Julia - a programming language

Uwe Hernandez Acosta (CASUS/HZDR) - January 9th, 2025

Introduction
Uwe Hernandez Acosta

- Particle physicist by training
- PhD in Physics 2021 at TU Dresden/HZDR
  - Topic: Strong field QED
- Affiliation: Center of Advanced Systems Understanding
- Research interests:
  - Theoretical particle physics/Quantum Field Theory
  - Strong field physics
  - X-ray diagnostics in matter under extreme conditions
  - Monte-Carlo Event Generation
  - Julia programming language

Survey says!

Survey says!

- I am able to write something useful in a scripting/interpreted programming language. (something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)
-

Survey says!

- I am able to write something useful in a scripting/interpreted programming language. (something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)
- I can write competitive programs in one these scripting languages. (e.g. in Python, competitive means, you know what a meta-class is)
-

Survey says!

- I am able to write something useful in a scripting/interpreted programming language. (something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)

- I can write competitive programs in one these scripting languages. (e.g. in Python, competitive means, you know what a meta-class is)

- I can write something useful in a systems-level programming language. (e.g. C, C++, Fortran, Rust, Go, Kotlin, Java, etc.)

Survey says!

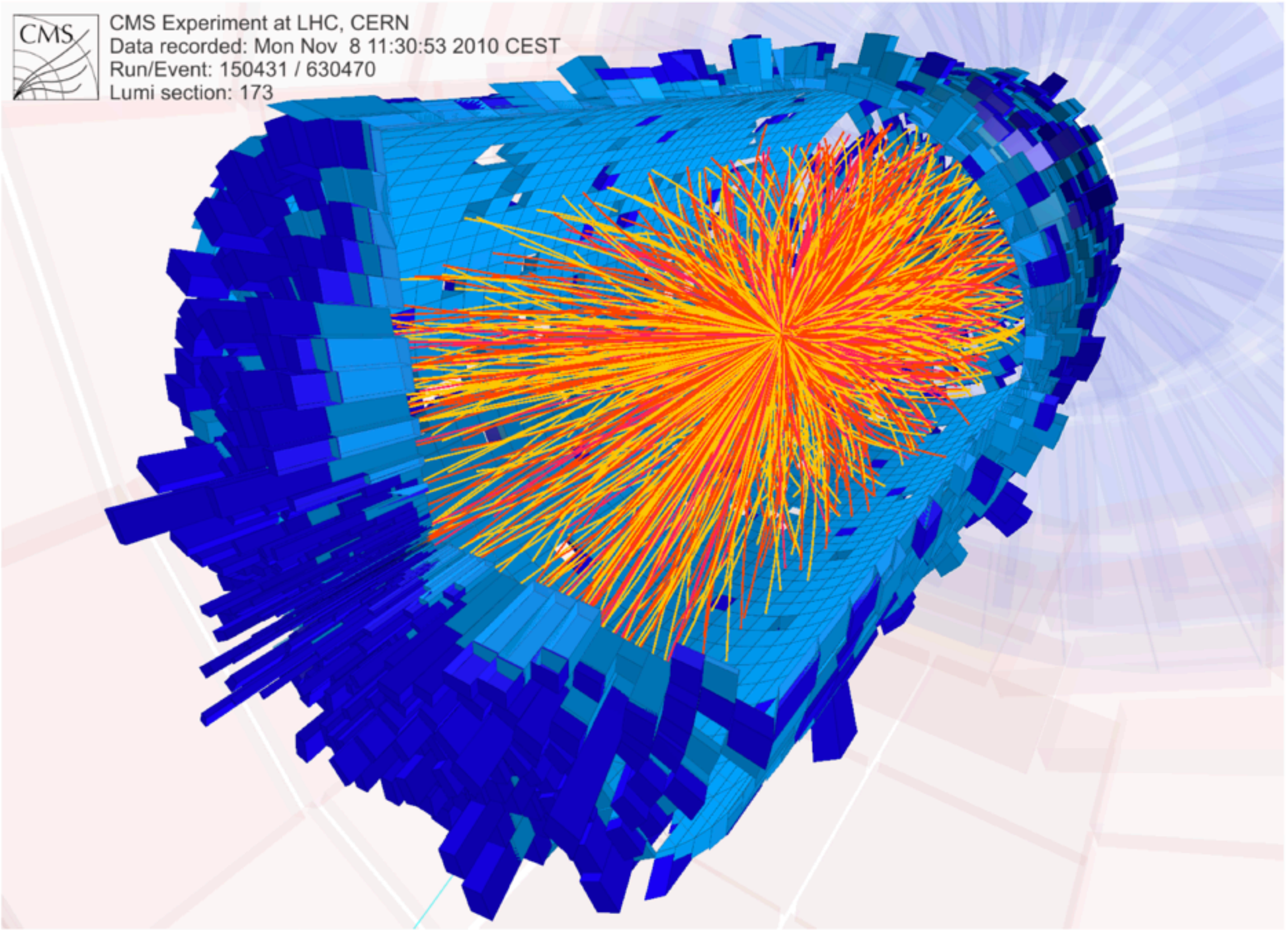
- I am able to write something useful in a scripting/interpreted programming language. (something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)
- I can write competitive programs in one these scripting languages. (e.g. in Python, competitive means, you know what a meta-class is)
- I can write something useful in a systems-level programming language. (e.g. C, C++, Fortran, Rust, Go, Kotlin, Java, etc.)
- I can write competitive code in one of these systems-level languages. (e.g. in C++, competitive means, you can write a template class without getting mad)

Survey says!

- I am able to write something useful in a scripting/interpreted programming language. (something like Python, R, Octave, Matlab, Shell, Javascript, Ruby, etc.)
- I can write competitive programs in one these scripting languages. (e.g. in Python, competitive means, you know what a meta-class is)
- I can write something useful in a systems-level programming language. (e.g. C, C++, Fortran, Rust, Go, Kotlin, Java, etc.)
- I can write competitive code in one of these systems-level languages. (e.g. in C++, competitive means, you can write a template class without getting mad)
- I can write library code in one of these systems-level languages. (e.g. you are one of the authors of Boost)

Software requirements in HEP

- Efficiency
  - Fast execution
  - High data throughput
  - Scalability
- Developer-friendly
  - Quick bug fixes
  - Newest algorithms implemented
  - Good tooling
- User-friendly
  - Rapid development cycles
  - Low entry points
  - Interactivity



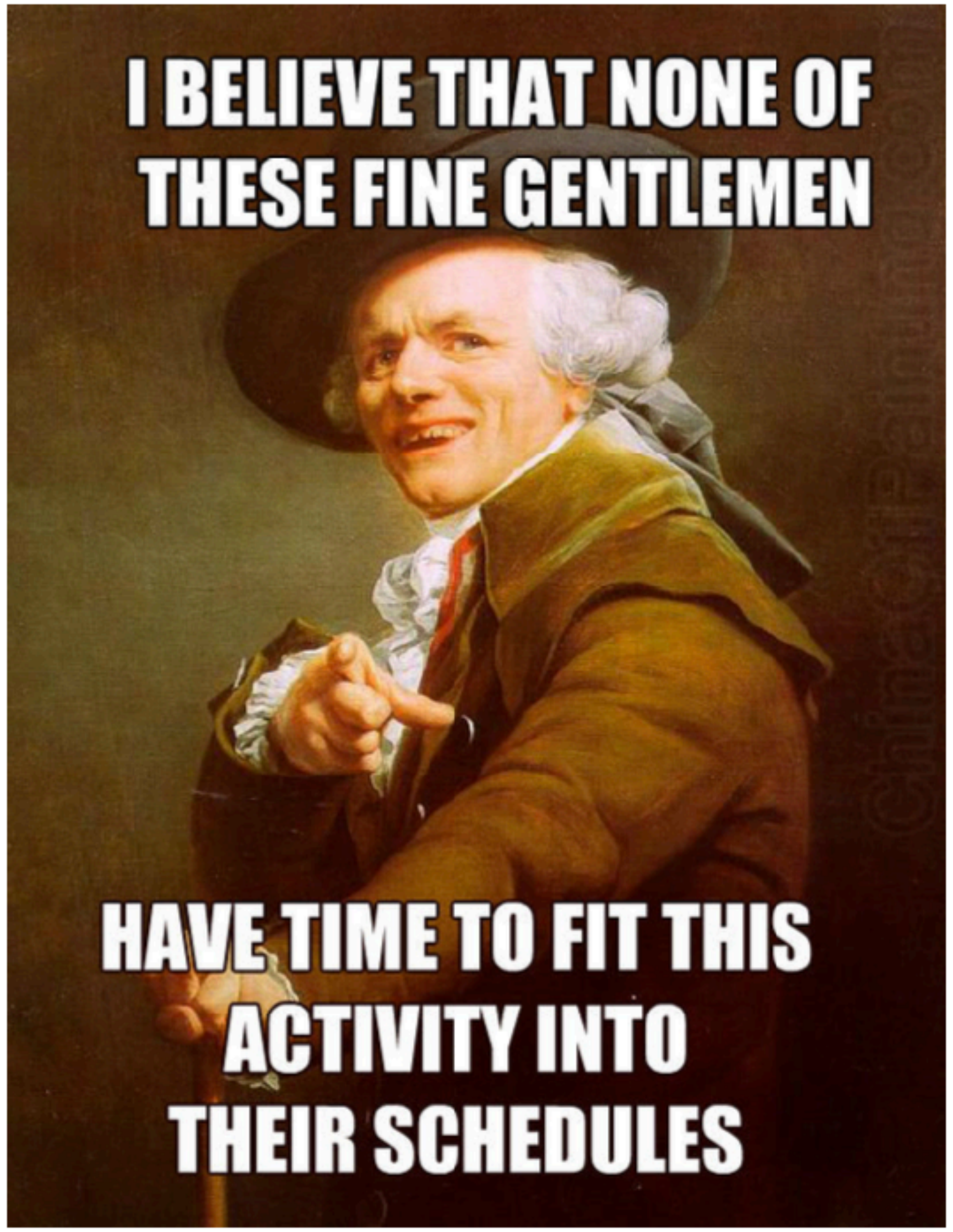
" [I propose that] you should use *two* languages for large software system: one, such as C or C++, for manipulating the complex internal data structures where performance is key and another, such as Tcl, for writing small-ish scripts that tie together the C pieces and are used for extensions."

[Ousterhout. "Re: Why you should not use Tcl" 1994] [Ousterhout. IEEE Computer magazine 31.3 (1998)]

Why is this problematic?

The two languages ~~problem~~ annoyance

- Rewriting parts == refactoring
- Different languages == different logics
- Need for glue code
- Extending is a mess
- Debugging is a mess
- Scientists need to be polyglot
- Multithreading? Anyone?



Established Solutions*?

*personal opinion

Why not use …

… only systems-level languages?

- Take years to learn…
- …decades to master
- Boilerplate code
- Hardware specific
- Mostly non-interactive
- Missing tools/libraries

Why not use …

… third-party libraries?

- “Use C/C++ under the hood”
- Valid in their scope
- Hard to do something outside the box
- Interoperability? Anyone?
- The vendor decides what is performance-critical

Why not use …

… Numba, PyPy, Pythran, etc?

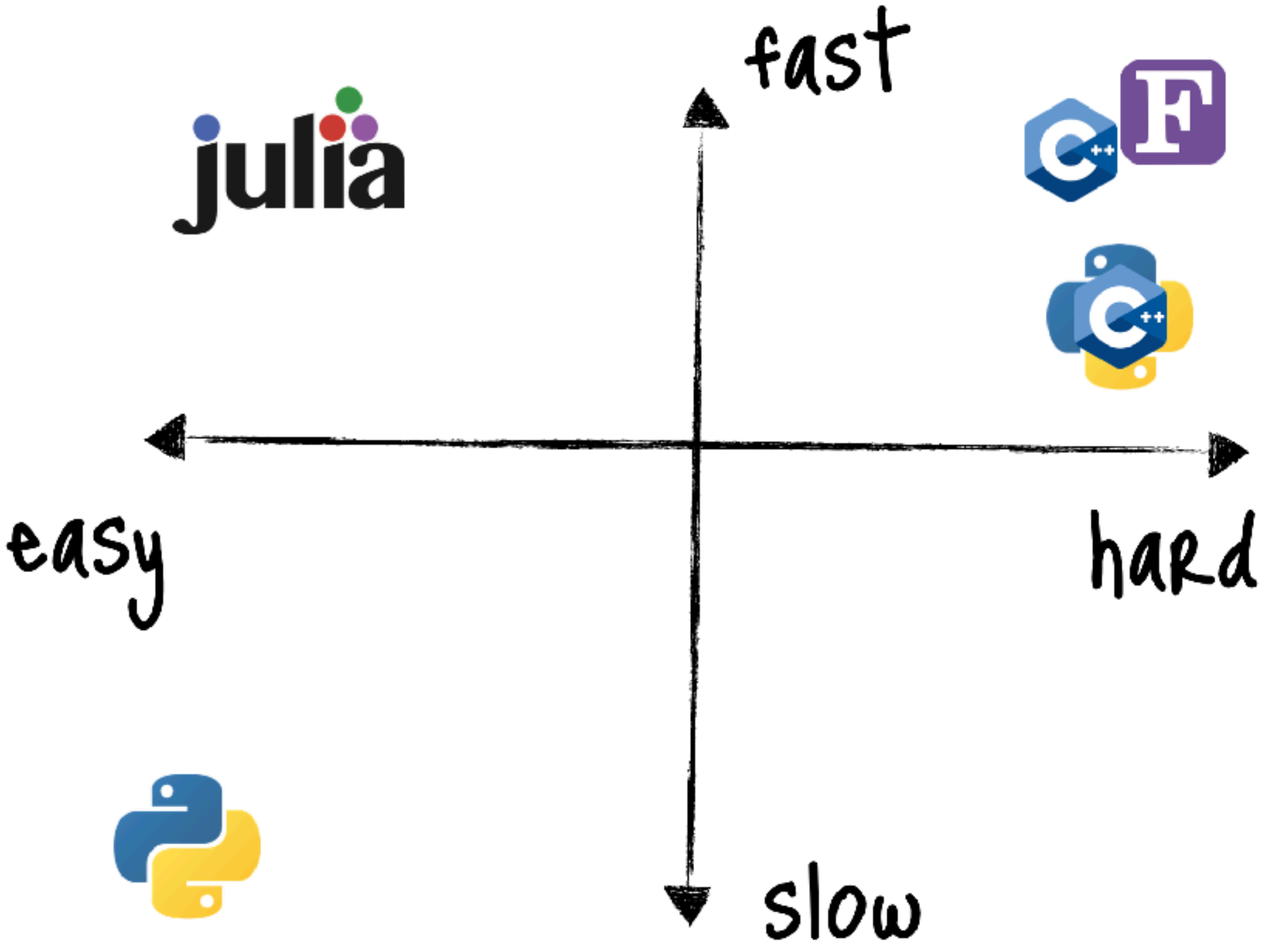
- Sufficient for small code pieces
- These *are* second languages
  - Support only a subset of the host language(s) …
  - … and/or add new commands/ logic/concepts
- Usually not a systems-level language
  - e.g Numba is neither Python nor C

Proposal of a solution

Introduction

The Julia programming language

- Invented 2012 at MIT (mostly)
- Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman
- Design goals
  - Open source
  - Speed like C, dynamic like Ruby
  - Obvious mathematical notation
  - General purpose like Python
  - As easy for statistics as R
  - Powerful linear algebra like in Matlab
  - Good for gluing programs together like the shell



"Something that is dirt simple to learn, yet keeps the most serious hackers happy."

[Bezanson, Karpinski, Shah, Edelman - "Why We Created Julia" (2012)]

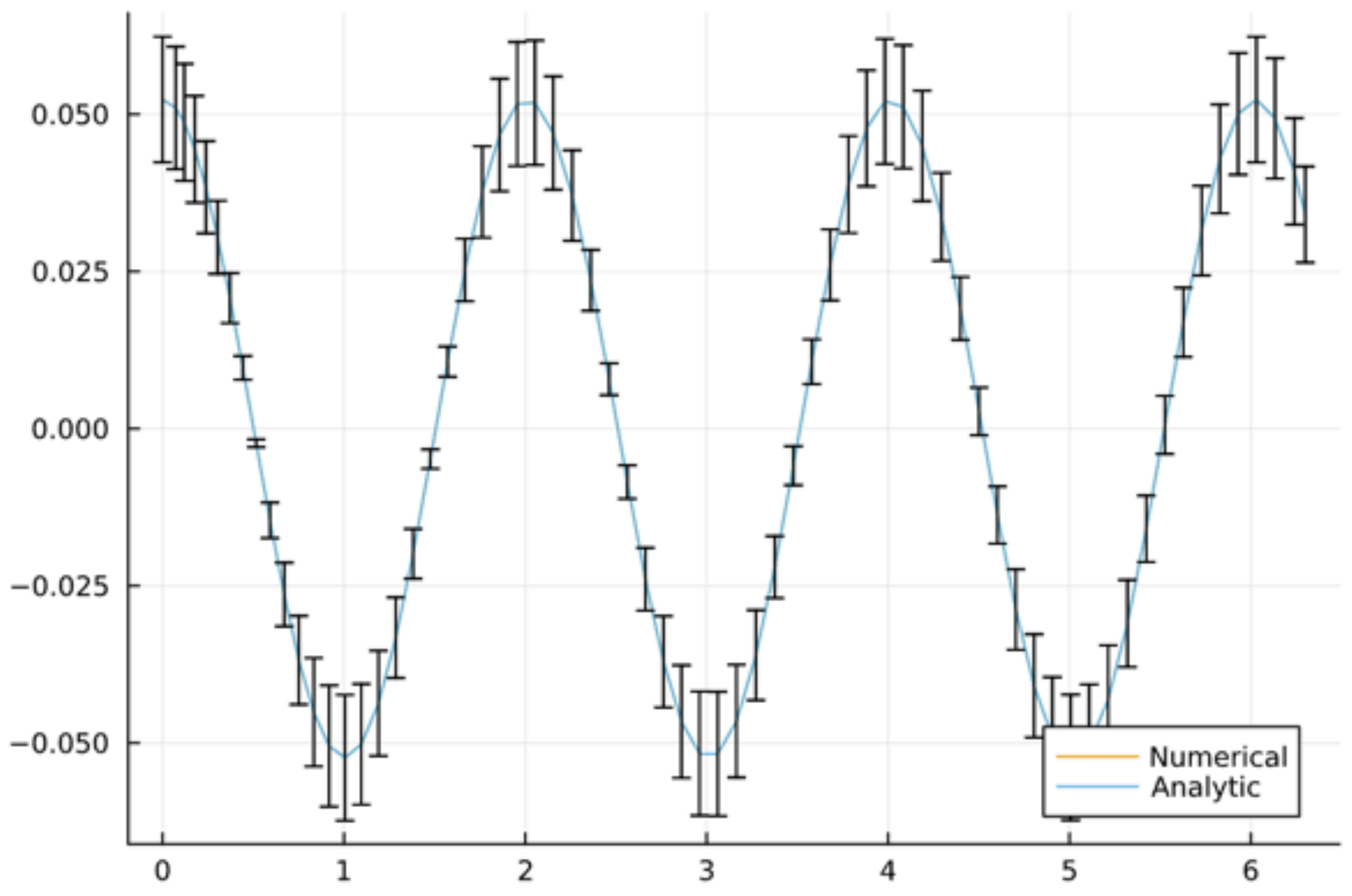
Julia is easy

Ease of use

- Dynamically typed
- Powerful type system
- Garbage collection
- Extensive standard library
    - Mostly written in Julia
    - Math included
    - Performant
- Multiple dispatch for the win!

You can write Julia code as far away from the metal as you want!

```julia
using DifferentialEquations, Measurements, Plots

g = 9.79 ± 0.02; # Gravitational constants
L = 1.00 ± 0.01; # Length of the pendulum

#Initial Conditions
u₀ = [0 ± 0, π / 60 ± 0.01] # Initial speed and initial angle
tspan = (0.0, 6.3)

#Define the problem
function pendulum(du,u,p,t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

#Pass to solvers
prob = ODEProblem(pendulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```



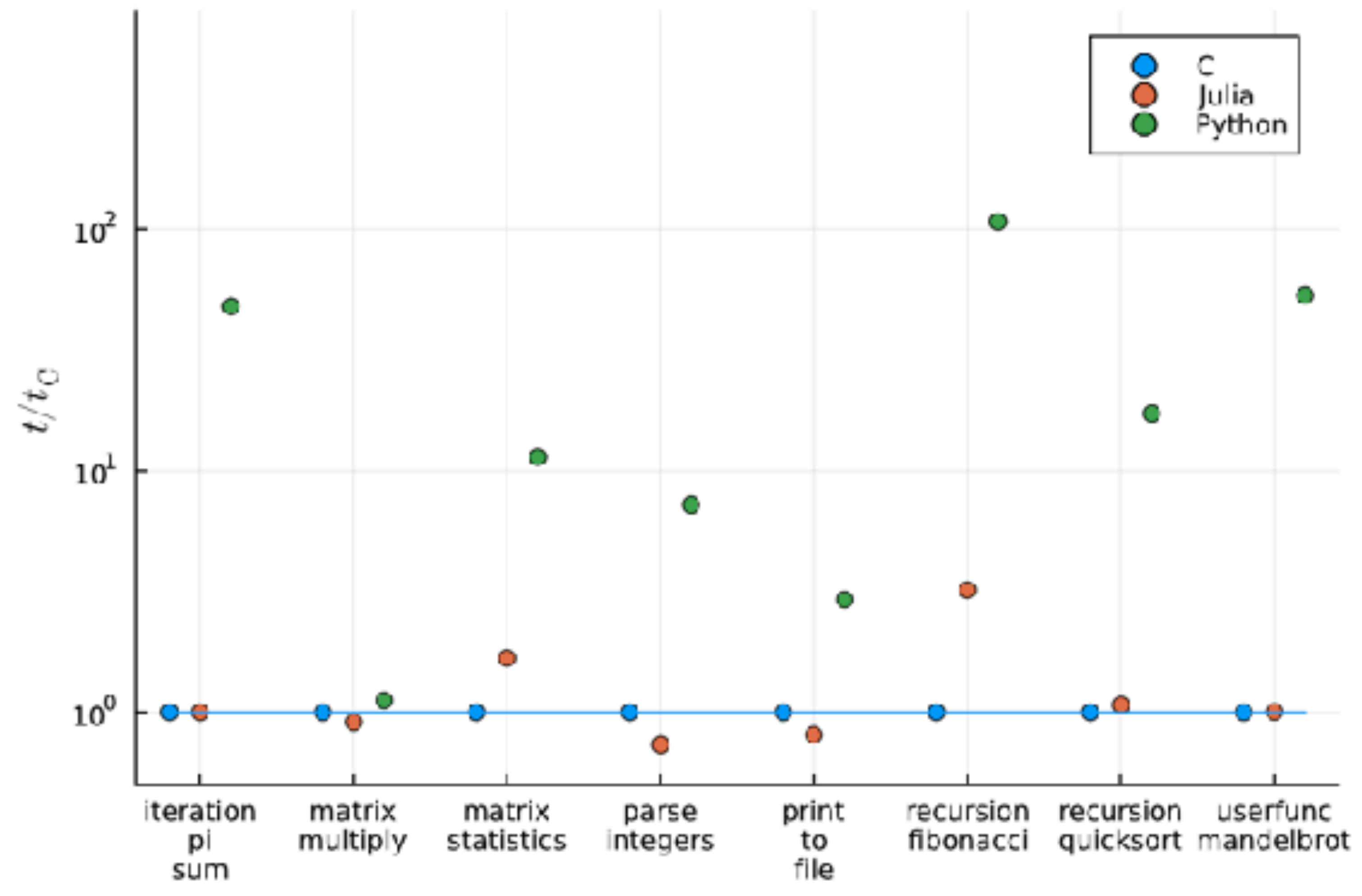
Julia is fast

Not an interpreter

- Just-ahead-of-time compiler
- LLVM empowered
- Statically sizes arrays
- Built-in vector/matrix types
- Arbitrary optimization
- Compiler reflections available
- Native thread support

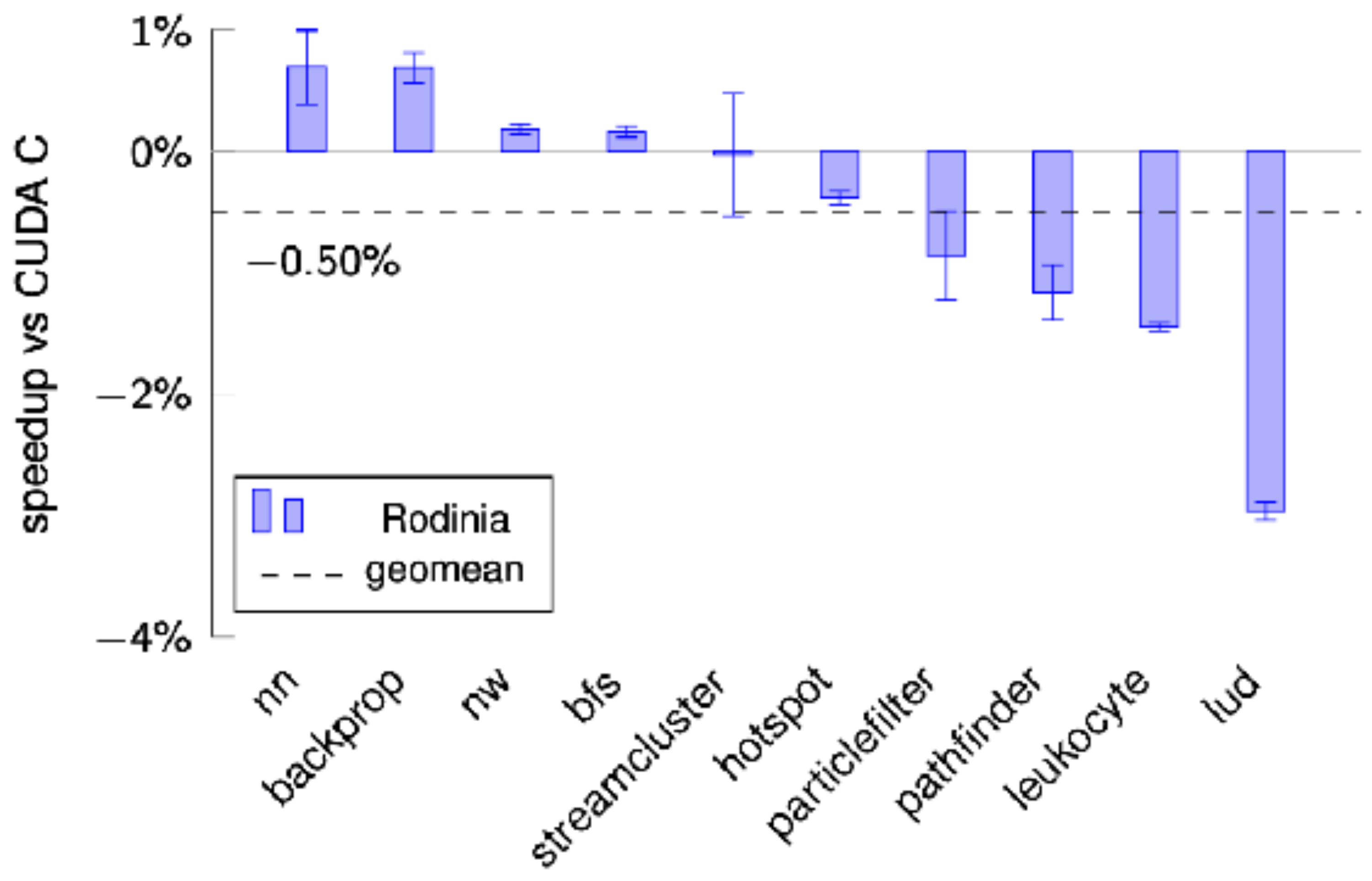
You can write Julia code as close to the metal as you want!

CPU performace



Data taken from [https://julialang.org/benchmarks/]

GPU performance



Taken from [Besard et al. IEEE Trans. Parallel Distrib. Syst. 30.4 (2018)]

Julia is a modern language

Development tooling

```
QEDcore.jl
├── CHANGELOG.md
├── LICENSE
├── Manifest.toml
├── Project.toml
├── README.md
├── docs
├── src
└── test
```

Packaging system

```
name = "QEDcore"
uuid = "35dc0263-cb5f-4c33-a114-1d7f54ab753e"
authors = [
    "Uwe Hernandez Acosta <u.hernandez@hzdr.de>",
    "Anton Reinhard <a.reinhard@hzdr.de>",
]
version = "0.1.1"

[deps]
DocStringExtensions = "ffbed154-4ef7-542d-bbb7-c09d3a79fcae"
QEDbase = "10e22c08-3ccb-4172-bfcf-7d7aa3d04d93"
Reexport = "189a3867-3050-52da-a836-e630ba90ab69"
SimpleTraits = "699a6c99-e7fa-54fc-8d76-47d257e15c1d"
StaticArrays = "90137ffa-7385-5640-81b9-e52037218182"

[compat]
DocStringExtensions = "^0.9"
QEDbase = "0.2.2"
Reexport = "^1.2"
SimpleTraits = "^0.9"
StaticArrays = "^1.9"
julia = "1.6"
```

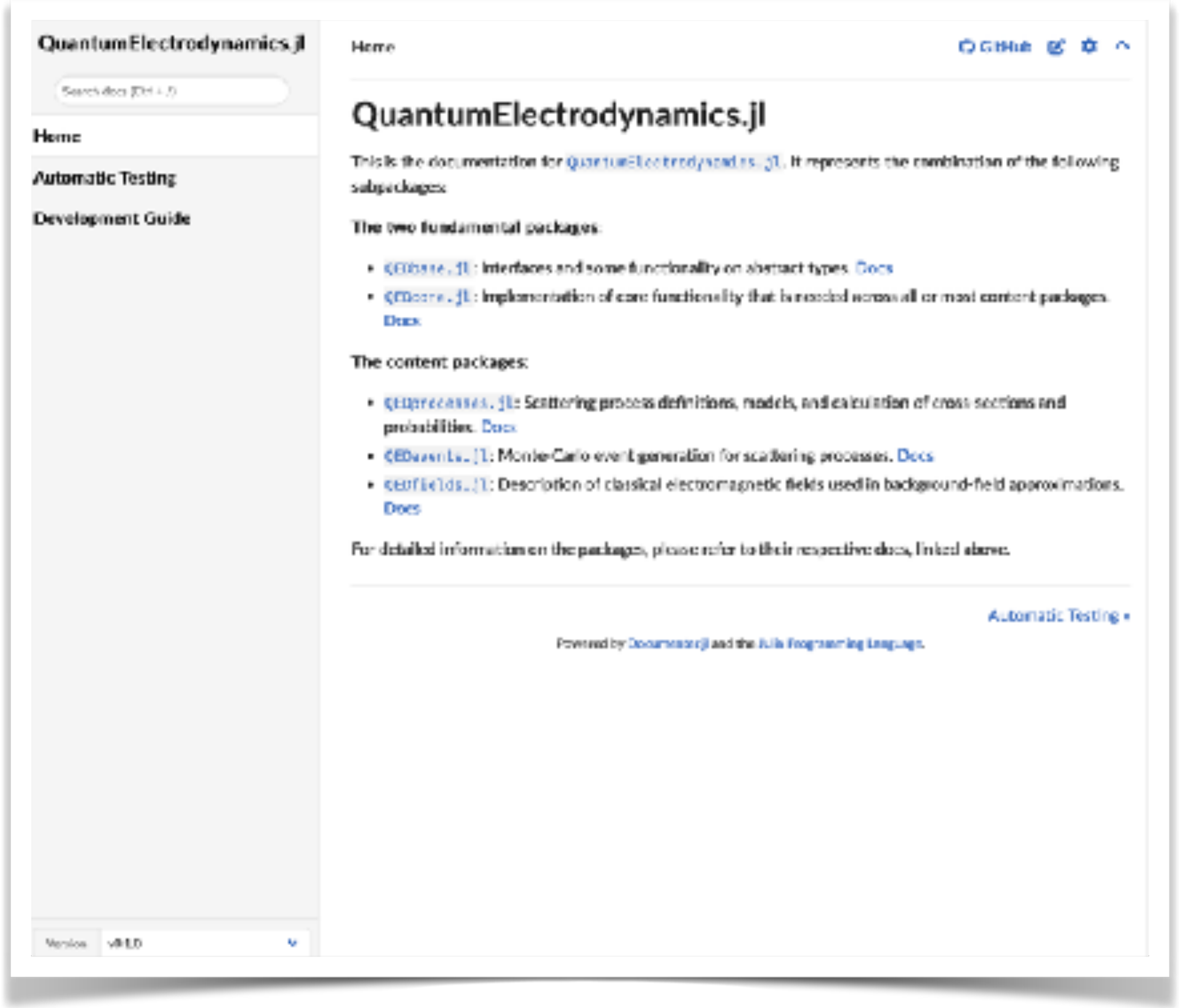
Project.toml

```
(@v1.11) pkg> add QEDcore
   Resolving package versions...
   Installed QEDcore ─ v0.1.1
    Updating `~/.julia/environments/v1.11/Project.toml`
  [35dc0263] + QEDcore v0.1.1
    Updating `~/.julia/environments/v1.11/Manifest.toml`
  [7d9f7c33] + Accessors v0.1.38
  [dce04be8] + ArgCheck v2.3.0
  [49dc2e85] + Calculus v0.5.1
  [38540f10] + CommonSolve v0.2.4
  [a33af91c] + CompositionsBase v0.1.2
  [187b0558] + ConstructionBase v1.5.8
  [3587e190] + InverseFunctions v0.1.17
  [eff96d63] + Measurements v2.11.0
  [5ad8b20f] + PhysicalConstants v0.2.3
  [10e22c08] + QEDbase v0.2.2
  [35dc0263] + QEDcore v0.1.1
  [f2b01f46] + Roots v2.2.1
  [699a6c99] + SimpleTraits v0.9.4
  [90137ffa] + StaticArrays v1.9.7
  [1e83bf80] + StaticArraysCore v1.4.3
Precompiling project...
  4 dependencies successfully precompiled in 6 seconds.
```

Package manager (Pkg.jl)

```
     Testing Running tests...
Test Summary: | Pass  Total  Time
phase spaces  |  152    152  3.1s
Test Summary: | Pass  Total  Time
four momentum |  400    400  2.5s
Test Summary:  | Pass  Total  Time
gamma matrices |   92     92  1.4s
Test Summary:  | Pass  Total  Time
Lorentz vector |   69     69  1.4s
Test Summary: | Pass  Total  Time
Dirac tensors |   51     51  1.5s
Test Summary:  | Pass  Total  Time
particle types |   35     35  0.1s
Test Summary:   | Pass  Total  Time
particle states | 4367   4367  0.9s
Test Summary:    | Pass  Total  Time
particle spinors |   84     84  0.7s
Test Summary:        | Pass  Total  Time
particle base states | 4367   4367  0.4s
Test Summary:        | Pass  Total  Time
particle propagators |    3      3  0.2s
Test Summary:     | Pass  Total  Time
process interface |  148    148  1.3s
     Testing QEDcore tests passed
```

Testing (integrates with Pkg.jl)



Documenter.jl

Rich eco-system

>10k packages

Visualization

- Plots.jl
- Makie.jl
- PGFPlots.jl

Data and Statistics

- JuliaData
- JuliaStats
- BAT.jl (Bayesian Analysis Toolkit)
- Turing.jl

Machine learning

- MJL.jl
- Flux.jl
- SciML
- JuliaDiff

Notebooks

- Pluto.jl
- IJulia.jl

GPU support

- JuliaGPU
- CUDA.jl
- AMDGPU.jl
- oneAPI.jl
- Metal.jl

CUDA.jl, AMDGPU.jl, oneAPI.jl, Metal.jl → KernelAbstractions.jl

Interoperability

- JuliaInterop
- CxxWrap.jl
- PyCall.jl
- RCall.jl
- MathLink.jl

Loading data

HEP data formats

UnROOT: an I/O library for the CERN ROOT file format written in Julia

Tamás Gál[1,2], Jerry (Jiahong) Ling[3], and Nick Amin[4]

High-performance end-user analysis in pure Julia programming language

Jerry Ling[1,*] and *Tamás* Gál[2,**]

File formats/standards

- ROOT Data Analysis Framework
- APACHE ARROW
- openPMD
- HDF
- Les Houches Event Format
- LCIO

Julia implementations/wrapper

- Arrow.jl
- UnROOT.jl
- LCIO.jl
- HDF5.jl
- LHEF.jl
- RootIO.jl
- JLD2.jl
- UpROOT.jl
- openPMD.jl

Drawbacks of using Julia?

Julia should be better

or shouldn’t it?

- Formatter/Linter/LSP could be better
- Little scripts*
- Startup time*
- Vendor lock
  - Only LLVM and Clang
  - Only one reference implementation
- Building binaries*
- Calling Juila from other Languages*
- Context-based programming*
- Cumbersome static performance prediction
- Cumbersome static analysis/checking*

*solved (kinda)

Julia in the wild

Data throughput

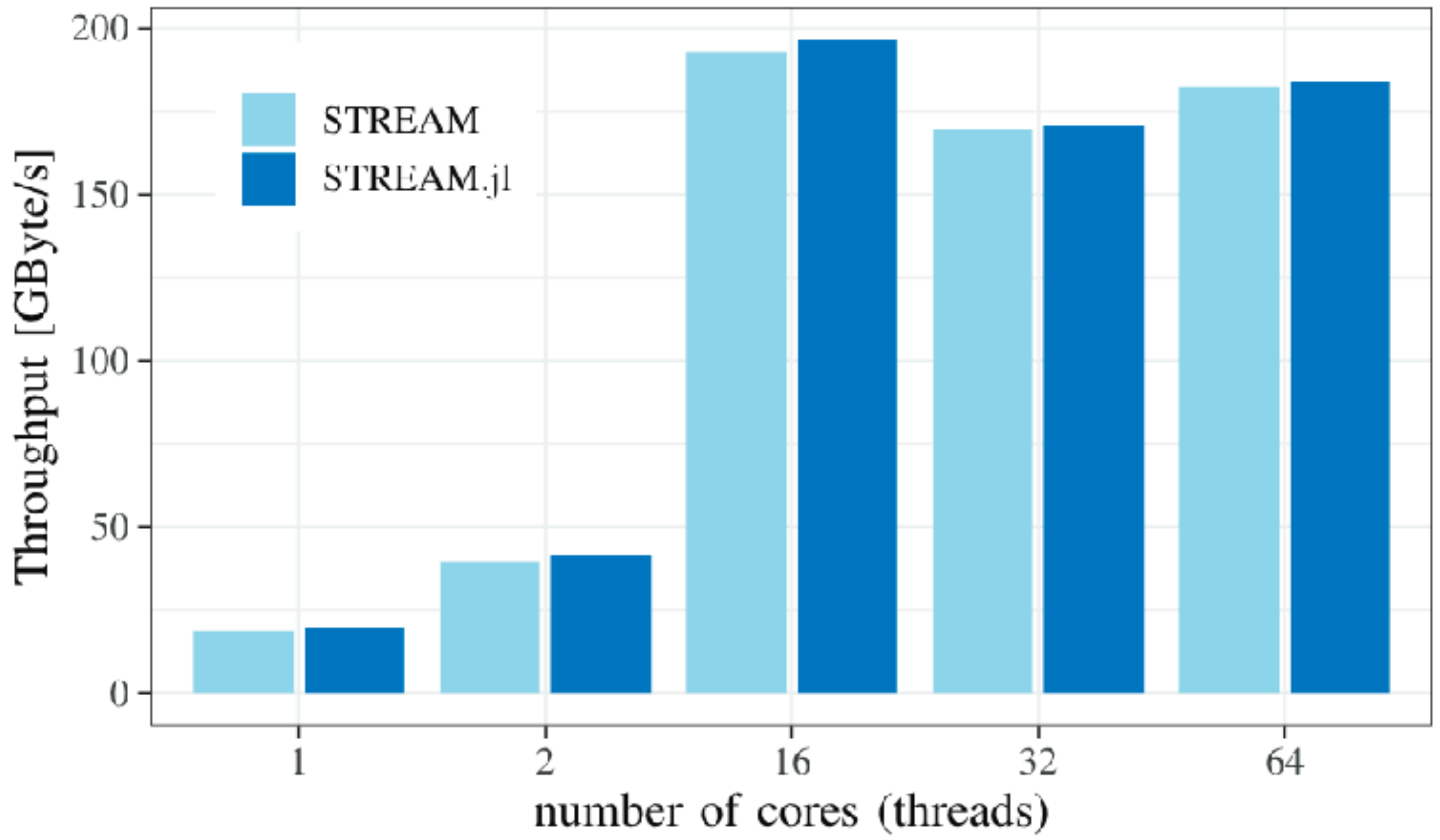
Memory bandwidth benchmarks

Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC?

Sascha Hunold
TU Wien, Faculty of Informatics
Vienna, Austria

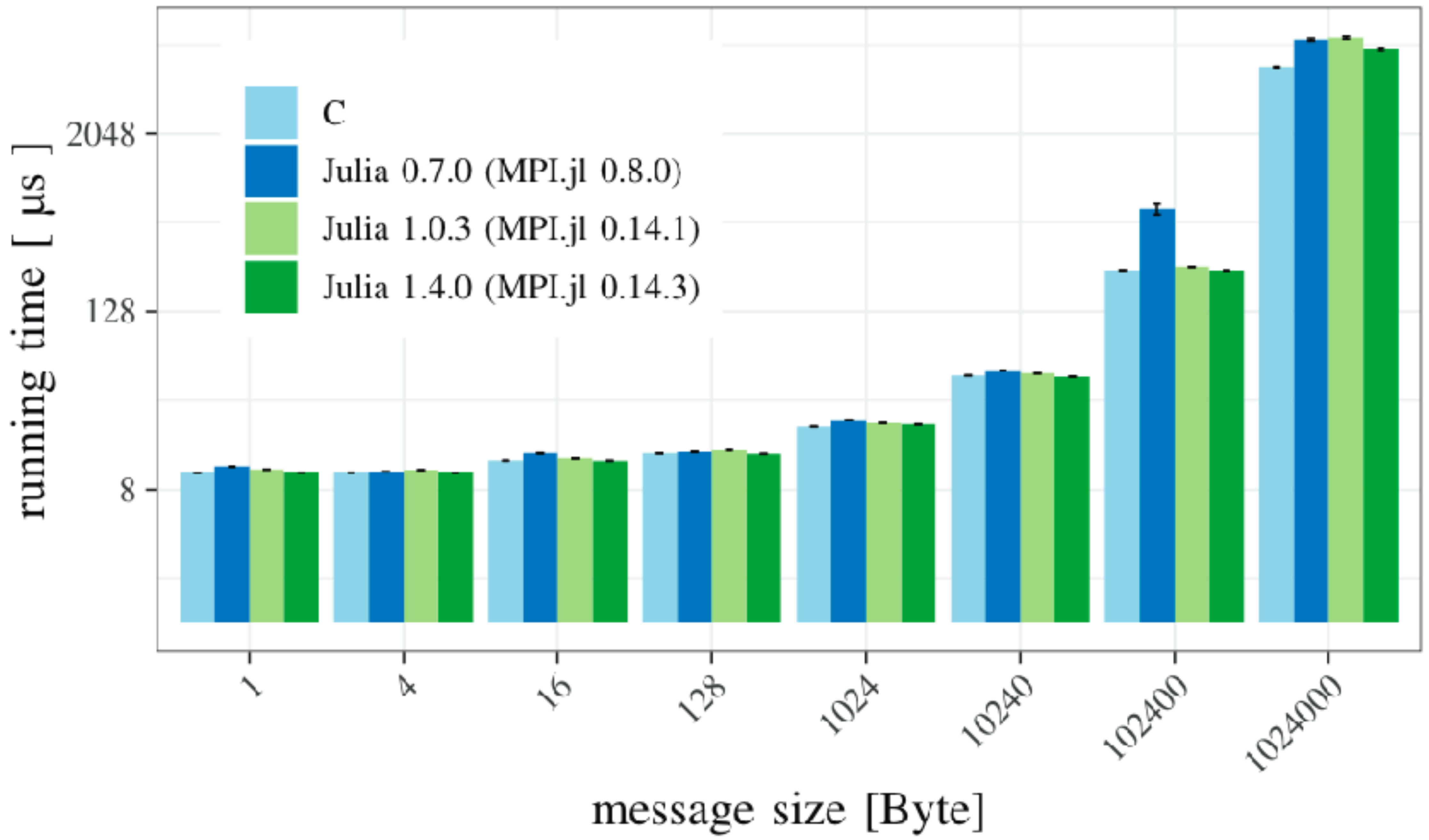
Sebastian Steiner
TU Wien, Faculty of Informatics
Vienna, Austria

Intra-node performance



STREAM benchmark up to 64 AMD CPU cores
LoC: 378 (C) vs 156 (Julia)

Inter-node performance

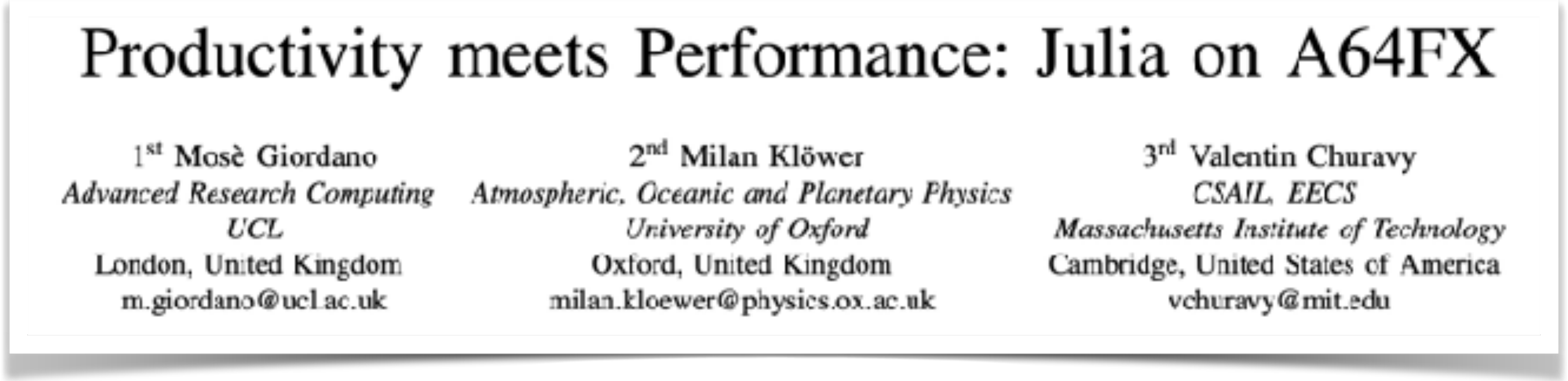


MPI broadcasting benchmark: 36 × 32 processes

Taken from [S. Hunold and S. Steiner, *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*]

Single-node performace

Single-thread axpy benchmarks on Fugaku (A64FX)



Productivity meets Performance: Julia on A64FX

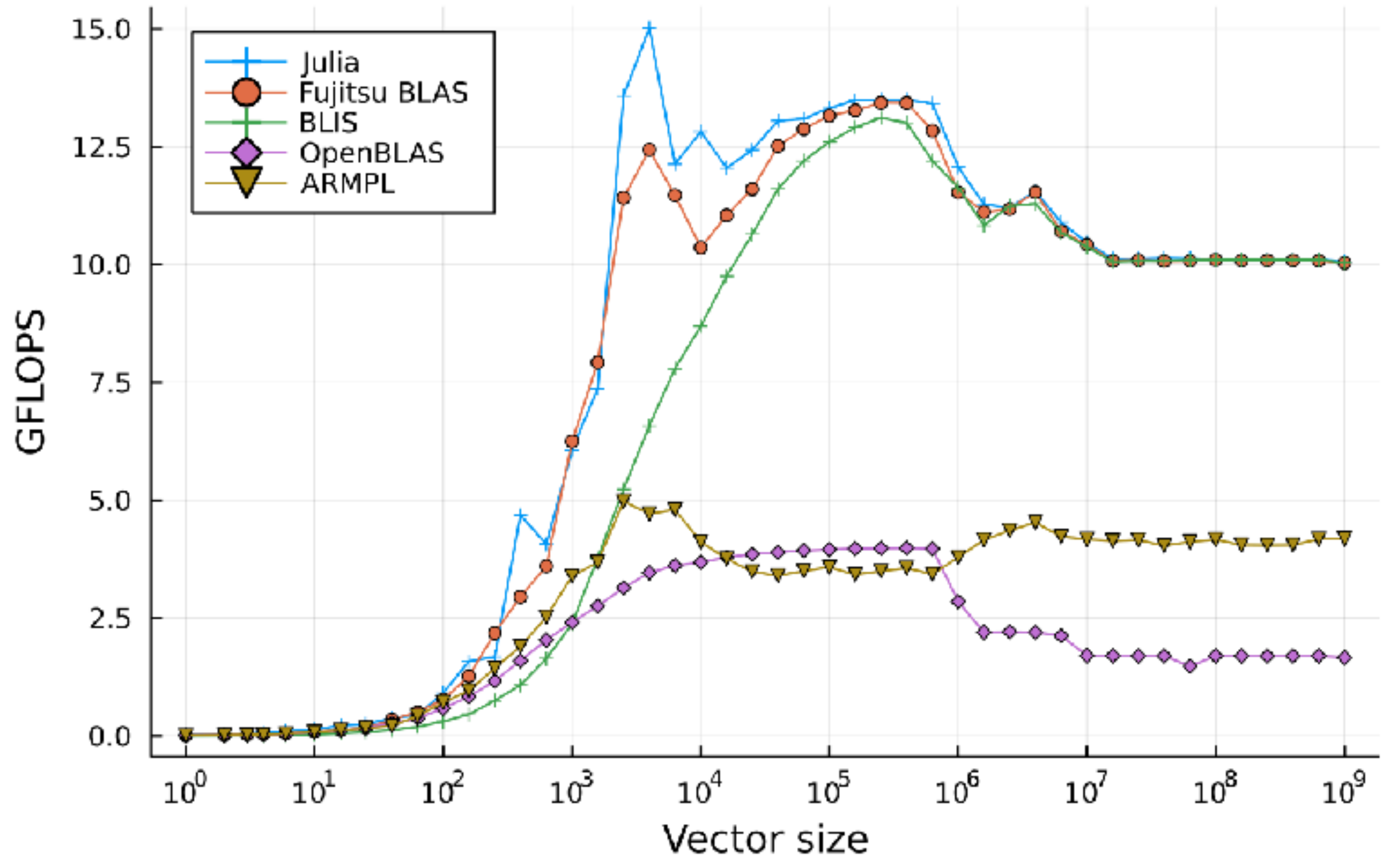
1st Mosè Giordano
Advanced Research Computing
UCL
London, United Kingdom
m.giordano@ucl.ac.uk

2nd Milan Klöwer
Atmospheric, Oceanic and Planetary Physics
University of Oxford
Oxford, United Kingdom
milan.kloewer@physics.ox.ac.uk

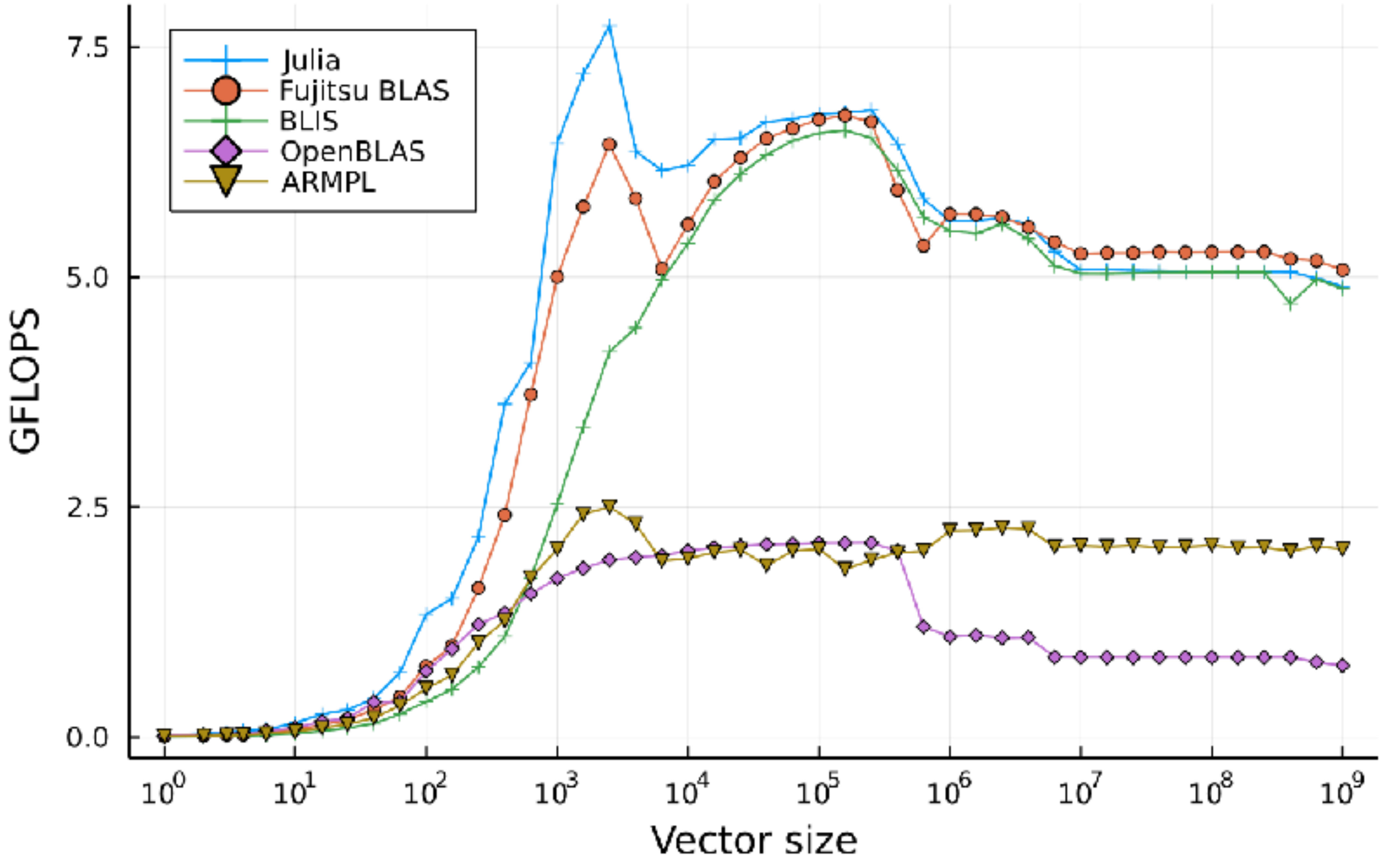
3rd Valentin Churavy
CSAIL, EECS
Massachusetts Institute of Technology
Cambridge, United States of America
vchuravy@mit.edu

```julia
function axpy!(a::T, x::Vector{T}, y::Vector{T}) where {T<:Number}
    @simd for i in eachindex(x, y)
        @inbounds y[i] = muladd(a, x[i], y[i])
    end
    return y
end
```

Single precision



Double precision



Taken from [M Giordano, M Klöwer, V Churavy 2022 IEEE International Conference on Cluster Computing (CLUSTER), 2022]

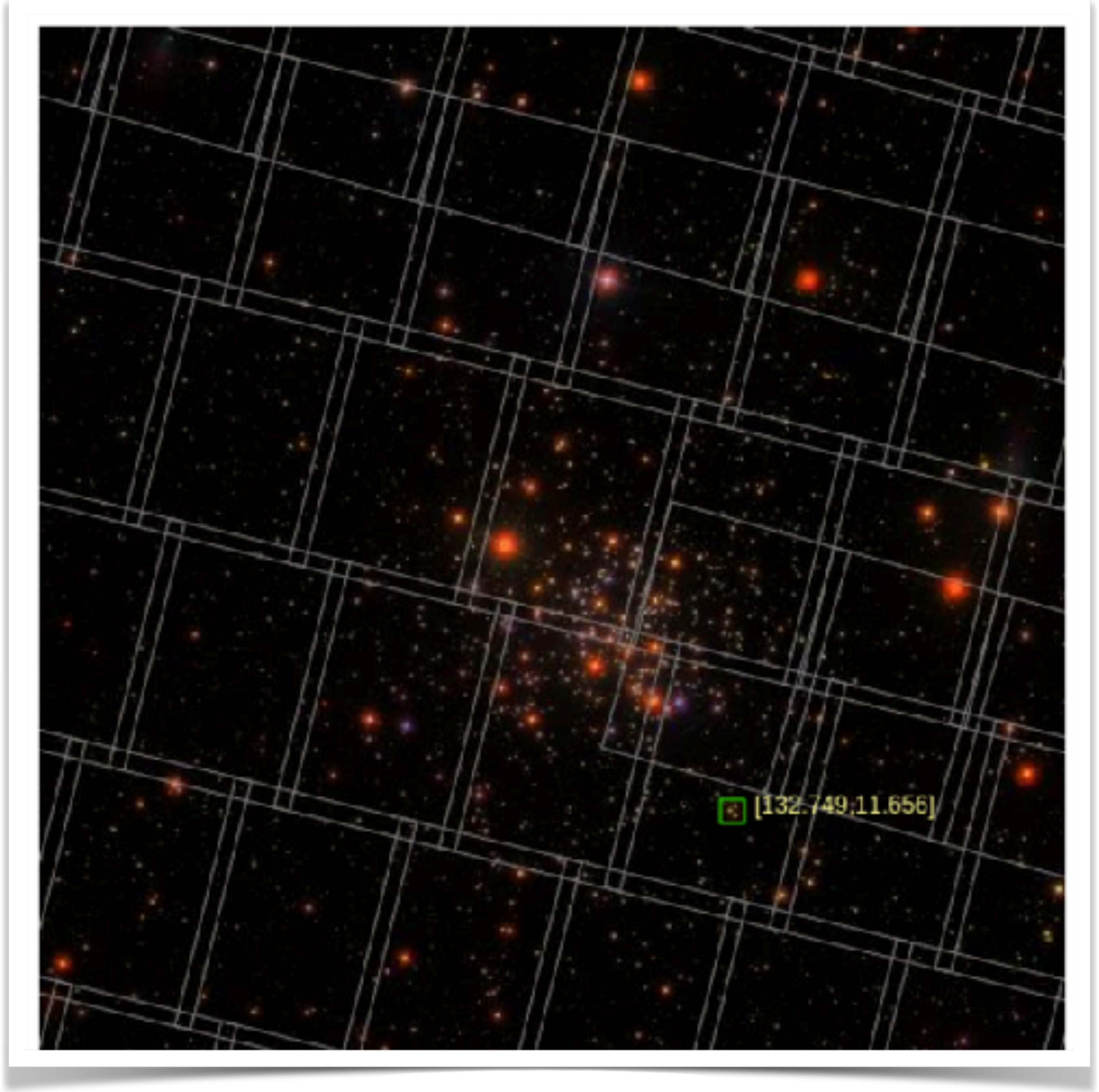
Julia on scale

Celeste.jl project

Cataloging the Visible Universe through Bayesian Inference at Petascale

Jeffrey Regier*, Kiran Pamnany†, Keno Fischer‡, Andreas Noack§, Maximilian Lam*, Jarrett Revels§, Steve Howard¶, Ryan Giordano¶, David Schlegel‖, Jon McAuliffe¶, Rollin Thomas‖, Prabhat‖

- 2017 at NERSC (Berkley)
  - Analysis of $178\,\mathrm{TB}$ telescope data
  - Inferred parameters of 1.88×10^8 stars
  - Done in $14.6\,\mathrm{min}$
  - 1.3×10^6 threads on 650.000 Intel Xeon Phi cores
  - $1.54\,\mathrm{PFLOPS}$ peak performance



Taken from [J. Regier, et al., *Journal of Parallel and Distributed Computing* 127 (2019): 89-104]

Community efforts

JuliaHEP working group in the HEP software foundation

hepsoftwarefoundation.org

HEP software foundation

- Founded around 2014
- facilitate coordination and common efforts in HEP software and computing
- Objectives
  - Share expertise
  - Raise awareness
  - Catalyse new common projects
  - Promote collaborations in new developments
  - Provide training
  - …

HEP Software Foundation

HEP computing collaborations for the challenges of the next decade

Contacts: Simone Campana (Simone.Campana@cern.ch), Zach Marshall (ZLMarshall@lbl.gov), Alessandro Di Girolamo (Alessandro.Di.Girolamo@cern.ch), Heidi Schellman (H… Stewart (grae…

A Roadmap for HEP Software and Computing R&D for the 2020s

The HEP Software Foundation[5] · Johannes Albrecht[69] · Antonio Augusto Alves Jr[81] · Guilherme Amadio[5] · Giuseppe Andronico[27] · Nguyen Anh-Ky[122] · Laurent Aphecetche[66] · John Apostolakis[5] · Makoto Asai[63] · Luca Atzori[5] · Marian Babik[5] · Giuseppe Bagliesi[32] · Marilena Bandieramonte[5] · Sunanda Banerjee[16] · Martin Barisits[5] · Lothar A. T. Bauerdick[16] · Stefano Belforte[35] · Douglas Benjamin[82] · Catrin Bernius[63] · Wahid Bhimji[46] · Riccardo Maria Bianchi[105] · Ian Bird[5] · Catherine Biscarat[52] · Jakob Blomer[5] · Kenneth Bloom[97] · Tommaso Boccali[32] · … Concezio Bozzi[28] · Ma…

Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC

The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · Stefano Frixione[12] · Francesco Giuli[13] · Alexander Grohsjean[5] · Christian Gütschow[14] · Stefan Höche[15] · Walter Hopkins[9] · Philip Ilten[16,17] · Dmitri Konstantinov[18] · Frank Krauss[19] · Qiang Li[20] · Leif Lönnblad[11] · Fabio Maltoni[21,22] · Michelangelo Mangano[1] · Zach Marshall[3] · Olivier Mattelaer[22] · Javier Fernandez Menendez[23] · Stephen Mrenna[15] · Servesh Muralidharan[1,9] · Tobias Neumann[14,24] · Simon Plätzer[25] · Stefan Prestel[11] · Stefan Roiser[1] · Marek Schönherr[19] · Holger Schulz[17] · Markus Schulz[1] · Elizabeth Sexton-Kennedy[15] · Frank Siegert[26] · Andrzej Siódmok[27] · Graeme A. Stewart[1] juliahep.org

JuliaHEP @ HSF

- HSF working group founded in 2022
- JuliaHEP annual workshop
  - 2023: ECAP in Erlangen
  - 2024: CERN
- Monthly community calls
- Monitoring/Supporting development
- Tutorial material + example project

Potential of the Julia Programming Language for High Energy Physics Computing

Jonas Eschle[1] · Tamás Gál[2] · Mosè Giordano[3] · Philippe Gras[4] · Benedikt Hegner[5] · Lukas Heinrich[6] · Uwe Hernandez Acosta[7,8] · Stefan Kluth[6] · Jerry Ling[9] · Pere Mato[5] · Mikhail Mikhasenko[10,11] · Alexander Moreno Briceño[12] · Jim Pivarski[13] · Konstantinos Samaras-Tsakiris[5] · Oliver Schulz[6] · Graeme Andrew Stewart[5] · Jan Strube[14,15] · Vassil Vassilev[13]

github.com/JuliaHEP

#hep

Next JuliaHEP workshop

July 28 - 31, 2025

Princeton University

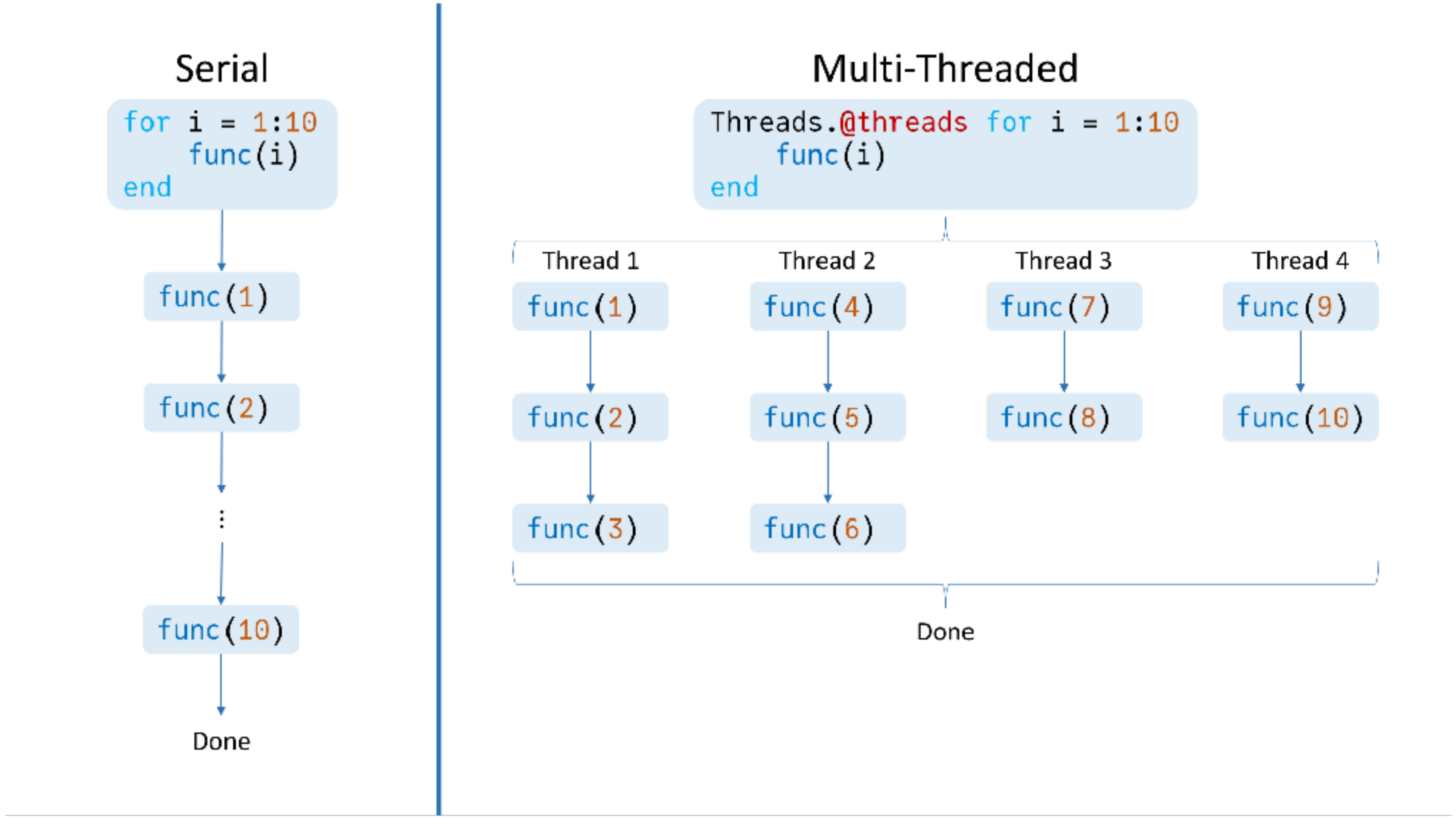
Abstract submissions are now open!

Backup

Parallel computing

Native Threading support

- Support for OpenMP-like models
  - Parallelization of loops
- Support for M:N threading
  - M user threads are mapped onto N kernel threads
- Support for task migration
  - Tasks can be started, suspended, and resumed again



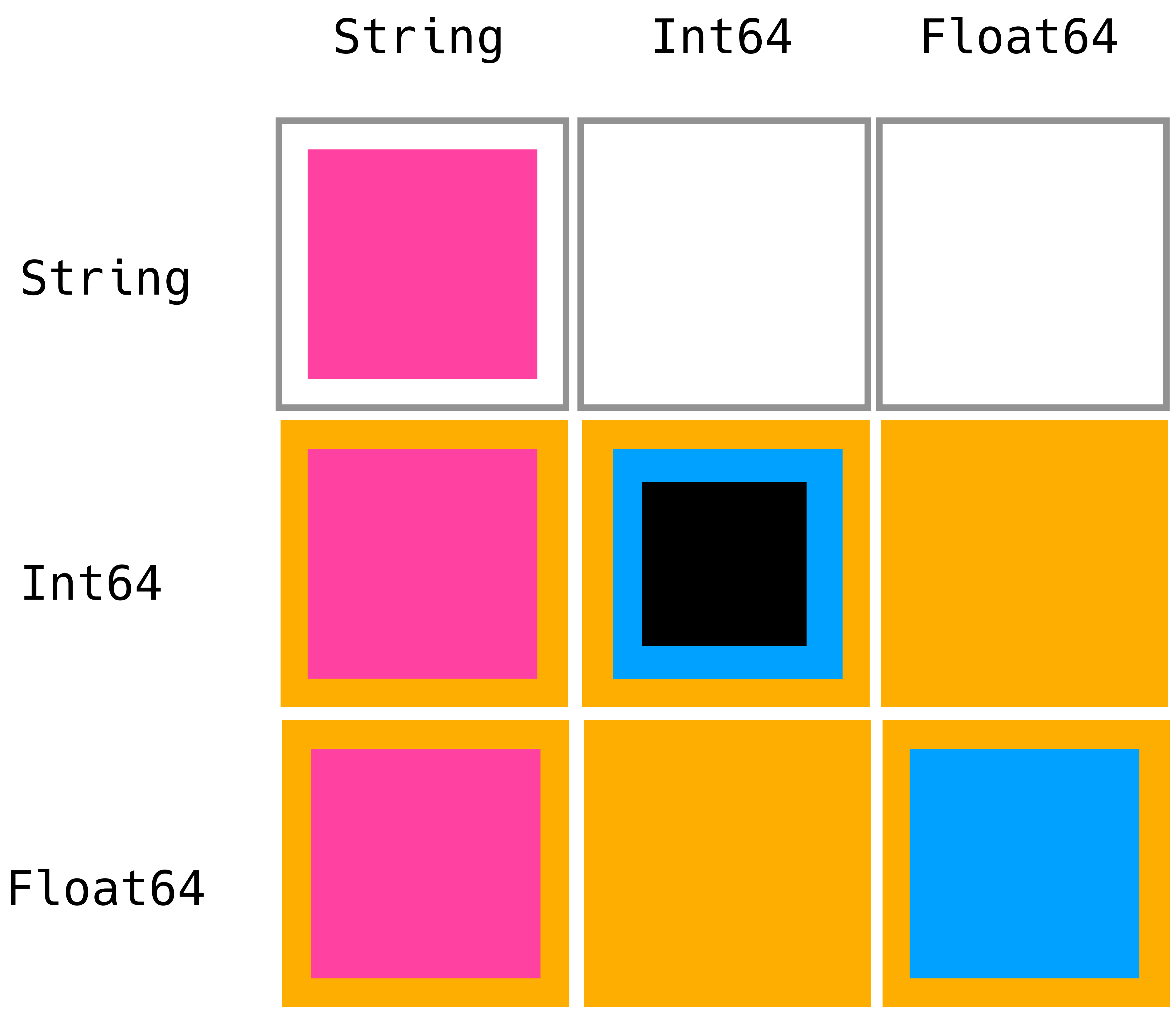
Taken from [https://blog.glcs.io/parallel-processing]

Multiple dispatch

Function and methods

`Float64<:AbstractFloat<:Real<:Number<:Any`

```
f(::Any, ::Number)

f(::T, ::T) where {T<:Number}

f(::Int64, ::Int64)

f(::String, ::Any)
```



Reproduced from [https://scientificcoder.com/the-art-of-multiple-dispatch]

Multiple dispatch II

Expressiveness

Dispatch degree	Syntax	Dispatched on	Selection power
None	`f(x,y,z)`	`{ }`	`1`
Single	`x.f(y,z)`	`{x}`	`|X|`
Multiple	`f(x::X,y::Y,z::Z)`	`{x,y,z}`	`|X|·|Y|·|Z|`

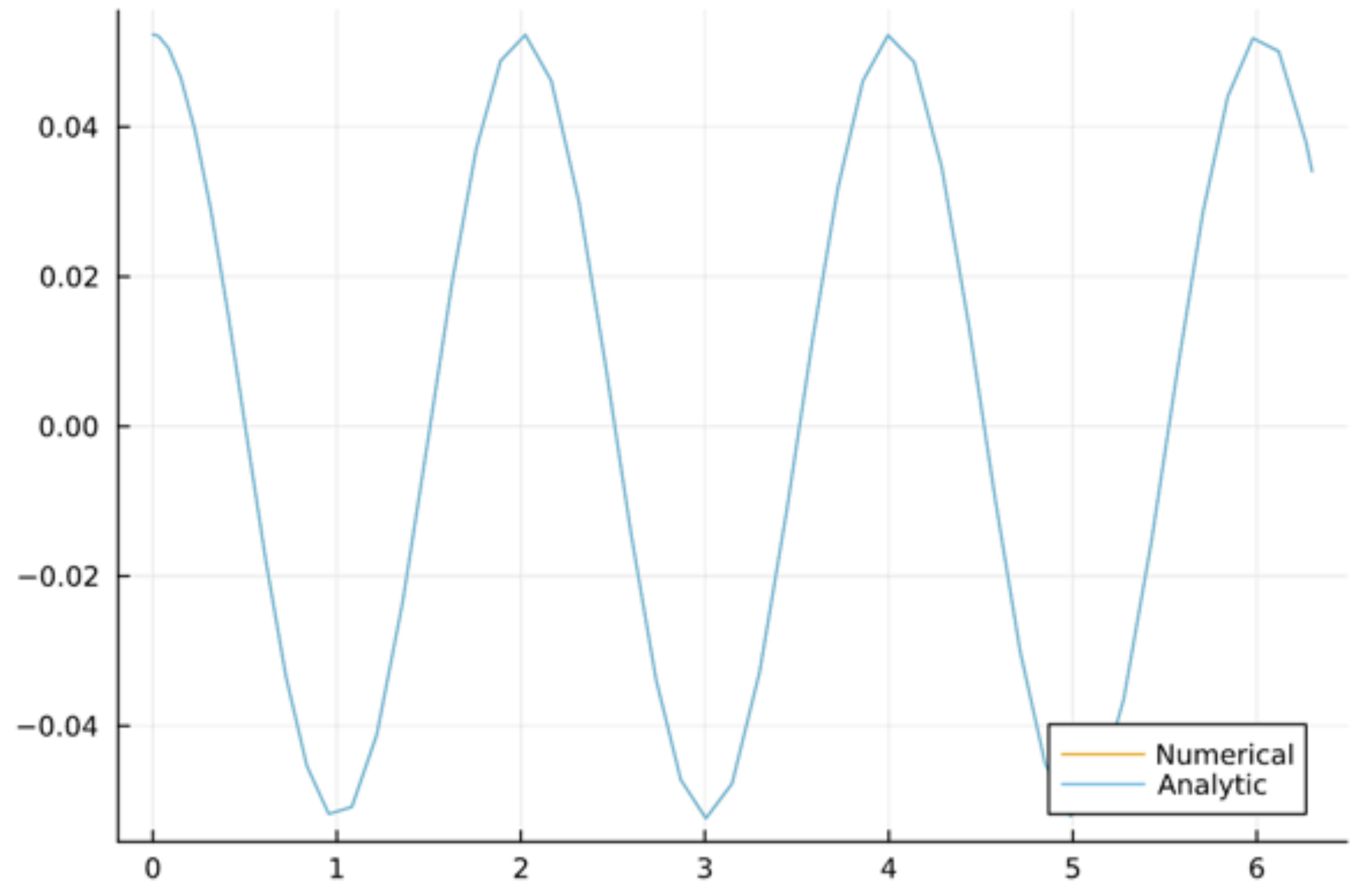
Reproduced from [S. Karpinski, “The unreasonable effectiveness of multiple dispatch”, JuliaCon2019]

Multiple dispatch III

Unreasonable effectiveness

- Allows generic code based on abstract types
- Allows arbitrary optimization
- Orthogonal development
- Solves the expression problem

```julia
using DifferentialEquations, Plots

g = 9.79          # Gravitational constants
L = 1.00          # Length of the pendulum

#Initial Conditions
u₀ = [0, π / 60]           # Initial speed and initial angle
tspan = (0.0, 6.3)

#Define the problem
function pendulum(du,u,p,t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

#Pass to solvers
prob = ODEProblem(pendulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```

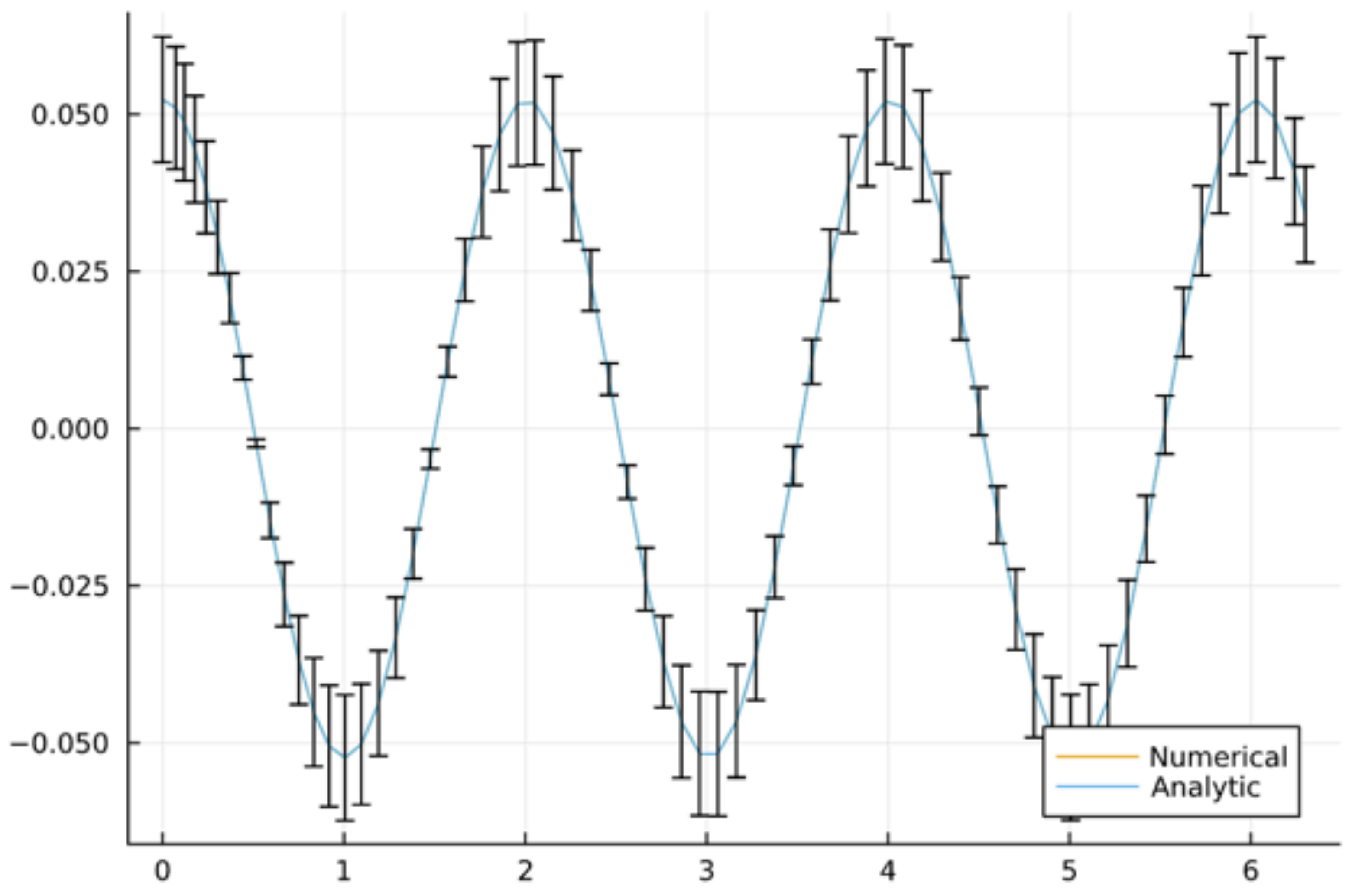


Multiple dispatch III

Unreasonable effectiveness

- Allows generic code based on abstract types
- Allows arbitrary optimization
- Orthogonal development
- Solves the expression problem

```
using DifferentialEquations, Measurements, Plots

g = 9.79 ± 0.02; # Gravitational constants
L = 1.00 ± 0.01; # Length of the pendulum

#Initial Conditions
u₀ = [0 ± 0, π / 60 ± 0.01] # Initial speed and initial angle
tspan = (0.0, 6.3)

#Define the problem
function pendulum(du,u,p,t)
    θ  = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

#Pass to solvers
prob = ODEProblem(pendulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```

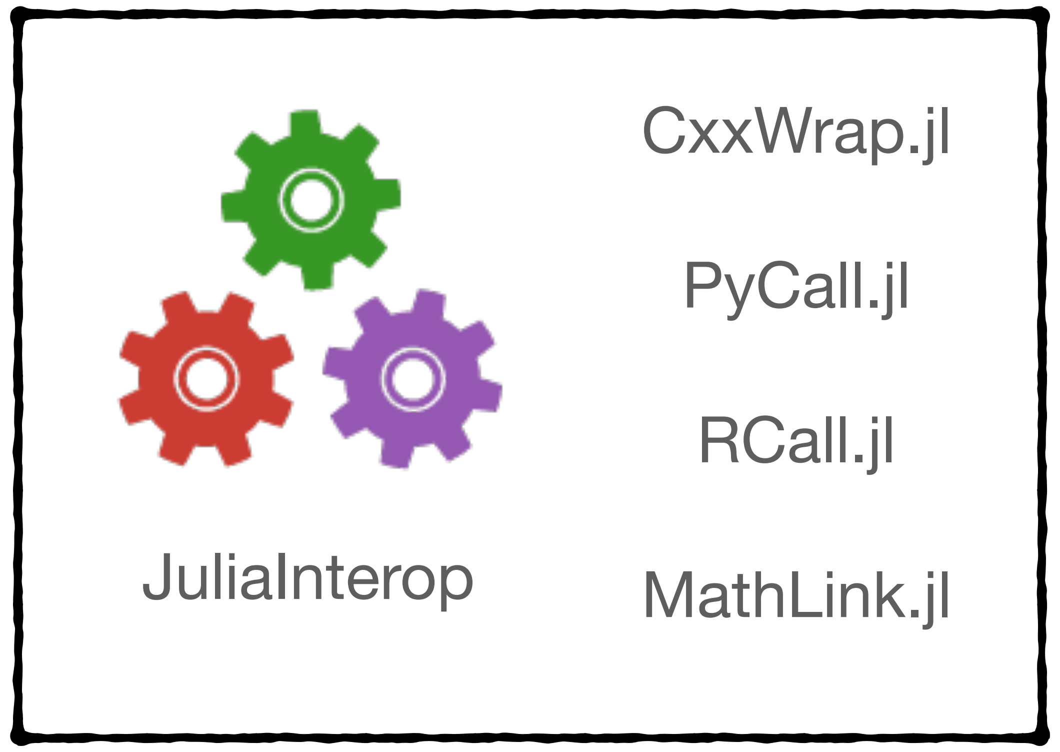


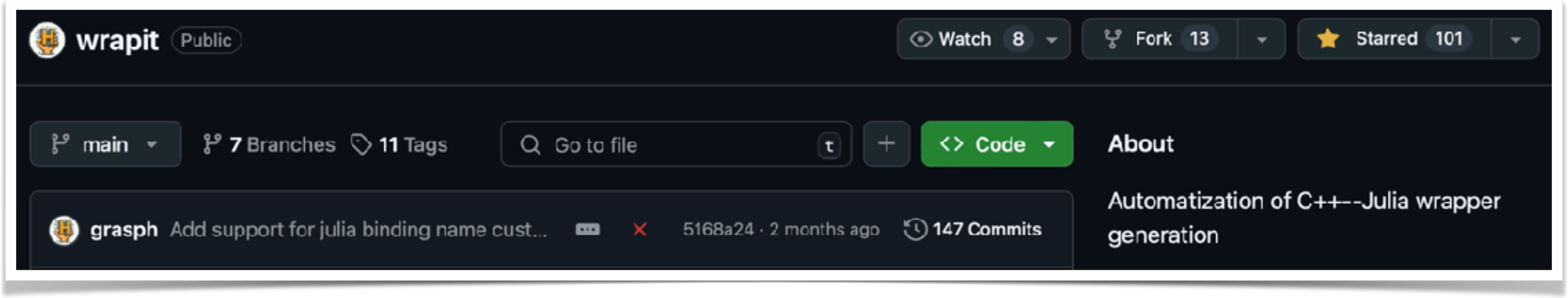
Interoperability and Legacy code

Everything is wrapped

- Use foreign code from Julia
- Wrapit and CxxWrap.jl for (semi-) automatic building of bindings
- non-exhausted list of wrapped libraries
  - Geant4.jl
  - ROOT.jl
  - XRootD.jl
  - Pythia8.jl
  - FastJet.jl
  - UpROOT.jl
  - Etc.

Interoperability





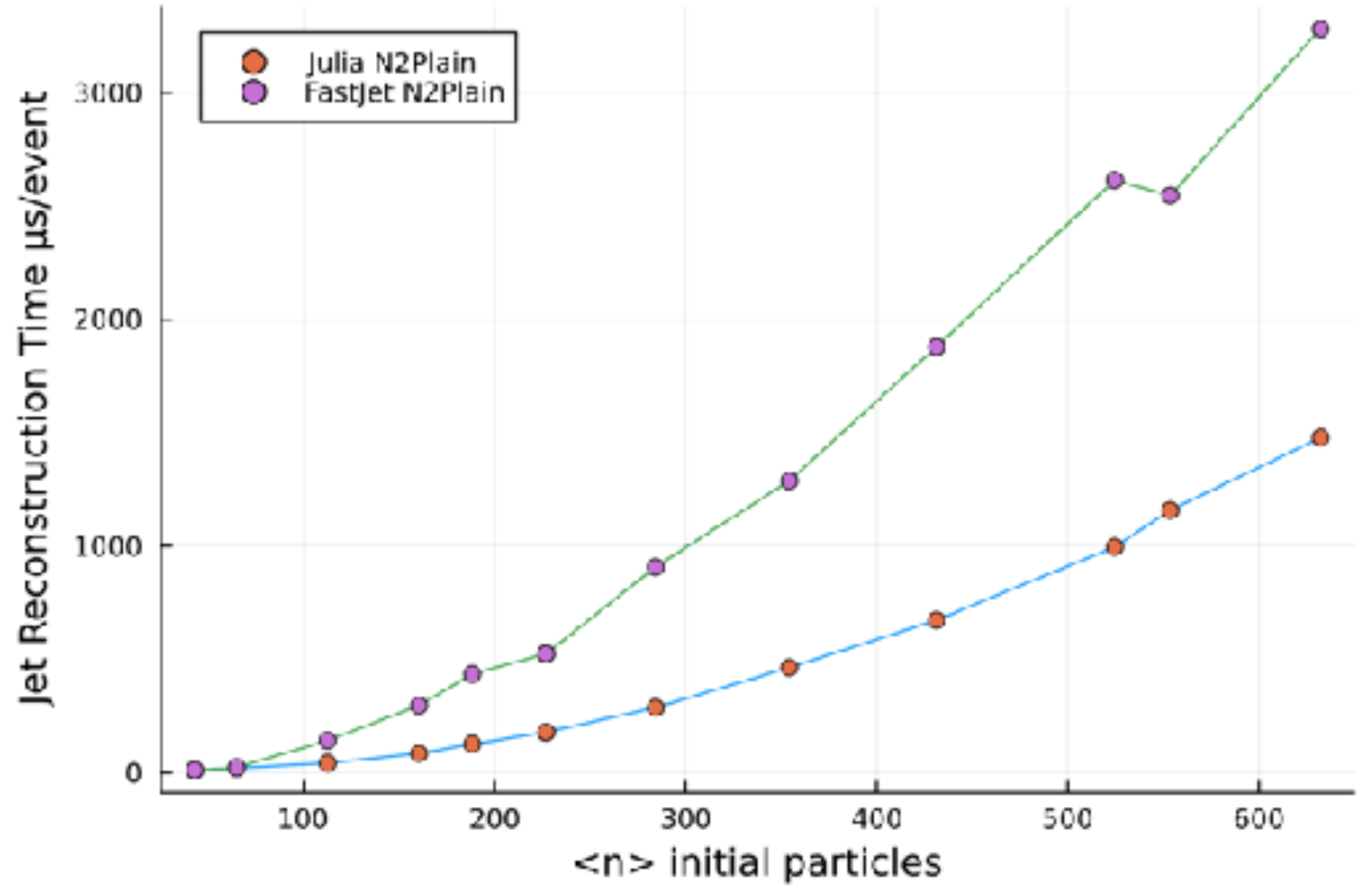
JetReconstruction.jl

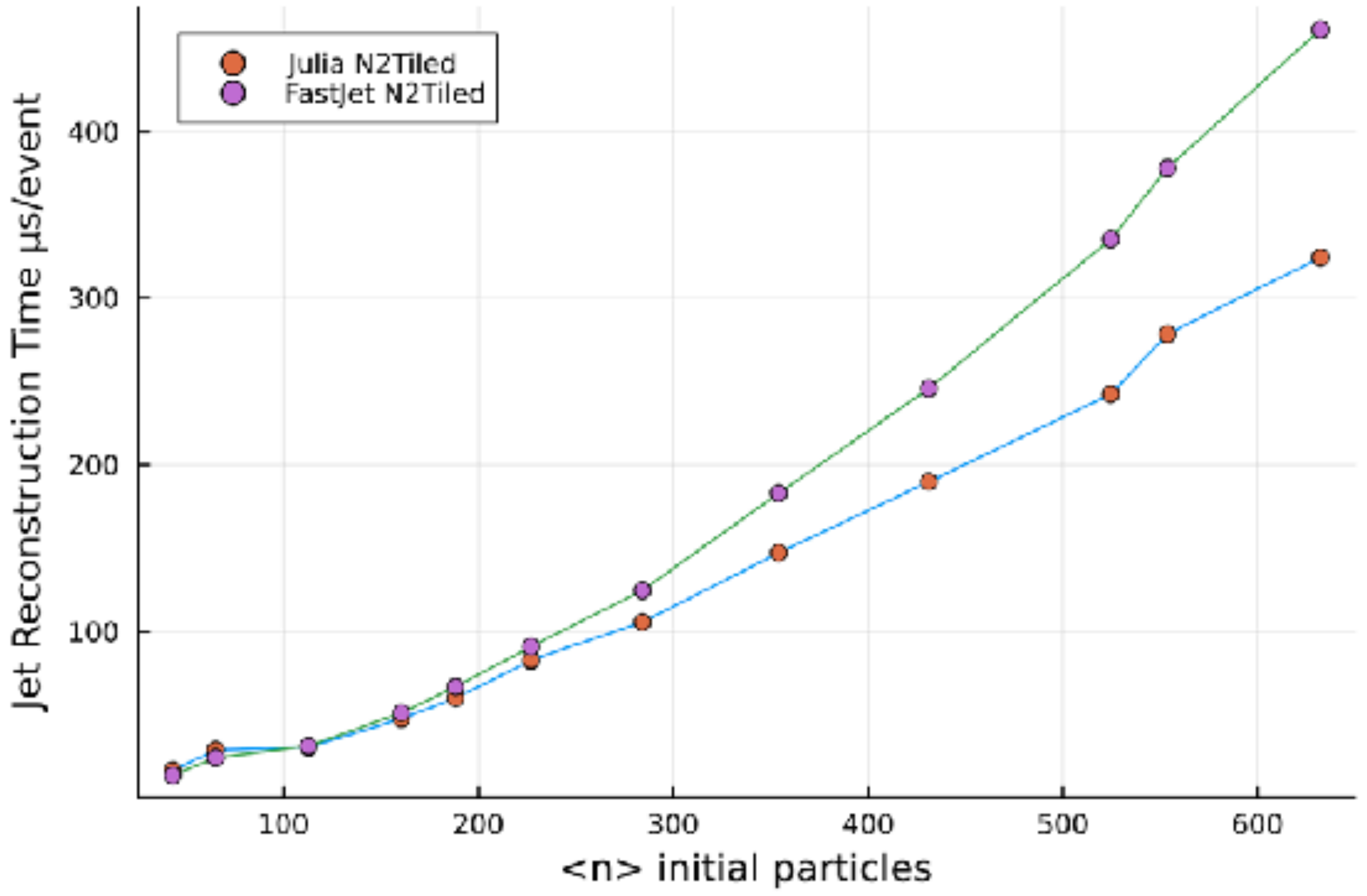
Example for rewriting

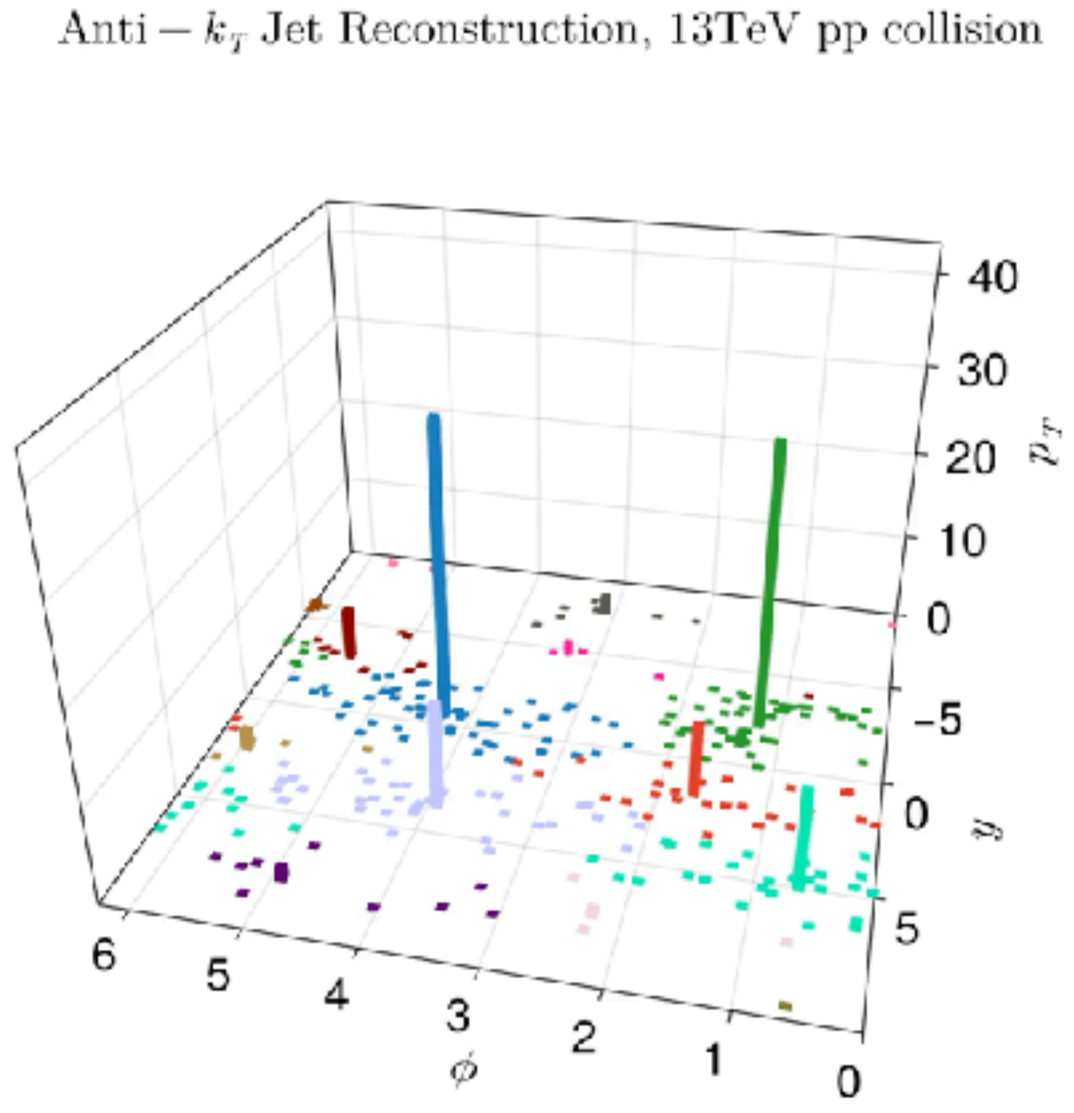
Polyglot Jet Finding

Graeme Andrew Stewart[1,*], *Philippe* Gras[2], *Benedikt* Hegner[1], and *Atell* Krasnopolski[3]

- Sequential jet clustering
  - Algorithms from FastJet
  - Fully written in Julia
  - Visualization included
- Lesson learned
  - Better ergonomics
  - Better tooling
  - Neat visualization
  - More flexible usage







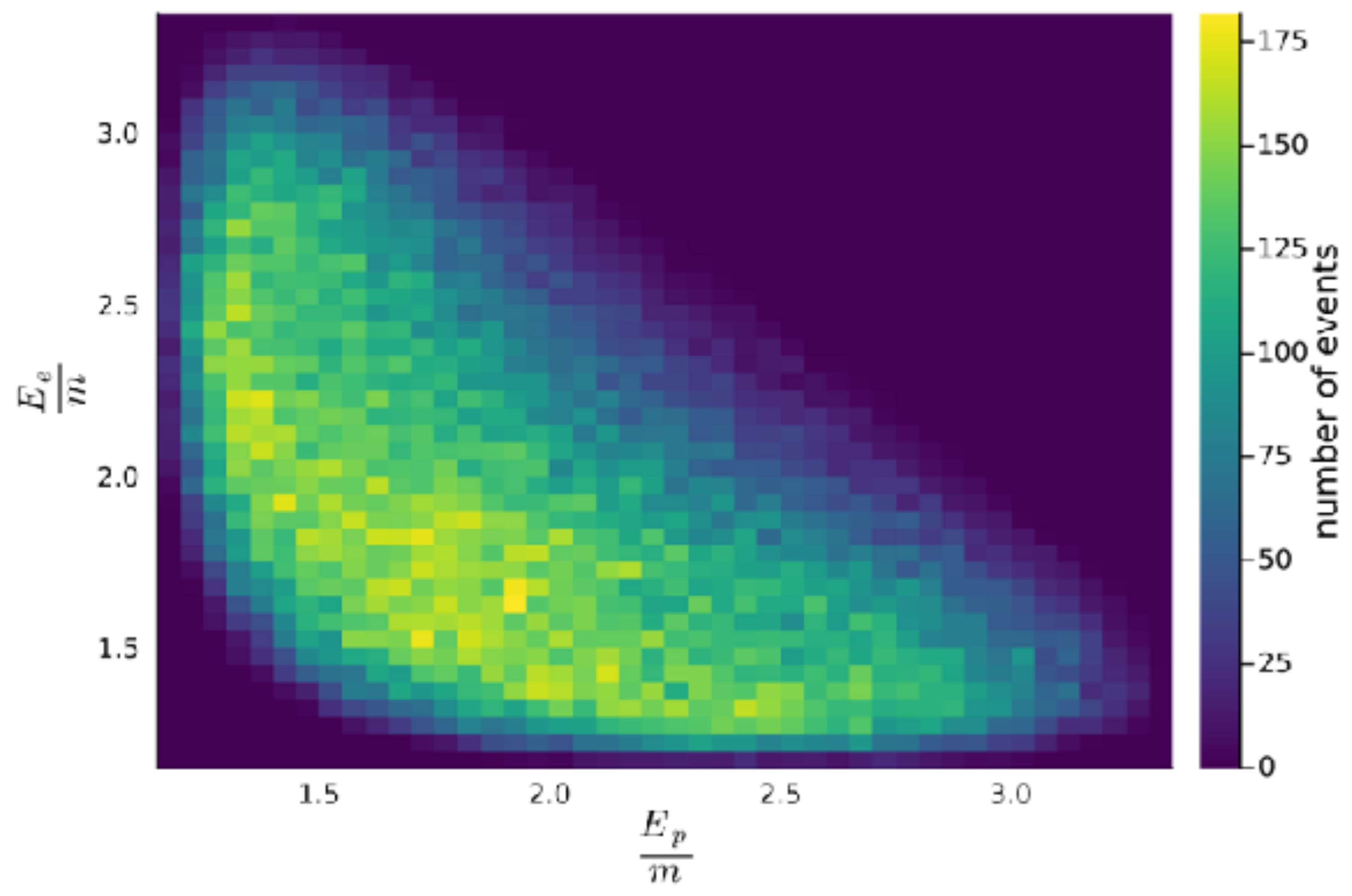
JetReconstruction.jl

QuantumElectrodynamics.jl

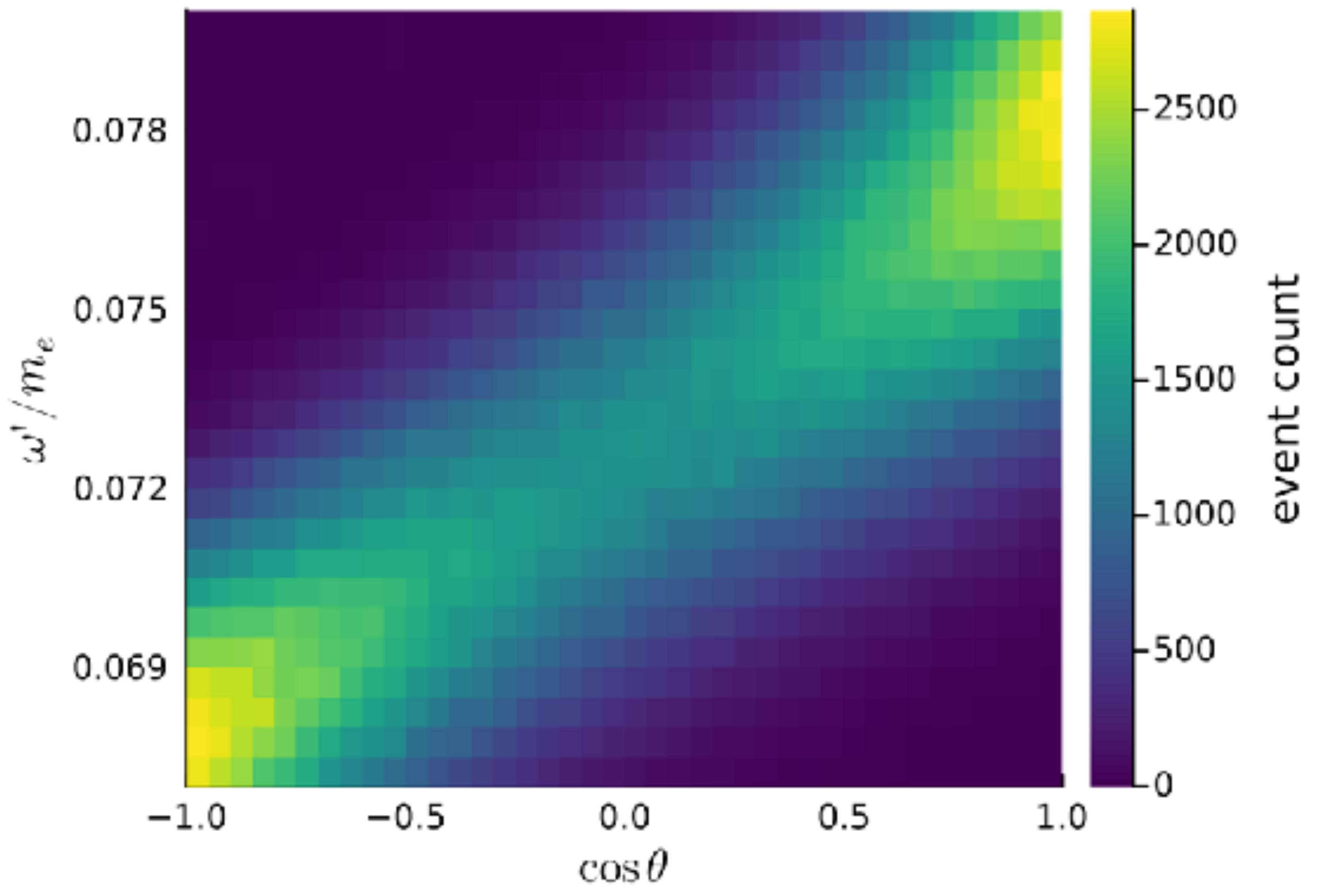
Interfaces and tools available

- Particles
- Lorentz Vectors
- Phase space points
- Computational models
- Scattering processes
- Particle distributions
- Laser fields
- Event generation

$e^- + \text{laser} \rightarrow e^- + (e^+e^-)$



$e^- + \text{laser} \rightarrow e^- + \gamma$



QuantumElectrodynamics.jl