universität innsbruck

Native GPU computing in Julia

A High-level language with low-level performance

Peter Kandolf

Disclaimer

The sources for this overview and some additional references:

- Julia basics: `julialang.org` and `https://docs.julialang.org/`
- GPU specifics: Presentations by Tim Besard, Valentin Churavy,...
  `https://juliagpu.org/`
- JuliaCon21 GPU Workshop material
  `https://github.com/maleadt/juliacon21-gpu_workshop`
- `https://juliaacademy.com/` (Introduction to Julia (for programmers), Parallel Computing, . . .)

Julia in a Nutshell

Fast

Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM.

Dynamic

Julia is dynamically typed, feels like a scripting language, and has good support for interactive use.

Reproducible

Reproducible environments make it possible to recreate the same Julia environment every time across platforms, with pre-built binaries.

Composable

Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The talk on the Unreasonable Effectiveness of Multiple Dispatch explains why it works so well.

General

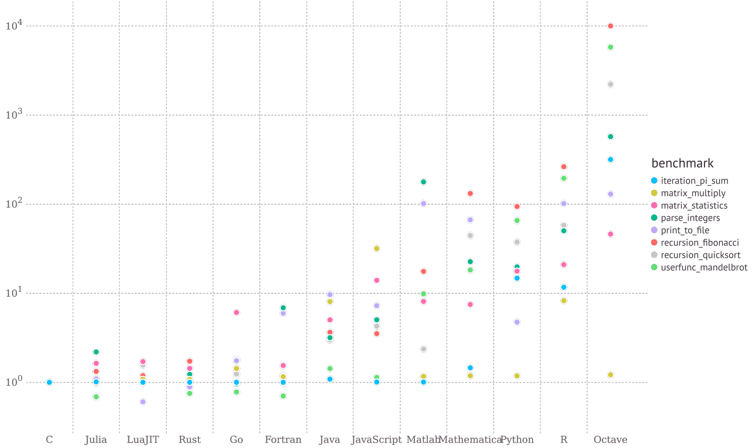
Julia provides asynchronous I/O, metaprogramming, debugging, logging, profiling, a package manager, and more. One can build entire Applications and Microservices in Julia.

Open source

Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub.

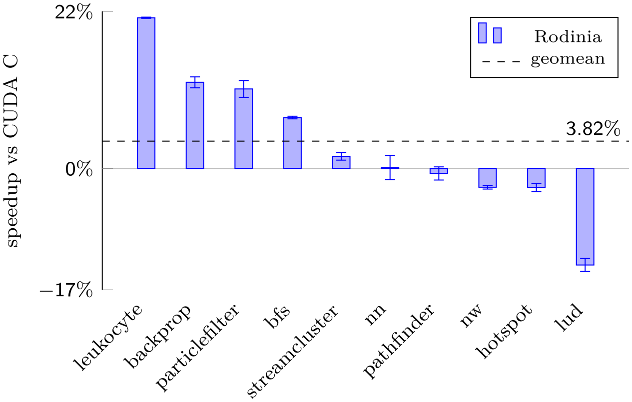
`https://julialang.org`

Julia - Fast



https://julialang.org/benchmarks/

Why should you care?



Relative performance of Rodinia benchmarks implemented in Julia with CUDA.jl.

https://juliagpu.org/cuda/, https://doi.org/10.1016/j.advengsoft.2019.02.002

GPU support for Julia - JuliaGPU

juliagpu.org unifies the many packages for programming GPUs in Julia.

Supported platforms - Frontend framework:

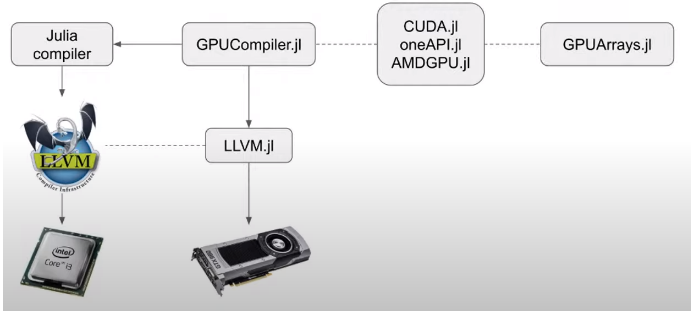
- CUDA.jl - NVIDIA CUDA (`https://juliagpu.org/cuda/`)
- AMDGPU.jl - AMD ROCm (`https://juliagpu.org/rocm/`)
- oneAPI.jl - Intel oneAPI (`https://juliagpu.org/oneapi/`)
- other
  - OpenCL
  - ArrayFire

`https://juliagpu.org/`

How?

Powerful array programming interfaces

- Natural mapping of `map`, `broadcast`, `scan`, `reduce`, ...
- Features like multiple dispatch, specialisation, ... allows for generic, reusable applications



`https://www.youtube.com/watch?v=Hz9IMJuW5hU`

Supported parallel programming models

Parallel abstraction - implicit parallelism

```
reduce(op, data)
data * data

# Example
a = CuArray.ones(10)
reduce(+,a)
10.0f0
```

Supported parallel programming models

Parallel abstraction - implicit parallelism

```julia
reduce(op, data)
data * data

# Example
a = CuArray.ones(10)
reduce(+,a)
10.0f0
```

Single Program Multiple Data - explicit parallelism

```julia
function vadd(c, a, b)
 i = threadIdx().x
 c[i] = a[i] + b[i]
 return
end

a = CuArray(1:10)
b = CuArray(2:2:20)
c = similar(a)
@cuda threads=length(a) vadd(c, a, b)
```

Installation

```
julia> using Pkg
julia> Pkg.add("CUDA")

julia> using CUDA
julia> CUDA.versioninfo()
```

```
Downloaded artifact: CUDA
CUDA toolkit 11.4.1, artifact installation
CUDA driver 11.2.0
NVIDIA driver 460.32.3

Libraries:
- CUBLAS: 11.5.4
- CURAND: 10.2.5
- CUFFT: 10.5.1
- CUSOLVER: 11.2.0
- CUSPARSE: 11.6.0
- CUPTI: 14.0.0
- NVML: 11.0.0+460.32.3
  Downloaded artifact: CUDNN
- CUDNN: 8.20.2 (for CUDA 11.4.0)
  Downloaded artifact: CUTENSOR
- CUTENSOR: 1.3.0 (for CUDA 11.2.0)

Toolchain:
- Julia: 1.6.3
- LLVM: 11.0.1
- PTX ISA support: 3.2, 4.0, 4.1, 4.2, 4.3, 5.0, 6.0, 6.1, 6.3, 6.4, 6.5,
- Device capability support: sm_35, sm_37, sm_50, sm_52, sm_53, sm_60, sm_

2 devices:
  0: A40 (sm_86, 44.562 GiB / 44.565 GiB available)
  1: A40 (sm_86, 44.562 GiB / 44.565 GiB available)
```

Array programming

```
julia> a = CuArray([1,2,3,4])'
1×4 adjoint(::CuArray{Int64, 1, CUDA.Mem.DeviceBuffer}) with eltype Int64:
 1  2  3  4

julia> a .+= 1
1×4 adjoint(::CuArray{Int64, 1, CUDA.Mem.DeviceBuffer}) with eltype Int64:
 2  3  4  5

julia> a * 2
1×4 adjoint(::CuArray{Int64, 1, CUDA.Mem.DeviceBuffer}) with eltype Int64:
 4  6  8  10

julia> sum(a)
14
```

Array programming 2

```
julia> a'*a
4×4 CuArray{Int64, 2, CUDA.Mem.DeviceBuffer}:
  4   6   8  10
  6   9  12  15
  8  12  16  20
 10  15  20  25

julia> d = CUDA.rand(length(a))
4-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 0.920334
 0.6714725
 0.66469926
 0.31437585
```

Vendor library

```
# CUBLAS
julia> using LinearAlgebra
julia> mul!(d, CUDA.rand(4,4), d)
4-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 1.2517724
 0.996908
 1.1825426
 1.3225887

# CURAND
julia> using Random
julia> rand!(d)
4-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 0.16366833
 0.91374624
 0.53687394
 0.5916346
```

Vendor library 2

```
# CUSOLVER
julia> qr(d);
Float32[-0.1336354 -0.7460745 -0.438358 -0.48307008; -0.7460745 0.5089892 -0.28849465
Float32[-1.2247386]

# CUFFT
julia> using AbstractFFTs
julia> fft(d)
4-element CuArray{ComplexF32, 1, CUDA.Mem.DeviceBuffer}:
   2.205923f0 + 0.0f0im
 -0.3732056f0 - 0.32211167f0im
 -0.8048386f0 + 0.0f0im
 -0.3732056f0 + 0.32211167f0im


julia> out = Ref{Cint}()
julia> CUBLAS.cublasIsamin_v2(CUBLAS.handle(), length(d), d, stride(d, 1), out)
julia> out[]
1
```

Higher-order abstractions:

```
julia> a = CuArray([1 2 3])
1×3 CuArray{Int64, 2, CUDA.Mem.DeviceBuffer}:
 1  2  3

julia> broadcast(a) do x
        sin(x)
       end
1×3 CuArray{Float64, 2, CUDA.Mem.DeviceBuffer}:
 0.841471  0.909297  0.14112

julia> a .+ 1
1×3 CuArray{Int64, 2, CUDA.Mem.DeviceBuffer}:
 2  3  4

julia> map(a) do x
         x + 1
       end
1×3 CuArray{Int64, 2, CUDA.Mem.DeviceBuffer}:
 2  3  4
```

Higher-order abstractions 2

```
julia> reduce(+, a)
6

julia> accumulate(+, a)
1×3 Matrix{Int64}:
 1  3  6

julia> findfirst(isequal(2), a)
CartesianIndex(1, 2)

julia> a = CUDA.ones(10, 10);
julia> broadcast(eachcol(a)) do x
        sum(x)
       end
10-element Vector{Float32}:
 10.0
 10.0
 10.0
 10.0
 10.0
 10.0
 10.0
 10.0
```

Pitfalls - Be aware of scalar operations:

```
julia> a = CUDA.rand(Int, 1000)
julia> for i in eachindex(a)
         a[i] += 1
       end
┌ Warning: Performing scalar indexing on task Task (runnable) @0x00002ba8512f8010.
│ Invocation of getindex resulted in scalar indexing of a GPU array.
│ This is typically caused by calling an iterating implementation of a method.
│ Such implementations *do not* execute on the GPU, but very slowly on the CPU,
│ and therefore are only permitted from the REPL for prototyping purposes.
│ If you did intend to index this array, annotate the caller with @allowscalar.
└
julia> CUDA.allowscalar(false)
julia> a[1]
ERROR: Scalar indexing is disallowed.
Invocation of getindex...

julia> CUDA.@allowscalar a[1]
```

Kernel programming

```julia
function vadd(c, a, b)
    i = threadIdx().x
   c[i] = a[i] + b[i]
   return
end

a = CuArray(1:10)
b = CuArray(2:2:20)
c = similar(a)
@cuda threads=length(a) vadd(c, a, b)
```

Kernel programming 2

```
function vadd(c, a, b)
    i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
    if i <= length(a)
        c[i] = a[i] + b[i]
    end
    return
end
@cuda threads=1024 blocks=cld(length(a),1024) vadd(c, a, b)
```

Benchmarking

```
julia> a = CUDA.ones(10, 10)
julia> broadcast(eachcol(a)) do x
         sum(x)
       end                                   #one kernal launch per column --> slow
julia> sum(a; dims=2)                        #optimized version

julia> using BenchmarkTools
```

Benchmarking - Results

```
julia> @benchmark CUDA.@sync broadcast(eachcol($a)) do x
         sum(x)
       end

BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  287.147 μs … 39.300 ms  ┊ GC (min … max): 0.00% … 33.75%
 Time  (median):     342.401 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   368.793 μs ±  1.082 ms  ┊ GC (mean ± σ):  2.95% ±  1.00%
```

```
  287 μs          Histogram: frequency by time          372 μs <

 Memory estimate: 32.09 KiB, allocs estimate: 740.
```

Benchmarking - Results

```
julia> @benchmark CUDA.@sync broadcast(eachcol($a)) do x
           sum(x)
       end

BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  287.147 μs … 39.300 ms  ┊ GC (min … max): 0.00% … 33.75%
 Time  (median):     342.401 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   368.793 μs ±  1.082 ms  ┊ GC (mean ± σ):  2.95% ±  1.00%
```

```
  287 μs          Histogram: frequency by time          372 μs <

 Memory estimate: 32.09 KiB, allocs estimate: 740.
julia> @benchmark CUDA.@sync sum(a; dims=2)

BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  15.879 μs … 537.396 μs  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     24.296 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   24.428 μs ±   6.074 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  15.9 μs         Histogram: frequency by time         29.5 μs <

Memory estimate: 2.27 KiB, allocs estimate: 50.
```

Profiling - NVIDIA Tools

```
$ nsys launch julia

julia> CUDA.@profile a.+reverse(a)
# Report file moved to "/temp/report1.qdrep"

$ nsys-ui /temp/report1.qdrep
```

CUDA.jl 3.0 - Concurrent GPU computing

asynchronous operations

- Kernel launches - `@cuda`
- Array operations - `broadcast`, `LinearAlgebra calls`, ...
- Most memory operations - `CuArray(), copyto!`, ...

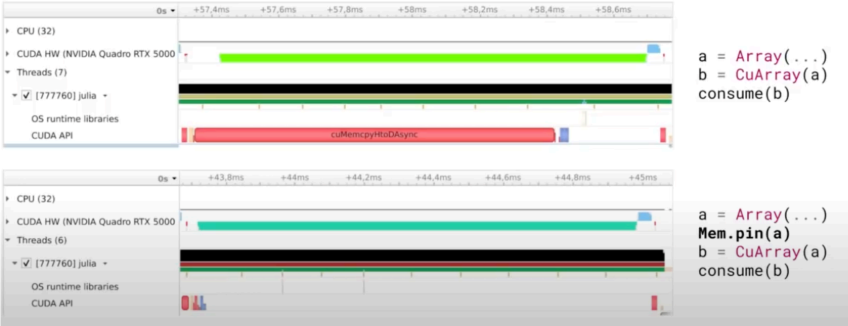
synchronous operations

- Stream synchronization - `synchronize()`
- Memory copies to the host - `Array(::CuArray)`
- Most memory operations - `CuArray(), copyto!`, ...

https://www.youtube.com/watch?v=fw0R5G8pB0U

CUDA.jl 3.0 - Concurrent GPU computing

Some GPU operations block in CUDA itself



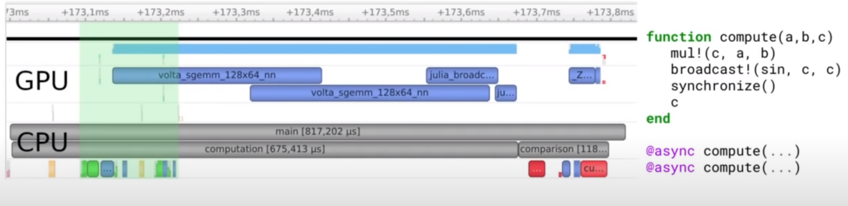
https://www.youtube.com/watch?v=fw0R5G8pB0U

CUDA.jl 3.0 - Overlap GPU computing

```
julia> @sync begin
          @async begin
             println("Start GPU work 1.")
             ...
             synchronize()
             println("GPU work 1 done.")
          end
           @async begin
             println("Start GPU work 2.")
             ...
             synchronize()
             println("GPU work 2 done.")
          end
       end

Start GPU work 1.
Start GPU work 2
GPU work 1 done.
GPU work 2 done.
```

CUDA.jl 3.0 - Overlap GPU computing



https://www.youtube.com/watch?v=fw0R5G8pB0U

CUDA.jl 3.0 - Multiple GPUs

```julia
julia> @sync begin
          @async begin
             device!(0)
             println("Start work on GPU 1.")
             ...
             synchronize()
             println("GPU 1 work done.")
          end
           @async begin
             device!(1)
             println("Start work on GPU 2.")
             ...
             synchronize()
             println("GPU 2 work done.")
          end
       end
```

This is also possible to do on different threads (experimental).