The Julia Programming Language

E.M. Stoudenmire

FLATIRON INSTITUTE

SIMONS FOUNDATION

Oct 2021 - Flatiron FWAM

Modern Scientific Computing

Speed

Productivity

Libraries

Package manager

Interactivity / notebooks

Extensibility

⋱

Two Categories of Languages

Systems	Productivity
C/C++	Python
Fortran	MATLAB
Rust	R
D	Javascript
Go	

The Two-Language Problem

C/C++ and Fortran are 💨 *fast*

But drawbacks:

- steep learning curve
- limited library ecosystem
- limited productivity features

The Two-Language Problem

Provide Python 🐍 frontend / wrapper to C++

Great from user perspective!

Not as great for developers 👩🏻‍💻💦

- work in two languages
- compromise design & performance
- wrapping is technical, limited

The Two-Language Problem

How to escape the two-language problem?

Need a <u>single language</u>

- *productive* as Python (& libraries, package manager)
- *high-performance* as C/C++/Fortran

Julia claims both – does it succeed?

Our Experience

Initially I was skeptical about Julia

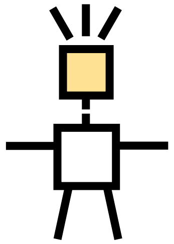
"looks like MATLAB, which is much slower than C++"

"claims are too good to be true"

Type system was appealing compared to Python...

It was attracting a lot of scientific users...

A few years later, we have completely ported our flagship library, ITensor, and are quite happy with the results



The Julia Language

How does it look?

```
N = 10
n = 0

for j=1:N
  n = n+j
end

if n ≈ N*(N+1)/2
  println("yes")
end
```

- Syntax like MATLAB
- Features like Python
- Easy to write, productive language

How does it look?

```
:: julia
julia> √(1+im)
1.09868411346781 + 0.45508986056222733im

julia> using BenchmarkTools

julia> f(z) = abs(z)^2
f (generic function with 1 method)

julia> @benchmark f(1.0+im)
BechmarkTools.Trial: 10000 samples with 999 evaluations.
 Range (min … max):   7.383 ns … 54.879 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):      7.443 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):    7.955 ns ±  2.182 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  7.38 ns      Histogram: log(frequency) by time     14.8 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.

julia> 
```

- Interactive sessions similar to Python or MATLAB
- Works with Jupyter notebooks

Arrays, Matrices, Tensors

```
a = [1,2,3,4]
typeof(a) == Vector{Int64}

M = randn(4,4)
H = M*M'
U,S,V = svd(H)
```

Array of integers

Random 4x4 matrix,
square it,
take its SVD

- Julia arrays are *fast* (customize to type inside)
- Matrices, tensors – like "built in Numpy"

Functions & Types

type of n is Int

```
function f(x,n::Int)
  z = cos(n*π*x) + im*sin(n*π*x)
  return z
end
```

- Variables have types
- Type-constrain function arguments, though not required

Compilation

```julia
function f(x,n::Int)
  z = cos(n*π*x) + im*sin(n*π*x)
  return z
end
```

```
julia> @code_native f(1.0,2)
        .section        __TEXT,__text,regular,pure_instructions
; ┌ @ REPL[2]:1 within `f'
        pushq   %rbx
        subq    $16, %rsp
        movq    %rdi, %rbx
; │ @ REPL[2]:2 within `f'
; │┌ @ operators.jl:560 within `*' @ promotion.jl:322
; ││┌ @ promotion.jl:292 within `promote'
; │││┌ @ promotion.jl:269 within `_promote'
; ││││┌ @ number.jl:7 within `convert'
; │││││┌ @ float.jl:94 within `Float64'
        vcvtsi2sd       %rsi, %xmm1, %xmm1
        movabsq $5401584520, %rax               ## imm = 0x141F5A388
; ││└└└└
; ││ @ operators.jl:560 within `*' @ promotion.jl:322 @ float.jl:332
        vmulsd  (%rax), %xmm1, %xmm1
; ││ @ operators.jl:560 within `*' @ float.jl:332
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, (%rsp)
```

assembly code

- Julia is a compiled language (== fast)
- Functions are compiled multiple times, once for each set of input types

Installing Libraries

```
julia>
(@v1.6) pkg> add CUDA
    Updating registry at `~/.julia/registries/General`
    Updating git-repo `https://github.com/JuliaRegistries/General.git`
   Resolving package versions...
   Installed LLVMExtra_jll ─ v0.0.11+0
   Installed GPUCompiler ─── v0.12.9
   Installed GPUArrays ───── v8.1.2
   Installed LLVM ────────── v4.6.0
   Installed CUDA ────────── v3.3.6
  Downloaded artifact: LLVMExtra
  Downloaded artifact: LLVMExtra
    Updating `~/.julia/environments/v1.6/Project.toml`
  [052768ef] + CUDA v3.3.6
    Updating `~/.julia/environments/v1.6/Manifest.toml`
  [ab4f0b2a] + BFloat16s v0.1.0
  [052768ef] + CUDA v3.3.6
  [0c68f7d7] + GPUArrays v8.1.2
  [61eb1bfa] + GPUCompiler v0.12.9
  [929cbde3] + LLVM v4.6.0
  [dad2f222] + LLVMExtra_jll v0.0.11+0
  Progress [========================================>]  6/6
6 dependencies successfully precompiled in 40 seconds (398 already precompiled)
```

- Built-in package manager
- Lots of great math, science & visualization libraries
- High interoperability between libraries

Composable Multithreading

https://julialang.org/blog/2019/07/multithreading/

```julia
import Base.Threads.@spawn

# sort the elements of `v` in place, from indices `lo` to `hi` inclusive
function psort!(v, lo::Int=1, hi::Int=length(v))
    if lo >= hi                       # 1 or 0 elements; nothing to do
        return v
    end
    if hi - lo < 100000               # below some cutoff, run in serial
        sort!(view(v, lo:hi), alg = MergeSort)
        return v
    end

    mid = (lo+hi)>>>1                 # find the midpoint

    half = @spawn psort!(v, lo, mid)  # task to sort the lower half; will run
    psort!(v, mid+1, hi)              # in parallel with the current call sorting
                                      # the upper half
    wait(half)                        # wait for the lower half to finish

    temp = v[lo:mid]                  # workspace for merging
```

⋮

Parallel merge sort

Spawn recursion as parallel task

Scales to many threads (even odd #'s of threads)

```
$ for n in 1 2 4 8 16; do    JULIA_NUM_THREADS=$n ./julia psort.jl; done
  2.813555 seconds (3.08 k allocations: 153.448 MiB, 1.44% gc time)
  1.731088 seconds (3.28 k allocations: 192.195 MiB, 0.37% gc time)
  1.028344 seconds (3.30 k allocations: 221.997 MiB, 0.37% gc time)
  0.750888 seconds (3.31 k allocations: 267.298 MiB, 0.54% gc time)
  0.620054 seconds (3.38 k allocations: 298.295 MiB, 0.77% gc time)
```

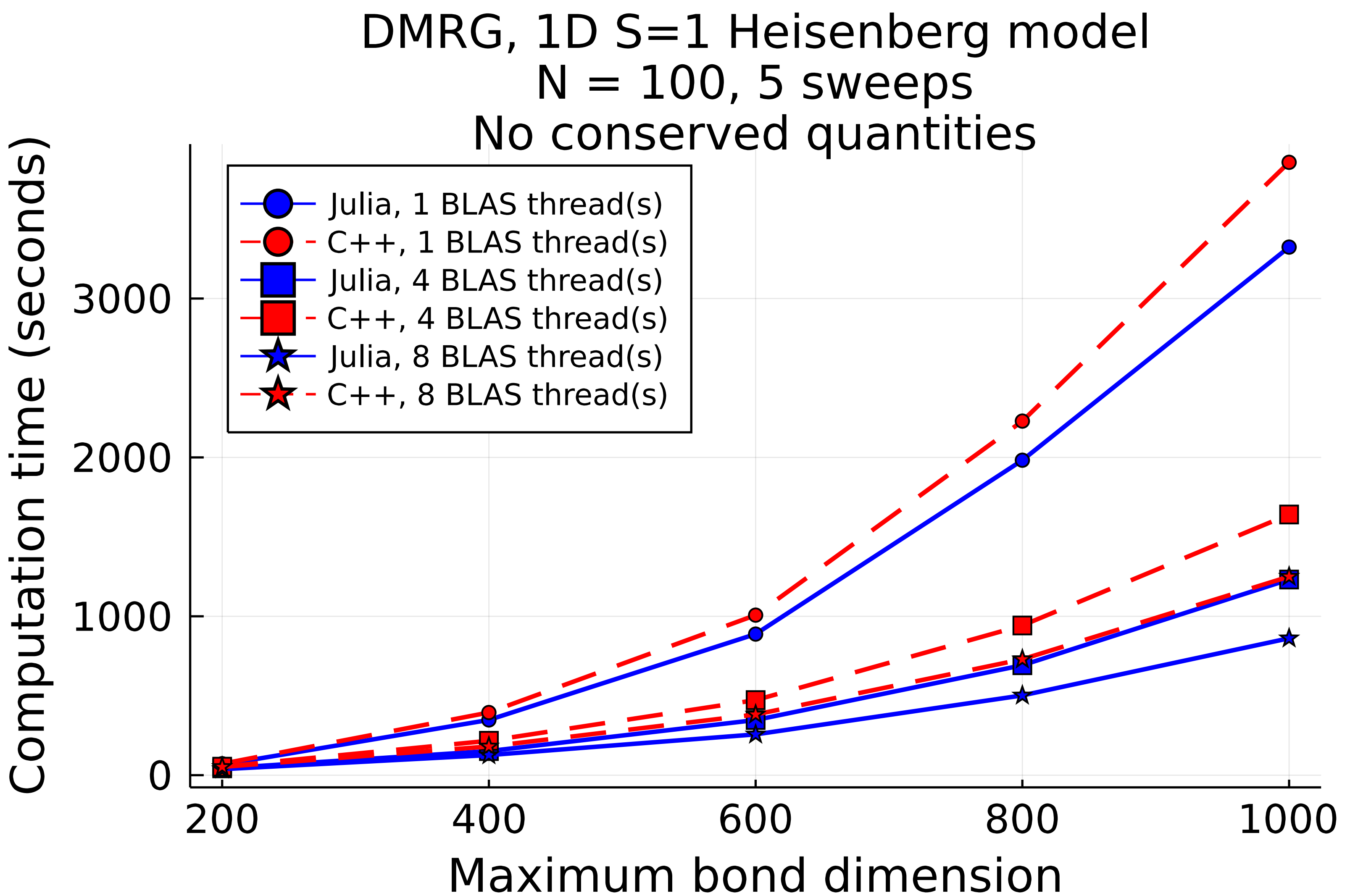
- Multithreading support with libraries for parallel algs
- Threading is <u>composable</u>: automatically handles nested cases

Benchmarks

ITensor DMRG* Benchmarks

C++ code state-of-the-art,

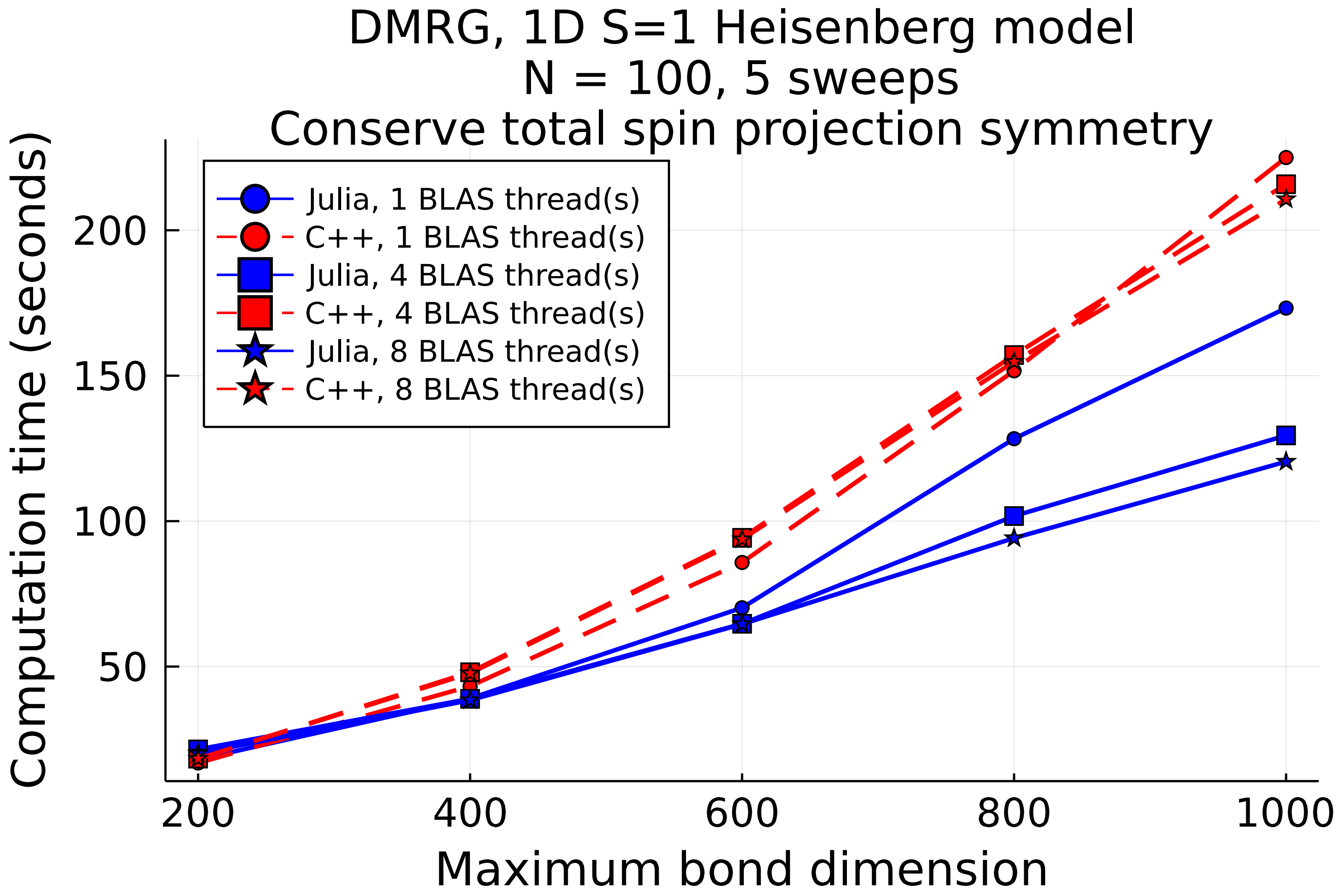
Julia version is a little faster!



*workhorse physics algorithm

ITensor DMRG Benchmarks

Version of code that spends less time in BLAS – Julia even further ahead

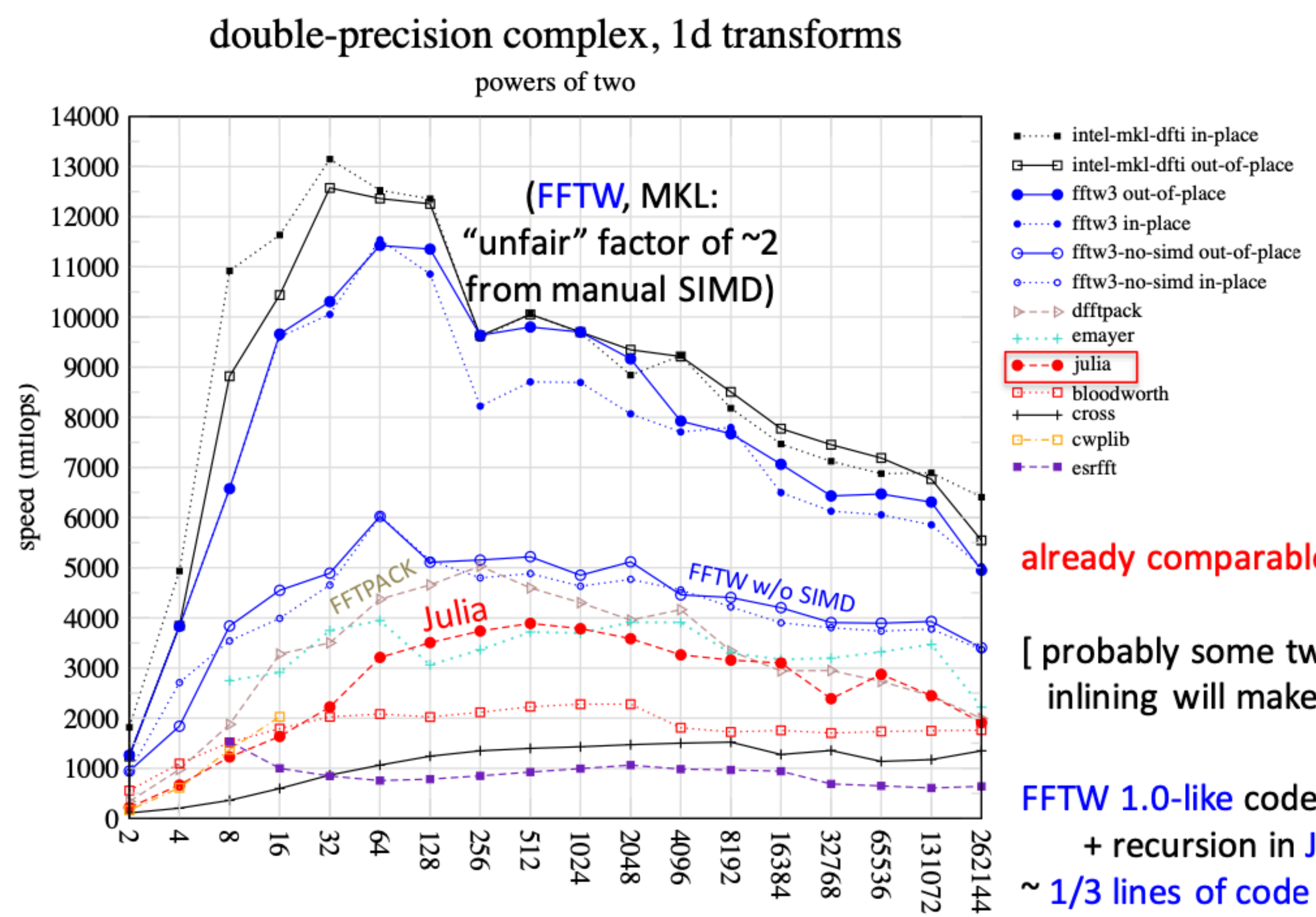


Other Benchmarks

Don't just take it from me!

Steven Johnson (co-creator, FFTW)

Pure-Julia FFT performance*



already comparable to FFTPACK

[probably some tweaks to inlining will make it better]

FFTW 1.0-like code generation + recursion in Julia

~ 1/3 lines of code compared to FFTPACK, more functionality

* from 2017

Julia As a C++ Alternative

Julia As a C++ Alternative

Some similarities

- compiled (fast)
- variables have types
- generic programming with constraints

Generic ("templated") functions

C++

```
template<typename Number>
f(Number x, int j)
{
    return x+j;
}
```

Julia

```
function f(x, j::Int)
  return x+j
end
```

same assembly code!

Julia As a C++ Alternative

Advantages of Julia

- package manager, library ecosystem
- many productivity features (linear algebra, modules, can print anything)
- macro system (code writing code)
- interactive mode, notebooks
- multiple dispatch
- introspection
- ...

Julia As a C++ Alternative

Advantages of C++

- control over memory management
- static analysis (helpful compiler errors)
- program starts immediately (no latency)

Julia As a Python Alternative

Julia As a Python Alternative

Some similarities

- productive & good for prototyping
- "batteries included" standard libraries
- useful anonymous types (tuples, lists)

```
julia> v = [2n for n=1:4]
4-element Vector{Int64}:
 2
 4
 6
 8

julia> ("v" => v, "x" => 1.0)
("v" => [2, 4, 6, 8], "x" => 1.0)
```

Julia As a Python Alternative

Advantages of Julia

- high-performance *without* C/C++ underneath (no two-language problem)
- multithreading
- macro system / compiler interaction (AD, GPU)
- standard, built-in package manager
- multiple dispatch, types
- ...

Julia As a Python Alternative

Advantages of Python

- mature, extensive library ecosystem
- widespread adoption

What's Not So Good about Julia

What's Not So Good about Julia

Just-in-time compilation (JIT) latency

```
julia> f(x)
  ...
```

ranges from seconds...to minutes

Though very fast thereafter!

What's Not So Good about Julia

Type instability

```julia
function unstable(n::Int)
  r = 0
  for i = 1:n
    r += sin(0.3*n)
  end
  return r
end
```

```
julia> @code_warntype unstable(3)
Variables
  #self#::Core.Const(unstable)
  n::Int64
  @_3::Union{Nothing, Tuple{Int64, Int64}}
  r::Union{Float64, Int64}
  i::Int64

Body::Union{Float64, Int64}
1 ─       (r = 0)
│   %2  = (1:n)::Core.PartialStruct(UnitRange{Int64}, Any[Core.Const(1),
│         (@_3 = Base.iterate(%2))
│   %4  = (@_3 === nothing)::Bool
│   %5  = Base.not_int(%4)::Bool
└──       goto #4 if not %5
2 ┄ %7  = @_3::Tuple{Int64, Int64}::Tuple{Int64, Int64}
│         (i = Core.getfield(%7, 1))
│   %9  = Core.getfield(%7, 2)::Int64
│   %10 = r::Union{Float64, Int64}
```

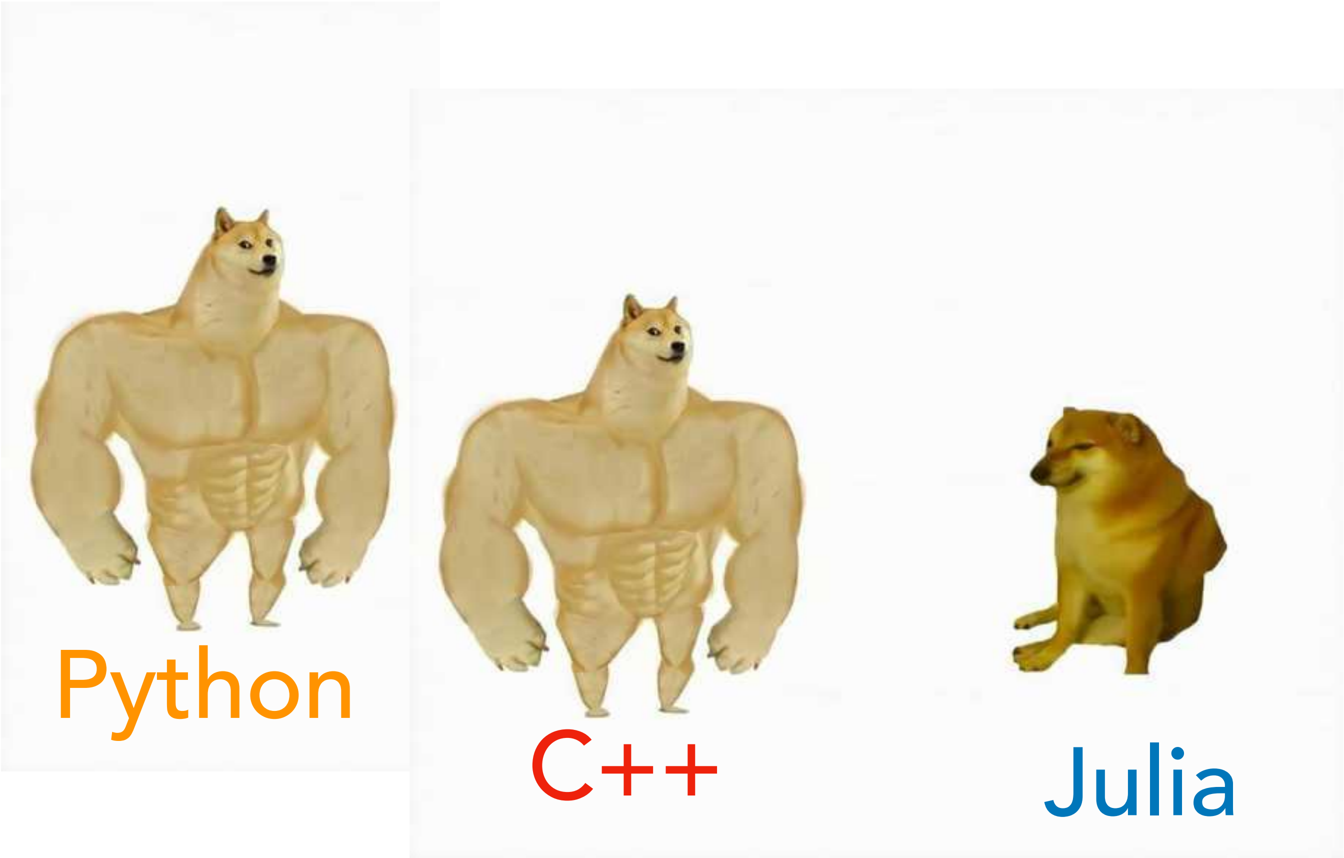
tradeoff of dynamism – need improved tools to prevent and detect

What's Not So Good about Julia

Garbage collection, performance issue? (versus C++)

Less mature library ecosystem (versus Python)

Julia still developing on cluster environments (MPI, etc.)



The Opportunity of Julia

The Opportunity of Julia

A single language we could use for

- training
- code contribution & development
- high-performance

The Opportunity of Julia

A single language we could use for

- training
- code contribution & development
- high-performance

Julia libraries highly interoperable
Vision of Flatiron codes in Julia, sharing libraries?

The Opportunity of Julia

A single language we could use for

- training
- code contribution & development
- high-performance

Julia libraries highly interoperable
Vision of Flatiron codes in Julia, sharing libraries?

Port even BLAS to Julia? (this is happening)

Composable, multithreaded BLAS of non-standard numeric types (tropical numbers, etc.)

Summary

Julia is a modern, compiled, fast language

Scientific-computing oriented

- linear algebra built in
- science-oriented community
- high library interoperability

Combines many best aspects of C++, Python
in a single language