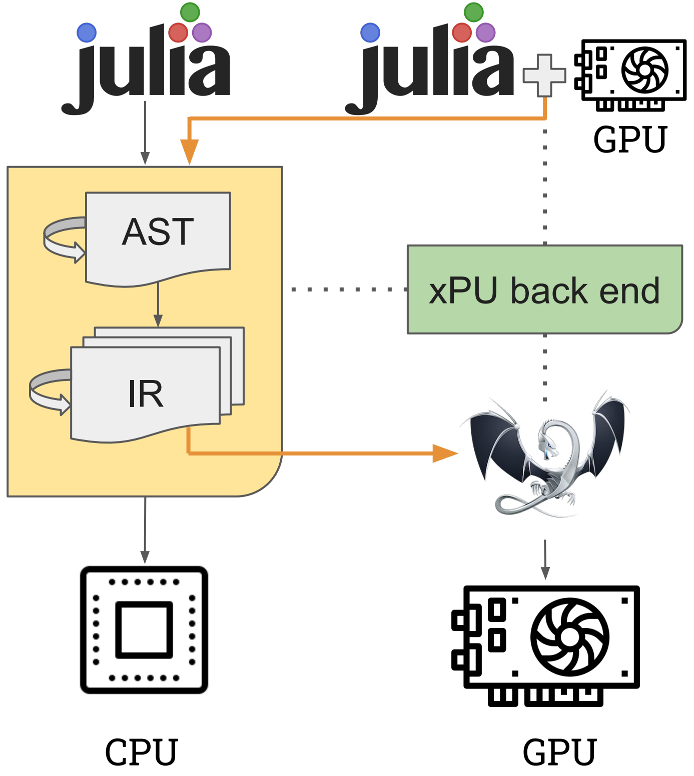
How to write device portable Code

Valentin Churavy (@vchuravy)

julia gets its Power from Extensible Compiler Design

Language design

Efficient execution



Julia: Dynamism and Performance Reconciled by Design (doi:10.1145/3276490)

Effective Extensible Programming: Unleashing Julia on GPUs (doi:10.1109/TPDS.2018.2872064)

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

```
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
```

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

Parallel programming with 3 character changes: Array type programming model

Array types — **where** memory resides and **how** code is executed

`A = Matrix{Float64}(64,32)`	CPU (Intel, IBM, Apple)
`A = CuMatrix{Float64}(64,32)`	NVIDIA (CUDA) GPU
`A = ROCMatrix{Float64}(64,32)`	AMD (ROCm) GPU

Think: Coloured pointers. We now semantically where data is stored.

Array programming

```
julia> CuArray{Float32,2}(undef, 2, 2)
2×2 CuArray{Float32,2}:
0.0  0.0
0.0  0.0

julia> similar(a)
1×3 CuArray{Int64,2}:
0  0  0
```

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

julia> b = Array(a)
1×3 Array{Int64,2}:
1  2  3
```

Goal: API compatibility with `Base.Array`

Array programming

```
julia> CUDA.ones(2)
2-element CuArray{Float32,1}:
1.0
1.0

julia> CUDA.zeros(Float32, 2)
2-element CuArray{Float32,1}:
0.0
0.0
```

```
julia> CUDA.fill(42, (3,4))
3×4 CuArray{Int64,2}:
42  42  42  42
42  42  42  42
42  42  42  42

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

Goal: API compatibility with `Base.Array`

```
julia> a = CuArray([1 2 3])

julia> view(a, 2:2)
1-element CuArray{Int64,1,...}:
2

julia> sum(a)
6

julia> a * 2
1×3 CuArray{Int64,2}:
2  4  6
```

```
julia> a'
3×1 Adjoint{Int64,CuArray{Int64,2}}:
1
2
3

julia> a * CuArray([1,2,3])
1-element CuArray{Int64,1}:
14
```

Goal: API compatibility with `Base.Array`

```
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
```

```
julia> a = CuArray([1 2 3])
julia> b = CuArray([4 5 6])

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

julia> a .+ 2b
1×3 CuArray{Int64,2}:
9  12  15
```

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

julia> accumulate(+, b; dims=2)
1×3 CuArray{Int64,2}:
4  9  15

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

Powerful array language: obviates need for custom kernels
makes it possible to write generic code

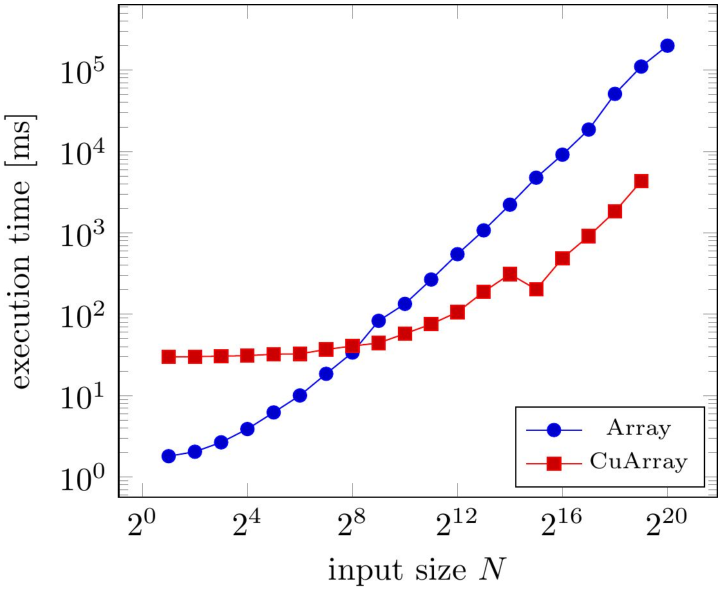
```
using LinearAlgebra

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

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



```
julia> a = CUDA.rand(Int, 1000)
julia> using LinearAlgebra

julia> norm(a)
Warning: Performing scalar operations on GPU arrays
3.74165738677394e20

julia> @btime norm($a)
 9.799 ms (18 allocations: 768 bytes)

julia> @btime norm($(Array(a)))
 2.908 μs (0 allocations: 0 bytes)

julia> CUDA.allowscalar(false)

julia> norm(a)
ERROR: scalar getindex is disallowed
```

Fallback functionality performing iteration

```
function generic_normInf(x)
   (v, s) = iterate(x)::Tuple
   maxabs = norm(v)
   while true
       y = iterate(x, s)
       y === nothing && break
       ...
   end
   return float(maxabs)
end
```

Portable Julia for numerical kernels

Lowest Level Abstraction for multiple hardware: GPU-focused semantics + CPU execution

```julia
using KernelAbstractions
using CUDA

@kernel function transpose_kernel!(b, @Const(a))
    i, j = @index(Global, NTuple)
    @inbounds b[i, j] = a[j, i]
end

a = CuArray(rand(Float32, (1024, 8192)))
b = similar(a, Iterators.reverse(size(a))...)

# Instantiate kernel with static information
const kernel! = transpose_kernel!(CUDADevice(), (32, 32))

# Execute kernel -- event based dynamic task-graph
event = kernel!(b, a, ndrange=size(a))
wait(event)
```

oneAPI.jl CUDA.jl AMDGPU.jl

Type parameters

- Type parameters allow for generic functionality

- We might want to be generic over data-types. Support `Float16`, `Float32`, `Float64`

- For device portability:
  Be generic over storage type

```julia
struct Model
    data::Array{Float64, 2}
end
```

```julia
struct Model{T<:Number}
    data::Array{T, 2}
end
```

```julia
struct Model{T<:Number, AT<:AbstractArray{T}}
    data::AT
end
```

Adapt.jl

- https://github.com/JuliaGPU/Adapt.jl is a lightweight dependency that you can use to convert complex structures from CPU to GPU.

```
using Adapt
adapt(CuArray, ::Adjoint{Array})::Adjoint{CuArray}


struct Model{T<:Number, AT<:AbstractArray{T}}
   data::AT
end

Adapt.adapt_structure(to, x::Model) = Model(adapt(to, x.data))


cpu = Model(rand(64, 64));
using CUDA

gpu = adapt(CuArray, cpu)
Model{Float64, CuArray{Float64, 2, CUDA.Mem.DeviceBuffer}}(...)
```

PSA: Coming in 1.9

- Weak dependencies and package extensions
  - GPU backends are often heavy dependencies
  - Ideally user would only need one backend, but we often need to add methods to support different backends.
  - https://pkgdocs.julialang.org/dev/creating-packages/#Conditional-loading-of-code-in-packages-(Extensions)

```toml
name = "FastCode"
version = "0.1.0"
uuid = "..."

[weakdeps]
CUDA = "052768ef-5323-5732-b1bb-66c8b64840ba"

[extensions]
# name of extension to the left
# extension dependencies required to load the extension to the right
# use a list for multiple extension dependencies
CUDAExt = "CUDA"

[compat]
CUDA = "4"
```