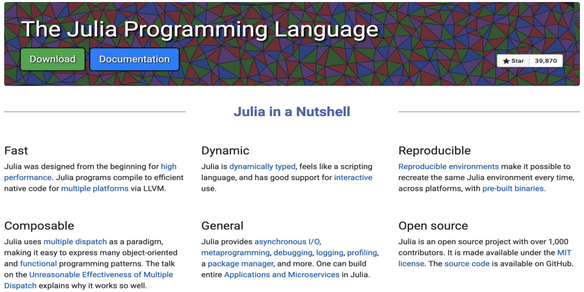
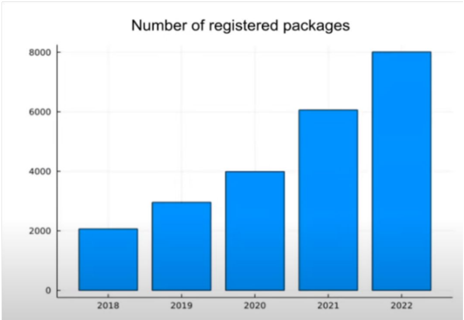
Introduction to julia

Johannes Blaschke, LBNL, USA

Julia Brief Walkthrough

- History: started at MIT in the early 2010s (predates Python Numba) https://julialang.org/blog/2022/02/10years/
- JuliaHub (formely Julia Computing) and MIT are major contributors: https://juliacomputing.com/case-studies
- First stable release v1.0 in 2018, v1.11 as of 2024 https://julialang.org/
- Open-source GitHub-hosted packages and ecosystem with MIT permissive license: https://github.com/JuliaLang/julia
- Community: annual JuliaCon summer conference: https://juliacon.org/2024/

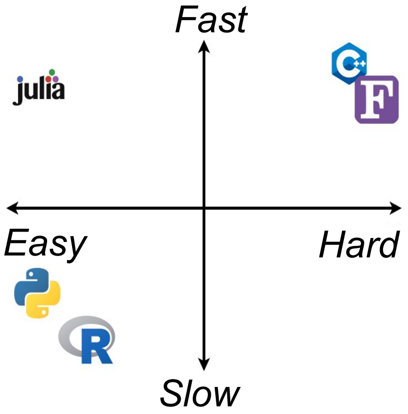




95% of Julia packages in the registry had some form of CI (youtube.com/watch?v=9YWwiFbaRx8)

2

Julia's value proposition for science

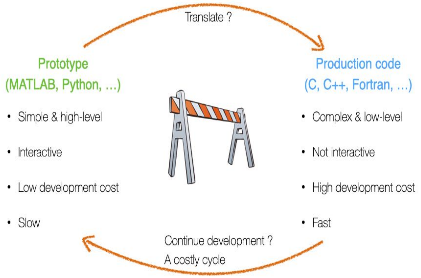


- Designed for "scientific computing" (Fortran-like) and "data science" (Python-like) with **performant kernel code via LLVM compilation**
- Lightweight **interoperability with existing Fortran and C libraries**
- Julia is a **unifying workflow language** with a **coordinated ecosystem**

"Julia **does not** replace Python, but the costly workflow process around **Fortran+Python+X**, **C+X**, **Python+X** or **Fortran+X** (e.g. GPUs, simulation + data analysis)"
where X = { conda, pip, pybind11, cython, Python, C, Fortran, C++, OpenMP, OpenACC, CUDA, HIP, CMake, numpy, scipy, matplotlib, Jupyter, …**}**

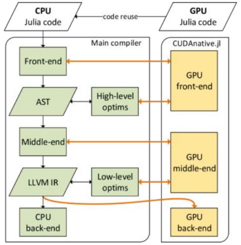
https://juliadatascience.io/

Rich data science ecosystem



https://pde-on-gpu.vaw.ethz.ch/lecture7

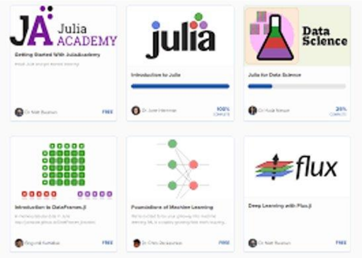
LLVM



https://developer.nvidia.com/blog/gpu-computing-julia-programming-language/

Pkg.jl

https://quantumzeitgeist.com/learning-the-julia-programming-language-for-free/



Julia and HPC

Why Julia for HPC?

~~Walks like Python~~, talks like Lisp, runs like Fortran

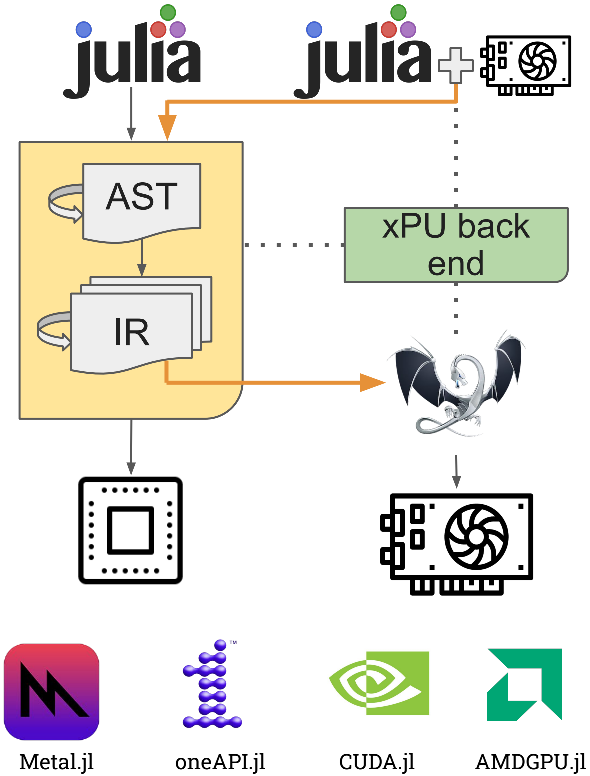
HPC suffers from the many language problem;
Domain experts and performance engineers use different programming languages:
Communication and Collaboration bottleneck
https://www.nature.com/articles/d41586-019-02310-3

Julia: come for the syntax, stay for the speed

Katie Hyatt
@kslimes

My port of our research code from CPU-based C++ to GPU-accelerated #julialang is so much faster (1 week -> 1 hour walltimes) and so much easier to add stuff to... it's a nice holiday gift to myself ❄️☀️.

Rich GPU Ecosystem



Slide shamelessly nicked from: Valentin Churavy

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

Multiple Dispatch => Situation-Dependent Code Generation

- Let’s take a closer look at Multiple Dispatch:
  - A function’s implementation (method) is selected depending on the input types of the

In [40]:
```
function double_int(x::Int)
    return 2*x
end

function double_int(x::AbstractFloat)
    y = floor(Int, x)
    r = x - y
    return 2*y + r
end
```

Out[40]:
```
double_int (generic function with 2 methods)
```

In [30]:
```
double_int(10)
```

Out[30]:
```
20
```

In [31]:
```
double_int(10.1)
```

Out[31]:
```
20.1
```

We can list the methods for a function using the `methods` function:

In [32]:
```
methods(double_int)
```

Out[32]:

2 methods for generic function **double_int**:

- double_int(x::**AbstractFloat**) in Main at In[29]:5
- double_int(x::**Int64**) in Main at In[29]:1

Multiple Dispatch => Situation-Dependent Code Generation

- Let's take a closer look at Multiple Dispatch:
  - A function's implementation (method) is selected depending on the input types of the

In [40]:
```julia
function double_int(x::Int)
    return 2*x
end

function double_int(x::AbstractFloat)
    y = floor(Int, x)
    r = x - y
    return 2*y + r
end
```

Out[40]:
```
double_int (generic function with 2 methods)
```

The `@code_lowered` macro gives is a (still somewhat abstract) idea what Julia actually *does*.

In [33]:
```julia
@code_lowered double_int(2)
```

Out[33]:
```
CodeInfo(
1 ─ %1 = 2 * x
└──      return %1
)
```

This picks up the method for `x` as an integer, and similarly we can see what Julia does when `x` is a float:

In [34]:
```julia
@code_lowered double_int(2.1)
```

Out[34]:
```
CodeInfo(
1 ─      y = Main.floor(Main.Int, x)
│        r = x - y
│   %3 = 2 * y
│   %4 = %3 + r
└──      return %4
)
```

NERSC

Multiple Dispatch => Situation-Dependent Code Generation

- Let's take a closer look at Multiple Dispatch:
  - Following dispatch, Typed IR is translated to LLVM IR

```
In [40]: function double_int(x::Int)
             return 2*x
         end

         function double_int(x::AbstractFloat
             y = floor(Int, x)
             r = x - y
             return 2*y + r
         end

Out[40]: double_int (generic function with 2
```

tually *does*.

s when `x` is a float:

Julia is a REPL for LLVM:

```
In [41]: @code_llvm double_int(2)

;  @ In[40]:1 within `double_int`
define i64 @julia_double_int_2028(i64 signext %0) #0 {
top:
;  @ In[40]:2 within `double_int`
; ┌ @ int.jl:88 within `*`
   %1 = shl i64 %0, 1
; └
  ret i64 %1
}
```

NERSC

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

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}
```

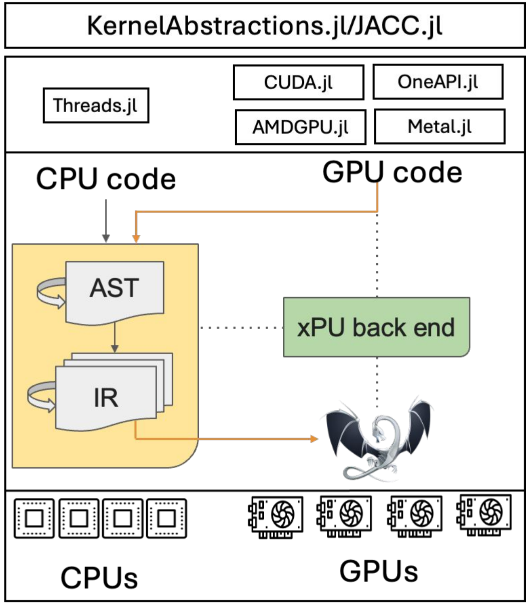
compiles to

```
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
```

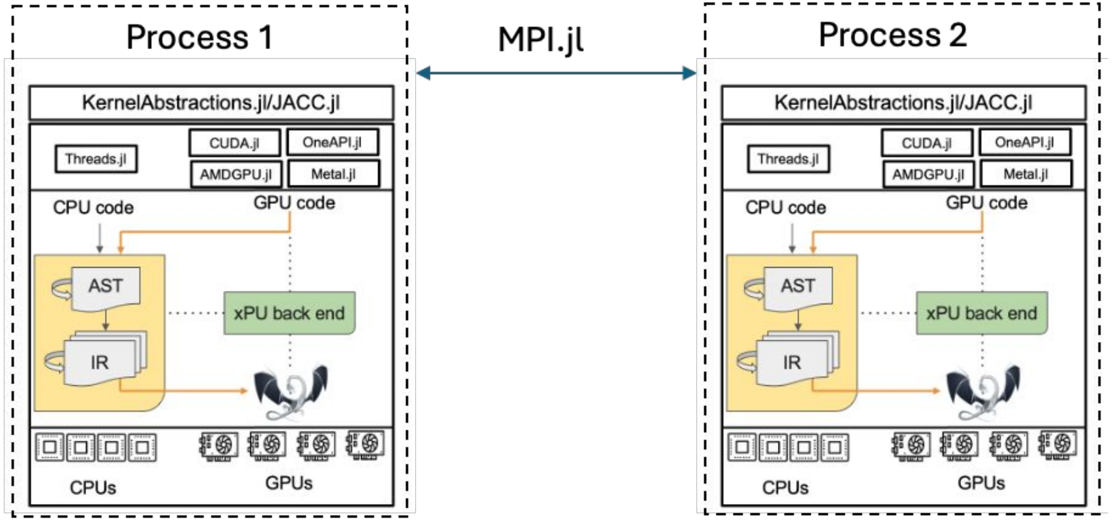

Slide shamelessly nicked from: Valentin Churavy

Julia Parallel Paradigm

Shared-memory



Distributed-memory

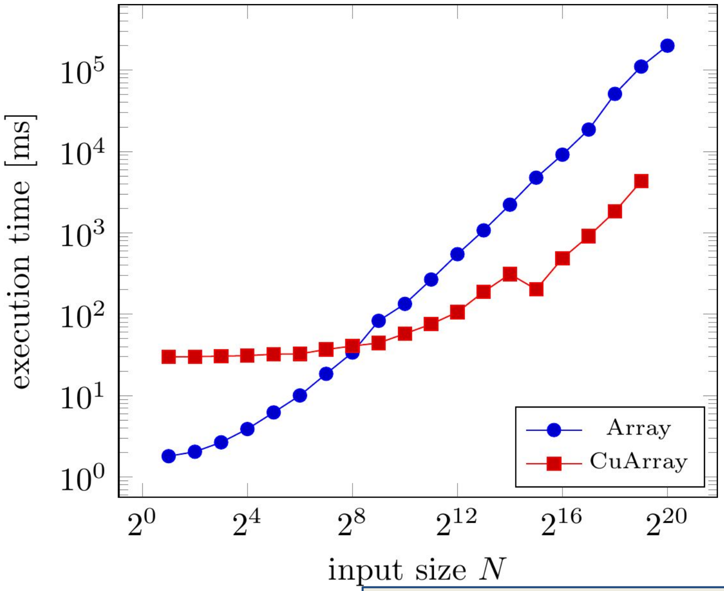


Array programming

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))

```julia
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) = ...

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
```

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



Slide shamelessly nicked from: Valentin Churavy

NERSC
BERKELEY LAB
Bringing Science Solutions to the World

KernelAbstractions.jl

- **GPU-centric** programming model
- SIMT execution model
- Multiple GPU backends
- Implementation using **macros** and **method overlay tables**
- GPU backends:
  - CUDA, AMDGPU, oneAPI, Metal
  - 172 - 188 LOC
- CPU execution as a fallback and for debugging
- Widely adopted in the JuliaHPC and JuliaGPU ecosystem

```julia
@kernel function lmem_copy_kernel!(output, @Const(input))
    I, J = @index(Global, NTuple)
    i, j = @index(Local,  NTuple)
    N = @uniform @groupsize()[1]
    M = @uniform @groupsize()[2]
    tile = @localmem eltype(output) (N, M)

    @inbounds tile[i, j] = input[I, J]
    @synchronize
    @inbounds output[I, J] = tile[i, j]
end
```


Slide shamelessly nicked from: Valentin Churavy

(advanced) LLVM + Julia

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
```

In a Nutshell…Why Julia??

- ***Think in Python/Fortran, but now imagine that it works "well" on HPC***

- A JIT "technical" language on top of LLVM
  - Easy-to-use and agile interface with expected performance

- Julia syntax is for Science and is Open Source

- Native syntax (no manual implemented wrappers of wrapper of …) for HPC support, e.g. www.juliagpu.org
  - Threads, CUDA, AMDGPU, OneAPI, MPI, DAGGER, etc.

JuliaGPU

High-performance GPU programming in a high-level language.

- Interoperability: C and Fortran, support for AI: FluxML

- Integrated and efficient support for packaging, reproducibility, CI/CD, …

- All this makes Julia's ecosystem and community motivated by performance and productivity -> e.g. **rapid prototyping, "throw-away" code, scientific CPU/GPU access, HPC stack**

Why NOT Julia??

- Existing large investments in other languages
- Long-term ROI: Package support? Stable not standard
- ~~Ecosystem is not mature: Tooling? (HPC)~~
- Out of scope for my application needs
- I'm simply more comfortable in another language

https://julialang.org/

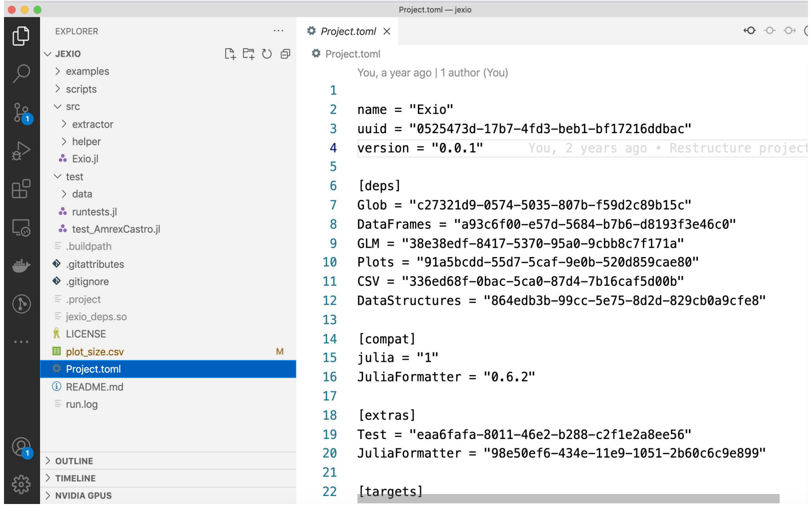
Julia Brief Walkthrough

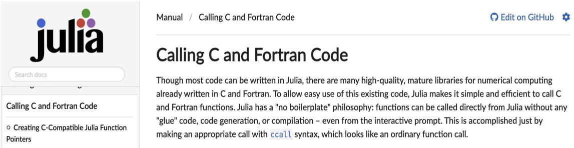
- Reproducibility is in the core of the language:
  - Interactive: Jupyter, Pluto.jl
  - Packaging Pkg.jl
  - Environment Project.toml
  - Testing Test.jl

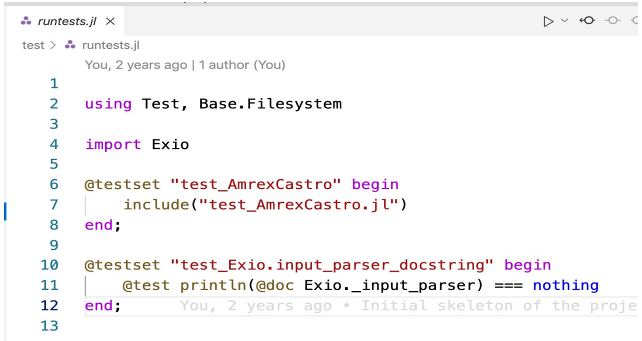
- Just-in-time or Ahead-of-time compilation with PackageCompiler.jl

- Powerful metaprogramming for code instrumentation: @profile, @time, @testset, @test, @code_llvm, @code_native, @inbounds,

- Interoperability is key: @ccall, @cxx, PyCall, CxxWrap.jl







https://github.com/ornl-training/julia-basics

BERKELEY LAB
Bringing Science Solutions to the World
U.S. DEPARTMENT OF ENERGY
Office of Science