

# Statically Compile Julia code

~~to surpass~~ `numpy` for fun and profit

Jerry Ling<sup>1</sup>

PyHEP 2025 @ Seattle, WA

---

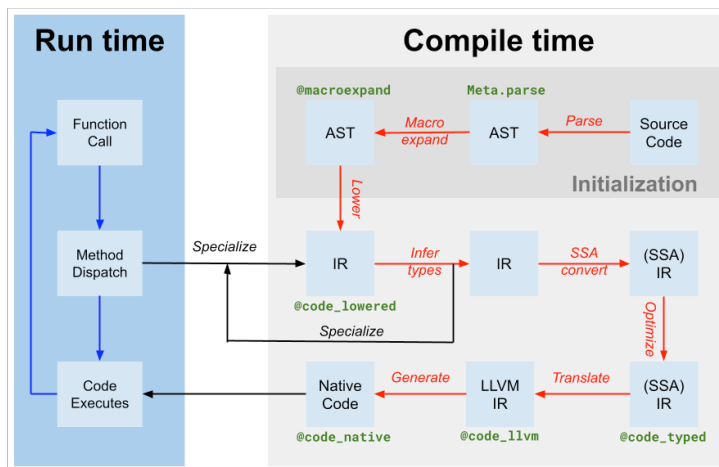
<sup>1</sup>Harvard University

# Motivation

- We want to write more code in Julia

# Motivation

- ~~We want to write more code in Julia~~
- Julia has excellent runtime performance and development experience thanks to [Just-In-Time \(JIT\) compilation](#), but static compilation hasn't been a focus so far,



## Motivation cont.

- Some applications needs embedding of Julia into other languages/frameworks
  1. Integrate as dynamic library in CMSSW/Athena
  2. Julia backend for Python packages (latency-sensitive use pattern)
- Although, even **without static compilation**, Julia has found success in being backend of Python libraries
  - E.g. `pysr`, `diffeqpy`

*PySR & SymbolicRegression.jl*



[github.com/MilesCranmer/pysr\\_paper](https://github.com/MilesCranmer/pysr_paper)

### Interpretable Machine Learning for Science with PySR and SymbolicRegression.jl

Miles Cranmer<sup>1,2</sup>

<sup>1</sup>*Princeton University, Princeton, NJ, USA*

<sup>2</sup>*Flatiron Institute, New York, NY, USA*

# This talk covers

1. Detailed example of statically compile `sum()` to a library in Julia
2. Demo wrapping `FHist.jl` + benchmark against `numpy.histogram`

## This talk covers

1. Detailed example of statically compile `sum()` to a library in Julia
2. Demo wrapping `FHist.jl` + benchmark against `numpy.histogram`

## This talk does NOT cover

1. How the Julia compilation pipeline works
2. How `--trim` works in static compilation
3. If specific dynamism will work or not with static compilation

## This talk covers

1. Detailed example of statically compile `sum()` to a library in Julia
2. Demo wrapping `FHist.jl` + benchmark against `numpy.histogram`

## This talk does NOT cover

1. How the Julia compilation pipeline works
2. How `--trim` works in static compilation
3. If specific dynamism will work or not with static compilation

Finally, code for this talk can be found at [C\\_FHist branch of FHist.jl repo](#)

## Julia code to a `.so` library

Let's use Julia's built-in `sum` to replace `numpy.sum`:

```
$ cat jlsum.jl
Base.@ccallable function jlsum(p::Ptr{Cdouble}, N::Clong)::Cdouble
    np_input = unsafe_wrap(Array{Float64}, p, N, own=false)
    return sum(np_input)
end
```



Tell Julia this function must be C-compatible.



```
Base.@ccallable function jlsum(p::Ptr{Cdouble}, N::Clong)::Cdouble
    np_input = unsafe_wrap(Array{Float64}, p, N, own=false)
    return sum(np_input)
end
```



takes an array by pointer, and return a double.

```
Base.@ccallable function jlsum(p::Ptr{Cdouble}, N::Clong)::Cdouble
    np_input = unsafe_wrap(Array{Float64}, p, N, own=false)
    return sum(np_input)
end
```



Make a Julia array from the pointer.

```
Base.@ccallable function jlsum(p::Ptr{Cdouble}, N::Clong)::Cdouble
    np_input = unsafe_wrap(Array{Float64}, p, N, own=false)
    return sum(np_input)
end
```



This memory is owned by Python, Julia should not free it later.

## Compile with `juliac` script

Assume you're running a Julia version that can work with the [juliac.jl script](#):

```
$ julia juliac.jl --output-lib libjlsum.so --compile-ccallable --  
experimental --trim jlsum.jl
```



output `.so` library file

```
$ julia juliac.jl --output-lib libjlsum.so --compile-ccallable --  
experimental --trim jlsum.jl
```

make functions marked with `@ccallable` callable




```
$ julia juliac.jl --output-lib libjlsum.so --compile-ccallable --  
experimental --trim jlsum.jl
```



trim output by reachable code analysis

```
$ julia juliac.jl --output-lib libjlsun.so --compile-ccallable --  
experimental --trim jlsum.jl
```

```
$ ls -lh libjlsun.so  
1.0M Jul  6 13:14 libjlsun.so*
```



not the smallest `.so`, but much better than 150MB if untrimmed

This also compiles faster with trim enabled, about 2 seconds.

# Calling from Python

At this point, the `.so` produced by Julia is no different from any C shared library. Which means it's as easy (or hard) to warp them from Python.



# Calling from Python

At this point, the `.so` produced by Julia is no different from any C shared library. Which means it's as easy (or hard) to warp them from Python.

For this talk, we use `ctypes` in Python stdlib. It doesn't have the most fancy tooling, but gets the job done for us.

```
import ctypes
```

```
lib = ctypes.CDLL('./libjlsun.so')
```

# Calling from Python

At this point, the `.so` produced by Julia is no different from any C shared library. Which means it's as easy (or hard) to warp them from Python.

For this talk, we use `ctypes` in Python stdlib. It doesn't have the most fancy tooling, but gets the job done for us.

```
import ctypes
```

```
lib = ctypes.CDLL('./libjlsun.so')
```

```
lib.jlsun.argtypes = [  
    ctypes.POINTER(ctypes.c_double), ctypes.c_long,  
    ]
```

```
lib.jlsun.restype = ctypes.c_double
```

set input/output types



# Calling from Python

Finally, we can make a Python function as user-friendly wrapper:

```
def jlsum(a):  
    res = lib.jlsum(  
        a.ctypes.data_as(ctypes.POINTER(ctypes.c_double)),  
        ctypes.c_long(len(a)),  
    )  
    return res
```

# Benchmark?

```
In [1]: import numpy as np
```

```
In [2]: ary = np.random.rand(10**6)
```

```
In [3]: np.isclose(np.sum(ary), jlsum(ary))
```

```
Out[3]: np.True_
```

```
In [4]: %timeit np.sum(ary)
```

```
114 µs ± 154 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

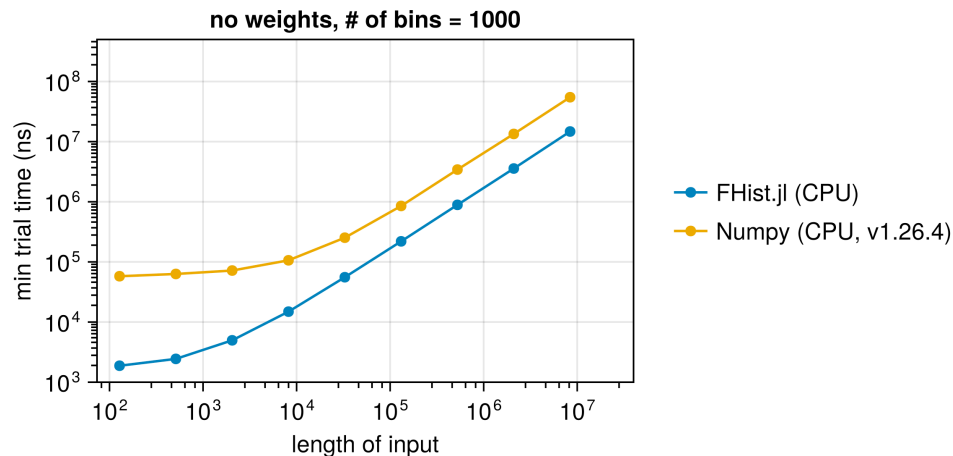
```
In [5]: %timeit jlsum(ary)
```

```
84.8 µs ± 319 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Results agree, and Julia version is slightly faster.

## A slightly more complex example: histogram


For the simplest kind of histogram (uniform binning, no weights), we can do significantly better than `numpy.histogram`:



Let's wrap an existing Julia histogram package, `FHist.jl`.

# Wrapping FHist.jl

```
using FHist: Hist1D, _fast_bincounts_1d!  
Base.@ccallable function hist1d(  
    input::Ptr{Cdouble}, Ninput::Clong,  
    bincounts::Ptr{Cdouble}, Nbincounts::Clong,  
    start::Cdouble, step::Cdouble, stop::Cdouble  
)::Cvoid  
  
# ...  
  
end
```



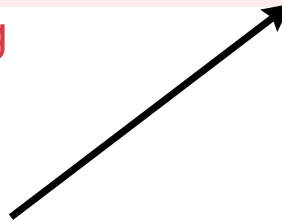
We can import functions from other packages.  
These are not specially made for this demo.

# Wrapping FHist.jl

```
using FHist: Hist1D, _fast_bincounts_1d!  
Base.@ccallable function hist1d(  
    input::Ptr{Cdouble}, Ninput::Clong,  
    bincounts::Ptr{Cdouble}, Nbincounts::Clong,  
    start::Cdouble, step::Cdouble, stop::Cdouble  
)::Cvoid  
# ..  
    gonna modify an array managed by Python to avoid benchmarking allocator  
end
```

# Wrapping FHist.jl

```
using FHist: Hist1D, _fast_bincounts_1d!  
Base.@ccallable function hist1d(...)::Cvoid  
    np_input = unsafe_wrap(...)  
    np_bincounts = unsafe_wrap(...)  
    binedges = start:step:stop  
    h = Hist1D(; bincounts=np_bincounts, binedges)  
    _fast_bincounts_1d!(h, np_input, binedges)  
    return nothing  
end
```



Make a histogram object and modify it



## Wrapping `FHist.jl` – Python side

```
lib = ctypes.CDLL('./libfhistjl.so')
lib.hist1d.argtypes = [
    ctypes.POINTER(ctypes.c_double), ctypes.c_long, # input and length
    ctypes.POINTER(ctypes.c_double), ctypes.c_long, # bincounts
    ctypes.c_double, ctypes.c_double, ctypes.c_double # binedges
]
lib.hist1d.restype = ctypes.c_voidp

def jlhist(a, bins, range):
    bincounts = np.zeros(bins) # manage this memory in Python
    step = (range[1] - range[0]) / bins
    lib.hist1d(...)
    return bincounts
```

# Benchmark histograms

```
In [1]: import numpy as np; ary=np.random.rand(10**5)
```

```
In [2]: np.allclose(np.histogram(ary, bins=10, range=(0,1))[0],  
    jlhist(ary, bins=10, range=(0,1))  
    ) # True
```

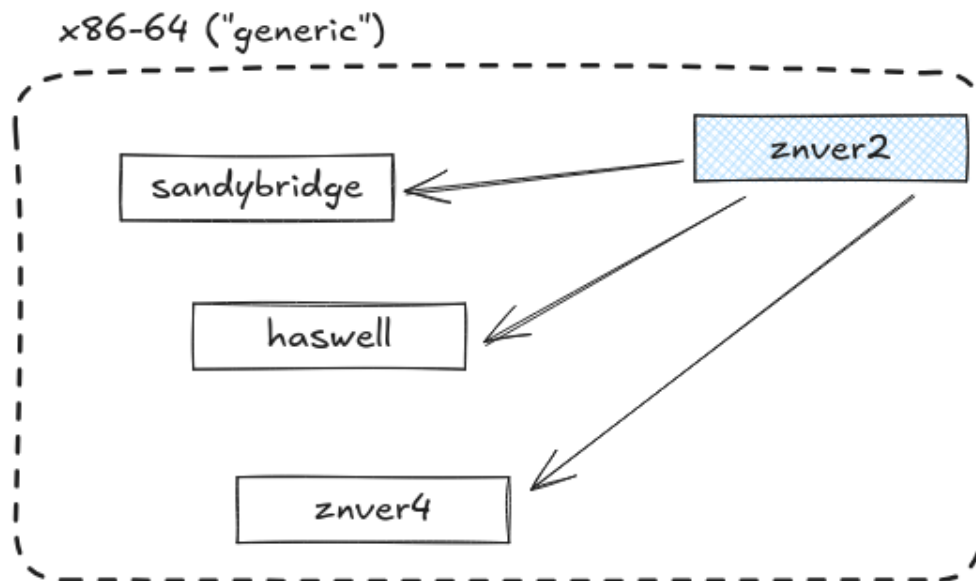
```
In [3]: %timeit np.histogram(ary, bins=10, range=(0,1))  
468 µs ± 8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
In [4]: %timeit jlhist(ary, bins=10, range=(0,1))  
91.3 µs ± 144 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops  
each)
```

About 4x faster!

# Function Multi-versioning

Although no cross-compilation, Julia supports [multi versioning](#), which means you can compile optimally for different CPU models given you have access to one system in the same architecture:



**Without** multi-versioning, one has to compile against a generic `x86_64` target:

```
# compiling on Intel, running on AMD Zen2
# JULIA_CPU_TARGET='generic;' julia juliac.jl ...
Numpy      time (μs): 6.538515799911693
FHist.jl   time (μs): 2.6081674004672095
```

**Without** multi-versioning, one has to compile against a generic `x86_64` target:

```
# compiling on Intel, running on AMD Zen2
# JULIA_CPU_TARGET='generic;' julia juliac.jl ...
Numpy      time (μs): 6.538515799911693
FHist.jl   time (μs): 2.6081674004672095
```

**With** multi-versioning:

```
# JULIA_CPU_TARGET='znver2;generic' julia juliac.jl ...
Numpy      time (μs): 6.641940600820817
FHist.jl   time (μs): 1.5721895993920043
```

# Real-life application in HEP

- [JetReconstruction.jl](#) is FastJet but written in Julia, potential integration as a statically compiled library.
  - “The performance of the statically compiled JetReconstruction.jl is compared with both native Julia code and C++ FastJet.”

For more info:

- [JuliaCon 2024: New Ways to Compile Julia Code](#)
- [JuliaHEP 2024: Julia as a Statically-Compiled Language](#)

# Summary

- Julia's upcoming 1.12 will have static compilation support
- Now possible to compile Julia packages to static executable or shared library
- Function multi-versioning allows shipping a single binary that is optimized for multiple CPUs (within the same architecture)

# Backup



# M4 benchmark

```
> julia +1.12 -e "using InteractiveUtils; versioninfo()"
```

```
Julia Version 1.12.0-beta4
```

```
Commit 600ac61d3d2 (2025-06-05 07:03 UTC)
```

```
Build Info:
```

```
  Official https://julialang.org release
```

```
Platform Info:
```

```
  OS: macOS (arm64-apple-darwin24.0.0)
```

```
  CPU: 10 × Apple M4
```

```
  WORD_SIZE: 64
```

```
  LLVM: libLLVM-18.1.7 (ORCJIT, apple-m1)
```

```
  GC: Built with stock GC
```

```
> pixi run python test.py
```

```
=====
Input size: 1000
All close: True
Numpy      time (ms): 0.021641701459884644
FHist.jl   time (ms): 0.003958120942115784
=====

Input size: 10000
All close: True
Numpy      time (ms): 0.05824156105518341
FHist.jl   time (ms): 0.011725164949893951
=====

Input size: 100000
All close: True
Numpy      time (ms): 0.438716821372509
FHist.jl   time (ms): 0.0859750434756279
=====
```

Input size: 1000000

All close: True

Numpy time (ms): 3.94724179059267

FHist.jl time (ms): 0.829133577644825

## x86\_64 benchmark

```
> julia +1.12 -e "using InteractiveUtils; versioninfo()"
```

```
Julia Version 1.12.0-beta4
```

```
Commit 600ac61d3d2 (2025-06-05 07:03 UTC)
```

```
Build Info:
```

```
  Official https://julialang.org release
```

```
Platform Info:
```

```
  OS: Linux (x86_64-linux-gnu)
```

```
  CPU: 24 × AMD Ryzen 9 3900X 12-Core Processor
```

```
  WORD_SIZE: 64
```

```
  LLVM: libLLVM-18.1.7 (ORCJIT, znver2)
```

```
  GC: Built with stock GC
```

```
> pixi run python test.py
```

```
=====
Input size: 1000
All close: True
Numpy      time (ms): 0.05090880440548062
FHist.jl   time (ms): 0.007097609341144562
=====

Input size: 10000
All close: True
Numpy      time (ms): 0.19860140746459365
FHist.jl   time (ms): 0.020549201872199774
=====

Input size: 100000
All close: True
Numpy      time (ms): 1.3177349930629134
FHist.jl   time (ms): 0.19442759221419692
=====
```

Input size: 1000000

All close: True

Numpy time (ms): 6.48591200588271

FHist.jl time (ms): 1.8390380078926682

## More comments on benchmark

- We can't cross-compile Julia code for different architectures yet due to LLVM.
- There's also no way to compile GPU code in this manner yet, despite Julia itself has excellent support for writing vendor-agnostic [high-level](#) and [kernel-level](#) code.