Julia: From multicore to GPU parallelism

presented by Steffen Haug

What is Julia?

(Briefly)

1. High level programming language

1. High level programming language
2. Dynamically typed

1. High level programming language
2. Dynamically typed
3. For HPC

1. High level programming language
2. Dynamically typed
3. For HPC
4. **Novel compilation infrastructure**
   (Central on CPU and GPU)

Untyped AST specialized at runtime

1. High level programming language
2. Dynamically typed
3. For HPC
4. **Novel compilation infrastructure**
   (Central on CPU and GPU)

Untyped AST specialized at runtime

"Looks like Python – runs like Fortran"

Example

```
function square!(xs)
  for i in eachindex(xs)
    xs[i] ^= 2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

Function that squares (mutates) a vector.

Example

```
function square!(xs)
  for i in eachindex(xs)
    xs[i] ^= 2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

Not a type in sight!

Example

```
function square!(xs)
  for i in eachindex(xs)
    xs[i] ^= 2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

When invoked with `V`, `square!` is specialized to **`Vector{Float64}`**.

Example

```
function square!(xs)
  for i in eachindex(xs)
    xs[i] ^= 2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

`@code_native` is a "macro" that allows us to inspect code generated for a particular specialization.

Example

```
function square!(xs)
  for i in eachindex(xs)
    xs[i] ^= 2
  end
end

V = rand(10000);
@code_native syntax=:intel square(V)
```

Simple function – surely the assembly code is reasonably simple? 

```
  .text
  .file  "square"
  .globl  julia_square_86
  .p2align  4, 0x90
  .type  julia_square_86,@function
julia_square_86:
; ┌ @ REPL[1]:1 within `square`
  .cfi_startproc
; │ @ REPL[1]:2 within `square`
; │┌ @ abstractarray.jl:285 within `eachindex`
; ││┌ @ abstractarray.jl:116 within `axes1`
; │││┌ @ abstractarray.jl:95 within `axes`
; ││││┌ @ array.jl:151 within `size`
  endbr64
  mov  rax, qword ptr [rdi + 8]
; │└└└└
; │┌ @ range.jl:879 within `iterate`
; ││┌ @ range.jl:654 within `isempty`
; │││┌ @ operators.jl:382 within `>`
; ││││┌ @ int.jl:83 within `<`
  test  rax, rax
; │└└└└
  je  .LBB0_7
  mov  rcx, qword ptr [rdi]
  mov  edx, 1
; │ @ REPL[1]:3 within `square`
; │┌ @ array.jl:924 within `getindex`
  cmp  rax, 16
  jb  .LBB0_5
  mov  rsi, rax
  and  rsi, -16
  lea  rdx, [rsi + 1]
  xor  edi, edi
  .p2align  4, 0x90
.LBB0_3:
  vmovups  xmm0, xmmword ptr [rcx + 8*rdi]
  vmovups  xmm1, xmmword ptr [rcx + 8*rdi + 32]
  vmovups  xmm2, xmmword ptr [rcx + 8*rdi + 64]
  vmovups  xmm3, xmmword ptr [rcx + 8*rdi + 96]
  vinsertf128  ymm0, ymm0, xmmword ptr [rcx + 8*rdi + 16], 1
  vinsertf128  ymm1, ymm1, xmmword ptr [rcx + 8*rdi + 48], 1
  vinsertf128  ymm2, ymm2, xmmword ptr [rcx + 8*rdi + 80], 1
  vinsertf128  ymm3, ymm3, xmmword ptr [rcx + 8*rdi + 112], 1
; │└
; │┌ @ intfuncs.jl:321 within `literal_pow`
; ││┌ @ float.jl:385 within `*`
  vmulpd  ymm0, ymm0, ymm0
  vmulpd  ymm1, ymm1, ymm1
  vmulpd  ymm2, ymm2, ymm2
  vmulpd  ymm3, ymm3, ymm3
; │└└
; │┌ @ array.jl:966 within `setindex!`
  vextractf128  xmmword ptr [rcx + 8*rdi + 16], ymm0, 1
  vmovupd  xmmword ptr [rcx + 8*rdi], xmm0
  vextractf128  xmmword ptr [rcx + 8*rdi + 48], ymm1, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 32], xmm1
  vextractf128  xmmword ptr [rcx + 8*rdi + 80], ymm2, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 64], xmm2
  vextractf128  xmmword ptr [rcx + 8*rdi + 112], ymm3, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 96], xmm3
  add  rdi, 16
  cmp  rsi, rdi
  jne  .LBB0_3
; │└
; │ @ REPL[1]:4 within `square`
  cmp  rax, rsi
  je  .LBB0_7
.LBB0_5:
  dec  rdx
  .p2align  4, 0x90
.LBB0_6:
; │ @ REPL[1]:3 within `square`
; │┌ @ array.jl:924 within `getindex`
  vmovsd  xmm0, qword ptr [rcx + 8*rdx]
; │└
; │┌ @ intfuncs.jl:321 within `literal_pow`
; ││┌ @ float.jl:385 within `*`
  vmulsd  xmm0, xmm0, xmm0
; │└└
; │┌ @ array.jl:966 within `setindex!`
  vmovsd  qword ptr [rcx + 8*rdx], xmm0
; │└
; │ @ REPL[1]:4 within `square`
  inc  rdx
  cmp  rdx, rax
  jb  .LBB0_6
.LBB0_7:                                # %L33
  vzeroupper
  ret
.Lfunc_end0:
  .size  julia_square_86, .Lfunc_end0-julia_square_86
  .cfi_endproc
  .section  ".note.GNU-stack","",@progbits
```

?

```
square:
  mov  rax, qword ptr [rdi + 8]
  test rax, rax
  je  .LBB0_7
  mov  rcx, qword ptr [rdi]
  mov  edx, 1
  cmp  rax, 16
  jb  .LBB0_5
  mov  rsi, rax
  and  rsi, -16
  lea  rdx, [rsi + 1]
  xor  edi, edi
.LBB0_3:
  vmovups  xmm0, xmmword ptr [rcx + 8*rdi]
  vmovups  xmm1, xmmword ptr [rcx + 8*rdi + 32]
  vmovups  xmm2, xmmword ptr [rcx + 8*rdi + 64]
  vmovups  xmm3, xmmword ptr [rcx + 8*rdi + 96]
  vinsertf128  ymm0, ymm0, xmmword ptr [rcx + 8*rdi + 16], 1
  vinsertf128  ymm1, ymm1, xmmword ptr [rcx + 8*rdi + 48], 1
  vinsertf128  ymm2, ymm2, xmmword ptr [rcx + 8*rdi + 80], 1
  vinsertf128  ymm3, ymm3, xmmword ptr [rcx + 8*rdi + 112], 1
  vmulpd  ymm0, ymm0, ymm0
  vmulpd  ymm1, ymm1, ymm1
  vmulpd  ymm2, ymm2, ymm2
  vmulpd  ymm3, ymm3, ymm3
  vextractf128  xmmword ptr [rcx + 8*rdi + 16], ymm0, 1
  vmovupd  xmmword ptr [rcx + 8*rdi], xmm0
  vextractf128  xmmword ptr [rcx + 8*rdi + 48], ymm1, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 32], xmm1
  vextractf128  xmmword ptr [rcx + 8*rdi + 80], ymm2, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 64], xmm2
  vextractf128  xmmword ptr [rcx + 8*rdi + 112], ymm3, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 96], xmm3
  add  rdi, 16
  cmp  rsi, rdi
  jne  .LBB0_3
  cmp  rax, rsi
  je  .LBB0_7
.LBB0_5:
  dec  rdx
.LBB0_6:
  vmovsd  xmm0, qword ptr [rcx + 8*rdx]
  vmulsd  xmm0, xmm0, xmm0
  vmovsd  qword ptr [rcx + 8*rdx], xmm0
  inc  rdx
  cmp  rdx, rax
  jb  .LBB0_6
.LBB0_7:
  vzeroupper
  ret
```

🤨

Remove some comments...

```
.LBB0_3:
  vmovups  xmm0, xmmword ptr [rcx + 8*rdi]
  vmovups  xmm1, xmmword ptr [rcx + 8*rdi + 32]
  vmovups  xmm2, xmmword ptr [rcx + 8*rdi + 64]
  vmovups  xmm3, xmmword ptr [rcx + 8*rdi + 96]
  vinsertf128  ymm0, ymm0, xmmword ptr [rcx + 8*rdi + 16], 1
  vinsertf128  ymm1, ymm1, xmmword ptr [rcx + 8*rdi + 48], 1
  vinsertf128  ymm2, ymm2, xmmword ptr [rcx + 8*rdi + 80], 1
  vinsertf128  ymm3, ymm3, xmmword ptr [rcx + 8*rdi + 112], 1
  vmulpd  ymm0, ymm0, ymm0
  vmulpd  ymm1, ymm1, ymm1
  vmulpd  ymm2, ymm2, ymm2
  vmulpd  ymm3, ymm3, ymm3
  vextractf128  xmmword ptr [rcx + 8*rdi + 16], ymm0, 1
  vmovupd  xmmword ptr [rcx + 8*rdi], xmm0
  vextractf128  xmmword ptr [rcx + 8*rdi + 48], ymm1, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 32], xmm1
  vextractf128  xmmword ptr [rcx + 8*rdi + 80], ymm2, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 64], xmm2
  vextractf128  xmmword ptr [rcx + 8*rdi + 112], ymm3, 1
  vmovupd  xmmword ptr [rcx + 8*rdi + 96], xmm3
  add  rdi, 16
  cmp  rsi, rdi
  jne  .LBB0_3
```

Julia has unrolled the loop and used SIMD-instructions!

```c
void square(double *xs, int N) {
  for (int i = 0; i < N; i++) {
    xs[i] *= xs[i];
  }
}
```

```asm
.L1:
  vmovupd ymm0,YMMWORD PTR [rdi+rsi*1]
  vmovupd ymm1,YMMWORD PTR [rdi+rsi*1+0x20]
  vmovupd ymm2,YMMWORD PTR [rdi+rsi*1+0x40]
  vmovupd ymm3,YMMWORD PTR [rdi+rsi*1+0x60]
  vmulpd ymm0,ymm0,ymm0
  vmulpd ymm1,ymm1,ymm1
  vmulpd ymm2,ymm2,ymm2
  vmulpd ymm3,ymm3,ymm3
  vmovupd YMMWORD PTR [rdi+rsi*1],ymm0
  vmovupd YMMWORD PTR [rdi+rsi*1+0x20],ymm1
  vmovupd YMMWORD PTR [rdi+rsi*1+0x40],ymm2
  vmovupd YMMWORD PTR [rdi+rsi*1+0x60],ymm3
  sub    rsi,0xffffffffffffff80
  cmp    rdx,rsi
  jne    .L1
```

This is what Clang does.

(*Slightly* better due to Julia-arrays being strided)

Julia uses LLVM

Julia can do the same optimizations we expect in C/C++

Julia uses LLVM

Julia can do the same optimizations we expect in C/C++

...because it is literally the same optimizer!

But there are some huge caveats

1. There is some *slight* compilation overhead at startup

However: As problem size grows, relative compilation time decreases.

But there are some <u>huge</u> caveats

1. There is some *slight* compilation overhead at startup
2. Julia is *garbage-collected* 🤮

But there are some huge caveats

1. There is some *slight* compilation overhead at startup
2. Julia is *garbage-collected* 🤮
3. We have no explicit control over memory
   (Allocation, copying, ...)

Accidental heap operations can really hurt!

But there are some <u>huge</u> caveats

1. There is some *slight* compilation overhead at startup
2. Julia is *garbage-collected* 🤮
3. We have no explicit control over memory
   (Allocation, copying, ...)
4. *The whole thing breaks* if Julia can't figure out the types

There are many ways to shoot yourself in the foot! 🙂

```julia
Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

```julia
const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

Two seemingly quite similar programs...

```julia
Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.010550 seconds
(499.49 k allocations)

```julia
const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.000175 seconds

One is $\sim 60\times$ faster than the other!

```julia
Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.010550 seconds
(499.49 k allocations)

```julia
const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.000175 seconds

And 500K allocations? 🤯

```julia
Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.010550 seconds
(499.49 k allocations)

```julia
const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.000175 seconds

And 500K allocations? 🤯

`Δx` could be reassigned $\implies$ Julia can't assume its type!

```julia
Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.010550 seconds
(499.49 k allocations)

```julia
const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.000175 seconds

And 500K allocations? 🤯

Leads to runtime type checks and dynamic dispatching...

```julia
Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.010550 seconds
(499.49 k allocations)

```julia
const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.000175 seconds

And 500K allocations? 🤯

...and forces intermediate calculations onto the heap!

```julia
Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.010550 seconds
(499.49 k allocations)

```julia
const Δx = 0.1

function riemann!(X, x)
    N = length(x)
    X[0] = 0.0
    for i in 2:N
        X[i] = X[i - 1] + Δx * X[i]
    end
end

v = rand(100000)
V = zeros(100000)
@time riemann!(V, v)
```

0.000175 seconds

And 500K allocations? 🤯

...and forces intermediate calculations onto the heap!

Julia *can* generate really fast code...

Julia *can* generate really fast code...

But the *lack of explicit control* requires a lot of attention.

Julia *can* generate really fast code...

But the *lack of explicit control* requires a lot of attention.

Even more so when we parallelize our programs.

Multicore

```
using .Threads: @threads
```

```julia
for i in eachindex(xs)

end
```

$\longrightarrow$

```julia
@threads for i in eachindex(xs)

end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

```julia
for i in eachindex(xs)

end
```

⟶

```julia
@threads for i in eachindex(xs)

end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

The normal caveats apply

1. Spawning threads involve some overhead

```julia
for i in eachindex(xs)

end
```

$\longrightarrow$

```julia
@threads for i in eachindex(xs)

end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

The normal caveats apply

1. Spawning threads involve some overhead
2. Memory is shared (Race conditions, deadlocks)

```julia
for i in eachindex(xs)

end
```

$\longrightarrow$

```julia
@threads for i in eachindex(xs)

end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

The normal caveats apply

1. Spawning threads involve some overhead
2. Memory is shared (Race conditions, deadlocks)
3. Possible speedup severely limited by core count

```julia
for i in eachindex(xs)

end
```

$\longrightarrow$

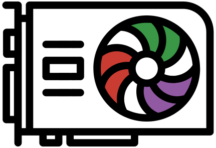
```julia
@threads for i in eachindex(xs)

end
```

Similar API to OpenMP: *Just annotate loops with a macro.*

The normal caveats apply

1. Spawning threads involve some overhead
2. Memory is shared (Race conditions, deadlocks)
3. Possible speedup severely limited by core count
4. Domain decomposition sometimes necessary to get speedup on simple problems

GPU Parallelism



Level 1: “Vectorized” Array API

(Vectorized in the sense that you only operate on arrays, Numpy style)

“Vectorized” Array API

```
# In-place operations.
v = CUDA.rand(10000000)
v *= 10.0
v ^= 2
v .= sqrt.(v)

# Parallel scan.
w = CUDA.zeros(10000000)
accumulate!(max, w, v)
```

`v`, `w` of type `CuArray` that manages a device-side buffer.

“Vectorized” Array API

```
# In-place operations.
v = CUDA.rand(10000000)
v *= 10.0
v ^= 2
v .= sqrt.(v)

# Parallel scan.
w = CUDA.zeros(10000000)
accumulate!(max, w, v)
```

Block- and gridsizes automatically determined.

“Vectorized” Array API

```julia
# FFT (10M-point FFT takes 200ms on 980)
using CUDA.CUFFT
ω = fft(v)

# Kernels get fused
w •= w.^2 .+ v

# Check out what happens on the GPU
@device_code_sass w •= w.^2 .+ v
```

Bindings to CUDA libraries like cuFFT and cuBLAS
(Also cuSPARSE, cuSOLVER, ...)

"Vectorized" Array API

```
# FFT (10M-point FFT takes 200ms on 980)
using CUDA.CUFFT
ω = fft(v)

# Kernels get fused
w •= w.^2 .+ v

# Check out what happens on the GPU
@device_code_sass w •= w.^2 .+ v
```

GPU code generated on the fly using LLVMs PTX/SASS backend...
(Same sort of infrastructure as specializing Julia functions)

“Vectorized” Array API

```
# FFT (10M-point FFT takes 200ms on 980)
using CUDA.CUFFT
ω = fft(v)

# Kernels get fused
w .= w.^2 .+ v

# Check out what happens on the GPU
@device_code_sass w .= w.^2 .+ v
```

Enables kernel fusion and global optimization (!)

"Vectorized" Array API

```
# FFT (10M-point FFT takes 200ms on 980)
using CUDA.CUFFT
ω = fft(v)

# Kernels get fused
w .= w.^2 .+ v

# Check out what happens on the GPU
@device_code_sass w .= w.^2 .+ v
```

Like with CPU code, you can inspect the generated GPU code.
(SASS/PTX code is too long/ugly to put on a slide)

Level 2: Julia on the GPU

Julia on the GPU

```
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil(Int, length(v) / Nt)
@cuda threads=Nt blocks=Nb cusquare!(v)
```

`cusquare!` is defined *as a regular Julia function*!

(It won't run if you call it; block- and thread-ID is not defined on the CPU)

Julia on the GPU

```julia
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil(Int, length(v) / Nt)
@cuda threads=Nt blocks=Nb cusquare!(v)
```

The `@cuda` macro transforms typed Julia IR into GPU code! 🤯

Julia on the GPU

```julia
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil(Int, length(v) / Nt)
@cuda threads=Nt blocks=Nb cusquare!(v)
```

Basically, almost *any* sensible Julia can just work on the GPU

Julia on the GPU

```julia
function cusquare!(xs)
    i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    if i < length(xs)
        xs[i] *= xs[i]
    end
    return nothing
end

Nt = 1024
Nb = ceil(Int, length(v) / Nt)
@cuda threads=Nt blocks=Nb cusquare!(v)
```

However, there is no escape hatch: *The code must type check, and can't allocate*

Pitfalls

Pitfalls

`CuArray` controls all memory transfer

Pitfalls

Accidentally reading back to the CPU is really painful!

Level 3: Multi-GPU / Multi-node

Future Outlook

Level 3: Multi-GPU / Multi-node

`CUDA.jl` composes with `DistributedArrays.jl`

`DistributedArrays.jl` : Domain decomposition and distribution algorithms

`CUDA.jl` : Manages device-side buffers and host-device communication

Both understands Julias abstract concept of
an "Array", so they work together. *

* Tim Besard et al. "Rapid software prototyping for heterogeneous and distributed platforms". In: *Advances in Engineering Software* (2019). URL: https://www.sciencedirect.com/science/article/pii/S0965997818310123, (Section 6.3.2. Distributed GPU Arrays).

IN SUMMARY

Julia makes bold claims

IN SUMMARY

Julia makes bold claims: *"Looks like Python – runs like* C/C++/Fortran"

IN SUMMARY

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

IN SUMMARY

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

It is a truth with a big asterisk.

IN SUMMARY

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

If there is *one thing* that is true, it is that you can't escape

Location, location, location

IN SUMMARY

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

If there is *one thing* that is true, it is that you can't escape

Location, location, location

Julia has *real* potential, but it is *not a free lunch!*
(Especially on the GPU)

IN SUMMARY

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

If there is *one thing* that is true, it is that you can't escape

Location, location, location

I spend a lot of time "fighting the abstraction"

IN SUMMARY

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

If there is *one thing* that is true, it is that you can't escape

LOCATION, LOCATION, LOCATION

Allocation, garbage-collection, copies, CUDA device$\leftrightarrow$host transfers, etc. are all *implicit!*

In summary

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

If there is *one thing* that is true, it is that you can't escape

Location, location, location

Allocation, garbage-collection, copies, CUDA device$\leftrightarrow$host transfers, etc.
are all *implicit!*

As Julia takes away your control, you have to be careful.

In summary

Julia makes bold claims: *"Only dynamic language to achieve petascale"*

If there is *one thing* that is true, it is that you can't escape

Location, location, location

Allocation, garbage-collection, copies, CUDA device↔host transfers, etc.
are all *implicit!*

The abstraction leaks!

IN SUMMARY

What niche does Julia even fill?

IN SUMMARY

What niche does Julia even fill?

If you *don't understand* the compilation process, Julia is basically unusable.

IN SUMMARY

What niche does Julia even fill?

Julia wont't replace lower-level languages.

IN SUMMARY

That being said...

IN SUMMARY

I quite like Julia

IN SUMMARY

What niche does Julia even fill?

IN SUMMARY

What niche does Julia even fill?

Julia makes HPC a *first-class citizen* in the "MATLAB workflow"

IN SUMMARY

What niche does Julia even fill?

Julia makes HPC a *first-class citizen* in the "MATLAB workflow"

1. Interactive REPL/Notebook-based workflow
2. Good for *fucking around* with data
3. Sensible package manager
4. "Math stuff" is easily accessible
5. **Visualization is made easy**

julia

julialang.org

Besard, Tim et al. "Rapid software prototyping for heterogeneous and distributed platforms". In: *Advances in Engineering Software* (2019). URL: https://www.sciencedirect.com/science/article/pii/S0965997818310123.