Julia on Ookami December 2022

Valentin Churavy & Mosè Giordano

Who?

Mosè Giordano
UCL

Valentin Churavy
MIT

Yet another high-level language?

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

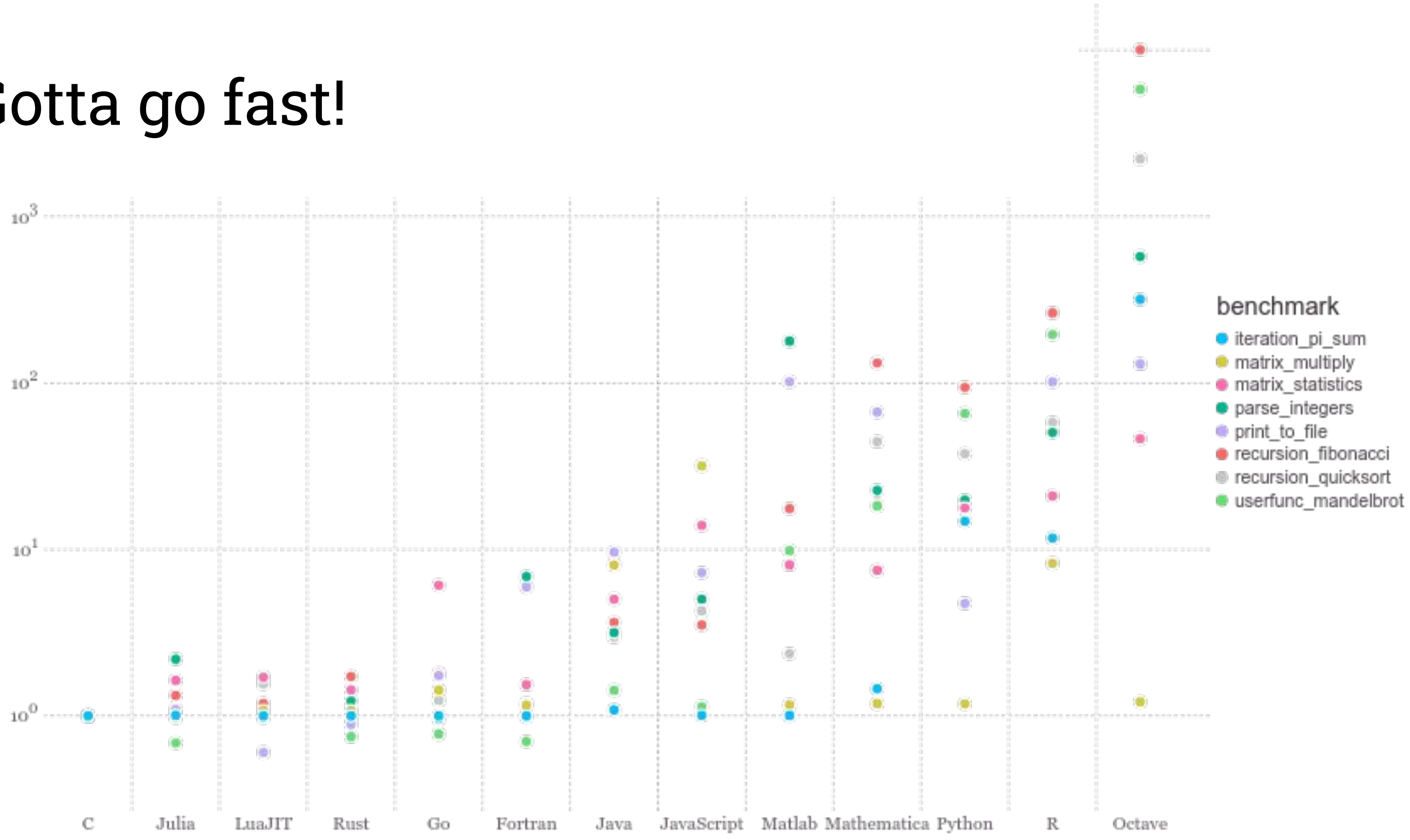
```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end

julia> mandel(complex(.3, -.6))
14
```

Yet another high-level language?

Typical features	**Unusual features**
Dynamically typed, high-level syntax	Great performance!
Open-source, permissive license	JIT AOT-style compilation
Built-in package manager	Most of Julia is written in Julia
Interactive development	Reflection and metaprogramming

Gotta go fast!



What makes a language dynamic?

- Commonly: Referring to the type system.
  - **Static:** Types are checked before run-time
  - **Dynamic:** Types are checked on the fly, during execution
  - Also: The type of a **variable** can change during execution
- Closed-world vs open-world semantics
  - The presence of **eval** (Can code be “added” at runtime)
- Struct layout
  - Can one change the fields of a object/class/struct at runtime?

```julia
x = true
if cond
  x = "String"
end
@show x
```

Dynamic semantics are a **spectrum**:

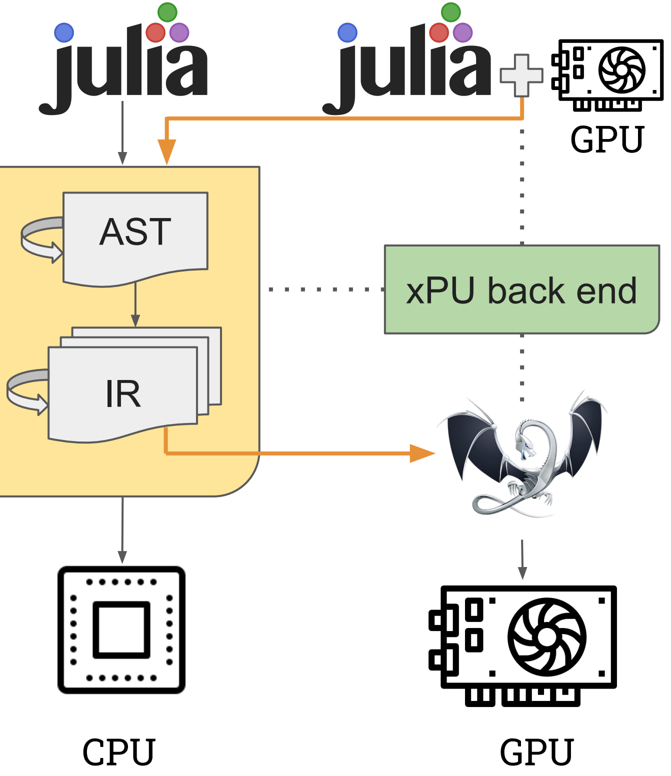
Julia has a dynamic type system and open-world semantics,

but struct layout is static.

julia gets its Power from Extensible Compiler Design

Language design

Efficient execution



Julia: Dynamism and Performance Reconciled by Design (doi:10.1145/3276490)

Effective Extensible Programming: Unleashing Julia on GPUs (doi:10.1109/TPDS.2018.2872064)

Magic of Julia

Abstraction, Specialization, and Multiple Dispatch

Did I really need to move memory for that transpose?

1. **Abstraction** to obtain generic behavior:

   Encode behavior in the type domain:

   `transpose(A::Matrix{Float64})::Transpose{Float64,Matrix{Float64}}`

2. **Specialization** of functions to produce optimal code

3. **Multiple-dispatch** to select optimized behavior

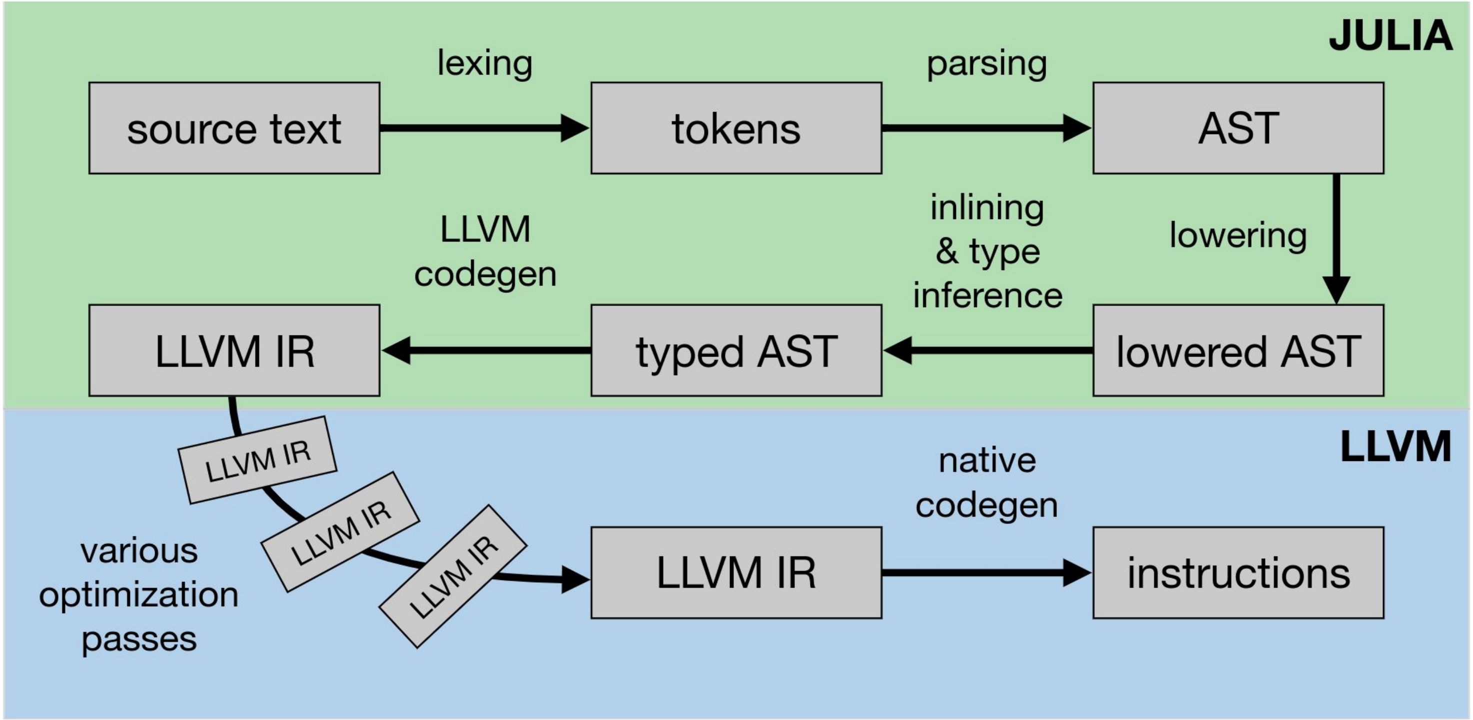
```
rand(N, M) * rand(K, M)'
Matrix * Transpose{Matrix}
```

compiles to

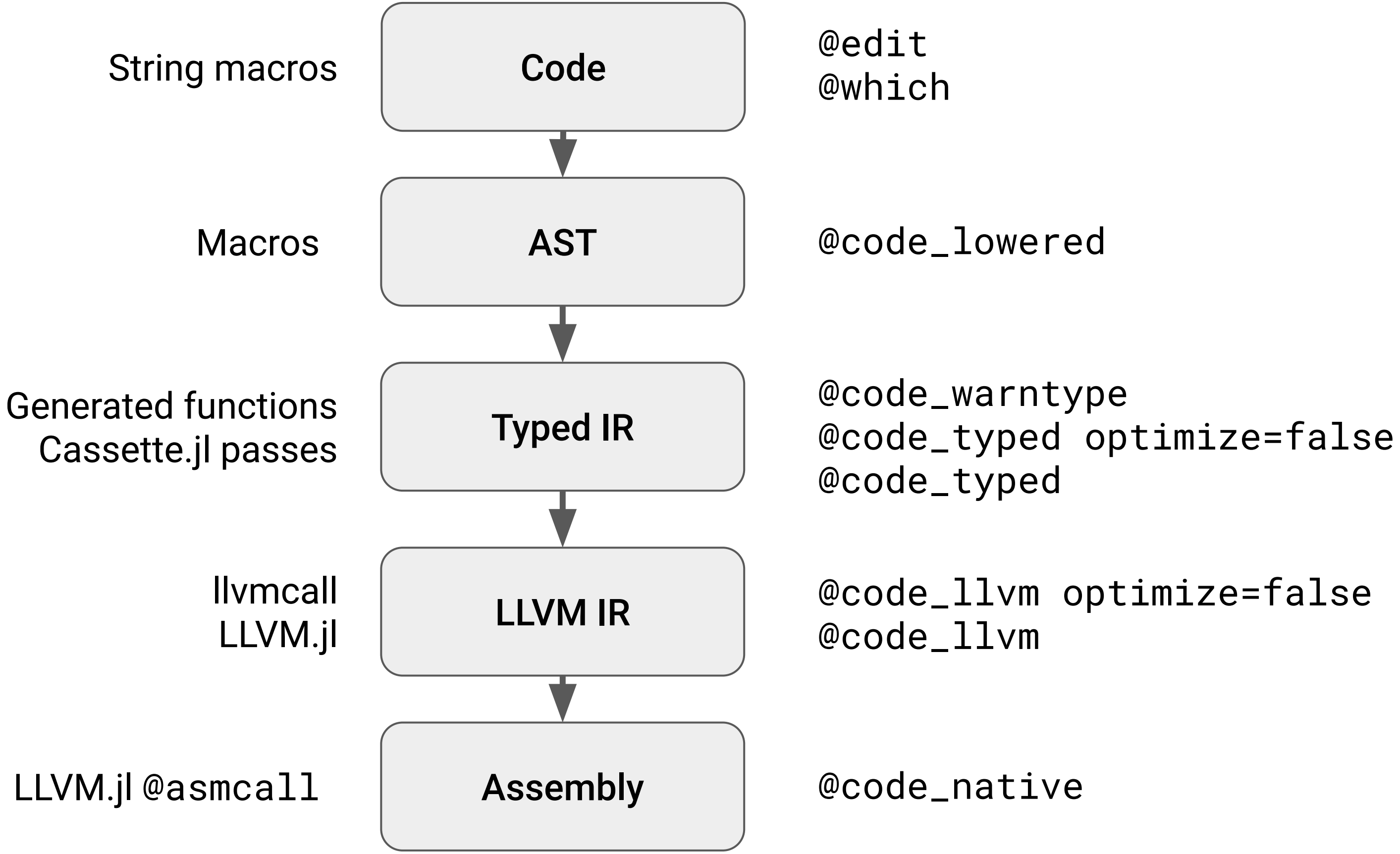
```
function mul!(C::Matrix{T}, A::Matrix{T}, tB::Transpose{<:Matrix{T}}, a, b) where {T<:BlasFloat}
    gemm_wrapper!(C, 'N', 'T', A, B, MulAddMul(a, b))
end
```

No I did not! I know AB^T is the dot product of every row of A with every row of B .

Compiling Julia



Introspection and staged metaprogramming



HPC – Interacting with the System

1. Julia has direct foreign call support for C & Fortran
    - https://docs.julialang.org/en/v1/base/c/#Base.@ccall
    - https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code

2. Automatic wrapper generation with Clang.jl

3. @cfunction creates a C-function pointer to use as a callback
    - Currently (-v1.8) can **only** be called from a Julia managed thread
      (Except for very carefully crafted exceptions)
    - Julia v1.9- will support callbacks from arbitrary foreign threads

4. Be careful around GC interactions!

Example UCX.jl

```julia
function ucp_put_nb(ep, buffer, length, remote_addr, rkey, cb)
    ccall(
        (:ucp_put_nb, libucp),
        ucs_status_ptr_t,
        (ucp_ep_h, Ptr{Cvoid}, Csize_t, UInt64, ucp_rkey_h, ucp_send_callback_t),
        ep, buffer, length, remote_addr, rkey, cb)
end


function send_callback(req::Ptr{Cvoid}, status::API.ucs_status_t, user_data::Ptr{Cvoid})
    @assert user_data !== C_NULL
    request = UCXRequest(user_data)
    request.status = status
    notify(request)
    API.ucp_request_free(req)
    nothing
end

function put!(ep::UCXEndpoint, request, data::Ptr, nbytes, remote_addr, rkey)
    cb = @cfunction(send_callback, Cvoid, (Ptr{Cvoid}, API.ucs_status_t, Ptr{Cvoid}))
    ptr = ucp_put_nb(ep, data, nbytes, remote_addr, rkey, cb)
    return handle_request(request, ptr)
end

function put!(ep::UCXEndpoint, buffer, nbytes, remote_addr, rkey)
    request = UCXRequest(ep, buffer) # rooted through ep.worker
    GC.@preserve buffer begin
        data = pointer(buffer)
        put!(ep, request, data, nbytes, remote_addr, rkey)
    end
end
```

What about binaries?

- We need to support:
  - Windows (32bit&64bit); Mac OS (x86_64&aarch64); Linux (Intel, ARM, PPC); FreeBSD
  - Can’t assume compiler available on user system
- Artifacts: Immutable, platform specific “archives”
- Doesn't overburden the Package manager (version control -> artifact selection)
- https://binarybuilder.org & https://github.com/JuliaPackaging/Yggdrasil
  - Cross-compilation toolchain for building binaries reliably
  - Repository of (user-submitted) recipes
  - Buildfarm + automatic release as JLL packages
  - https://www.youtube.com/watch?v=S__x3K31qnE
  - https://www.youtube.com/watch?v=_jI8CbN_-IE

_jll Packages

- A binary released as a Julia package that the package manager can install
- Transparently loading dependencies and makes libraries available

BUT! What about my bespoke HPC cluster? I must use HPE/Cray...

1. Everything is overwritable/exchangeable
2. JLLWrappers.jl uses Preferences.jl to make library location configurable
3. Extended platform tags supports things like MPI ABI
    a. MPI.jl 0.20 and MPIPreferences.jl

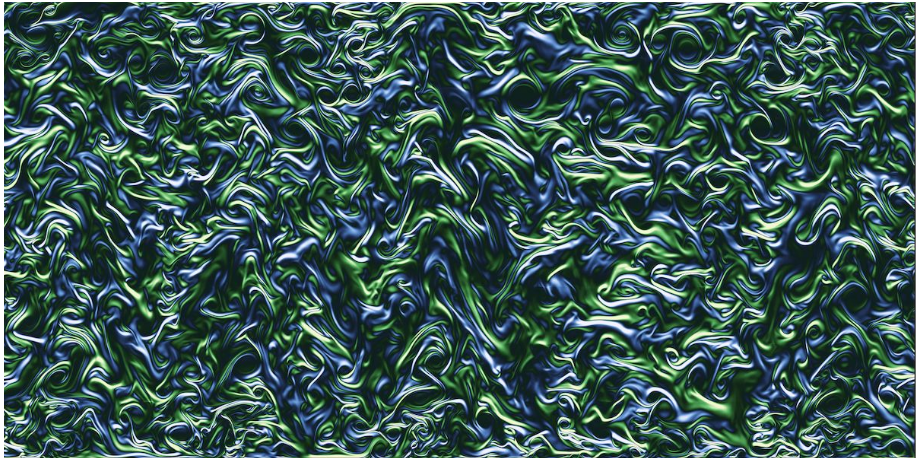
Long term plan to seamlessly integrate with HPC centric tools such as Spack! If you want to help with that reach out!

Mixed precision computing in Julia

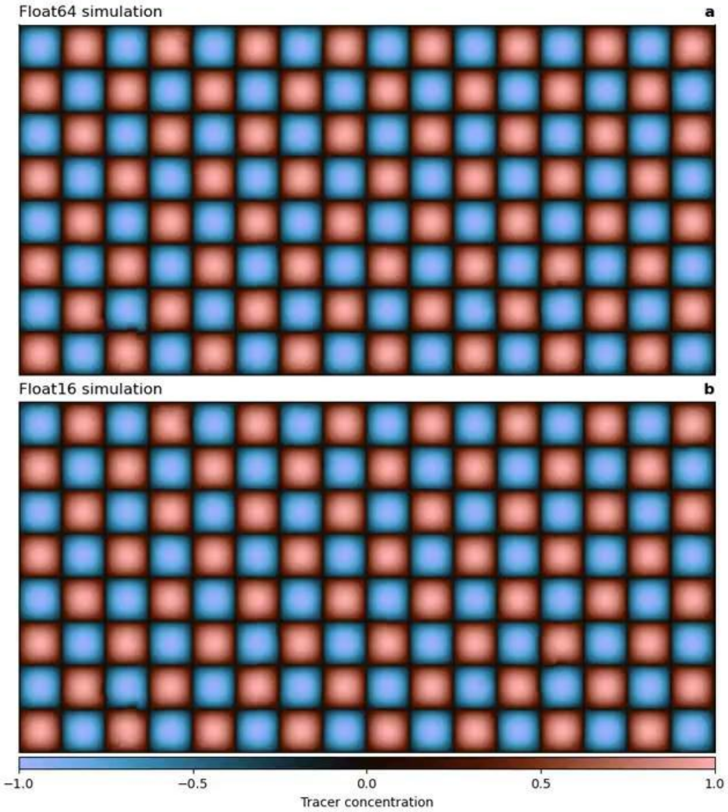
Milan Klöwer
MIT/Oxford

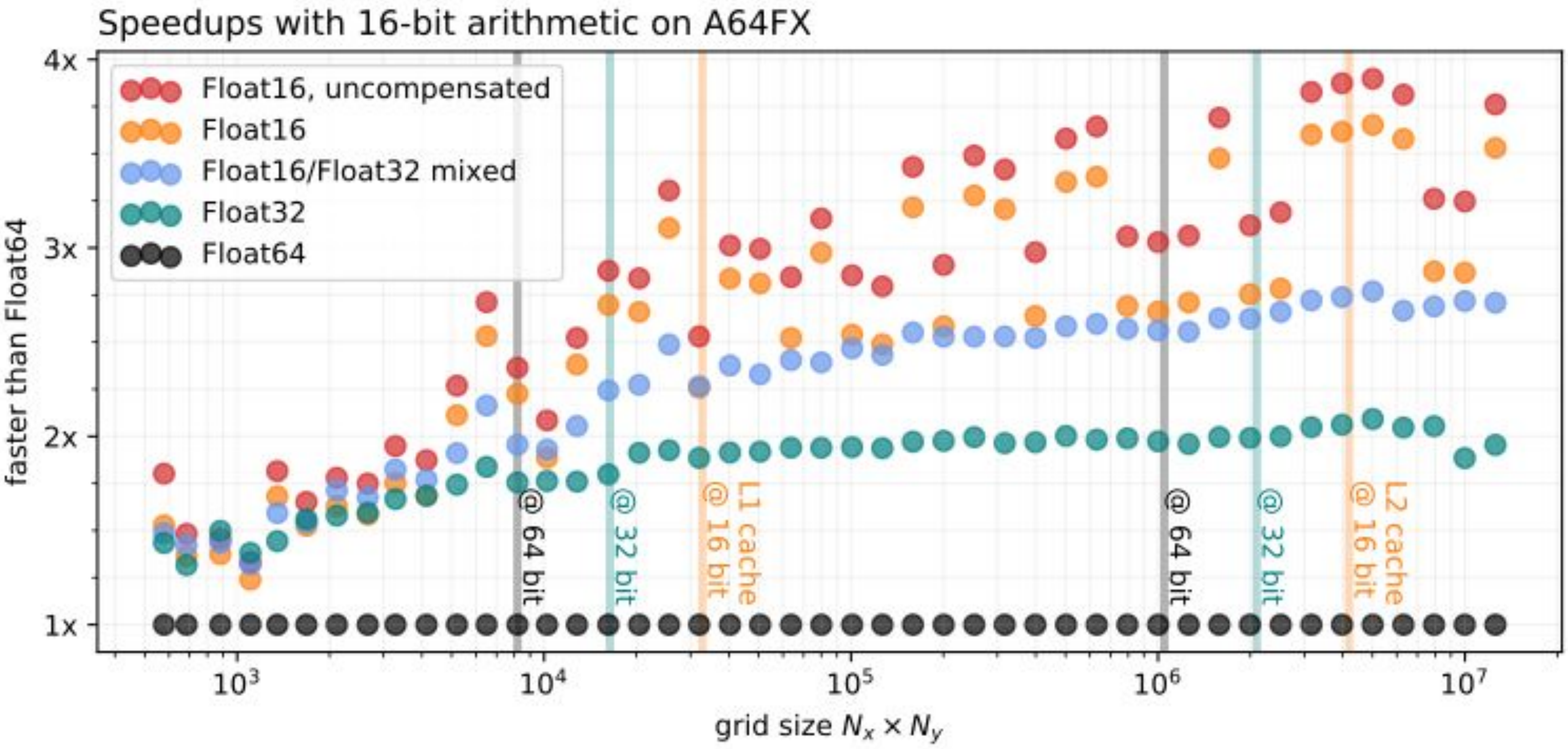
ShallowWaters.jl

- Open-Source CFD code written in Julia
- Type-agnostic/Type-flexible
  - Compensated summation for low-precision
- ~4x speedup with `Float16` and 2x speedup with `Float32` over `Float64`
- Qualitative results equivalent between `Float64` and `Float16`



ShallowWaters.jl — Fidelity comparison





Reproduced from https://doi.org/10.1029/2021MS002684

Float16 in Julia

```
abstract type Number end
abstract type Real <: Number end
abstract type AbstractFloat <: Real end
primitive type Float64 <: AbstractFloat 64 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float16 <: AbstractFloat 16 end
```

```
julia> methods(cbrt)
# 7 methods for generic function "cbrt":
[1] cbrt(x::Union{Float32, Float64}) in Base.Math at
special/cbrt.jl:142
[2] cbrt(a::Float16) in Base.Math at special/cbrt.jl:150
[3] cbrt(x::BigFloat) in Base.MPFR at mpfr.jl:626
[4] cbrt(x::AbstractFloat) in Base.Math at
special/cbrt.jl:34
[5] cbrt(x::Real) in Base.Math at math.jl:1352
```

Taking Float16 seriously

First attempt: Naively lowering `Float16` to LLVM's `half` type.

1. What to do on platforms with no/limited hardware support
2. Extended precision (thanks x87) rears it's ugly head

Lesson: In order to implement numerical routines that are portable we must be very careful in what semantics we promise.

Solution: On targets without hardware support for `Float16`, truncate after each operation.

GCC 12 supports this as: `-fexcess-precision=16`

```
define half @julia_muladd(half %0,
half %1, half %2) {
top:
  %3 = fmul half %0, %1
  %4 = fadd half %3, %2
  ret half %4
}
```

```
define half @julia_muladd(half %0, half %1, half %2){
top:
  %3 = fpext half %0 to float
  %4 = fpext half %1 to float
  %5 = fmul float %3, %4
  %6 = fptrunc float %5 to half
  %7 = fpext half %6 to float
  %8 = fpext half %2 to float
  %9 = fadd float %7, %8
  %10 = fptrunc float %9 to half
  ret half %10
```