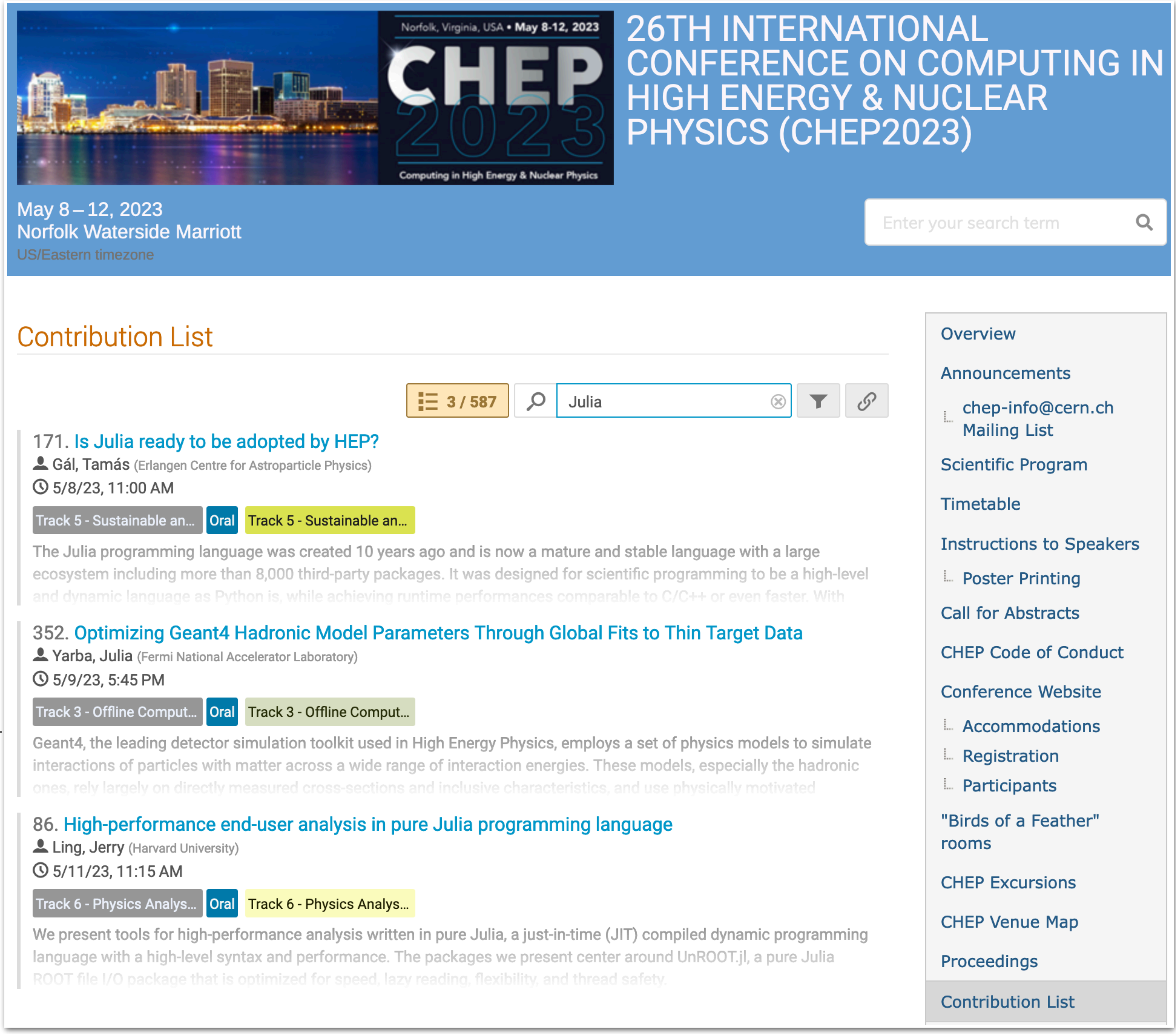
Python, Julia, and Awkward Array

Ianna Osborne, Jim Pivarski, Jerry 🦑 Ling

Julia at CHEP2024 and CHEP2023

The Emerging Trend

- Julia in HEP by Graeme A Stewart, 21 Oct 2024, Plenary session
- R&D towards heterogenous frameworks for Future Experiments by Mateusz Jakub Fila, 21 Oct 2024, Parallel (Track 3)
- ROOT RNTuple implementation in Julia programming language by Jerry Ling, 21 Oct 2024, Parallel (Track 5)
- Comparative efficiency of HEP codes across languages and architectures by Samuel Cadellin Skipsey, 21 Oct 2024, Parallel (Track 6)
- EDM4hep.jl: Analysing EDM4hep files with Julia by Pere Mato, 21 Oct 2024, Poster session
- Fast Jet Reconstruction in Julia by Graeme A Stewart, 23 Oct 2024, Parallel (Track 3)
- Navigating Phase Space for Event Generation – interfacing Sherpa with BAT.jl by Salvatore La Cagnina, 23 Oct 2024, Parallel (Track 5)
- BAT.jl, the Bayesian Analysis Toolkit in Julia by Oliver Schulz, 23 Oct 2024, Parallel (Track 5)
- Navigating the Multilingual Landscape of Scientific Computing: Python, Julia, and Awkward Array, by Ianna Osborne, 24 Oct 2024, Parallel (Track 9)


Norfolk, Virginia, USA • May 8-12, 2023
CHEP 2023
Computing in High Energy & Nuclear Physics
26TH INTERNATIONAL CONFERENCE ON COMPUTING IN HIGH ENERGY & NUCLEAR PHYSICS (CHEP2023)
May 8 – 12, 2023
Norfolk Waterside Marriott
US/Eastern timezone
Enter your search term

Contribution List

3 / 587 Julia

171. Is Julia ready to be adopted by HEP?
Gál, Tamás (Erlangen Centre for Astroparticle Physics)
5/8/23, 11:00 AM
Track 5 - Sustainable an... Oral Track 5 - Sustainable an...
The Julia programming language was created 10 years ago and is now a mature and stable language with a large ecosystem including more than 8,000 third-party packages. It was designed for scientific programming to be a high-level and dynamic language as Python is, while achieving runtime performances comparable to C/C++ or even faster. With

352. Optimizing Geant4 Hadronic Model Parameters Through Global Fits to Thin Target Data
Yarba, Julia (Fermi National Accelerator Laboratory)
5/9/23, 5:45 PM
Track 3 - Offline Comput... Oral Track 3 - Offline Comput...
Geant4, the leading detector simulation toolkit used in High Energy Physics, employs a set of physics models to simulate interactions of particles with matter across a wide range of interaction energies. These models, especially the hadronic ones, rely largely on directly measured cross-sections and inclusive characteristics, and use physically motivated

86. High-performance end-user analysis in pure Julia programming language
Ling, Jerry (Harvard University)
5/11/23, 11:15 AM
Track 6 - Physics Analys... Oral Track 6 - Physics Analys...
We present tools for high-performance analysis written in pure Julia, a just-in-time (JIT) compiled dynamic programming language with a high-level syntax and performance. The packages we present center around UnROOT.jl, a pure Julia ROOT file I/O package that is optimized for speed, lazy reading, flexibility, and thread safety.

Overview
Announcements
chep-info@cern.ch Mailing List
Scientific Program
Timetable
Instructions to Speakers
Poster Printing
Call for Abstracts
CHEP Code of Conduct
Conference Website
Accommodations
Registration
Participants
"Birds of a Feather" rooms
CHEP Excursions
CHEP Venue Map
Proceedings
Contribution List

Embedding Julia in Python

How easy is it to blend these languages?

More about configuration and run-time environment see “Power of Python and Julia for Advanced Data Analysis” talk at JuliaHEP2024 workshop

- We can use PythonCall for integrating Python’s vast ecosystem into Julia projects and JuliaCall for embedding high-performance Julia code into Python scripts.

```
[ ]: from juliacall import Main as jl
```

```
[ ]: %load_ext juliacall
```

```
[ ]: %%julia

     using Pkg
     Pkg.add("UnROOT")
     using UnROOT
```

Using Julia Packages from Python

[1]:
```
from juliacall import Main as jl
```

```
Detected IPython. Loading juliacall extension. See https://juliapy.github.io/PythonCall.jl/stable/compat/#IPython
```

[2]:
```
%load_ext juliacall
```

```
WARNING: replacing module _ipython.
```

[3]:
```
%%julia

using Pkg
Pkg.add("UnROOT")
using UnROOT
```

```
   Resolving package versions...
  No Changes to `~/anaconda3/envs/julia_hep_2024/julia_env/Project.toml`
  No Changes to `~/anaconda3/envs/julia_hep_2024/julia_env/Manifest.toml`
```

ROOT File as Julia Object in Python

Using UnROOT

- This dataset contains about 60 mio. data events from the CMS detector taken in 2012 during Run B and C. The original AOD dataset is converted to the NanoAOD format and reduced to the muon collections.
- Wunsch, Stefan; (2019). DoubleMuParked dataset from 2012 in NanoAOD format reduced on muons. CERN Open Data Portal. DOI:10.7483/OPENDATA.CMS.LVG5.QT81

```
[175]: %%julia

       using UnROOT

       @time big_tree = ROOTFile("../../../Run2012BC_DoubleMuParked_Muons.root")

         0.007673 seconds (4.65 k allocations: 10.119 MiB)

[175]: ROOTFile with 2 entries and 17 streamers.
       ../../../Run2012BC_DoubleMuParked_Muons.root
       └ Events (TTree)
           ├ "nMuon"
           ├ "Muon_pt"
           ├ "Muon_eta"
           ├ "Muon_phi"
           ├ "Muon_mass"
           └ "Muon_charge"
```

```
[176]: jl.big_tree

[176]: ROOTFile with 2 entries and 17 streamers.
       ../../../Run2012BC_DoubleMuParked_Muons.root
       └ Events (TTree)
           ├ "nMuon"
           ├ "Muon_pt"
           ├ "Muon_eta"
           ├ "Muon_phi"
           ├ "Muon_mass"
           └ "Muon_charge"
```

```
[174]: %%julia

       @time events = LazyTree(big_tree, "Events")

         0.000334 seconds (365 allocations: 31.703 KiB)
```

[174]: 61,540,413 rows × 6 columns (omitted printing of 61,540,403 rows)

	Muon_phi	nMuon	Muon_pt	Muon_eta	Muon_charge	Muon_mass
	SubArray{Float3	UInt32	SubArray{Float3	SubArray{Float3	SubArray{Int32,	SubArray{Float3
1	[-0.0343, 2.54]	2	[10.8, 15.7]	[1.07, -0.564]	[-1, -1]	[0.106, 0.106]
2	[-0.275, 2.54]	2	[10.5, 16.3]	[-0.428, 0.349]	[1, -1]	[0.106, 0.106]
3	[-1.22]	1	[3.28]	[2.21]	[1]	[0.106]
4	[-2.08, 0.251, -2.01, -1.85]	4	[11.4, 17.6, 9.62, 3.5]	[-1.59, -1.75, -1.59, -1.66]	[1, 1, 1, 1]	[0.106, 0.106, 0.106, 0.106]

PRINCETON UNIVERSITY

Julia ROOT Tree in Python

Faster way to read ROOT files

```
[7]: events = jl.Main.LazyTree(file, "Events")

[8]: %%timeit
     jl.Main.LazyTree(file, "Events")

     368 µs ± 23.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

- With viewing the data as AwkwardArray we can use either Julia or Python analysis code or even combine both languages.

AwkwardArray.jl

AwkwardArray.jl as Data Bridge

Between Julia and Python

- Using AwkwardArray in Julia to calculate Higgs mass:

Awkward Array.jl

for Julia!

Search docs

- Introduction
- Getting started
  - Installation
  - Using Julia Awkward Arrays from Python
  - Using Python Awkward Arrays from Julia
- Converting Arrays
- API
  - Types
  - Functions
- LICENSE

Version dev

Getting started   Edit on GitHub

Getting started

Let's assume that both Python and Julia are installed.

Note

If Julia is not install it is recomended to follow its official installation instructions described here.

Installation

It is recommended to use conda virtul environment.

Using Julia Awkward Arrays from Python

- To install Awkward Array Python package:

```
conda install -c conda-forge awkward
```

- To install JuliaCall:

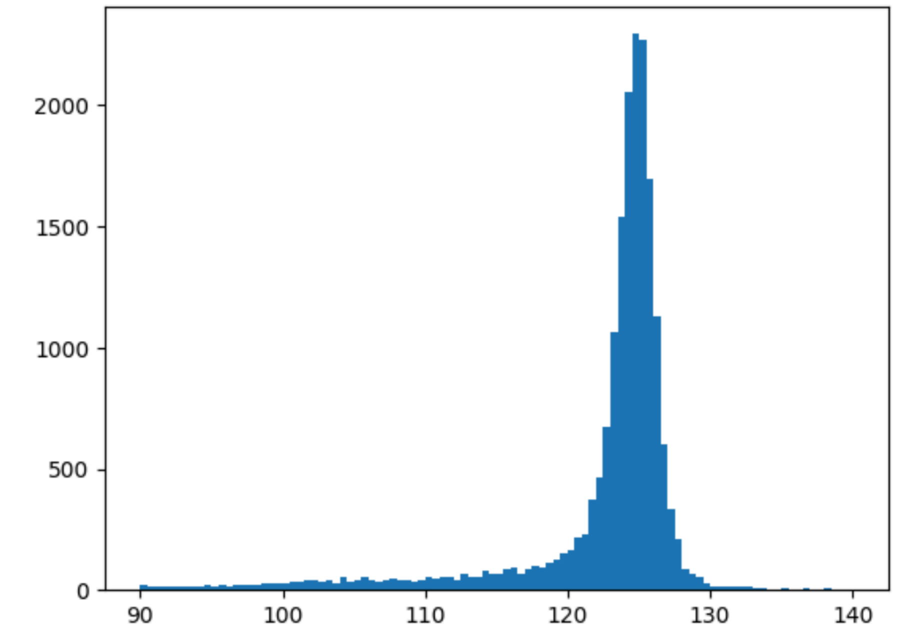
```
conda install pyjuliacall
```

JuliaCall takes care of installing all necessary Julia packages, including this package.

```
import awkward as ak
from juliacall import Main as jl

jl.seval("using AwkwardArray")
```

```
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
```

```
[16]: import matplotlib.pyplot as plt

[17]: %%timeit
      plt.hist(jl.array, bins=100, range=(90, 140));

66.9 ms ± 8.05 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```



Including Julia Code in Python

Notes on code organization

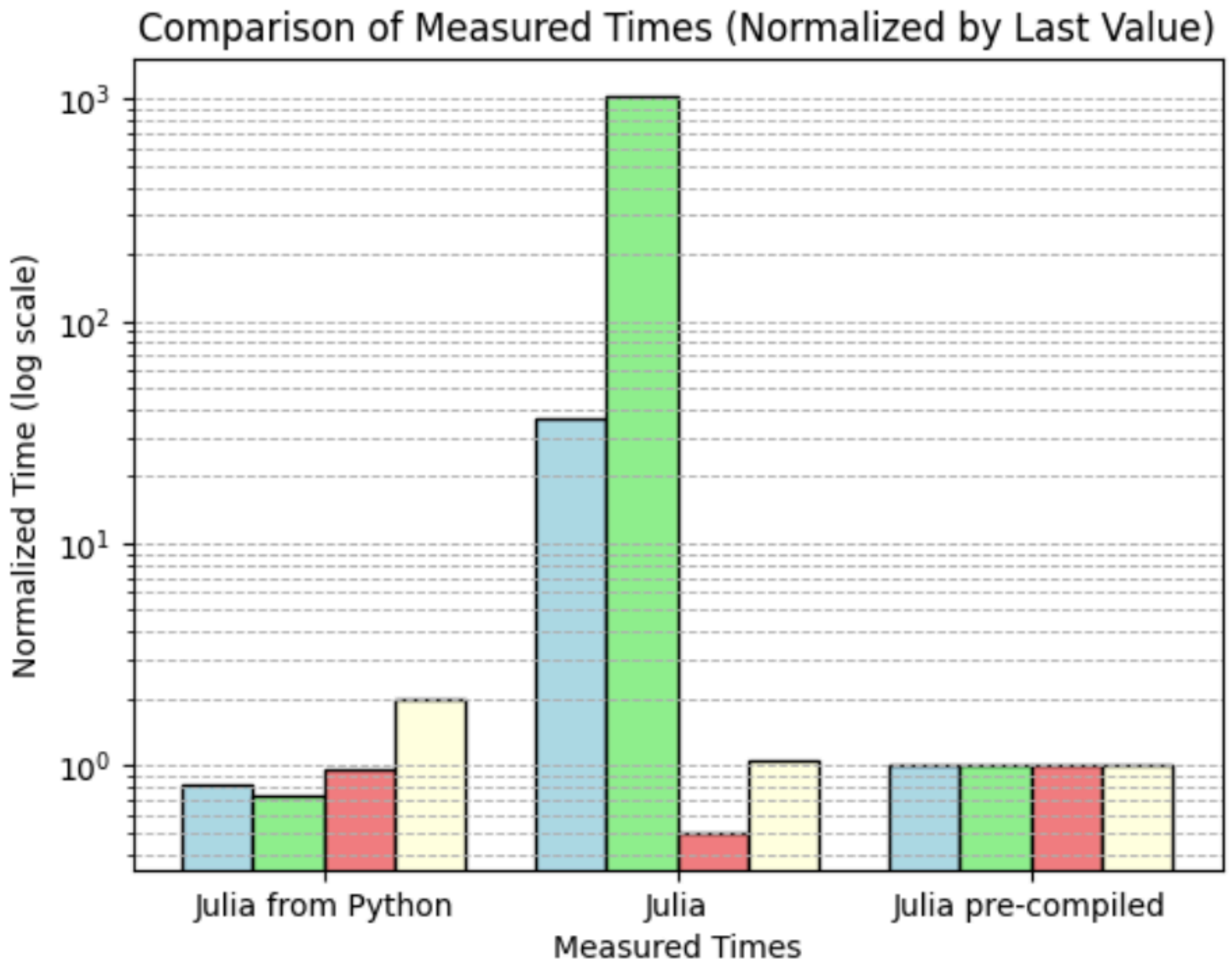
```julia
# Predefine the output structure with a concrete NamedTuple type
const RecordArrayType = NamedTuple{(:pt, :eta, :phi, :mass, :charge, :isolation)}

function make_record_array(
  events::NamedTuple{(:muon,),
  Tuple{
     NamedTuple{(:pt, :eta, :phi, :mass, :charge, :pfRelIso03_all),
       Tuple{
         Vector{T}, Vector{T}, Vector{T}, Vector{T}, Vector{T}, Vector{T}
       }
     }
   }
 }) where T

    # Convert the relevant fields into AwkwardArray arrays
    array = AwkwardArray.RecordArray(
      RecordArrayType((
        AwkwardArray.from_iter(events.muon.pt),
        AwkwardArray.from_iter(events.muon.eta),
        AwkwardArray.from_iter(events.muon.phi),
        AwkwardArray.from_iter(events.muon.mass),
        AwkwardArray.from_iter(events.muon.charge),
        AwkwardArray.from_iter(events.muon.pfRelIso03_all)
      ))
    )

    return array
end
```

```python
[20]: jl.include('awkward_analyzer_functions.jl');
```

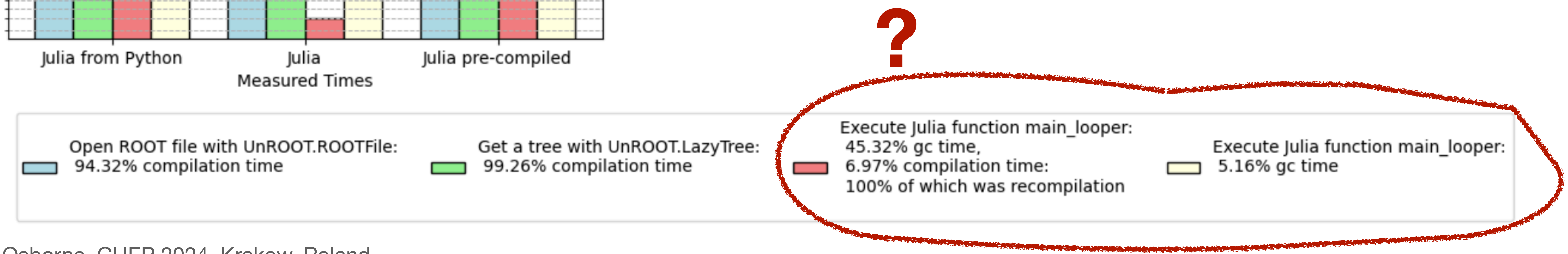
- Provide the correct path when using the include function.
- If your project grows larger, consider structuring your code into more modules and files for better organization:
  - It is generally a good practice to organize your code into modules. This helps with namespace management and reduces the likelihood of name collisions.
  - Use export to expose functions from a module. This makes it easy to access the desired functionality after including a module.

Calling Julia from Python Efficiency

with a very small overhead

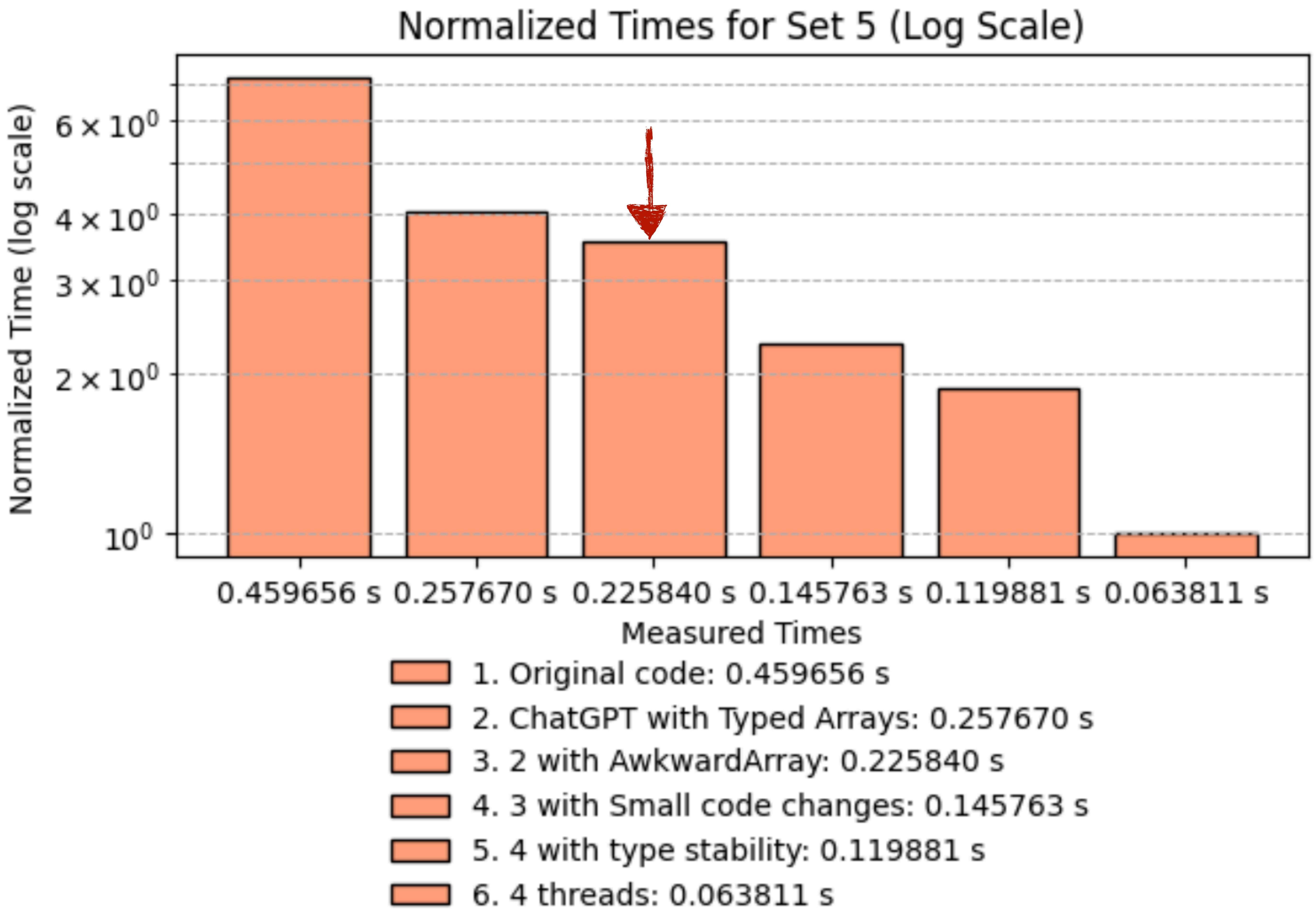


- Getting the best performance from Julia requires us to focus on type-stability and good practices for reducing unnecessary recompilation.



AwkwardArray.jl Overhead

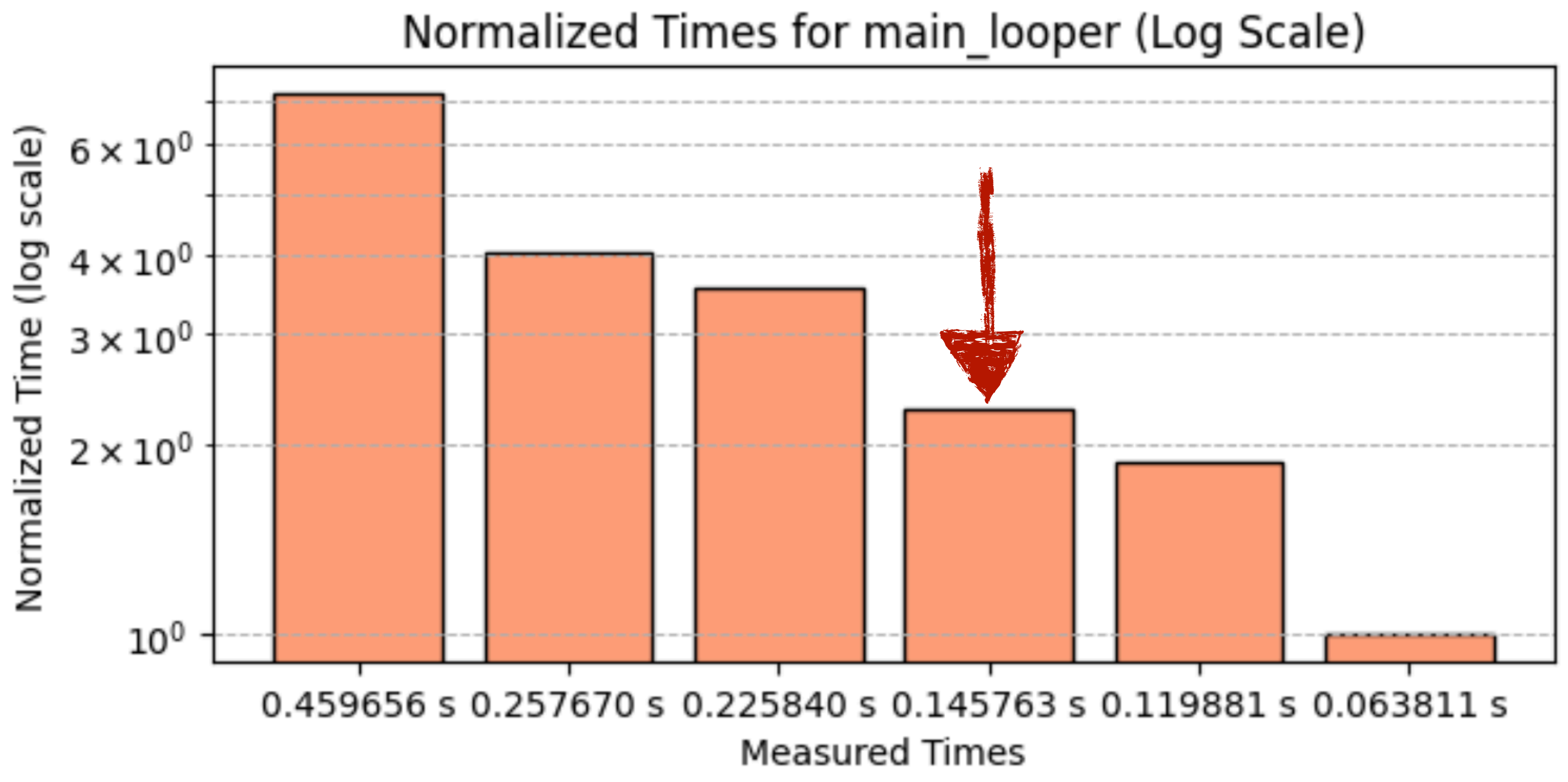
Compared with Using Typed Arrays



- Started with using AwkwardArray and compared it to a Julia native typed array: Vector
- **Takeaway**: no significant overhead seen after small changes to Julia main_looper code.

Small Code Changes

in Destructure and Skip



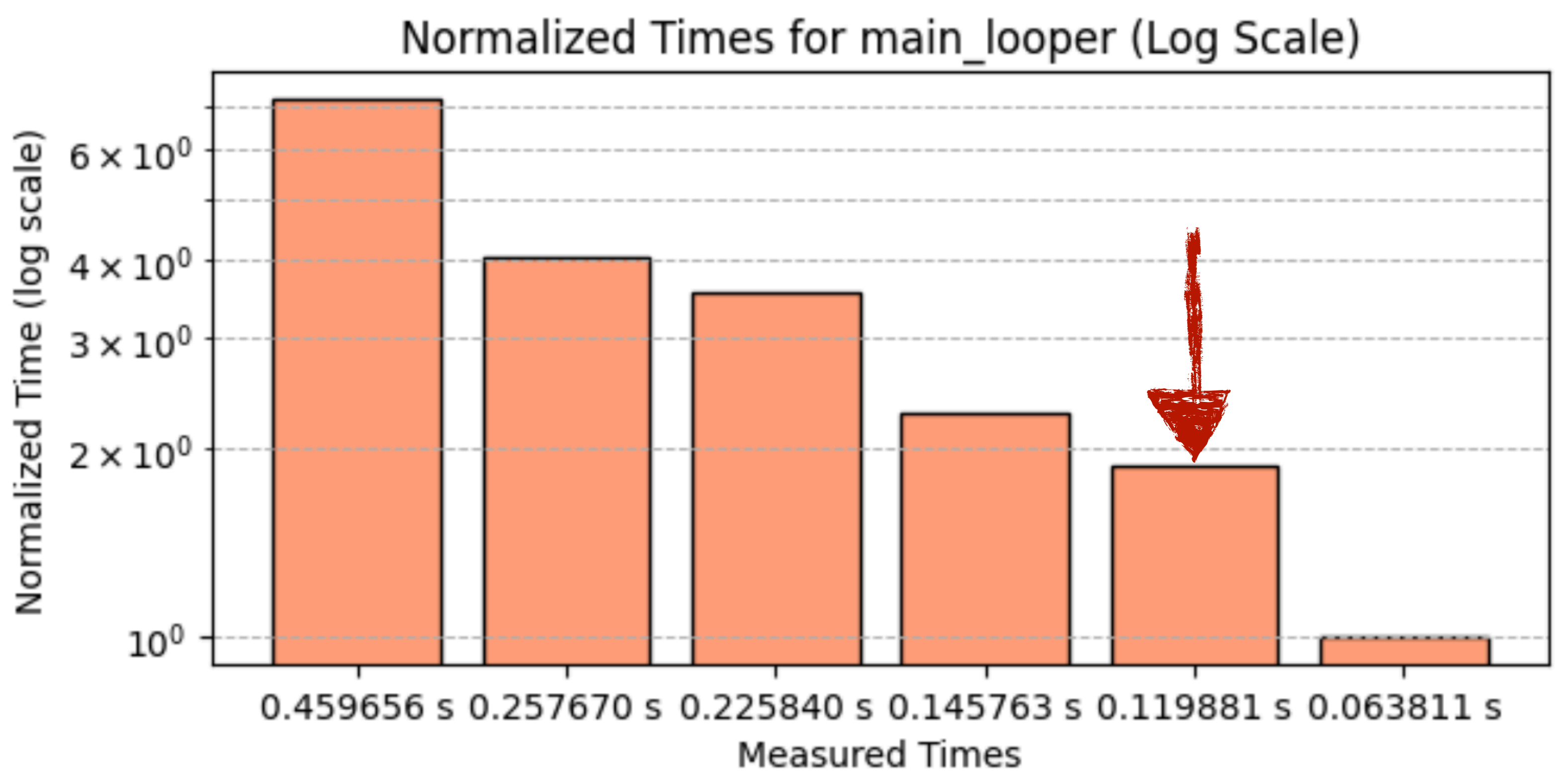
ChatGPT

```
function main_looper(events)
    # Create an empty AwkwardArray for storing the Higgs mass values
    array = AwkwardArray.PrimitiveArray{Float64}()

    # Loop over events and process only valid ones
    for evt in events
        # Destructure the necessary fields from the event
        (; Muon_charge, Muon_pt, Muon_eta, Muon_phi, Muon_mass) = evt

        # Skip event if it doesn't meet the required conditions
        if length(Muon_charge) != 4 || sum(Muon_charge) != 0
            continue
        end

        # Create Lorentz vectors for the muons and calculate the Higgs mass
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        # Add the result to the AwkwardArray
        push!(array, higgs_mass)
    end

    # Return the final AwkwardArray containing Higgs masses
    return array
end
```

PRINCETON UNIVERSITY

Ensuring Type Stability

ChatGPT help



- Avoid unnecessary recompilation!

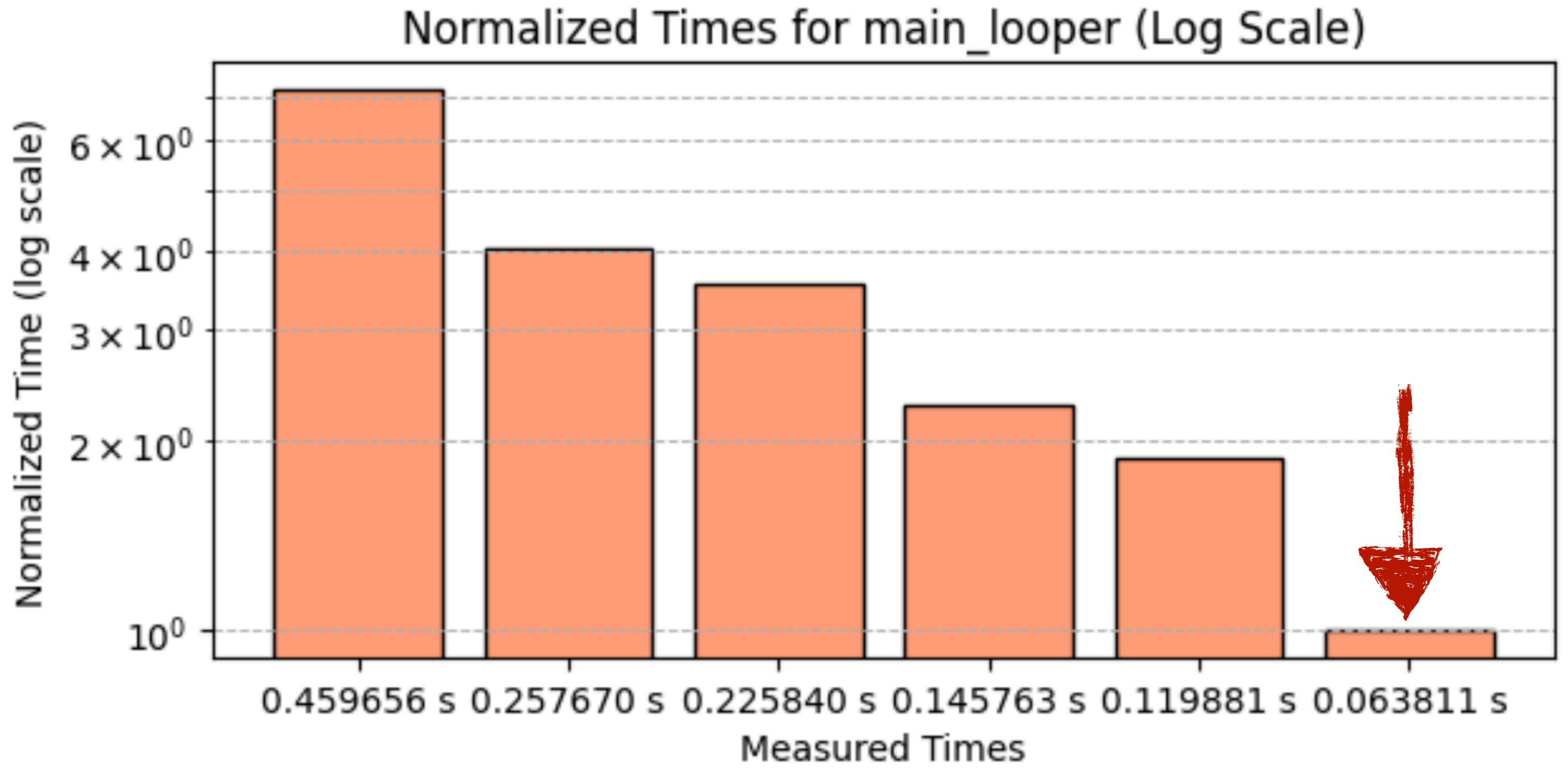
ChatGPT

```
function main_looper(events::AwkwardArray.RecordArray)
    # Pre-allocate an AwkwardArray to store Higgs mass values
    array = AwkwardArray.PrimitiveArray{Float64}(undef, length(events))
    count = 0  # To track valid entries

    for evt in events
        # Destructure the necessary fields from the event with concrete types
        (; Muon_charge::Vector{Float64}, Muon_pt::Vector{Float64}, Muon_eta::Vector{Float64},
           Muon_phi::Vector{Float64}, Muon_mass::Vector{Float64}) = evt

        # Check conditions with inlined logic
        if length(Muon_charge) != 4 || sum(Muon_charge) != 0
            continue
        end

        # Compute the Lorentz vector sum and Higgs mass
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        # Store the result in the pre-allocated array
        count += 1
        array[count] = higgs_mass
    end

    # Resize the array to only include valid entries
    return AwkwardArray.subarray(array, 1:count)
end
```

PRINCETON UNIVERSITY

Multithreading Support

Experimental in JuliaCall



- **Execution time reduced by 88%**, speeding up from 0.5 seconds to 0.06 seconds—a **8.33x performance improvement**.
- **Memory usage optimized**, cutting allocations from 398k to 24k, making the process much more efficient.
- Overall, the code is **much faster and leaner**, showing significant gains in both speed and memory management.

```
% export JULIA_NUM_THREADS=4
```

```
% export PYTHON_JULIACALL_HANDLE_SIGNALS=yes
```

```
[89]: %%julia

using AwkwardArray
using Base.Threads

function main_looper_awkward(events)
    array = AwkwardArray.PrimitiveArray{Float64}()
    lock_obj = ReentrantLock()  # Create a lock object to control access to shared array

    @threads for i in 1:length(events)
        evt = events[i]

        # Destructure the necessary fields from the event
        (; Muon_charge, Muon_pt, Muon_eta, Muon_phi, Muon_mass) = evt

        # Skip event if it doesn't meet the required conditions
        if length(Muon_charge) != 4 || sum(Muon_charge) != 0
            continue
        end

        # Create Lorentz vectors for the muons and calculate the Higgs mass
        higgs_4vector = sum(LorentzVectorCyl.(Muon_pt, Muon_eta, Muon_phi, Muon_mass))
        higgs_mass = mass(higgs_4vector)

        # Use lock to safely push! into the shared array
        lock(lock_obj)  # Explicitly lock before modifying shared data
        try
            push!(array, higgs_mass)
        finally
            unlock(lock_obj)  # Ensure the lock is always released
        end
    end

    return array
end
```

Can we do better?

Multi-processing and Distributed Computing

```
[7]: %%julia

     @everywhere include("DummyAwkwardModule.jl")

[8]: %%julia

     using .DummyAwkwardModule

[9]: %%julia

     MuMu = @spawnat :any invariant_mass(events);

[11]: %%julia

     @time fetch(MuMu)

       0.000005 seconds

[11]: 5638149-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default
      113.64686584472656
       88.29710388183594
       88.33483123779297
       91.27149963378906
       93.55725860595703
       90.91211700439453
       89.15238952636719
       82.29732513427734
       94.57678985595703
       89.23975372314453
        ⋮
```

```julia
module DummyAwkwardModule

export invariant_mass

using LorentzVectorHEP, AwkwardArray
using UnROOT

using Base.Threads

function invariant_mass(cms_events)

    array = AwkwardArray.PrimitiveArray{Float64}()
    lock_obj = ReentrantLock()  # Create a lock object to control access to shared array

    @threads for i in 1:length(cms_events)
        evt = cms_events[i]

        # Destructure the necessary fields from the event
        (; Muon_charge, Muon_pt, Muon_eta, Muon_phi, Muon_mass, nMuon) = evt

        # Skip event if it doesn't meet the required conditions
        if nMuon != 2 || Muon_charge[1] == Muon_charge[2]
            continue
        end

        # Calculate invariant mass using LorentzVectorHEP for clarity and accuracy
        muon1 = LorentzVectorCyl(Muon_pt[1], Muon_eta[1], Muon_phi[1], Muon_mass[1])
        muon2 = LorentzVectorCyl(Muon_pt[2], Muon_eta[2], Muon_phi[2], Muon_mass[2])
        result = mass(muon1 + muon2)

        # Only add masses greater than 70 GeV
        if result > 70
            # Use lock to safely push! into the shared array
            lock(lock_obj)  # Explicitly lock before modifying shared data
            try
```

https://docs.julialang.org/en/v1/manual/distributed-computing/

julia

Search docs (Ctrl + /)

Manual / Multi-processing and Distributed Computing

GitHub

Multi-processing and Distributed Computing

An implementation of distributed memory parallel computing is provided by module `Distributed` as part of the standard library shipped with Julia.

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache. Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI[1]. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Distributed programming in Julia is built on two primitives: *remote references* and *remote calls*. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

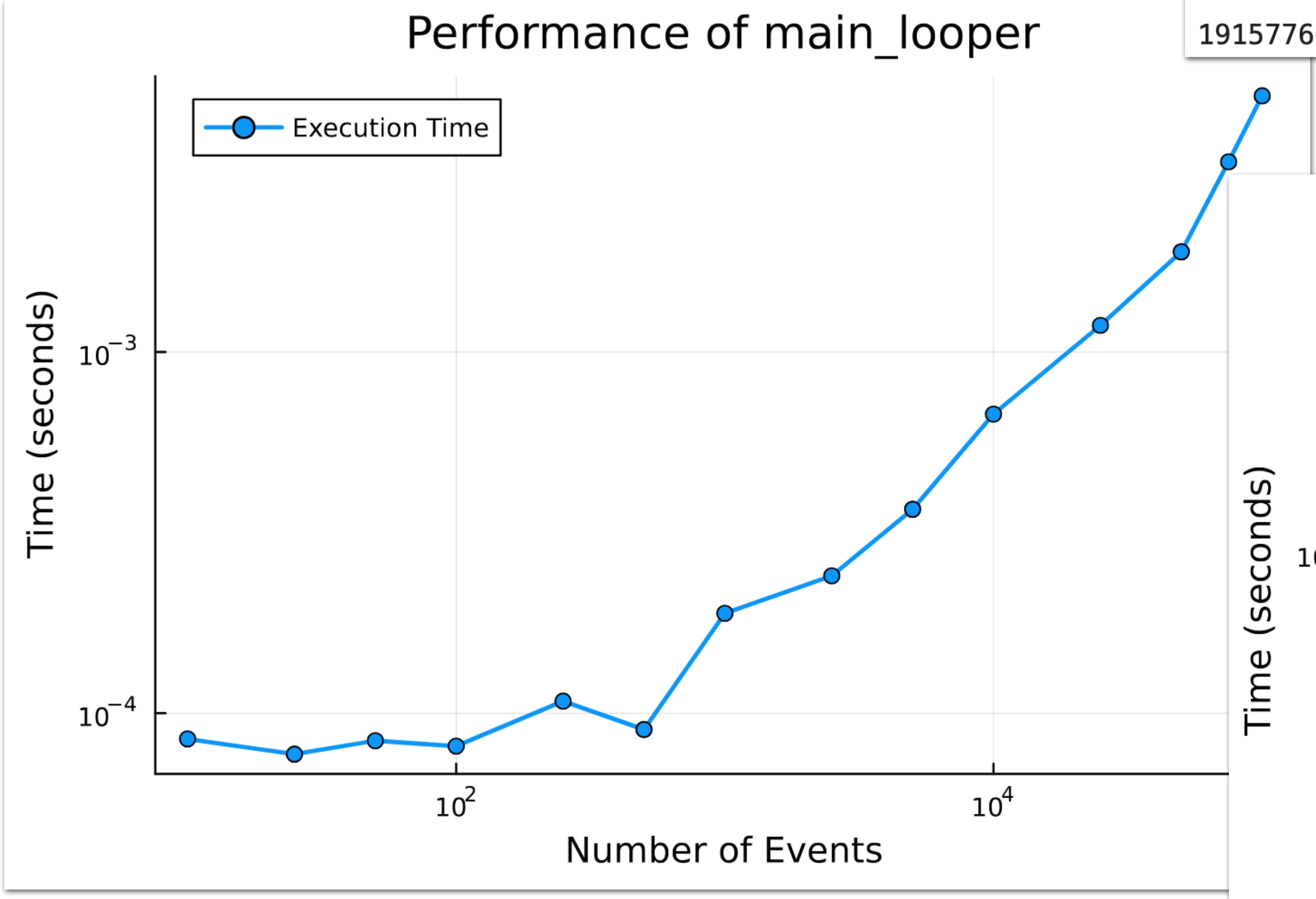
Remote references come in two flavors: `Future` and `RemoteChannel`.

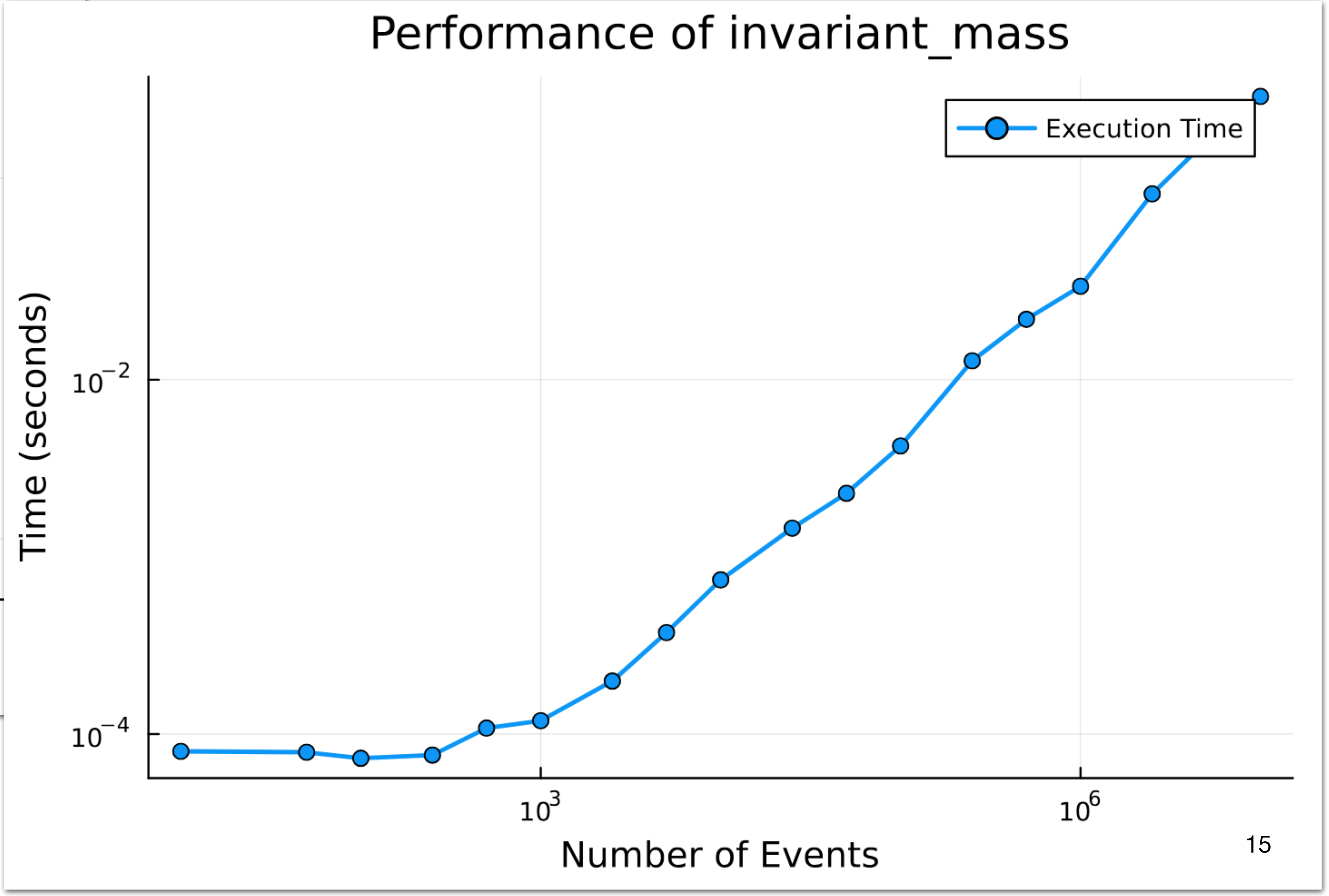
Performance Scaling

Increasing # Events

```
@time main_looper(events)
  9.897901 seconds (2.09 M allocations: 15.559 GiB, 16.83% gc time)
1915776-element AwkwardArray.PrimitiveArray{Float64, Vector{Float64}, :default}:
```

61,540,413

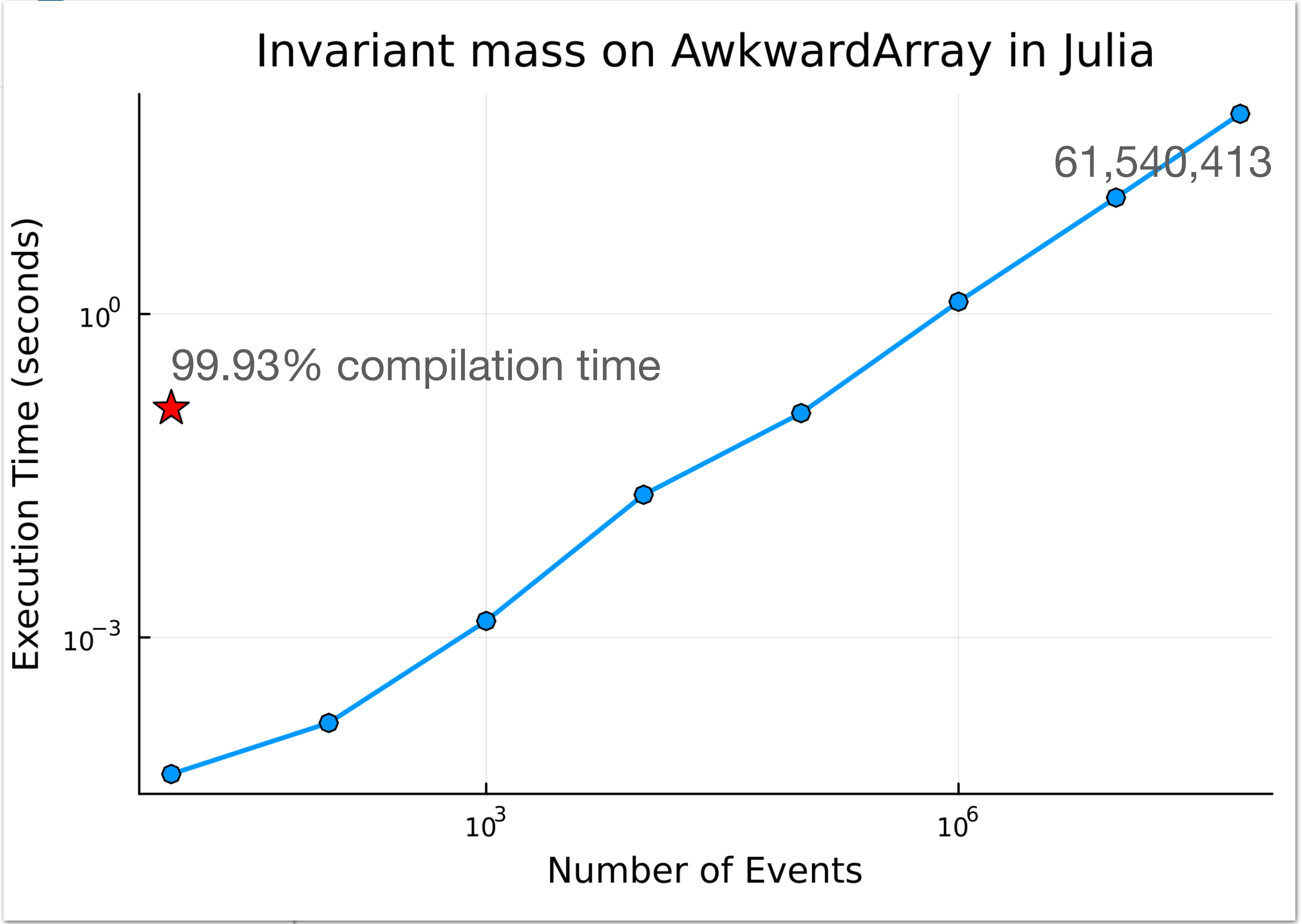




Performance Scaling

Increasing # Events

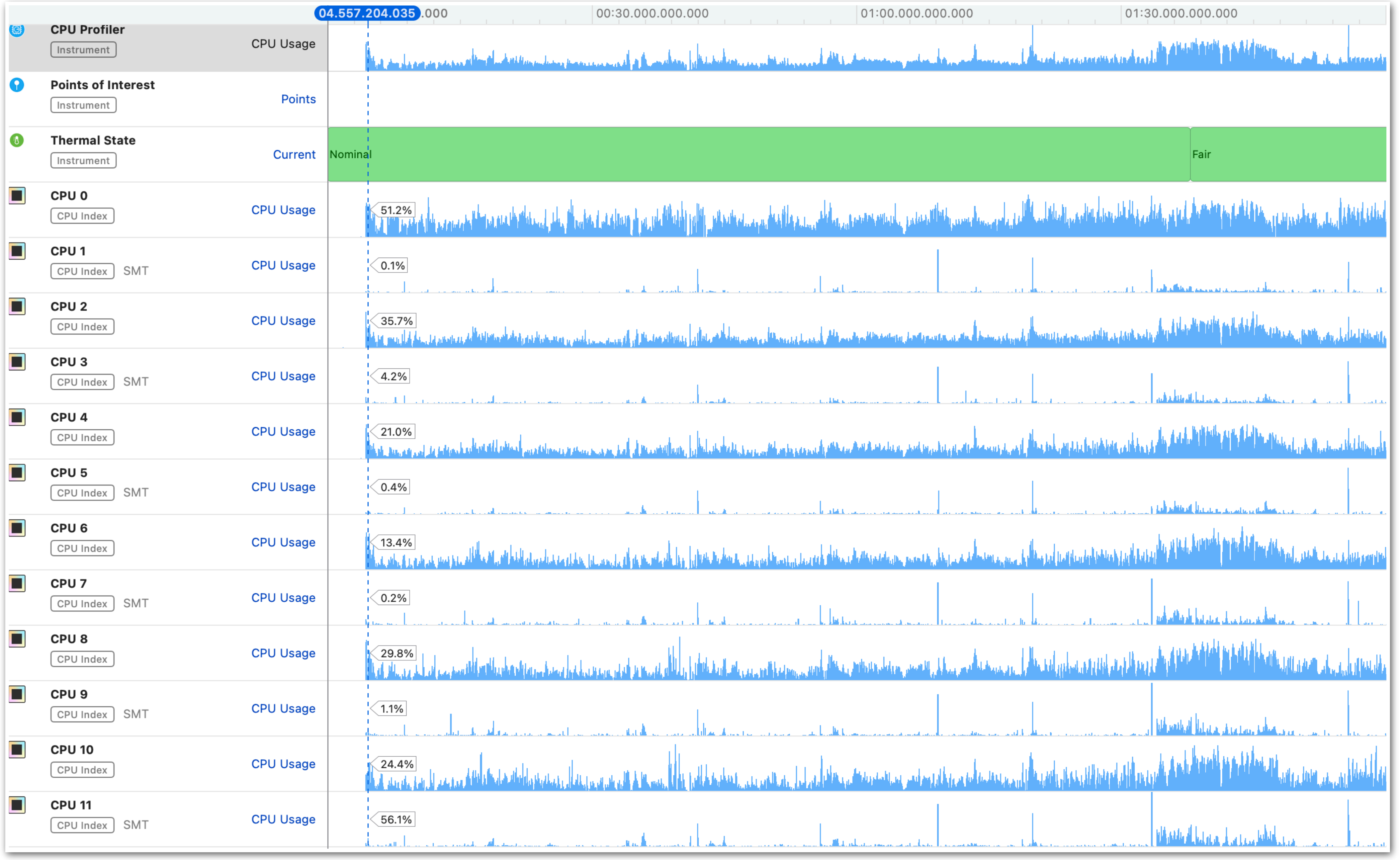
```
function invariant_mass(events::AwkwardArray.RecordArray)
    array = AwkwardArray.PrimitiveArray{Float64}()

    for i in 1:length(events)
        event = events[i]

        if event[:nMuon] != 2 ||
           event[:charge][1] == event[:charge][2]
            continue
        end

        muon1 = LorentzVectorCyl.(
            event[:pt][1], event[:eta][1], event[:phi][1],
        muon2 = LorentzVectorCyl.(
            event[:pt][2], event[:eta][2], event[:phi][2],
        result = mass(muon1 + muon2)

        if result > 70
            push!(array, result)
        end
    end
    return array
end
```

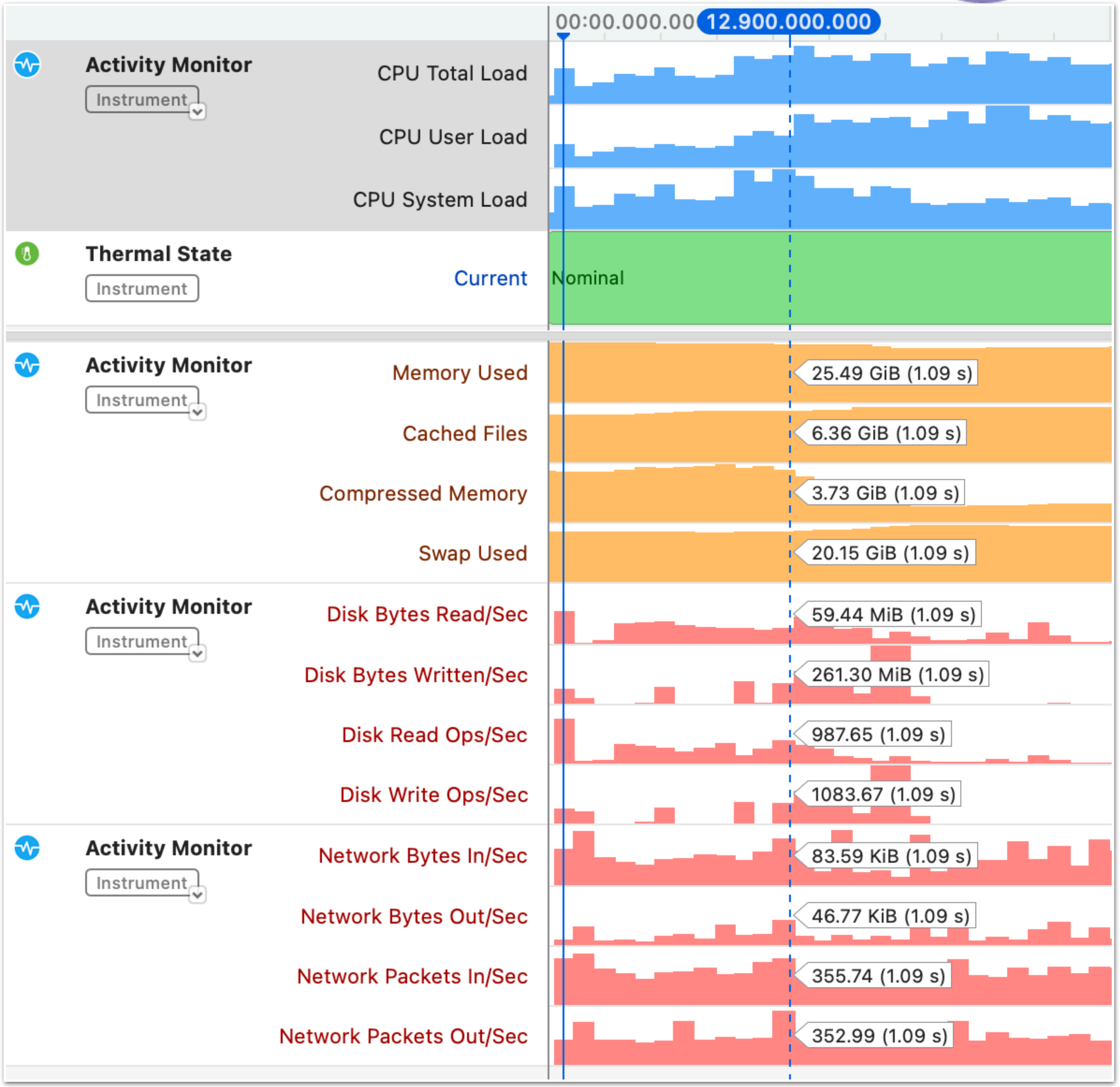


Instruments

Activity Monitor

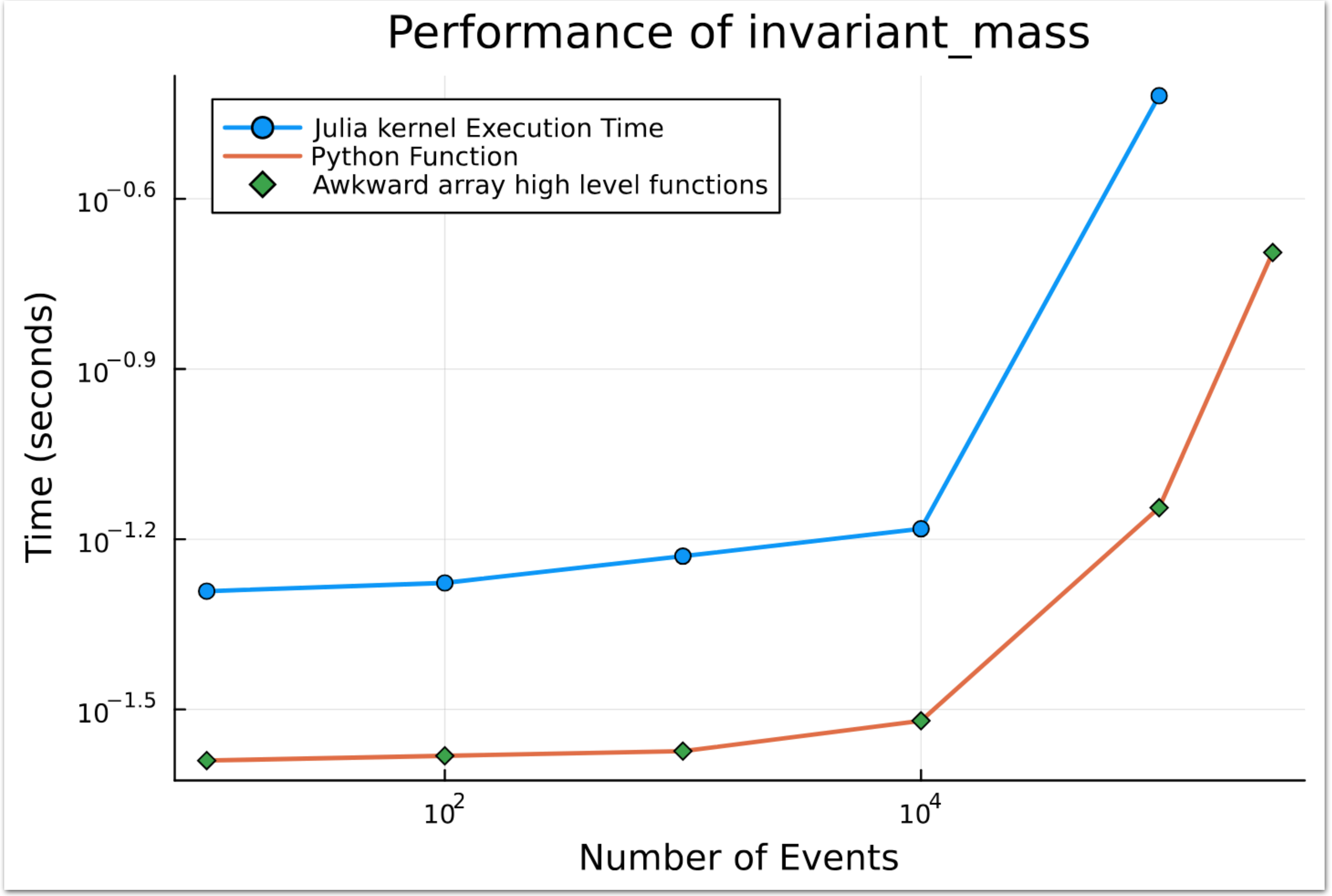
- macOS Big Sur version 11.6
- Processor 2.6 GHz 6-Core Intel Core i7
- Memory 32 GB 2667 MHz DDR4





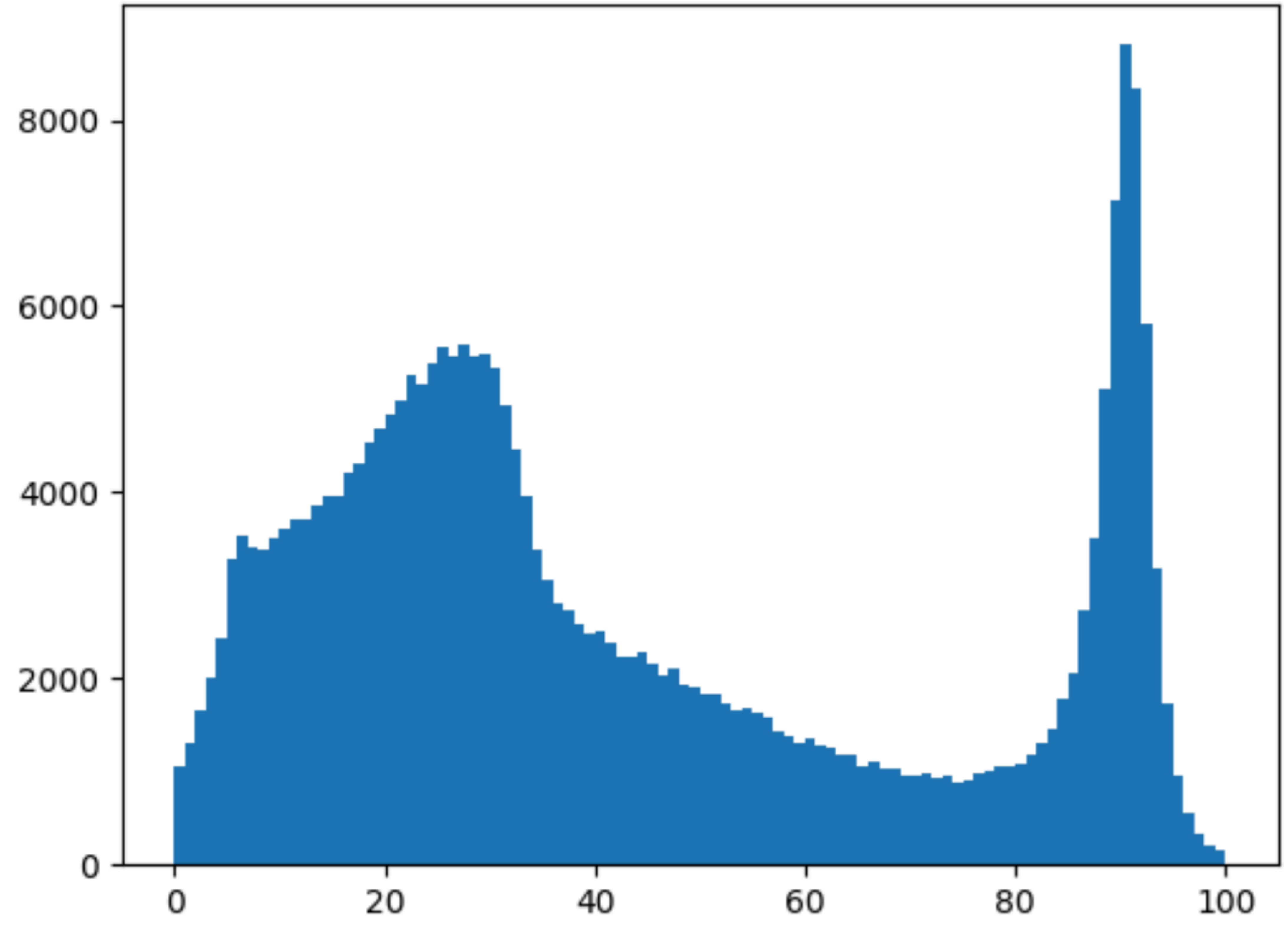
Performance Comparison

Increasing # Events




```
[28]: def mass_fun(events):
          muplus = events.muon[events.muon.charge > 0]
          muminus = events.muon[events.muon.charge < 0]
          mu1, mu2 = ak.unzip(ak.cartesian((muplus, muminus)))
          return ak.ravel((mu1 + mu2).mass)
```

```
[29]: import matplotlib.pyplot as plt

      plt.hist(mass_fun(events), bins=100, range=(0, 100));
```



Summary

Optimizing Performance with Julia

- While we may not see a significant speedup from replacing NumPy, Awkward, or Numba with Julia in vectorized operations, identifying tasks that don't fit well with these libraries can unlock Julia's true potential.
- Developing custom kernels for specific problems may lead to innovative solutions, even if it’s not immediately obvious.
- Despite challenges in multilingual runtime environments and experimental thread support, the ongoing evolution of Julia offers exciting opportunities for performance enhancement.