
Jeff Hammond https://c.im/@jeffscience
@science_dot
Replying to @science_dot, @miguelraz_ and @JuliaLanguage
Julia is of course great because it's basically Fortran for people who are too lazy to declare types and has an interpreter.
1:41 pm · 22 Feb 2023 · 1,257 Views
1 Quote Tweet 22 Likes

Why I think Julia is THE Language for Scientific Computing

A language should not “force you to rethink the way you solve problems”

Not everything needs to be a f-ing for loop!

Don’t keep secrets from your compiler

OpenMP version:

```
// Begin the target region for GPU execution
#pragma omp target teams distribute parallel for \
    reduction(+:d_sum) \
    map(to: d_arr[0:N])
for (int i = 0; i < N; i++) {
    d_sum += d_arr[i] * d_arr[i];
}
```

Julia version:

```
d_squared = map(x -> x^2, d_arr)
d_sum = reduce(+, d_squared)
```

Pipeliend Julia version:

```
d_sum = map(x -> x^2, d_arr) |> reduce(+)
```

Why I think Julia is THE Language for Scientific Computing

A language should not “force you to rethink the way you solve problems”

There is no “Julian” way of programming

Not everything needs to be a f-ing for loop!

Julia natively contains structures, and concepts for parallel and pipelined work

Don’t keep secrets from your compiler

Multiple dispatch, JIT, LLVM, and introspection tools make Julia a powerful “LLVM frontend”

OpenMP version:

```
// Begin the target region for GPU execution
#pragma omp target teams distribute parallel for \
    reduction(+:d_sum) \
    map(to: d_arr[0:N])
for (int i = 0; i < N; i++) {
    d_sum += d_arr[i] * d_arr[i];
}
```

Julia version:

```
d_squared = map(x -> x^2, d_arr)
d_sum = reduce(+, d_squared)
```

Pipelined Julia version:

```
d_sum = map(x -> x^2, d_arr) |> reduce(+)
```

A Template System that Allows you to Focus on Science

Rapid software prototyping for heterogeneous and distributed platforms
*Besard T., **Churavy V.**, Edelman A., De Sutter B.*
([doi:10.1016/j.advengsoft.2019.02.002](https://doi.org/10.1016/j.advengsoft.2019.02.002))

```
using LinearAlgebra

loss(w,b,x,y) = sum(abs2, y - (w*x .+ b)) / size(y,2)
loss∇w(w, b, x, y) = ...
lossdb(w, b, x, y) = ...
```

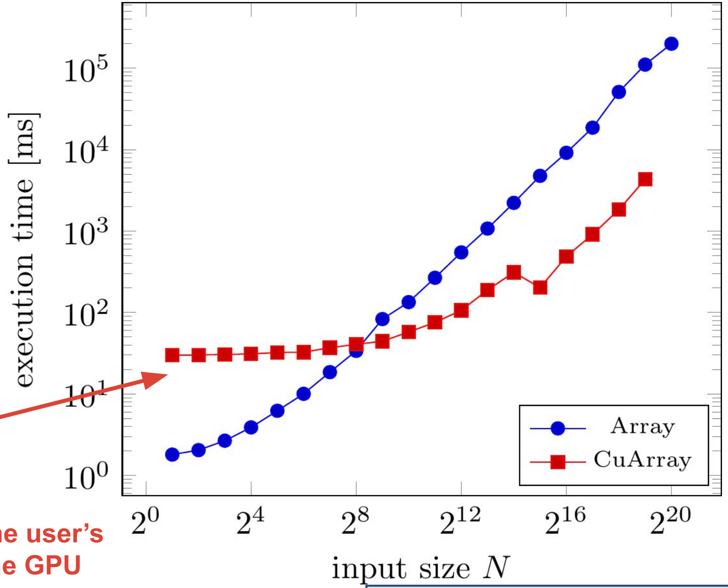
These don't have to be array-ish functions (e.g. Numpy), they can contain if, for, etc statements.

```
function train(w, b, x, y ; lr=.1)
    w -= lmul!(lr, loss∇w(w, b, x, y))
    b -= lr * lossdb(w, b, x, y)
    return w, b
end

n = 100; p = 10
x = randn(n,p)'
y = sum(x[1:5,:]; dims=1) .+ randn(n)'*0.1
w = 0.0001*randn(1,p)
b = 0.0

for i=1:50
    w, b = train(w, b, x, y)
end
```

```
x = CuArray(x)
y = CuArray(y)
w = CuArray(w)
```



By moving your data to the GPU, the user's of the algorithm "Just works" on the GPU

Slide shamelessly nicked from: Valentin Churavy

A Template System that Allows you to Focus on Science

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

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

```
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
```


Slide shamelessly nicked from: Valentin Churavy

```julia
using CUDA
function gemm!(A,B,C)
    row = (blockIdx().x - 1) * blockDim().x + threadIdx().x
    col = (blockIdx().y - 1) * blockDim().y + threadIdx().y

    sum = zero(eltype(C))
```

```julia
using Metal
function gemm!(A,B,C)
    row, col = thread_position_in_grid_2d()

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
```

```julia
using AMDGPU
function gemm!(A,B,C)
    row = (workgroupIdx().x - 1) * workgroupDim().x + workitemIdx().x
    col = (workgroupIdx().y - 1) * workgroupDim().y + workitemIdx().y

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
        for i = 1:size(A, 2)
            @inbounds sum += A[row, i] * B[i, col]
```

```julia
using oneAPI
function gemm!(A,B,C)
    row = get_global_id(0)
    col = get_global_id(1)

    sum = zero(eltype(C))

    if row <= size(A, 1) && col <= size(B, 2)
        for i = 1:size(A, 2)
            @inbounds sum += A[row, i] * B[i, col]
        end
        @inbounds C[row, col] = sum
    end

    return
end
```

- Leverage Julia's Toolchain for low-level portable code!
- Julia's xPU LLVM-backend also can generate Kernel code using `CUDA.jl`, `AMDGPU.jl`, or `oneAPI.jl`
- `KernelAbstractions.jl` wraps all of these up into a single `@kernel` macro

```julia
using KernelAbstractions
@kernel function gemm!(A, B, C)
    row, col = @index(Global, NTuple)

    sum = zero(eltype(C))
    for i = 1:size(A, 2)
        @inbounds sum += A[row, i] * B[i, col]
    end
    @inbounds C[row, col] = sum
end
```

BERKELEY LAB
Bringing Science Solutions to the World

Slide shamelessly adapted from: Valentin Churavy

Bonus slides

Beware of the Glue-Code Pattern

- Some languages are not ABI-compatible
  - Calling from one language to another will incur a cost due to type conversion
- This will can get very expensive if calls come from the “inner loop” (eg. AI inference call for every grid cell)

Function signature	Pybind11		ccall		speedup
`int fn0()`	132	±14.9	2.34	±1.24	56×
`int fn1(int)`	217	±20.9	2.35	±1.33	92×
`double fn2(int, double)`	232	±11.7	2.32	±0.189	100×
`char* fn3(int, double, char*)`	267	±28.9	6.27	±0.396	42×

all times in ns

Introspection and staged metaprogramming

String macros	**Code**	`@edit` `@which`
Macros	**AST**	`@code_lowered`

Generated functions
Cassette.jl passes
	Typed IR	`@code_warntype`
		`@code_typed optimize=false`
		`@code_typed`

llvmcall LLVM.jl	**LLVM IR**	`@code_llvm optimize=false` `@code_llvm`
LLVM.jl `@asmcall`	**Assembly**	`@code_native`

Julia my favorite LLVM frontend

Slide shamelessly nicked from: Valentin Churavy

Julia is basically a REPL for LLVM

Julia provides interfaces to the LLVM backend.

Eg.:

- `loopinfo`
- `llvmcall`

```
[16]: macro unroll(expr)
          expr = loopinfo("@unroll", expr, (Symbol("llvm.loop.unroll.full"),))
          return esc(expr)
      end

      for (jlf, f) in zip((:+, :*, :-), (:add, :mul, :sub))
          for (T, llvmT) in ((:Float32, "float"), (:Float64, "double"))
              ir = """
                  %x = f$f contract nsz $llvmT %0, %1
                  ret $llvmT %x
              """
              @eval begin
                  # the @pure is necessary so that we can constant propagate.
                  @inline Base.@pure function $jlf(a::$T, b::$T)
                      Base.llvmcall($ir, $T, Tuple{$T, $T}, a, b)
                  end
              end
          end
          @eval function $jlf(args...)
              Base.$jlf(args...)
          end
      end
```

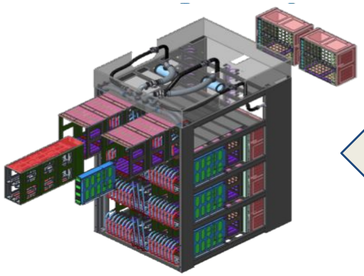
No Annoying "Paradigm Shifts"

`pip install …`

Rapid scripting

C/C++

Python



Workflows

No “Nibbling Around The Edges”

Function signature	Pybind11		ccall		speedup
`int fn0()`	132	±14.9	2.34	±1.24	56×
`int fn1(int)`	217	±20.9	2.35	±1.33	92×
`double fn2(int, double)`	232	±11.7	2.32	±0.189	100×
`char* fn3(int, double, char*)`	267	±28.9	6.27	±0.396	42×

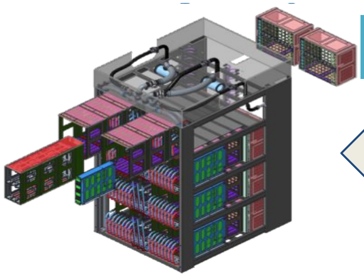
`pip install` …
Rapid scripting

C/C++

Python

Julia

Multiple Dispatch, “Designed for” Scientific Compute



Rust

Workflows

Immutability + pure functions, safe pointers, `cargo`