GPU Programming in julia

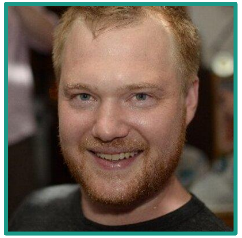
Neil Gutkin

Brief History

Alan Edelman

Jeff Bezanson

- Work started in 2009, first release in 2012
  - Developed and incubated at MIT
  - Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman
  - Open-source and free under MIT license
  - Julia 1.7.0 was released November 30, 2021


Stefan Karpinski

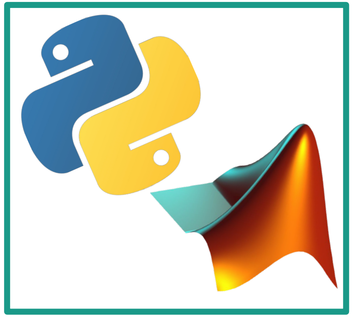
Viral B. Shah

- Why?
  - “In short, because we are greedy” [1]
  - Wanted a fast, high-level, all-purpose language

[1] https://julialang.org/blog/2012/02/why-we-created-julia/

Images: Twitter.com, Wikidata.org, Wikipedia.org

Why is Julia cool?



- Dynamically typed
- High-level syntax
- Built-in package manager
- Interactive development

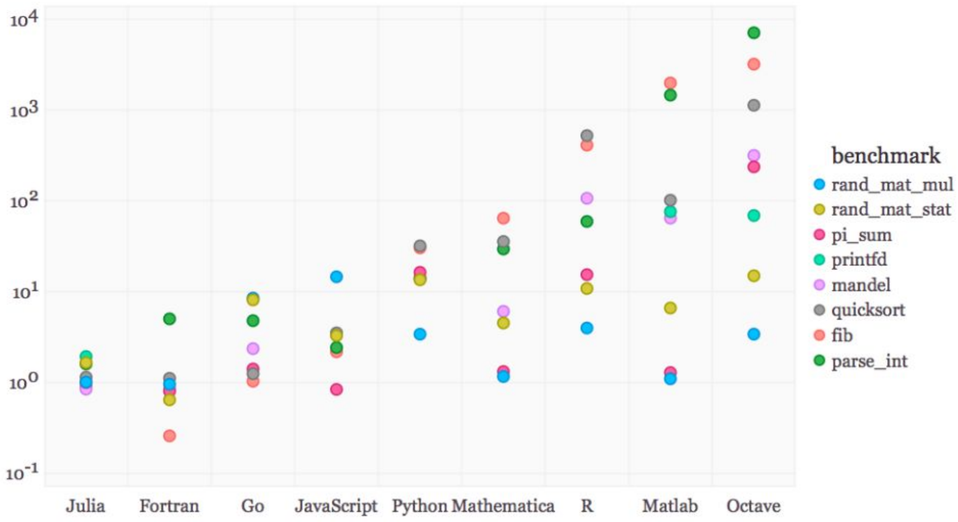
+

- Great performance
- Works well with GPUs

Images: Wikipedia.org

Some cool features are...

- JIT compilation
  - Compilation happens at runtime
- Dynamic dispatch
  - Functional behavior depends on arguments
- Reflection and metaprogramming
  - Most of Julia is written in Julia
- Effective debugging
  - Debugger is a package
- Unicode integration
  - Gimmick?

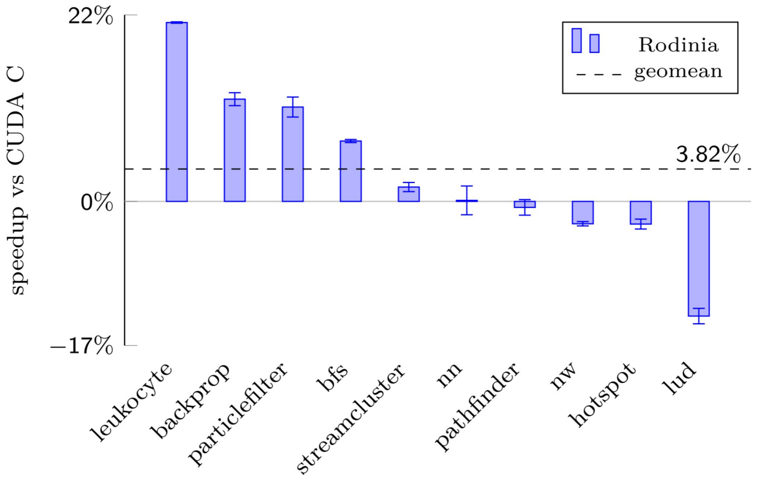


Source: MIT.edu

So, you mentioned GPUs?

- Mature support for NVIDIA
  - CUDA.jl
- Similar but newer support for Intel
  - oneAPI.jl
- Experimental support for AMD
  - AMDGPU.jl

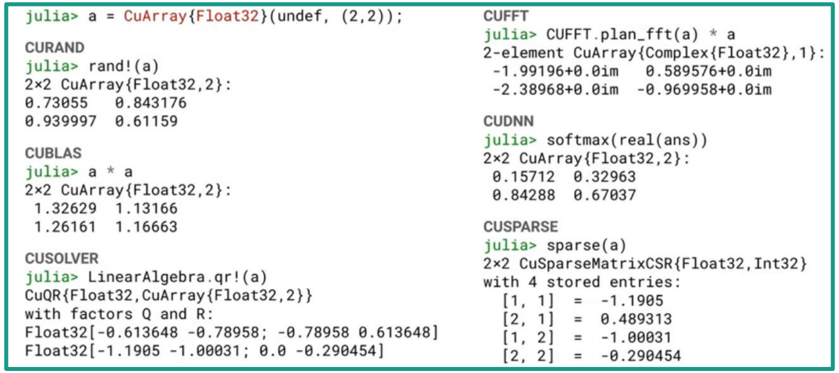
- **Kernels are Julia functions!**
  - *...they're not just translated into C code*



Source: JuliaGPU.org

Choose your style... or both

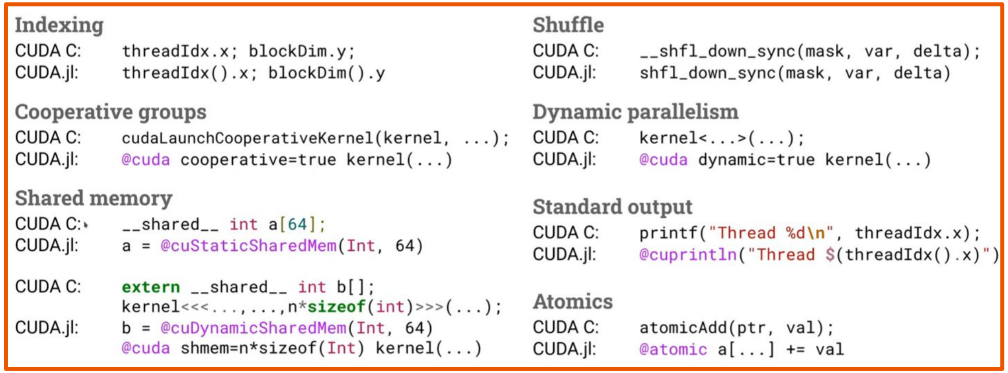
Array Programming



```
julia> a = CuArray{Float32}(undef, (2,2));

CURAND
julia> rand!(a)
2×2 CuArray{Float32,2}:
0.73055    0.843176
0.939997   0.61159

CUBLAS
julia> a * a
2×2 CuArray{Float32,2}:
 1.32629   1.13166
 1.26161   1.16663

CUSOLVER
julia> LinearAlgebra.qr!(a)
CuQR{Float32,CuArray{Float32,2}}
with factors Q and R:
Float32[-0.613648 -0.78958; -0.78958 0.613648]
Float32[-1.1905 -1.00031; 0.0 -0.290454]
```

```
CUFFT
julia> CUFFT.plan_fft(a) * a
2-element CuArray{Complex{Float32},1}:
 -1.99196+0.0im    0.589576+0.0im
 -2.38968+0.0im   -0.969958+0.0im

CUDNN
julia> softmax(real(ans))
2×2 CuArray{Float32,2}:
 0.15712   0.32963
 0.84288   0.67037

CUSPARSE
julia> sparse(a)
2×2 CuSparseMatrixCSR{Float32,Int32}
with 4 stored entries:
  [1, 1]  =  -1.1905
  [2, 1]  =  0.489313
  [1, 2]  =  -1.00031
  [2, 2]  =  -0.290454
```

Kernel Programming



Indexing

```
CUDA C:     threadIdx.x; blockDim.y;
CUDA.jl:    threadIdx().x; blockDim().y
```

Cooperative groups

```
CUDA C:     cudaLaunchCooperativeKernel(kernel, ...);
CUDA.jl:    @cuda cooperative=true kernel(...)
```

Shared memory

```
CUDA C:     __shared__ int a[64];
CUDA.jl:    a = @cuStaticSharedMem(Int, 64)

CUDA C:     extern __shared__ int b[];
            kernel<<<...,...,n*sizeof(int)>>>(...);
CUDA.jl:    b = @cuDynamicSharedMem(Int, 64)
            @cuda shmem=n*sizeof(Int) kernel(...)
```

Shuffle

```
CUDA C:     __shfl_down_sync(mask, var, delta);
CUDA.jl:    shfl_down_sync(mask, var, delta)
```

Dynamic parallelism

```
CUDA C:     kernel<...>(...);
CUDA.jl:    @cuda dynamic=true kernel(...)
```

Standard output

```
CUDA C:     printf("Thread %d\n", threadIdx.x);
CUDA.jl:    @cuprintln("Thread $(threadIdx().x)")
```

Atomics

```
CUDA C:     atomicAdd(ptr, val);
CUDA.jl:    @atomic a[...] += val
```

JuliaComputing.com

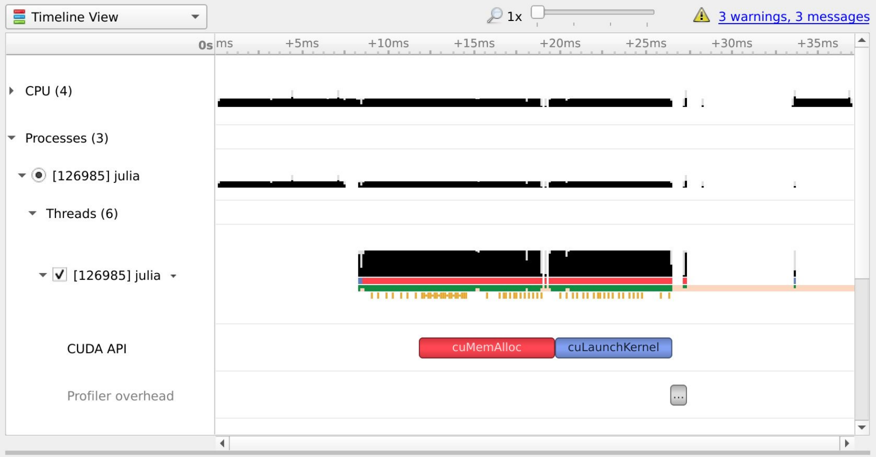
What about profiling and debugging?

Profiling

- Timing code is straightforward & robust
  - BenchmarkTools.jl
- nvprof
- nvvp
- Nsight tools

Debugging

- Code reflection macros
  - Interactively descend into call stack
    - Chthulhu.jl
- cuda-memcheck



JuliaGPU.org

Okay, so what's the catch?

- There's not really a catch…
  - I've had a positive experience working with Julia

However...

- When writing kernels…
  - Some Julia functionality is disabled
    - Dynamic typing, garbage collection, I/O
- Compilation + REPL start-up takes a while
  - Constant
- 1-based indexing is annoying
  - Personal preference

StickPNG.com

Code Demo

- The repo is linked in my project thread on Slack
    - https://code.vt.edu/neilg99/julia_exploration

- Live code?
- **Questions?**