High-performance end-user analysis with julia

CHEP 2023, May 11th @ Norfolk, Virginia

Jerry Ling (Harvard University / ATLAS Experiment)

Tamas Gal (Erlangen Centre for Astroparticle Physics)

Who are end users?

Physicists (mostly students) who write analysis code, make plots etc.

What do end users want from a programming language?

- Expressiveness – write less code and do more physics

What do end users want from a programming language?

- ❖ Expressiveness – write less code and do more physics
- ❖ Performance – shorter time-to-insight, more iterations on analysis ideas

What do end users want from a programming language?

- ❖ Expressiveness – write less code and do more physics
- ❖ Performance – shorter time-to-insight, more iterations on analysis ideas
- ❖ **In short: "A language that's easy to write but runs fast".**

Conventional wisdom: trade-offs

- Expressiveness/Performance often thought as trade-offs

Conventional wisdom: trade-offs

- Expressiveness/Performance often thought as trade-offs
- Q: Why “faster” languages require us to write more verbose, less flexible code?
- E.g. when we write C++:
  - variable type
  - function argument types
  - function return type

Conventional wisdom: trade-offs

- Expressiveness/Performance often thought as trade-offs
- Q: Why “faster” languages require us to write more verbose, less flexible code?
- E.g. when we write C++:
    - variable type
    - function argument types
    - function return type
- A: The more information you write down for the compiler, the easier it is to optimize the emitted native code.

Beyond the traditional trade-offs

- Julia claims it’s “easy to write and fast to run”, how can it workaround the trade-off?
- One of the ingredients**: Specialization** in compilation.

Beyond the traditional trade-offs: Specialization

- Consider this function that sums all the elements in an array.
- No type annotation in source code.

```
julia> function mysum(ary)
           s = zero(eltype(ary))
           for x in ary
               s += x
           end
           return s
       end
```

Beyond the traditional trade-offs: Specialization

- Julia compiles specialized native code for different argument types.
- # of types executed on: 0
- # of compiled native code: 0

```
julia> using MethodAnalysis

julia> function mysum(ary)
           s = zero(eltype(ary))
           for x in ary
               s += x
           end
           return s
       end

julia> methodinstances(mysum)
[]
```

Beyond the traditional trade-offs: Specialization

- Julia compiles specialized native code for different argument types.
- # of types executed on: 1
- # of compiled native code: 1

```
julia> using MethodAnalysis

julia> function mysum(ary)
           s = zero(eltype(ary))
           for x in ary
               s += x
           end
           return s
       end

julia> methodinstances(mysum)
[]

julia> mysum([1, 2, 3])
6

julia> methodinstances(mysum)
1-element:
 MethodInstance for mysum(::Vector{Int64})
```

Beyond the traditional trade-offs: Specialization

- Julia compiles specialized native code for different argument types.
- # of types executed on: 2
- # of compiled native code: 2

```julia
julia> using MethodAnalysis

julia> function mysum(ary)
           s = zero(eltype(ary))
           for x in ary
               s += x
           end
           return s
       end

julia> methodinstances(mysum)
[]

julia> mysum([1, 2, 3])
6

julia> methodinstances(mysum)
1-element:
 MethodInstance for mysum(::Vector{Int64})

julia> mysum([1.0, 2.0, 3.0])
6.0

julia> methodinstances(mysum)
2-element:
 MethodInstance for mysum(::Vector{Int64})
 MethodInstance for mysum(::Vector{Float64})
```

Beyond the traditional trade-offs: Specialization

- ❖ Julia compiles specialized native code for different argument types.
- ❖ # of types executed on: 2
- ❖ # of compiled native code: 2

- ❖ Just-In-Time (JIT) compile once and cache native code.
- ❖ This allows end users to write generic code and retain full performance.

```
julia> using MethodAnalysis

julia> function mysum(ary)
           s = zero(eltype(ary))
           for x in ary
               s += x
           end
           return s
       end

julia> methodinstances(mysum)
[]

julia> mysum([1, 2, 3])
6

julia> methodinstances(mysum)
1-element:
 MethodInstance for mysum(::Vector{Int64})

julia> mysum([1.0, 2.0, 3.0])
6.0

julia> methodinstances(mysum)
2-element:
 MethodInstance for mysum(::Vector{Int64})
 MethodInstance for mysum(::Vector{Float64})
```

Opportunity for **end users**

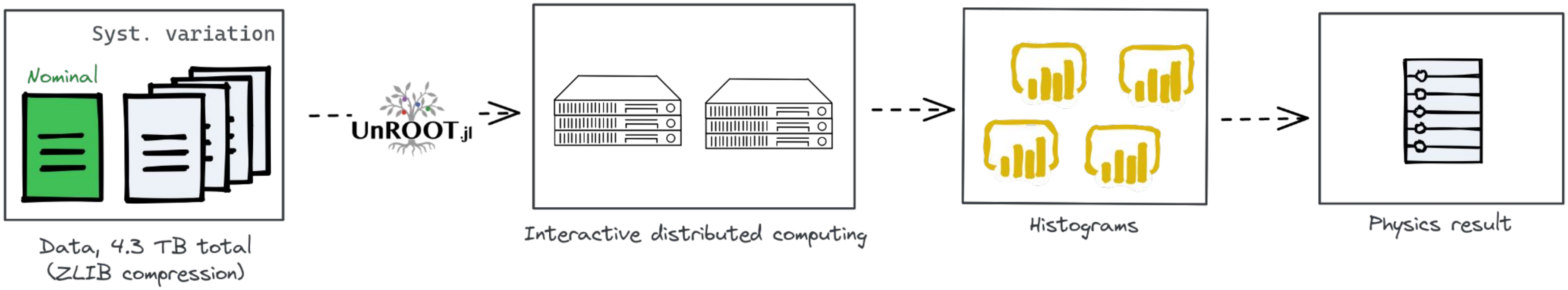
Some benefits of using one accessible and performant language:

- Lower barrier for physics students: learning -> production
- Reduce alternating languages for different tasks

How does Julia do in a real analysis?

Julia for end users - a typical workflow

Julia workflow in an ongoing ATLAS analysis, use many projects under JuliaHEP organization.



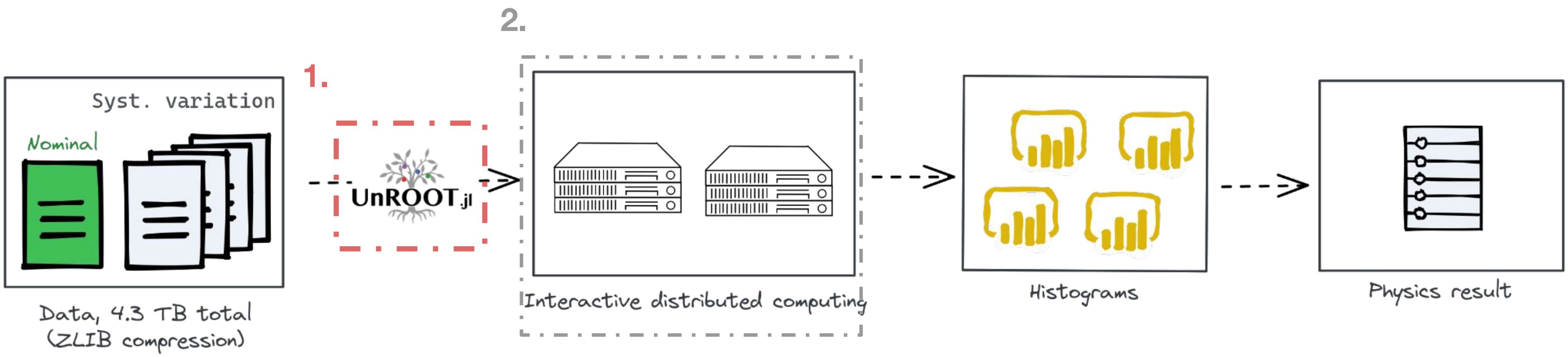
Data handling

Languages

- **Julia** 100.0%

I want to focus on two parts of the workflow:

1. Handling ROOT file – easy for human and fast for machine.
2. Scaling to cluster (HPC) – smooth transition and debug interactively.



Data handling

End users’ partial wish list for handling root files:

- ❖ No boilerplate code
- ❖ Fast
- ❖ Multi-threading

Data handling

End users' partial wish list for handling root files:

- ❖ No boilerplate code ✅
- ❖ Fast
- ❖ Multi-threading

```julia
using UnROOT
tree = LazyTree("./data.root", "Events")
for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```

Data handling

End users' partial wish list for handling root files:

- ❖ No boilerplate code ✅
- ❖ Fast ❓

Recall specialization is the source of performance, Julia's job here seems hard:

1. Know the type of `evt.Muon_pt`
2. Compile specialized `sum()`
3. Infer type of `muon_HT`
4. Compile the best `<` native code.

```julia
using UnROOT
tree = LazyTree("./data.root", "Events")
for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```

Data handling

End users' partial wish list for handling root files:

- ❖ No boilerplate code ✅
- ❖ Fast ❓

Recall specialization is the source of performance, Julia's job here seems hard:

1. Know the type of `evt.Muon_pt`
2. Compile specialized `sum()`
3. Infer type of `muon_HT`
4. Compile the best `<` native code.

```
using UnROOT
tree = LazyTree("./data.root", "Events")
for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```

Data handling

End users' partial wish list for handling root files:

- ❖ No boilerplate code ✅
- ❖ Fast ❓

```julia
using UnROOT
tree = LazyTree("./data.root", "Events")
for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```

Recall specialization is the source of performance, Julia's job here seems hard:

1. Know the type of `evt.Muon_pt`
2. Compile specialized `sum()`
3. Infer type of `muon_HT`
4. Compile the best `<` native code.

Actually, if compiler is smart, 1 should imply 2,3,4!

Data handling

End users' partial wish list for handling root files:

❖ No boilerplate code ✅
❖ Fast ❓

Recall specialization is the source of performance, Julia's job here seems hard:

1. Know the type of `evt.Muon_pt`

Developer's job:

❖ encode "Branch name" <--> "type" information in the variable types of `evt` and `tree` when parsing the file.

```julia
using UnROOT
tree = LazyTree("./data.root", "Events")
for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```

Data handling

End users' partial wish list for handling root files:

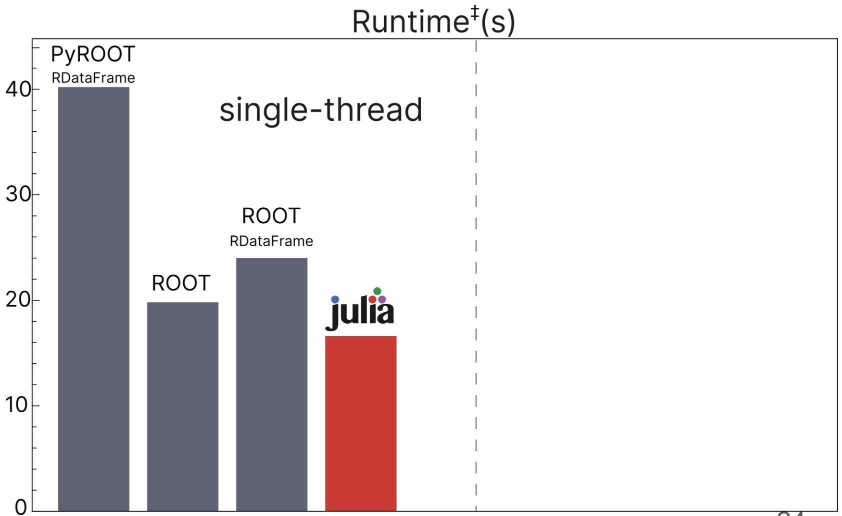
- No boilerplate code ✅
- Fast ✅
- Multi-threading

Benchmark: CMS Open Data, 4l Higgs analysis

*Initially ROOT loop was slower than RDataFrame, fixed after discussion with Enrico Guiraud from ROOT.

*not code from benchmark

```
using UnROOT
tree = LazyTree("./data.root", "Events")
for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```



‡ More info: http://cern.ch/go/vhR6

Data handling

End users' partial wish list for handling root files:

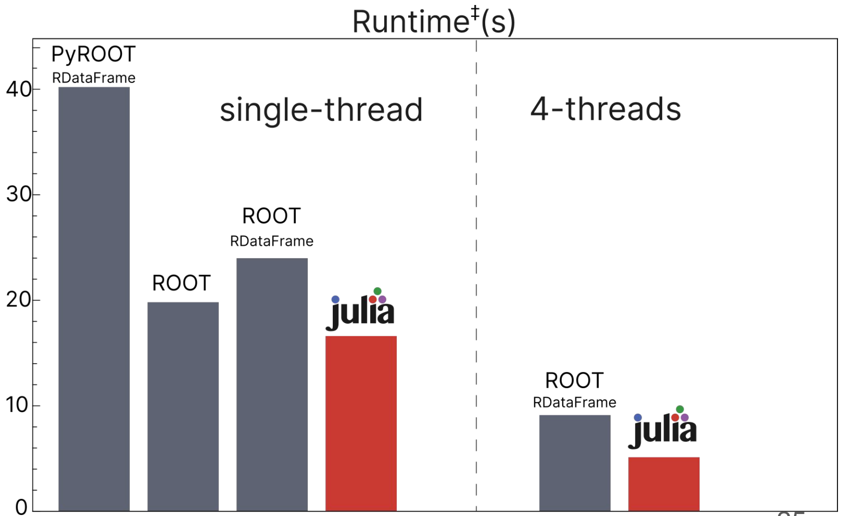
- ❖ No boilerplate code ✅
- ❖ Fast ✅
- ❖ Multi-threading ✅

Benchmark: CMS Open Data, 4l Higgs analysis

- ❖ Julia as a language doesn't have "global lock" (e.g. Global Interpreter Lock in Python)
- ❖ UnROOT.jl is thread-safe.

*not code from benchmark

```
using UnROOT
tree = LazyTree("./data.root", "Events")
@threads for evt in tree
    muon_HT = sum(evt.Muon_pt)
    if muon_HT < 200
        continue
    end
    #...
end
```



‡ More info: http://cern.ch/go/vhR6

Data handling

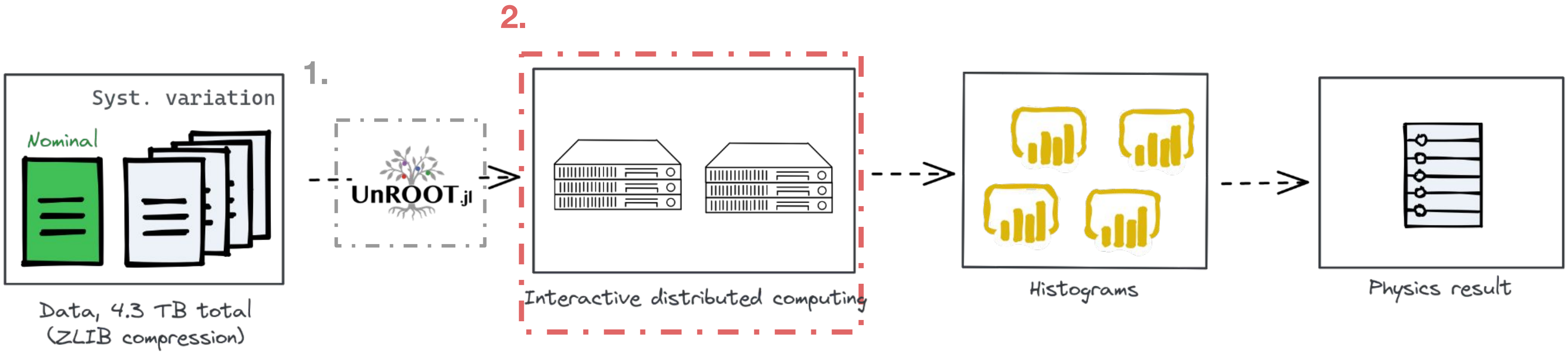
A few more things:

- ❖ Columnar manipulation: each branch follows Julia vector interface, native jagged support
- ❖ For each event, read from file only when branch is accessed – **lazy read**.
- ❖ Already support `RNTuple`, identical user code
  - ➢ I also implemented the `RNTuple` in uproot

```julia
using UnROOT
tree = LazyTree("./data.root", "Events")
@threads for evt in tree
    muon_HT = sum(evt.Muon_pt)
    muon_HT < 200 && continue
    #...
end
```

Interactive distributed analysis

1. Handling ROOT file – easy for human and fast for machine.
2. Scaling to cluster (HPC) – smooth transition and debug interactively.



Interactive distributed analysis

End users’ partial wish list for running analysis on cluster:

- ❖ Smooth local session -> cluster
- ❖ No wait for compilation
- ❖ Revise code without re-submitting

Interactive distributed analysis

End users' partial wish list for running analysis on cluster:

- ❖ Smooth local session -> cluster
- ❖ No wait for compilation
- ❖ Revise code without re-submitting

```
# [local code working!]
julia> using ClusterManagers, Distributed, Revise

julia> addprocs(HTCManager(4))
# Waiting for 4 workers: 1 2 3 4 .

julia> @fetchfrom 1 gethostname()
"login02.af.uchicago.edu" # <--- user's login node

julia> @fetchfrom 2 gethostname()
"c028.af.uchicago.edu" # <--- a HTCondor node
```

Interactive distributed analysis

End users' partial wish list for running analysis on cluster:

- Smooth local session -> cluster
- No wait for compilation
- Revise code without re-submitting

```
# [local code working!]
julia> using ClusterManagers, Distributed, Revise

julia> addprocs(HTCManager(4))
# Waiting for 4 workers: 1 2 3 4 .

julia> @fetchfrom 1 gethostname()
"login02.af.uchicago.edu" # <--- user's login node

julia> @fetchfrom 2 gethostname()
"c028.af.uchicago.edu" # <--- a HTCondor node

julia> @everywhere using WVZAnalysis
```

Interactive distributed analysis

End users' partial wish list for running analysis on cluster:

- ❖ Smooth local session -> cluster
- ❖ No wait for compilation
- ❖ Revise code without re-submitting

```
# [local code working!]
julia> using ClusterManagers, Distributed, Revise

julia> addprocs(HTCManager(4))
# Waiting for 4 workers: 1 2 3 4 .

julia> @fetchfrom 1 gethostname()
"login02.af.uchicago.edu" # <--- user's login node

julia> @fetchfrom 2 gethostname()
"c028.af.uchicago.edu" # <--- a HTCondor node

julia> @everywhere using WVZAnalysis

julia> run_analysis(..)

# Result looks wrong!
```

Interactive distributed analysis

End users' partial wish list for running analysis on cluster:

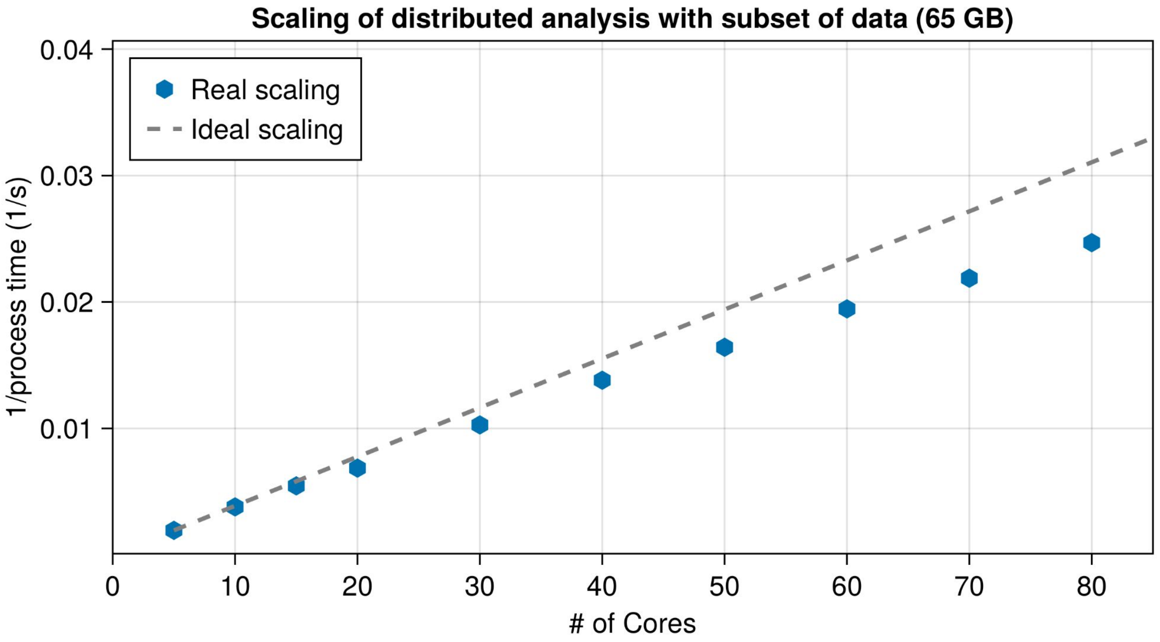
- ❖ Smooth local session -> cluster
- ❖ No wait for compilation
- ❖ Revise code without re-submitting

Modified code re-compiled

```
# [local code working!]
julia> using ClusterManagers, Distributed, Revise

julia> addprocs(HTCManager(4))
# Waiting for 4 workers: 1 2 3 4 .

julia> @fetchfrom 1 gethostname()
"login02.af.uchicago.edu" # <--- user's login node

julia> @fetchfrom 2 gethostname()
"c028.af.uchicago.edu" # <--- a HTCondor node

julia> @everywhere using WVZAnalysis

julia> run_analysis(..)

# Result looks wrong!

# [Edit source code]

julia> run_analysis(..)
```

Interactive distributed analysis

- ❖ Embarrassingly parallel workload scales nicely
- ❖ AF UChicago has **25** physical nodes, fall off when network/storage bottlenecked



Result & Future work

Feedback from ongoing ATLAS analysis:

- On Analysis Facility UChicago, all 4.3 TB data with full systematics can be processed in 30 minutes. Near real-time turnaround!
- Easy enough to maintain: a high school student[1] was able to efficiently iterate analysis ideas.

[1]: Rafael Jacobsen

Result & Future work

Feedback from ongoing ATLAS analysis:

- ❖ On Analysis Facility UChicago, all 4.3 TB data with full systematics can be processed in 30 minutes. Near real-time turnaround.
- ❖ Easy enough to maintain: a high school student was able to efficiently iterate analysis ideas.

Possible future exploration:

- ❖ Julia as a less thorny escape hatch for Python users (compared to C++)
- ❖ Explore Julia application upstream of “end-user analysis”
- ❖ Machine learning without flattening the data
- ❖ More featureful statistical tools

Backup

Numba also uses LLVM, performance?

- This example taken from Numba tutorial.
- (For Julia faster than Jax example, see Jax GitHub discussion.)

Python + Numba

```python
x = np.arange(100).reshape(10, 10)

@jit(nopython=True)
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

python> %timeit go_fast(x)
576 ns
```

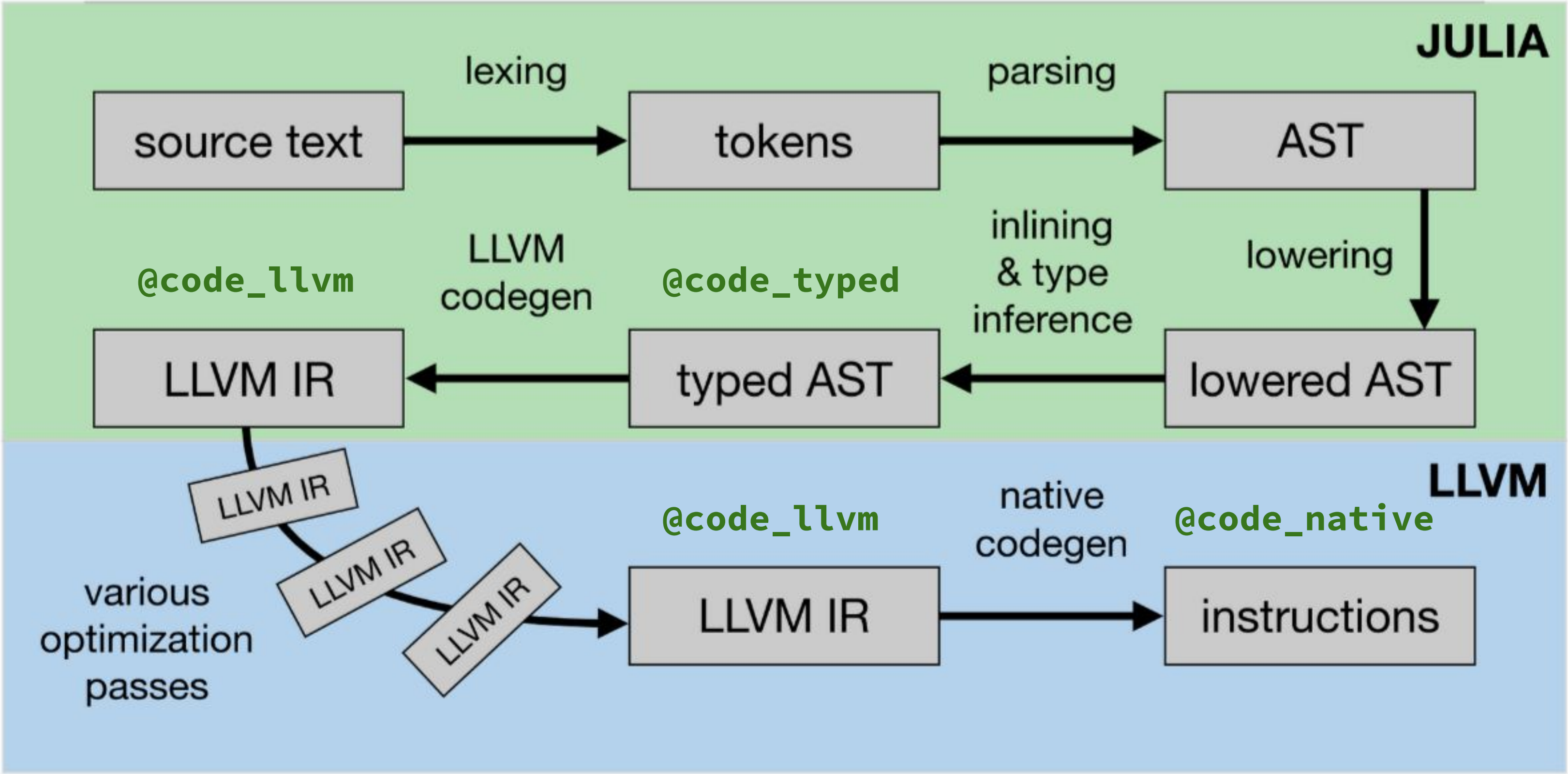
Julia

```julia
x = reshape(0:99, 10, 10)

function go_faster(a)
    trace = 0.0
    for i in axes(a, 1)
        trace += tanh(a[i, i])
    end
    return a .+ trace
end

julia> @btime go_faster(x)
122.176 ns
```

What’s non-Julia in the ATLAS analysis?

- Systematics derivation – need Athena; engineering / labor challenge, not technical.
- Likelihood fitting done in TRexFitter – the group has combined fit with other group in the end.
  - LiteHF.jl can provide statistical fitting, can load `pyhf` JSON workspace, use auto diff
  - LiteHF.jl + Turing.jl gives you Bayesian interpretation

From Source to Machine Code



Different input type compiles to different native code

```
julia> @code_llvm 2*3
  %2 = mul i64 %1, %0
  ret i64 %2

julia> @code_llvm 2.0*3.0
  %2 = fmul double %0, %1
  ret double %2
```

```
julia> @code_native 2*3
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, %rax
    imulq   %rsi, %rax
    popq    %rbp
    retq


julia> @code_native 2.0*3.0
    pushq   %rbp
    movq    %rsp, %rbp
    vmulsd  %xmm1, %xmm0, %xmm0
    popq    %rbp
    retq
```