Productivity meets Performance Julia on A64FX

Valentin Churavy
vchuravy@mit.edu

Who?

Mosè Giordano
UCL

Milan Klöwer
Oxford

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

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

Unusual features

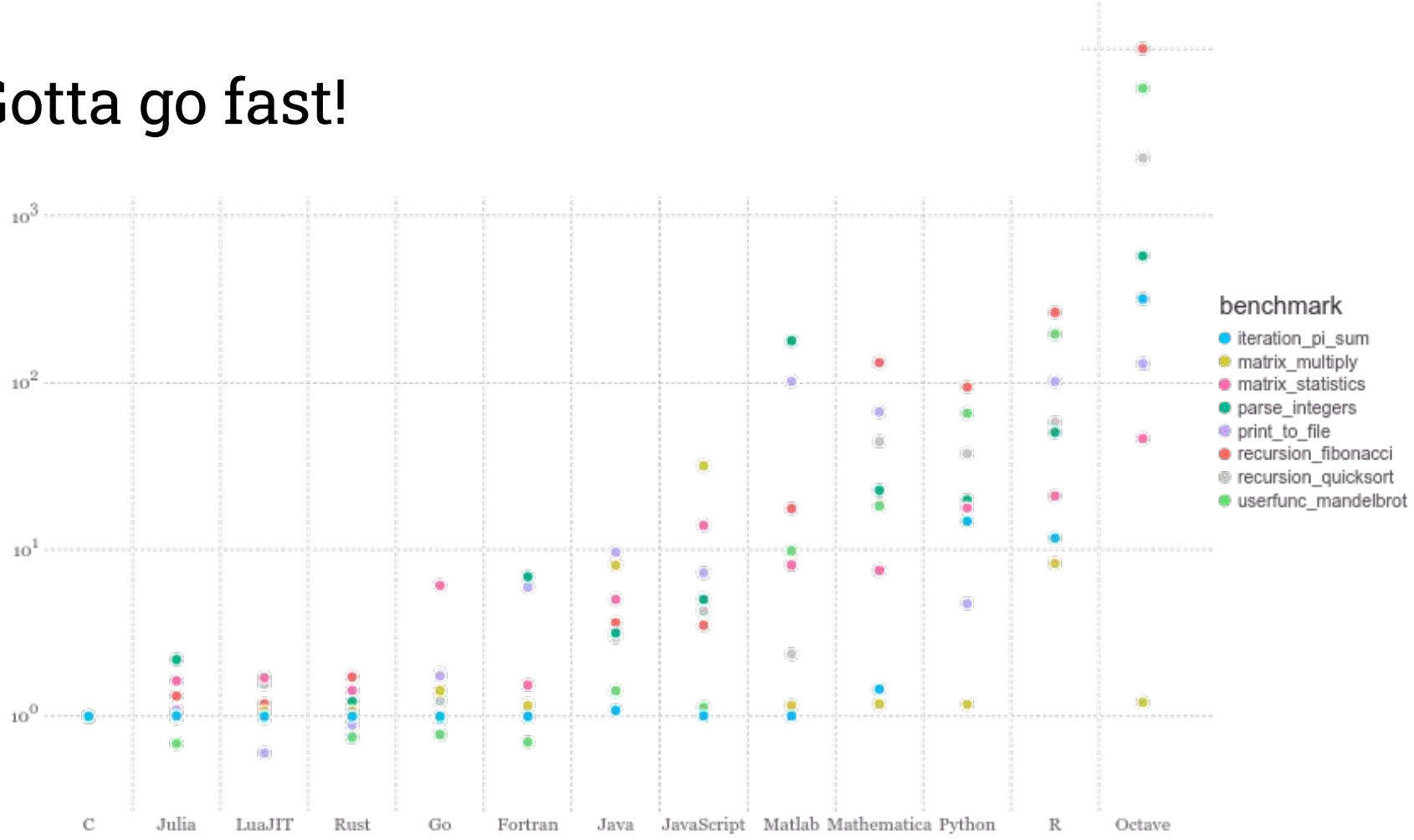
Great performance!

JIT AOT-style compilation

Most of Julia is written in Julia

Reflection and metaprogramming

Gotta go fast!



What makes a language dynamic?

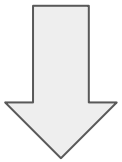
- Commonly: Referring to the type system.
  - **Static:** Types are checked before run-time
  - **Dynamic:** Types are checked on the fly, during execution
  - Also: The type of a **variable** can change during execution
- Closed-world vs open-world semantics
  - The presence of **eval** (Can code be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

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

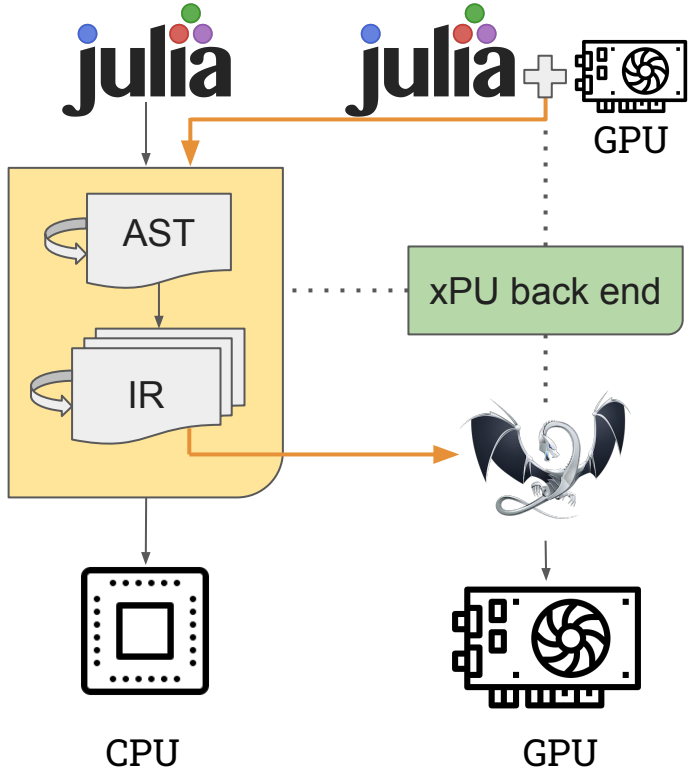
Dynamic semantics are a **spectrum**:
Julia has a dynamic type system and open-world semantics,
but struct layout is static.

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:

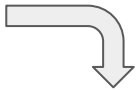
   Encode behavior in the type domain:

   ```
   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)'
```

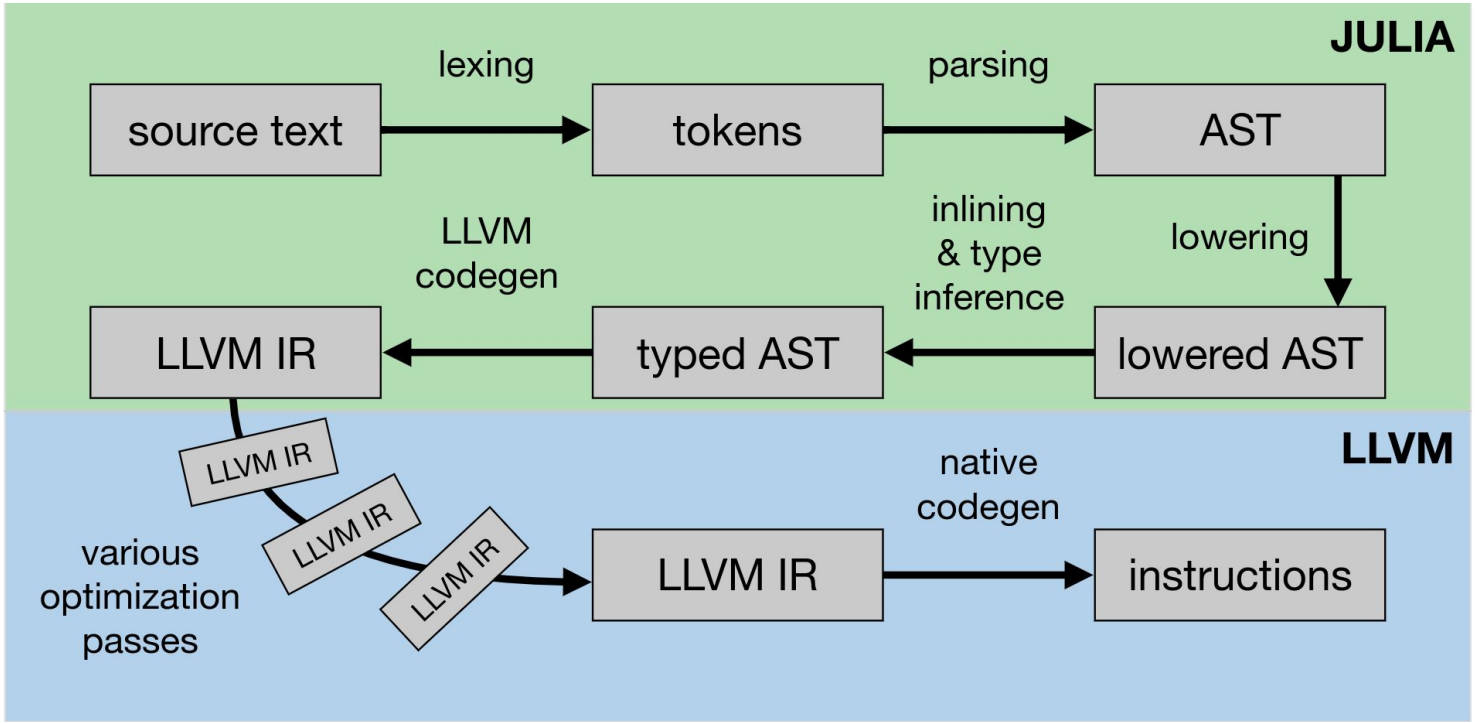


compiles to

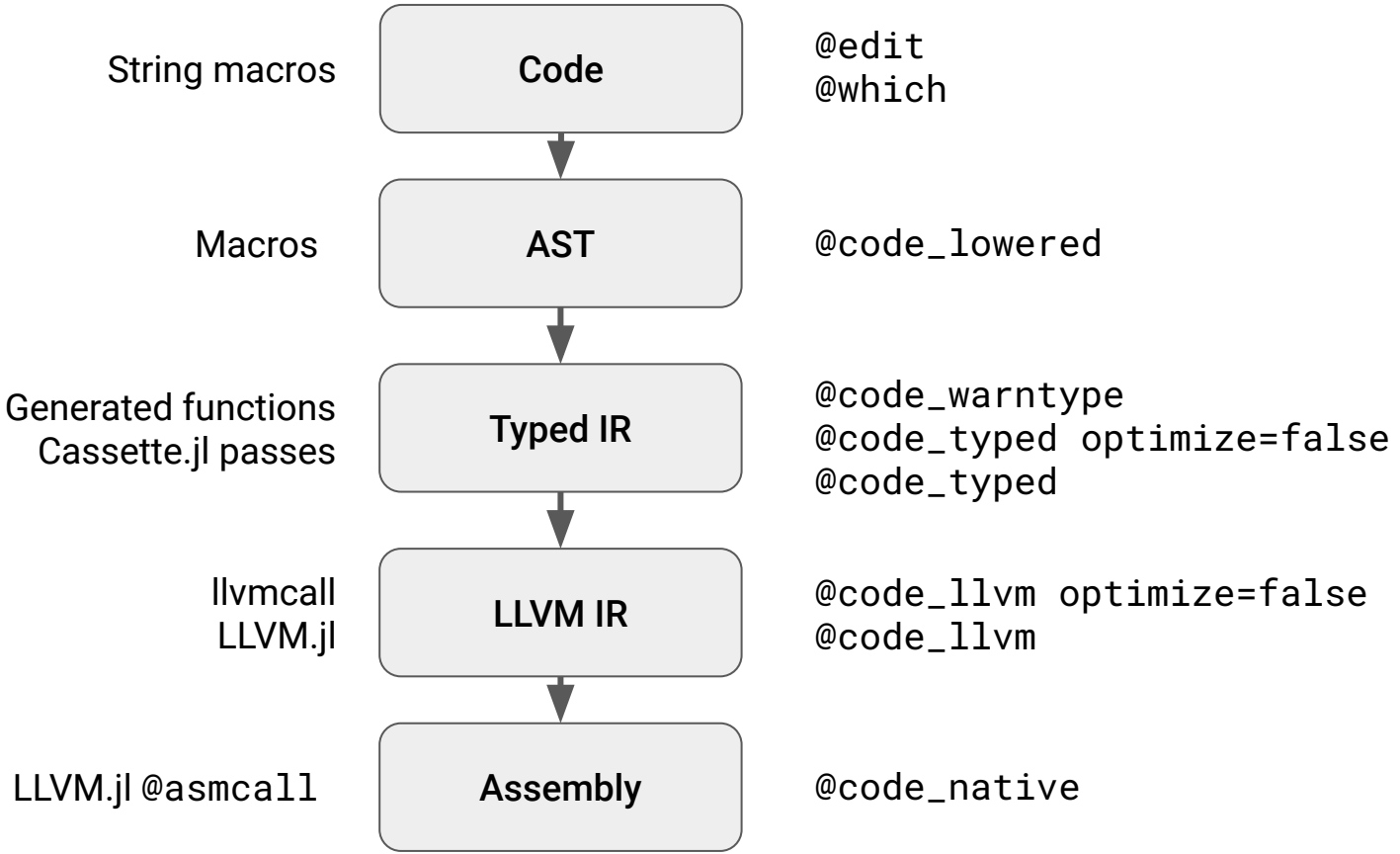
```
Matrix * Transpose{Matrix}
function mul!(C::Matrix{T}, A::Matrix{T}, tB::Transpose{<:Matrix{T}}, a, b) where {T<:BlasFloat}
    gemm_wrapper!(C, 'N', 'T', A, B, MulAddMul(a, b))
end
```

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

Compiling Julia

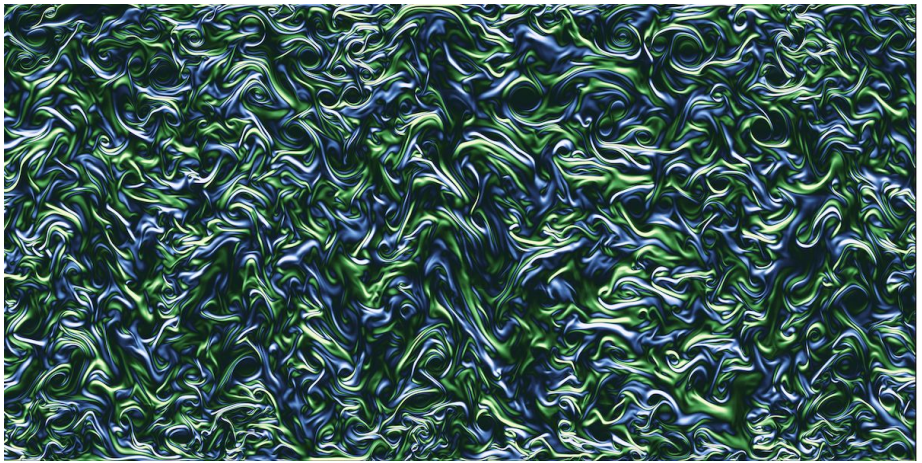


Introspection and staged metaprogramming

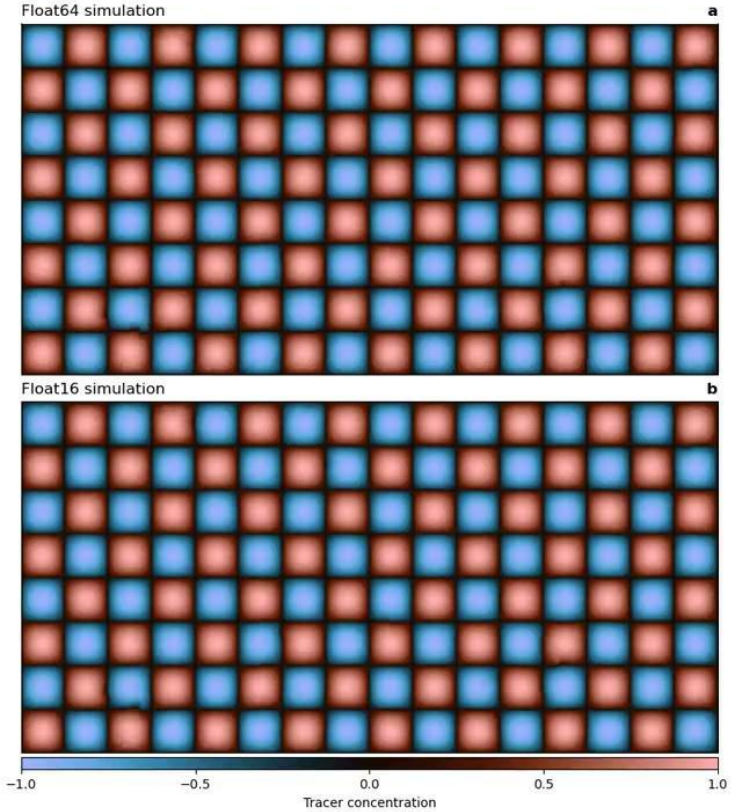


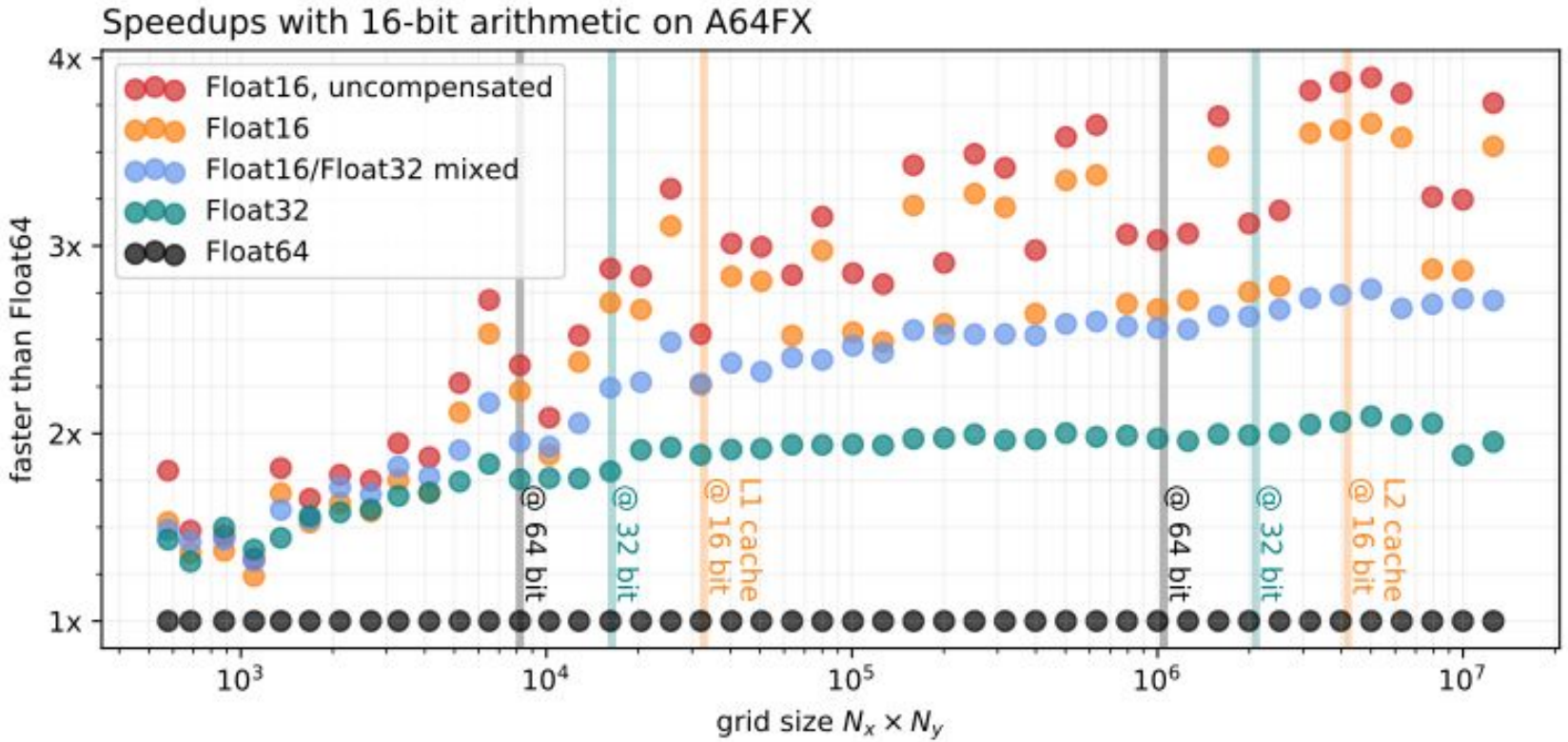
ShallowWaters.jl

- 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.

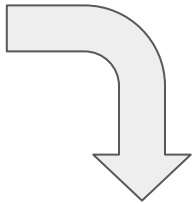
1. What to do on platforms with no/limited hardware support
2. Extended precision (thanks x87) rears it's ugly head

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
```

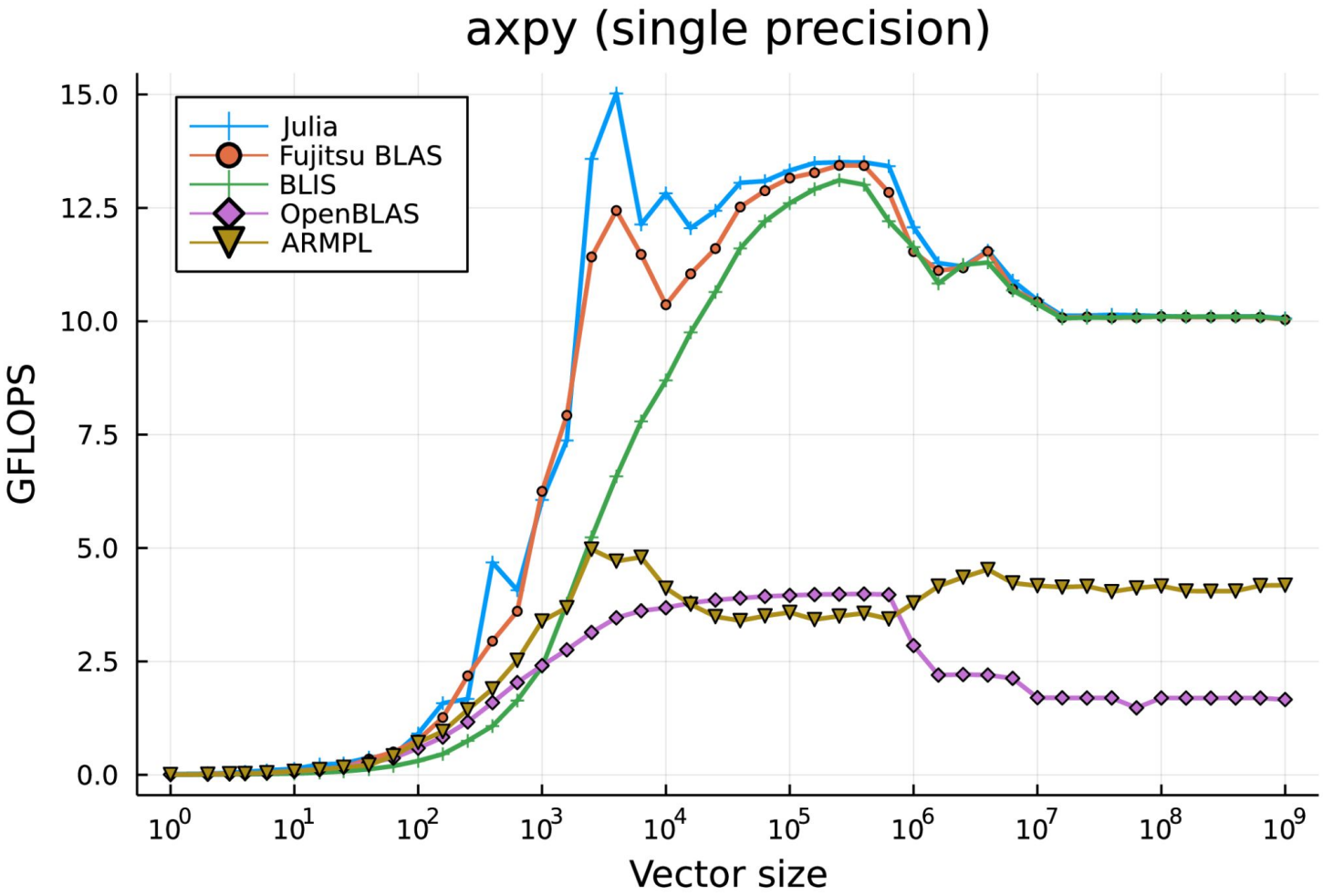
Performance and Scalability on Fugaku

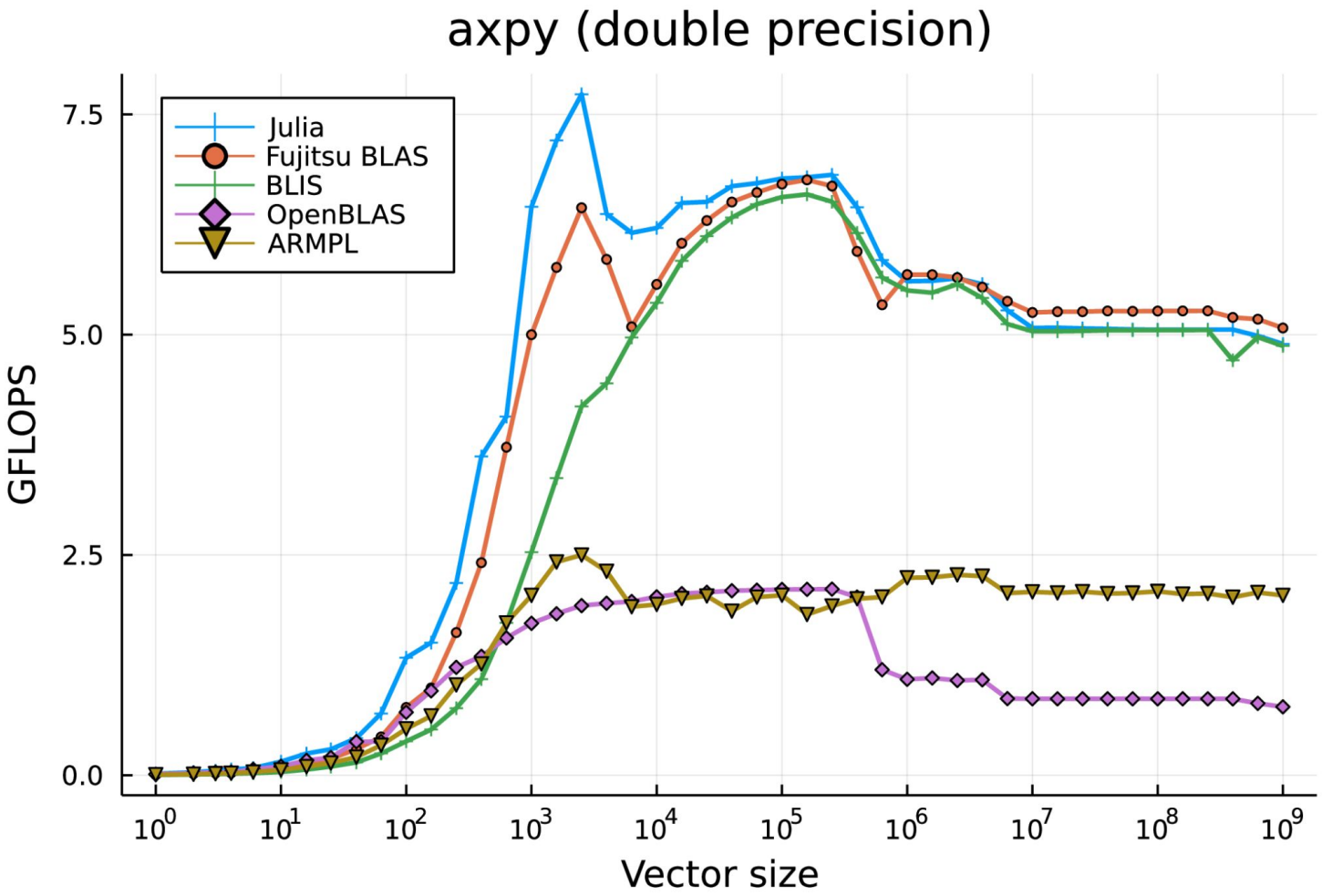
Level 1 BLAS showdown

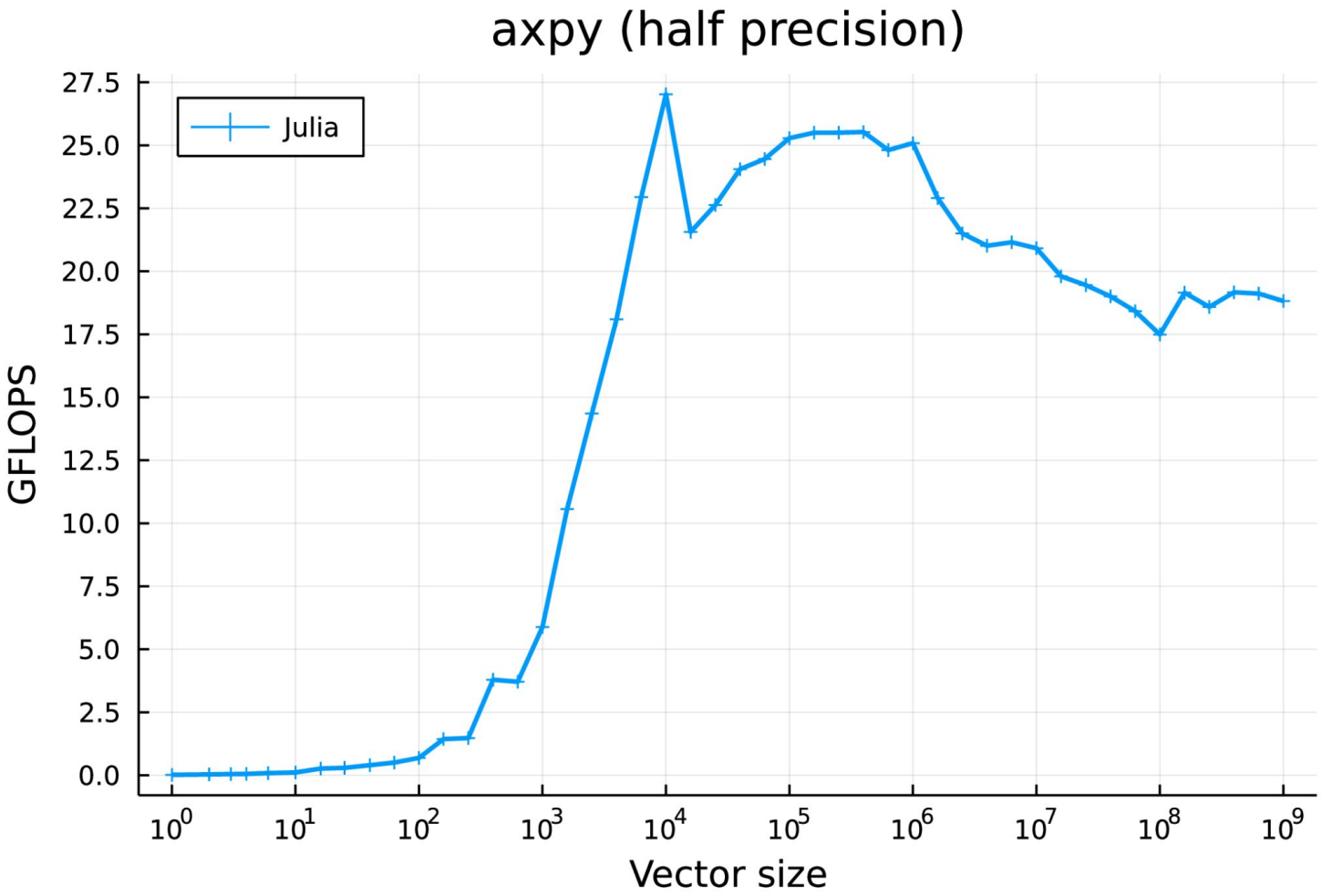
```
function axpy!(a, x, y)
    @simd for i in eachindex(x, y)
        @inbounds y[i] = muladd(a, x[i], y[i])
   end
   return y
end
```

vs

```
LinearAlgebra.BLAS.axpy!(a, x, y)
```


axpy (single precision)
Julia
Fujitsu BLAS
BLIS
OpenBLAS
ARMPL
GFLOPS
15.0
12.5
10.0
7.5
5.0
2.5
0.0
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9
Vector size


axpy (double precision)
Julia
Fujitsu BLAS
BLIS
OpenBLAS
ARMPL
GFLOPS
7.5
5.0
2.5
0.0
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9
Vector size

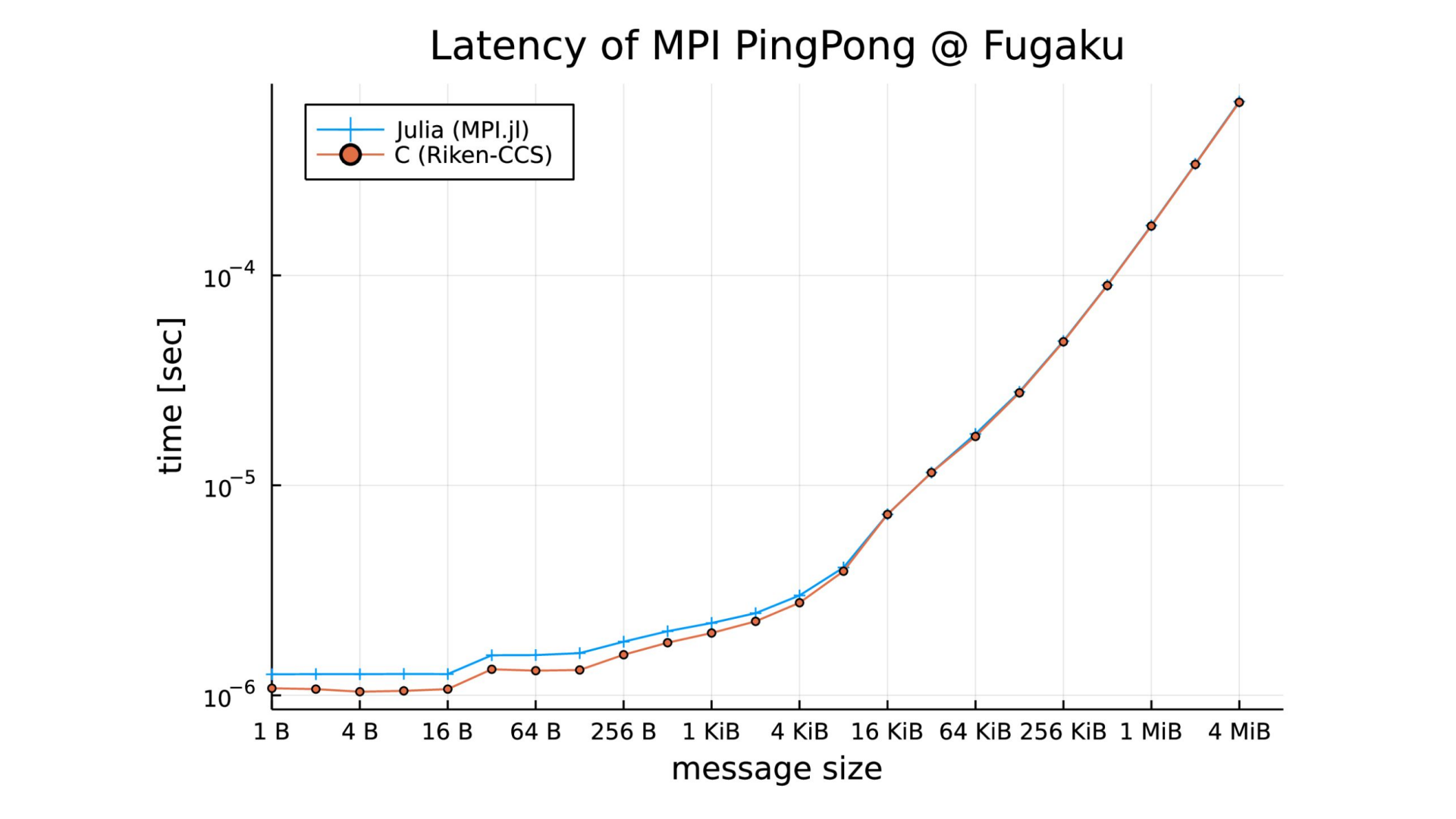

axpy (half precision)
Julia
GFLOPS
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9
Vector size

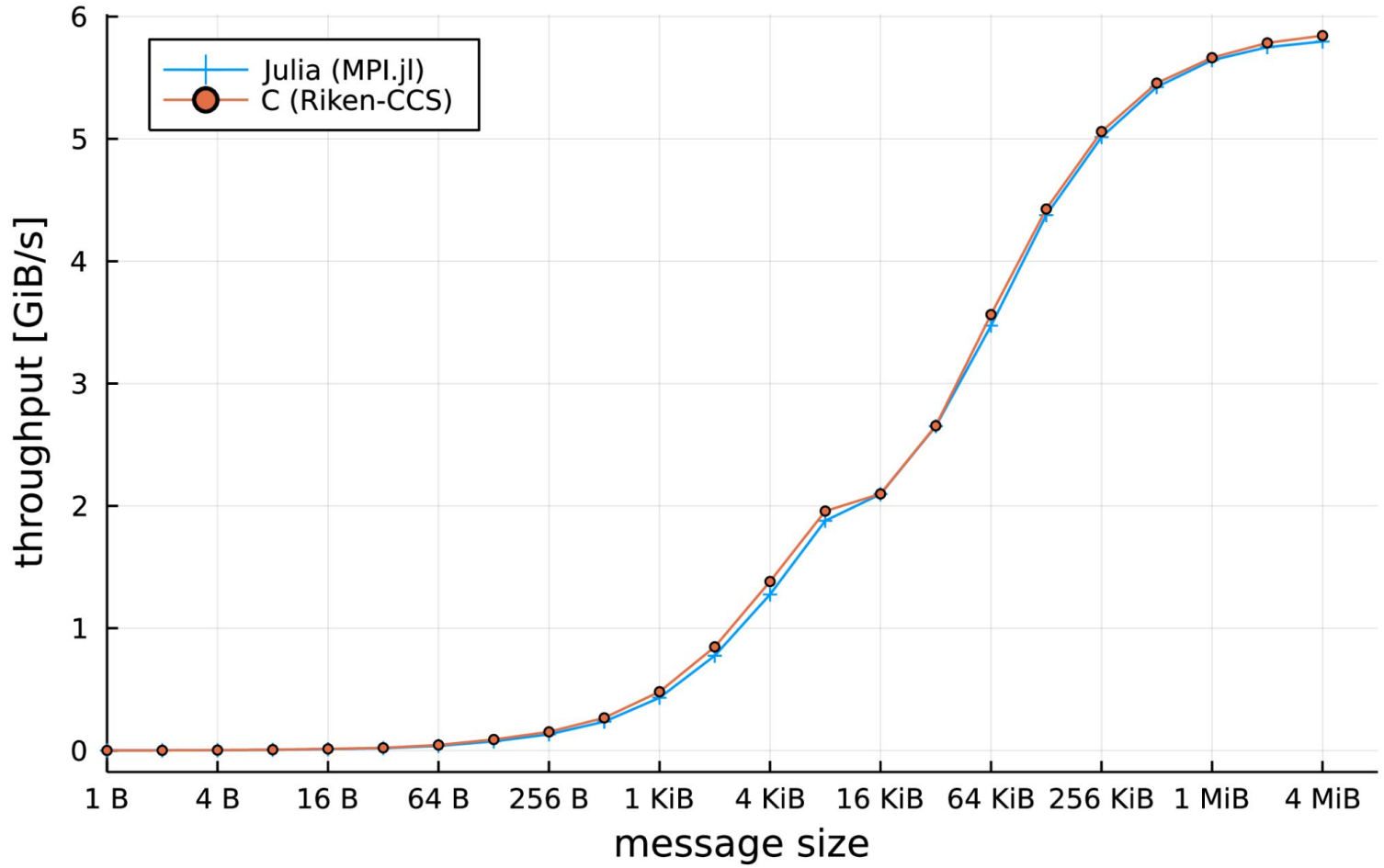
MPI.jl

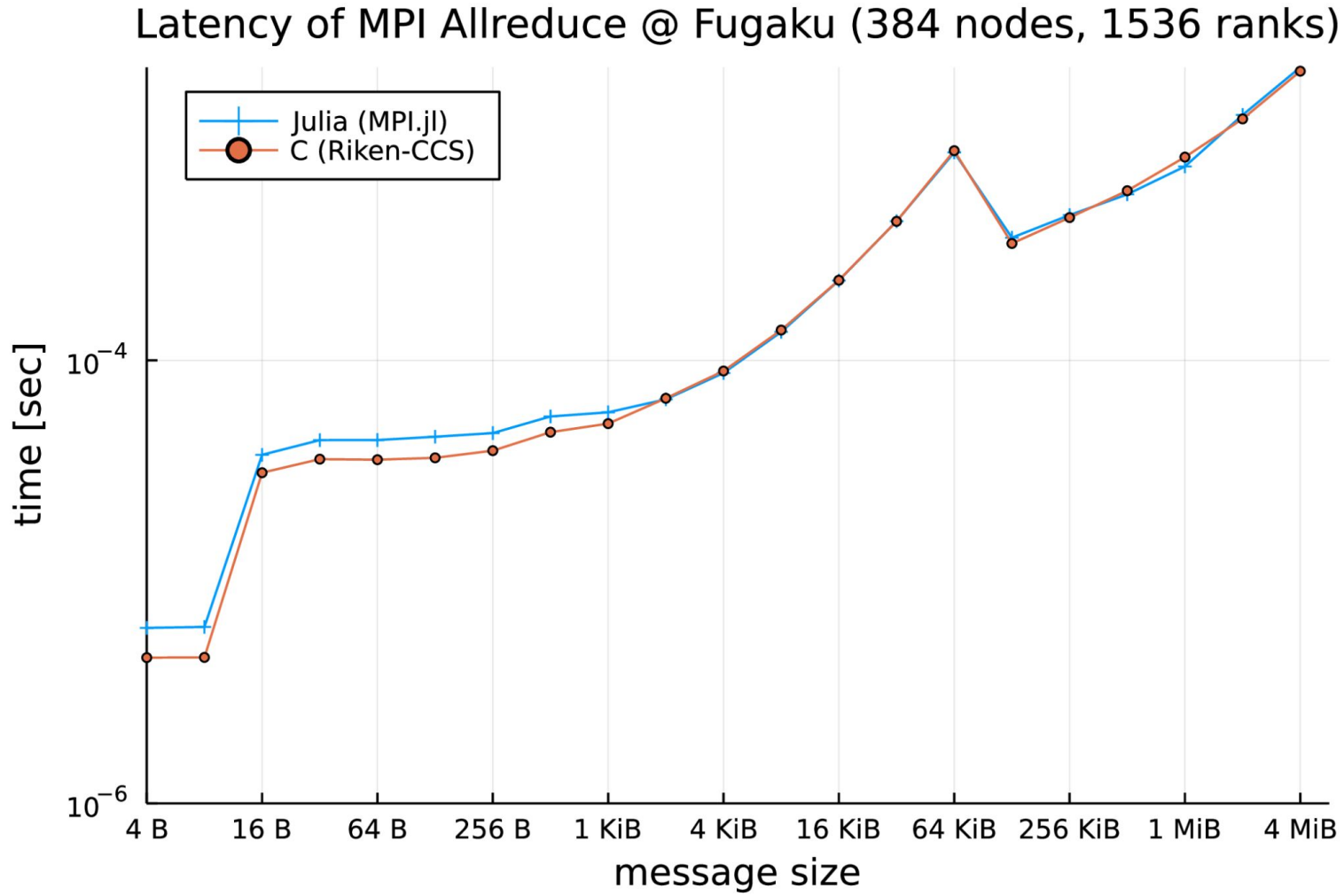
- Low-level access to MPI
- High-level convenience wrappers
- Deals with MPI ABI

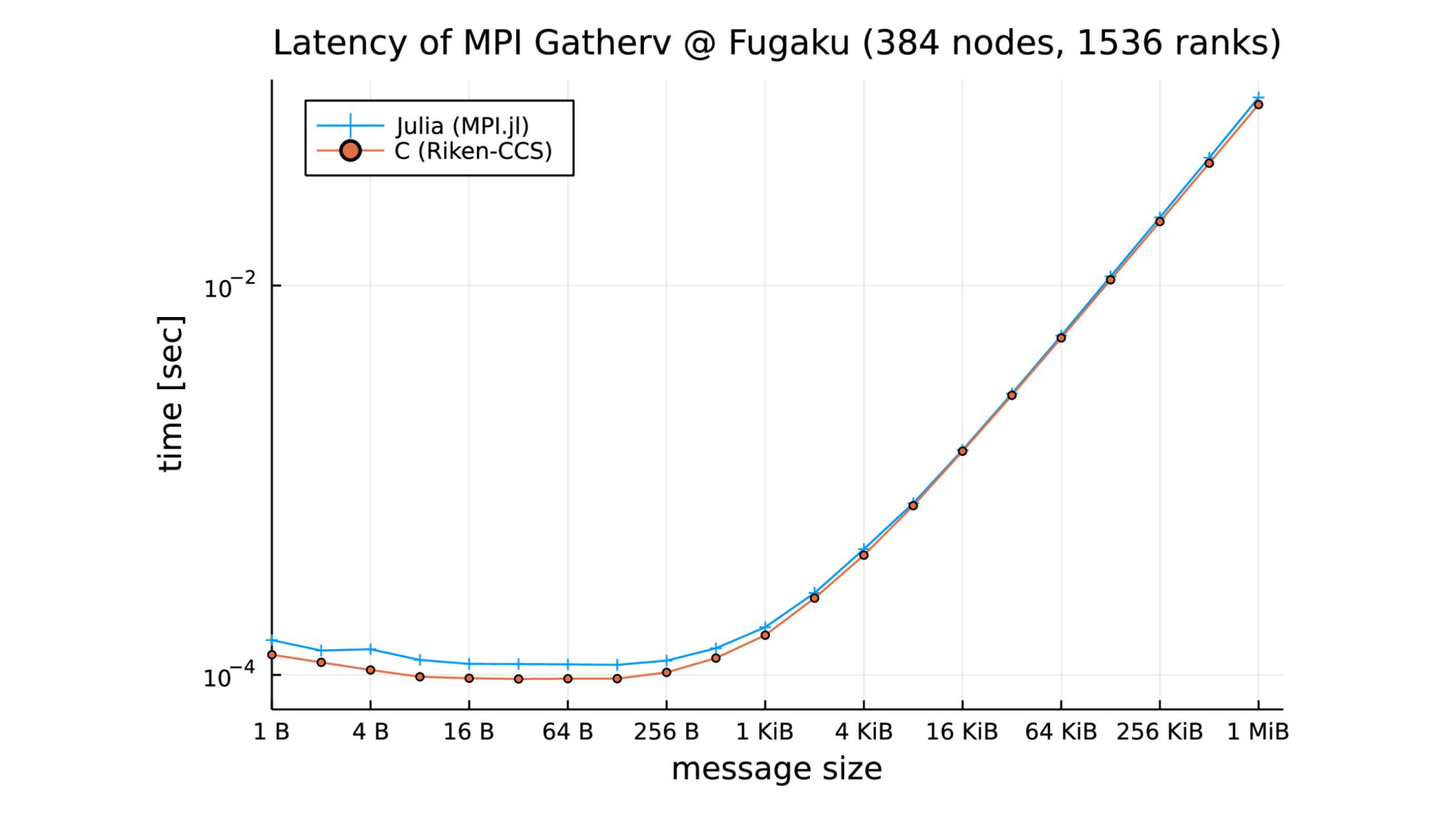
One of the oldest Julia packages (2012)

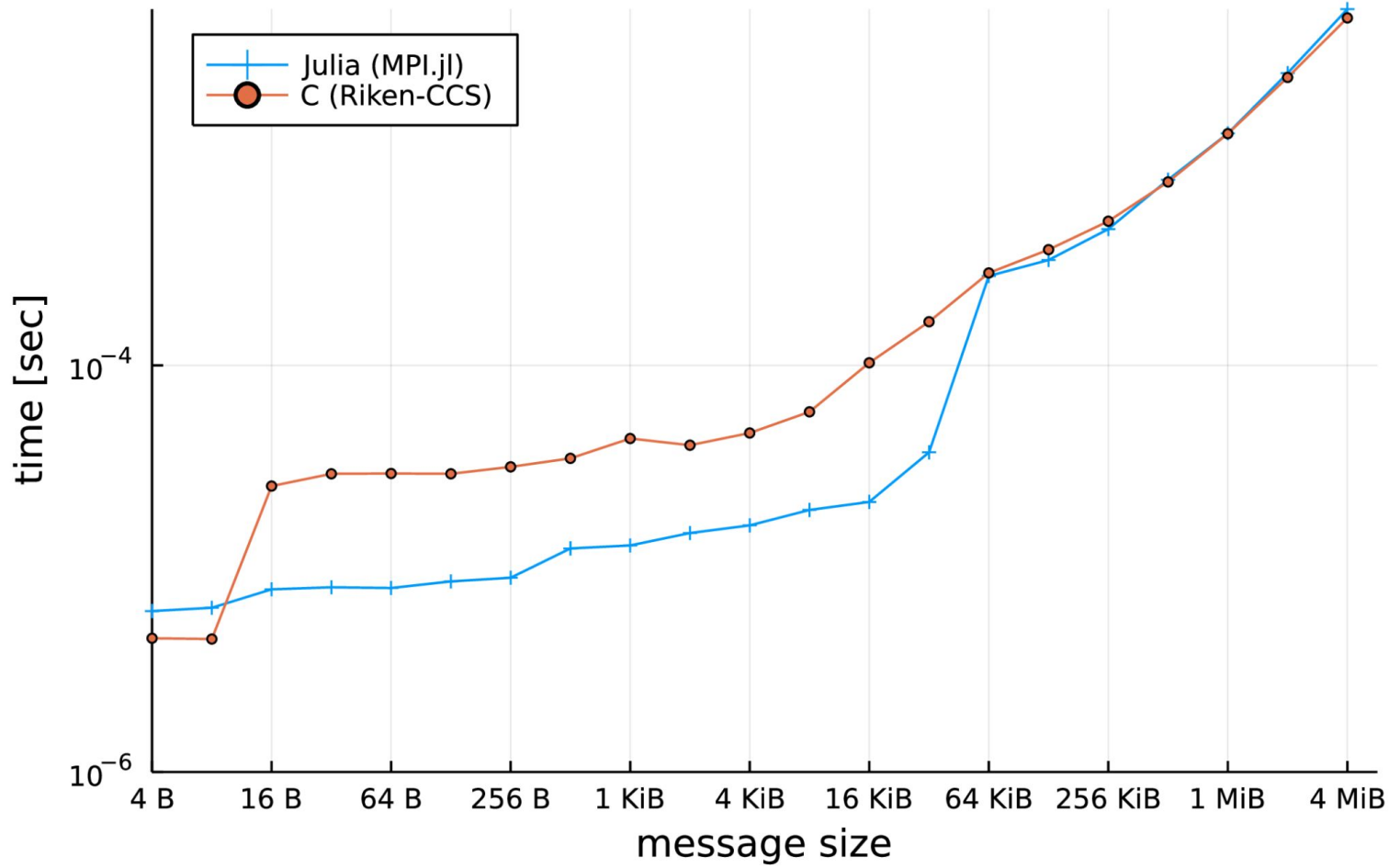
```
function pingpong(T::Type, bufsize::Int,
                  iters::Int, comm::MPI.Comm)
    rank = MPI.Comm_rank(comm)
    buffer = zeros(T, bufsize)
    tag = 0
    MPI.Barrier(comm)
    tic = MPI.Wtime()
    for i in 1:iters
        if iszero(rank)
            MPI.Send(buffer, comm; dest=1, tag)
            MPI.Recv!(buffer, comm; source=1, tag)
        elseif isone(rank)
            MPI.Recv!(buffer, comm; source=0, tag)
            MPI.Send(buffer, comm; dest=0, tag)
        end
    end
    toc = MPI.Wtime()
    return (toc - tic) / iters
end
```


Latency of MPI PingPong @ Fugaku
Julia (MPI.jl)
C (Riken-CCS)
time [sec]
10^{-4}
10^{-5}
10^{-6}
1 B
4 B
16 B
64 B
256 B
1 KiB
4 KiB
16 KiB
64 KiB
256 KiB
1 MiB
4 MiB
message size


Throughput of MPI PingPong @ Fugaku
Julia (MPI.jl)
C (Riken-CCS)
throughput [GiB/s]
0
1
2
3
4
5
6
1 B
4 B
16 B
64 B
256 B
1 KiB
4 KiB
16 KiB
64 KiB
256 KiB
1 MiB
4 MiB
message size


Latency of MPI Allreduce @ Fugaku (384 nodes, 1536 ranks)
Julia (MPI.jl)
C (Riken-CCS)
time [sec]
10^{-4}
10^{-6}
4 B
16 B
64 B
256 B
1 KiB
4 KiB
16 KiB
64 KiB
256 KiB
1 MiB
4 MiB
message size


Latency of MPI Gatherv @ Fugaku (384 nodes, 1536 ranks)
Julia (MPI.jl)
C (Riken-CCS)
time [sec]
10^{-2}
10^{-4}
1 B
4 B
16 B
64 B
256 B
1 KiB
4 KiB
16 KiB
64 KiB
256 KiB
1 MiB
message size


Latency of MPI Reduce @ Fugaku (384 nodes, 1536 ranks)
Julia (MPI.jl)
C (Riken-CCS)
time [sec]
10^{-4}
10^{-6}
4 B
16 B
64 B
256 B
1 KiB
4 KiB
16 KiB
64 KiB
256 KiB
1 MiB
4 MiB
message size

Opportunities for improvements

- Julia is not optimised for A64FX, but every version gets better thanks to upstream improvements in LLVM
  - Keep them coming!
- Compilation latency on A64FX hits particularly hard a JIT language
  - There are ongoing works to continue reducing compilation latency in Julia and to improve static compilation story
- Custom reductions don’t work in MPI.jl on non-Intel architectures
  - Can be fixed, but it needs someone to do the work
- Runtime detection of hardware support for Float16
  - Open pull request, recently tested also on AVX512-FP16