Growing a Compiler

Getting to machine learning from a general purpose compiler

CGO C4ML 2019

Keno Fischer and Jameson Nash

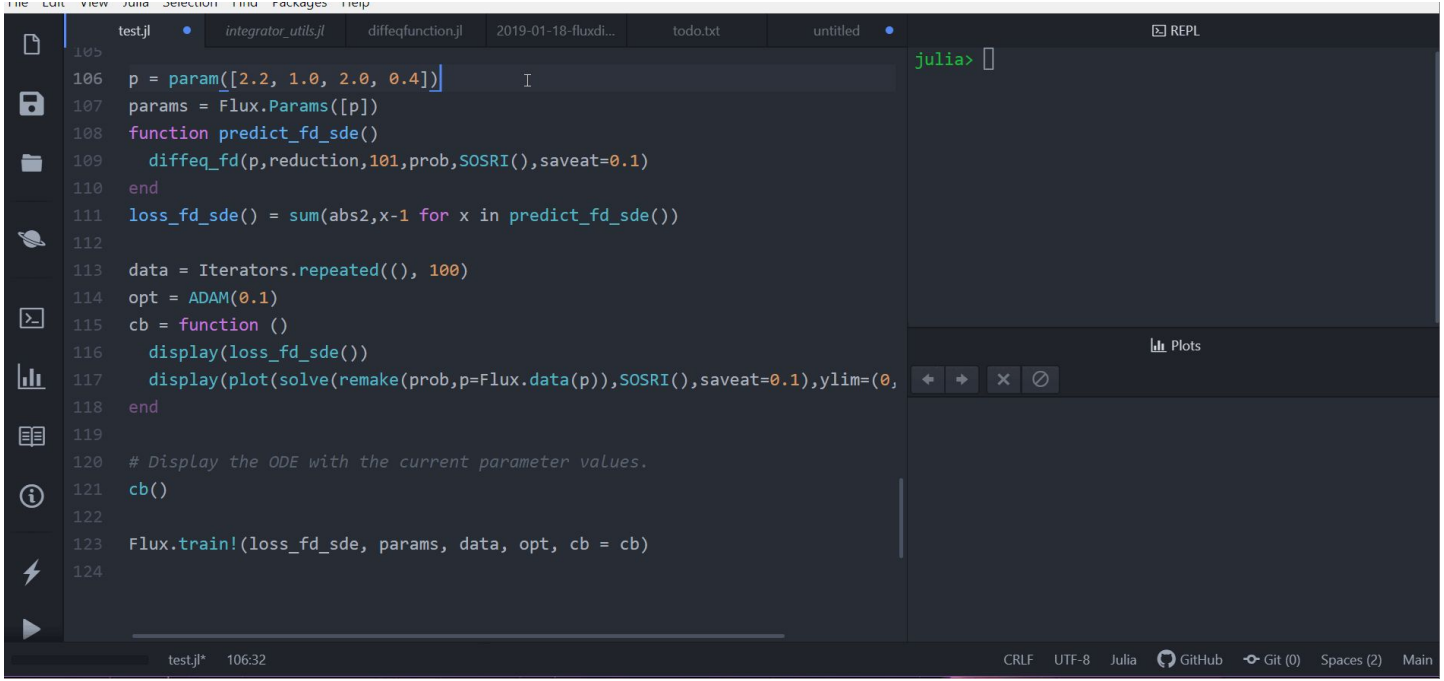
With Tim Besard, James Bradbury, Valentin Churavy, Dhairya Gandhi, Mike Innes, Neethu Joy, Tejan Karmali, Matt Kelley, Avik Pal, Chris Rackauckas, Marco Rudilosso, Elliot Saba, Viral Shah, and Deniz Yuret

julia

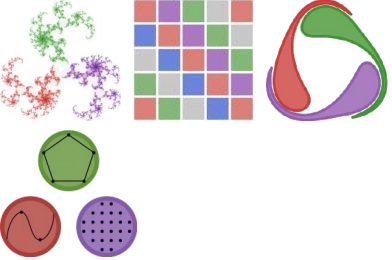
Julia computing

Neural ODEs

Combining ML and Differential Equation libraries



DiffEqFlux.jl — A Julia Library for Neural Differential Equations (arXiv:1902.02376)




Composability of Libraries

- Numbers (Integer, Floating Point, Complex, Quaternion, Rational, Fixed Point, non-standard precision, Intervals, Unitful, Symbolic, Dual, Grassmanian, Log-Domain, Polynomial, ...)
- Arrays (Matrices, Tensors, Offset, Named, Remote, ...)
- Linear Algebra (Factorizations, Arithmetic, Infinite, Finite Fields, ...)
- Differential Equations
- Convex Optimization
- Parallel Computing
- Data Science
- Graphics
- Machine Learning
- Image Processing Libraries

Composability of Libraries

- Numbers (Integer, Floating Point, Complex, Quaternion, Rational, Fixed Point, non-standard precision, Intervals, Unitful, Symbolic, Dual, Grassmanian, Log-Domain, Polynomial, ...)
- Arrays (Matrices, Tensors, Offset, Named, Remote, ...)
- Linear Algebra (Factorizations, Arithmetic, Infinite, Finite Fields, ...)
- Differential Equations
- Convex Optimization
- Parallel Computing
- Data Science
- Graphics
- Machine Learning
- Image Processing Libraries

Composability of Compiler Transforms?

Transforms / Programming Models	Optimizations	Code Generators
Automatic Differentiation SPMD Task Parallelism Data Query Interval Constraint Programming Disciplined Convex Programming Tracing/Profiling/Debugging Verification/Formal Methods . . .	Scalar Tensor Parallel/Distributed Data Layout Data Access (Polyhedral) Relational (Query Compilers) Symbolic . . .	CPU GPU TPU / ML Accelerators Virtual Machines (WebAssembly) FPGA / Silicon Quantum Computers Homomorphic Encryption . . .

Composability of Compiler Transforms?

Transforms / Programming Models	Optimizations	Code Generators
Automatic Differentiation SPMD Task Parallelism Data Query Interval Constraint Programming Disciplined Convex Programming Tracing/Profiling/Debugging Verification/Formal Methods ...	Scalar Tensor Parallel/Distributed Data Layout Data Access (Polyhedral) Relational (Query Compilers) Symbolic ...	CPU GPU TPU / ML Accelerators Virtual Machines (WebAssembly) FPGA / Silicon Quantum Computers Homomorphic Encryption ...

Inference Deep Dive

Extracting Information From Dynamic Programs

Extensive Standard Library

```
julia> @which sin(1.0)
sin(x::T) where T<:Union{Float32, Float64} in Base.Math at special/trig.jl:30

julia> using Base.Math: @horner

julia> begin
       # Coefficients in 13th order polynomial approximation on [0; π/4]
       #     sin(x) ≈ x + S1*x³ + S2*x⁵ + S3*x⁷ + S4*x⁹ + S5*x¹¹ + S6*x¹³
       # D for double, S for sin, number is the order of x-1
       const DS1 = -1.6666666666666324348e-01
       const DS2 = 8.333333333322489461 24e-03
       const DS3 = -1.984126982985794931 34e-04
       const DS4 = 2.755731370707006767 89e-06
       const DS5 = -2.505076025340686341 95e-08
       const DS6 = 1.589690995211550102 21e-10
       function sin_kernel_f64(y)
           y² =  y*y
           y⁴ =  y²*y²
           r  =  @horner(y², DS2, DS3, DS4) + y²*y⁴*@horner(y², DS5, DS6)
           y³ =  y²*y
           return y+y³*(DS1+y²*r)
       end
       end;
```

Avoiding “boilerplate” syntax: no types, interfaces, or lifetime constraints mentioned. The polynomial evaluation (Horner’s method) is also concisely and efficiently specified with a macro

Dynamic semantics

+

Static Analysis:

Just-Ahead-of-Time analysis to extract static properties of code

```
julia> code_lowered(sin_kernel_f64)
1 ─      nothing
   @ REPL[29]:12 within `sin_kernel_f64'
      y² = y * y
   @ REPL[29]:13 within `sin_kernel_f64'
      y⁴ = y² * y²
   @ REPL[29]:14 within `sin_kernel_f64'
      t@_3 = y²
   @ base/math.jl:101 within `@horner'
   %5 = t@_3
   %6 = Base.Math.muladd(t@_3, Main.DS4, Main.DS3)
      r@_4 = Base.Math.muladd(%5, %6, Main.DS2)

   %8  = r@_4
   %9  = y²
   %10 = y⁴
      t@_5 = y²
   @ base/math.jl:101 within `@horner'
      r@_6 = Base.Math.muladd(t@_5, Main.DS6, Main.DS5)

   %13 = r@_6
   %14 = %9 * %10 * %13
      r@_9 = %8 + %14
   @ REPL[29]:15 within `sin_kernel_f64'
      y³ = y² * y
   @ REPL[29]:16 within `sin_kernel_f64'
   %17 = y³
   %18 = y² * r@_9
   %19 = Main.DS1 + %18
   %20 = %17 * %19
   %21 = y + %20
      return %21
```

```
julia> methods(Base.Math.muladd)
# 12 methods for generic function "muladd":
[1] muladd(a::Float16, b::Float16, c::Float16)
      in Base at float.jl:406
[2] muladd(x::Float64, y::Float64, z::Float64)
      in Base at float.jl:404
[3] muladd(x::Float32, y::Float32, z::Flo
      in Base at float.jl:403
...
[10] muladd(x::T, y::T, z::T) where T<:Nu
      in Base at promotion.jl:397
[11] muladd(x::Number, y::Number, z::Numb
      in Base at promotion.jl:348
[12] muladd(x, y, z)
      in Base.Math at math.jl:1011

julia> methods(+)
# 161 methods for generic function "+":
[1] +(x::Bool, y::Bool)
      in Base at bool.jl:96
[2] +(a::Float16, b::Float16)
      in Base at float.jl:392
[3] +(x::Float32, y::Float32)
      in Base at float.jl:394
[4] +(x::Float64, y::Float64)
      in Base at float.jl:395
[5] +(c::BigInt, x::BigFloat)
      in Base.MPFR at mpfr.jl:408
[6] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt)
      in Base.GMP at gmp.jl:436
...
```

Everything is a virtual function call.

Oh no!

```
julia> code_lowered(sin_kernel_f64)
1 ─       nothing
   @ REPL[29]:12 within `sin_kernel_f64'
         y² = y * y
   @ REPL[29]:13 within `sin_kernel_f64'
         y⁴ = y² * y²
   @ REPL[29]:14 within `sin_kernel_f64'
         t@_3 = y²
   @ base/math.jl:101 within `@horner'
   %5 = t@_3
   %6 = Base.Math.muladd(t@_3, Main.DS4, Main.DS3)
         r@_4 = Base.Math.muladd(%5, %6, Main.DS2)
   %8  = r@_4
   %9  = y²
   %10 = y⁴
         t@_5 = y²
   @ base/math.jl:101 within `@horner'
         r@_6 = Base.Math.muladd(t@_5, Main.DS6, Main.DS5)
   %13 = r@_6
   %14 = %9 * %10 * %13
         r@_9 = %8 + %14
   @ REPL[29]:15 within `sin_kernel_f64'
         y³ = y² * y
   @ REPL[29]:16 within `sin_kernel_f64'
   %17 = y³
   %18 = y² * r@_9
   %19 = Main.DS1 + %18
   %20 = %17 * %19
   %21 = y + %20
         return %21
```

```
julia> @code_warntype sin_kernel_f64(0.3)
Body::Float64
1 ─       nothing
          (y² = y * y)::Float64
          (y⁴ = y² * y²)::Float64
          (t@_3 = y²)::Float64
   %5  = t@_3::Float64
   %6  = Base.Math.muladd(t@_3, Main.DS4, Main.DS3)::Float64
          (r@_4 = Base.Math.muladd(%5, %6, Main.DS2))::Float64
   %8  = r@_4::Float64
   %9  = y²::Float64
   %10 = y⁴::Float64
          (t@_5 = y²)::Float64
          (r@_6 = Base.Math.muladd(t@_5, Main.DS6, Main.DS5))::Float64
   %13 = r@_6::Float64
   %14 = (%9 * %10 * %13)::Float64
          (r@_9 = %8 + %14)::Float64
          (y³ = y² * y)::Float64
   %17 = y³::Float64
   %18 = (y² * r@_9)::Float64
   %19 = (Main.DS1 + %18)::Float64
   %20 = (%17 * %19)::Float64
   %21 = (y + %20)::Float64
         return %21::Float64

Variables
  #self#::Core.Compiler.Const(sin_kernel_f64)
  y::Float64 t@_3::Float64 r@_4::Float64 t@_5::Float64 r@_6::Float64
  y²::Float64 y⁴::Float64 r@_9::Float64 y³::Float64
```

But the compiler can devirtualize and simplify it.

And simple is *fast*.

Simple Auto-Differentiation: Forward Derivatives

```
julia> struct Dual{T<:Real} <: Real
         x::T    # the value
         dx::T   # the derivative at that point
       end;

# Some convenience constructors describing the derivative of a constant
julia> Dual(x) = Dual(x, zero(x));

julia> Dual{T}(x) where {T} = Dual{T}(x, zero(x));

# Compose with existing types of numbers
julia> Base.promote_rule(::Type{Dual{T}}, ::Type{S}) where {T, S} =
            Dual{Base.promote_type(T, S)};

# Define the base cases for arithmetic
julia> Base.:+(a::Dual{T}, b::Dual{T}) where {T} =
            Dual{T}(a.x + b.x, a.dx + b.dx);

julia> Base.:*(a::Dual{T}, b::Dual{T}) where {T} =
            Dual{T}(a.x * b.x, a.x*b.dx + a.dx*b.x);
```

Small abstraction to define a new subtype of real numbers

Simple Auto-Differentiation: Forward Derivatives

```
julia> wrt(x) = Dual(x, typeof(x)(1)) # helper for taking derivative “with-respect-to” this parameter

julia> sin_kernel_f64(wrt(.3)) |> dump # first derivative
Dual{Float64}
  x: Float64 0.29552020666133955
  dx: Float64 0.955336489125606

julia> sin_kernel_f64(wrt(wrt(.3))) |> dump # first and second derivatives
Dual{Dual{Float64}}
  x: Dual{Float64}
    x: Float64 0.29552020666133955
    dx: Float64 0.955336489125606
  dx: Dual{Float64}
    x: Float64 0.955336489125606
    dx: Float64 -0.2955202066613394

julia> sincos(0.3) # ground truth
(0.29552020666133955, 0.955336489125606)
```

And it works!

```
julia> @code_warntype sin_kernel_f64(wrt(0.3))
Body::Dual{Float64}
1 ─       nothing
          (y² = y * y)::Dual{Float64}
          (y⁴ = y² * y²)::Dual{Float64}
          (t@_3 = y²)::Dual{Float64}
    %5  = t@_3::Dual{Float64}
    %6  = Base.Math.muladd(t@_3, Main.DS4, Main.DS3)::Dual{Float64}
    %7  = Base.Math.muladd(%5, %6, Main.DS2)::Dual{Float64}
    %8  = y²::Dual{Float64}
    %9  = y⁴::Dual{Float64}
          (t@_4 = y²)::Dual{Float64}
    %11 = Base.Math.muladd(t@_4, Main.DS6, Main.DS5)::Dual{Float64}
    %12 = (%8 * %9 * %11)::Dual{Float64}
          (r = %7 + %12)::Dual{Float64}
          (y³ = y² * y)::Dual{Float64}
    %15 = y³::Dual{Float64}
    %16 = (y² * r)::Dual{Float64}
    %17 = (Main.DS1 + %16)::Dual{Float64}
    %18 = (%15 * %17)::Dual{Float64}
    %19 = (y + %18)::Dual{Float64}
          return %19::Dual{Float64}
```

Custom user types no different from “builtin” ones:
It’s still inferred fully!

```
julia> @code_native sin_kernel_f64(wrt(wrt(.3)))
        .text
        vmovupd (%rsi), %ymm21
        vmulsd  %xmm21, %xmm21, %xmm1
        vpermilpd      $1, %xmm21
        vmulsd  %xmm9, %xmm21, %x
        vaddsd  %xmm2, %xmm2, %xm
        vextractf32x4  $1, %ymm2
        vmulsd  %xmm11, %xmm21, %
        vpermilpd      $1, %xmm1
        vmulsd  %xmm8, %xmm21, %x
        vmulsd  %xmm11, %xmm9, %x
        vaddsd  %xmm5, %xmm4, %xm
        vaddsd  %xmm3, %xmm3, %xm
        vaddsd  %xmm4, %xmm4, %xm
        vmulsd  %xmm1, %xmm1, %xmm
        vmulsd  %xmm2, %xmm1, %xmm6
        vaddsd  %xmm6, %xmm6, %xmm10
        vmulsd  %xmm3, %xmm1, %xmm6
        vmulsd  %xmm16, %xmm1, %xmm7
        vmulsd  %xmm3, %xmm2, %xmm5
        vaddsd  %xmm7, %xmm5, %xmm5
        vaddsd  %xmm6, %xmm6, %xmm14
        vaddsd  %xmm5, %xmm5, %xmm13
        movabsq $140702207837800, %rax   # imm = 0x7FF7C91E0668
        vmovsd  (%rax), %xmm5            # xmm5 = mem[0],zero
        vmulsd  %xmm5, %xmm1, %xmm15
        vxorpd  %xmm20, %xmm20, %xmm20
        vmulsd  %xmm20, %xmm1, %xmm7
        vmulsd  %xmm5, %xmm2, %xmm4
        vaddsd  %xmm4, %xmm7, %xmm17
        vmulsd  %xmm20, %xmm2, %xmm4
        vaddsd  %xmm4, %xmm7, %xmm18
        vmulsd  %xmm5, %xmm3, %xmm4
```

```
movabsq  $140702207837800, %rax   # imm = 0x7FF7C91E0668
vmovsd   (%rax), %xmm5            # xmm5 = mem[0],zero
vmulsd   %xmm5, %xmm1, %xmm15
vxorpd   %xmm20, %xmm20, %xmm20
vmulsd   %xmm20, %xmm1, %xmm7
vmulsd   %xmm5, %xmm2, %xmm4
vaddsd   %xmm4, %xmm7, %xmm17
```

And it’s fast!

Eliminated runtime overhead of dynamic definition!

```
julia> function mysum(a)
           s = zero(eltype(a))
           for x in a
               s += x
           end
           return s
       end;
```

Just a “simple” for-loop

```
Variables
  #self#::Core.Compiler.Const(mysum)
  a::UnitRange{Int64}
  s::Int64
  @_4::Union{Nothing, Tuple{Int64,Int64}}
  x::Int64
```

But dynamic, like what

```
julia> @code_warntype mysum(1:10)
Body::Int64
    @ REPL[86]:2 within `mysum'
1 ─ %1  = Main.eltype(a)::Core.Compiler.Const(Int64)
│         (s = Main.zero(%1))::Int64
    @ REPL[86]:3 within `mysum'
    %3  = a::UnitRange{Int64}
          (@_4 = Base.iterate(%3))::Union{Nothing, Tuple{Int64,Int64}}
    %5  = (@_4 === nothing)::Bool
    %6  = Base.not_int(%5)::Bool
└──       goto #4 if not %6
2 ┄ %8  = @_4::Tuple{Int64,Int64}::Tuple{Int64,Int64}
│         (x = Core.getfield(%8, 1))::Int64
    %10 = Core.getfield(%8, 2)::Int64
    @ REPL[86]:4 within `mysum'
          (s = s + x)::Int64
          (@_4 = Base.iterate(%3, %10))::Union{Nothing, Tuple{Int64,Int64}}
    %13 = (@_4 === nothing)::Bool
    %14 = Base.not_int(%13)::Bool
└──       goto #4 if not %14
3 ─       goto #2
    @ REPL[86]:6 within `mysum'
4 ┄       return s::Int64
```

```
julia> sigma(n) = mysum(1:n);
julia> @code_llvm sigma(10)
define i64 @julia_sigma_12697(i64) {
top:
  %1 = icmp sgt i64 %0, 0
  br i1 %1, label %L7.L12_crit_edge, label %L29

L7.L12_crit_edge:                                 ; preds = %top
  %2 = shl nuw i64 %0, 1
  %3 = add nsw i64 %0, -1
  %4 = zext i64 %3 to i65
  %5 = add nsw i64 %0, -2
  %6 = zext i64 %5 to i65
  %7 = mul i65 %4, %6
  %8 = lshr i65 %7, 1
  %9 = trunc i65 %8 to i64
  %10 = add i64 %2, %9
  %11 = add i64 %10, -1
  br label %L29

L29:                                              ; preds = %L7.L12_crit_edge, %top
  %value_phi9 = phi i64 [ 0, %top ], [ %11, %L7.L12_crit_edge ]
  ret i64 %value_phi9
}

julia> f() = sigma(10);
julia> @code_llvm f()
define i64 @julia_f_12704() {
top:
  ret i64 55
}
```

Not even a loop, like woah!

Gone, all gone!

~~Machine Learning~~
Differentiable Programming

Fashionable Modelling with Flux (arXiv:1811.01457)

Building a Language and Compiler for Machine Learning (julialang:ml-language-compiler)

Zygote.jl - AD is a compiler problem

```
function foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c = tanh.(b)
    r = a + c
    return r
end
```


```
function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, 𝒥tanh = ∇tanh.(b)
    a + c, function (Δr)
        Δc = Δr, Δa = Δr
        (Δtanh, Δb) = 𝒥tanh(Δc)
        (ΔY, Δx) = (Δb * x', Y' * Δb)
        (ΔZ = Δa * x', Δx += Z' * Δa)
        (ΔW = ΔZ * Y', ΔY = W' * ΔZ)
        (nothing, ΔW, ΔY, Δx)
    end
end
```

Note: Simplified to assume *,+ are compiler primitives (Not the case in the original implementation)

Note: Reverse-AD model (actual implementation uses mixed)

Don't Unroll Adjoint: Differentiating SSA-Form Programs (arXiv:1810.07951)

Zygote.jl - AD is a compiler problem

```
function foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c = tanh.(b)
    r = a + c
    return r
end
```


```
function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, 𝒥tanh = ∇tanh.(b)
    a + c, function (Δr)
        Δc = Δr, Δa = Δr
        (Δtanh, Δb) = 𝒥tanh(Δc)
        (ΔY, Δx) = (Δb * x', Y' * Δb)
        (ΔZ = Δa * x', Δx += Z' * Δa)
        (ΔW = ΔZ * Y', ΔY = W' * ΔZ')
        (nothing, ΔW, ΔY, Δx)
    end
end
```

In the backwards pass

- Inputs become Outputs
- Outputs become Inputs

Zygote.jl - AD is a compiler problem

```
function foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c = tanh.(b)
    r = a + c
    return r
end
```


```
function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, 𝒥tanh = ∇tanh.(b)
    a + c, function (Δr)
        Δc = Δr, Δa = Δr
        (Δtanh, Δb) = 𝒥tanh(Δc)
        (ΔY, Δx) = (Δb * x', Y' * Δb)
        (ΔZ = Δa * x', Δx += Z' * Δa)
        (ΔW = ΔZ * Y', ΔY = W' * ΔZ')
        (nothing, ΔW, ΔY, Δx)
    end
end
```

The compiler builds the “tape” for us

Zygote.jl - AD is a compiler problem

```
struct 𝒥_foo
    W
    Y
    x
    Z
    𝒥tanh
end

(::𝒥_foo)(Δr) = ....

function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, 𝒥thanh = ∇tanh.(b)
    r = a + c
    (r, 𝒥_foo(W, Y, x, Z, 𝒥thanh))
end
```

Closure conversion

```
function ∇foo(W, Y, x)
    Z = W * Y
    a = Z * x
    b = Y * x
    c, 𝒥tanh = ∇tanh.(b)
    a + c, function (Δr)
        Δc = Δr, Δa = Δr
        (Δtanh, Δb) = 𝒥tanh(Δc)
        (ΔY, Δx) = (Δb * x', Y' * Δb)
        (ΔZ = Δa * x', Δx += Z' * Δa)
        (ΔW = ΔZ * Y', ΔY = W' * ΔZ')
        (nothing, ΔW, ΔY, Δx)
    end
end
```

AD as a compiler problem

Simple (Syntactic) - but requires optimizing compiler for performance

Partial Specialization (/DCE) => Partial Gradients

Better Compiler Optimizations ⇔ Faster AD

Nested AD for free

Even Control Flow!

```
function lstm(model, input, output)
    hidden = initial(model)
    total_loss = 0.
    for sample in take(zip(input, output), 50)
        (hidden, predicted) = model(sample, hidden)
        total_loss += loss(output, predicted)
    end
    total_loss
end
```

```
function ∇lstm(model, input, output)
    stack = Stack()
    hidden, 𝒥initial = ∇initial(model)
    total_loss = 0.
    for sample in take(zip(input, output), 50)
        (hidden, predicted), 𝒥model = ∇model(sample, hidden)
        (total_loss +), 𝒥loss = loss(output, predicted)
        push!(stack, (𝒥model, 𝒥loss))
    end
    total_loss, function(Δ)
        Δmodel_total = zero(typeof(model))
        Δhidden = nothing
        for (𝒥model, 𝒥loss) in reverse(stack)
            (Δloss, Δoutput, Δpredicted) = 𝒥loss(Δ)
            (Δmodel_it, Δsample, Δhidden) = 𝒥model(Δhidden, Δpredicted)
            Δmodel += Δmodel_it
            # (... For input and output, but let's
            # ignore those for simplicity)
        end
        Δmodel_total += 𝒥initial(Δhidden)
        (Δmodel_total, ...)
    end
end
```

~~Machine Learning~~
Compiler Backends

Fashionable Modelling with Flux (arXiv:1811.01457)

Building a Language and Compiler for Machine Learning (julialang:ml-language-compiler)

CUDAnative.jl - low level GPU Programming

```
function vadd(gpu, a, b, c)
  i = threadIdx().x + blockDim().x *
       ((blockIdx().x-1) + (gpu-1) * gridDim().x)
  @inbounds c[i] = a[i] + b[i]
  return
end

a, b, c = (CuArray(...) for _ in 1:3)
@cuda threads=length(a) vadd(1, a, b, c)
```

Provides:

- CUDA intrinsics
- SPMD Programming model
- GPU memory management

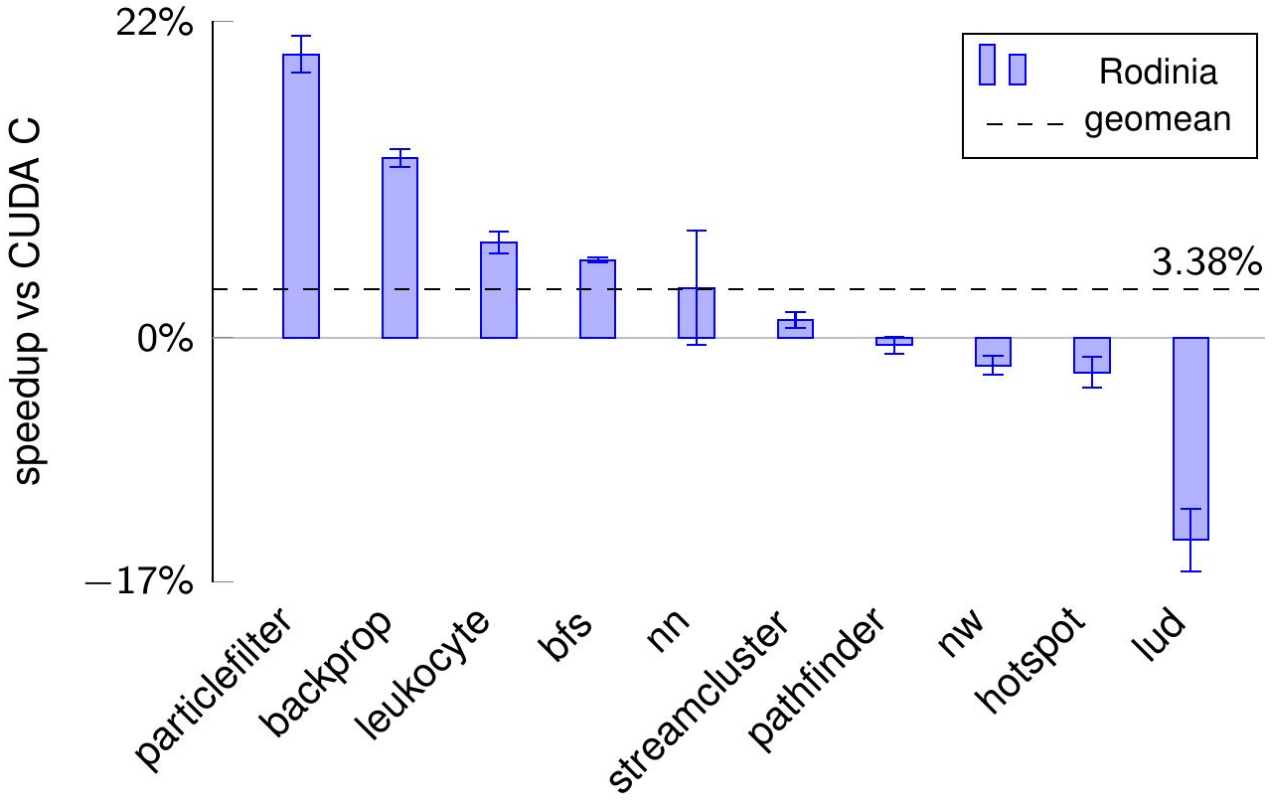
...

Effective Extensible Programming: Unleashing Julia on GPUs (arXiv:1712.03112)

```
julia> @device_code_ptx @cuda vadd(1, a, a, a)
//
// Generated by LLVM NVPTX Back-End
//

.visible .entry ptxcall_vadd_23(
        .param .u64 ptxcall_vadd_23_param_0,
        .param .align 8 .b8 ptxcall_vadd_23_param_1[16],
        .param .align 8 .b8 ptxcall_vadd_23_param_2[16],
        .param .align 8 .b8 ptxcall_vadd_23_param_3[16]
)
{
        mov.u32     %r1, %tid.x;
        mov.u32     %r2, %ntid.x;
        mov.u32     %r3, %ctaid.x;
        ...
}
```

Performance



Julia all the way down

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

```
W = randn(2, 10)
b = randn(2)

f(x) = softmax(W * x .+ b)
```

```
model = Chain(
  Dense(10, 5, σ),
  Dense(5, 2),
  softmax)
```

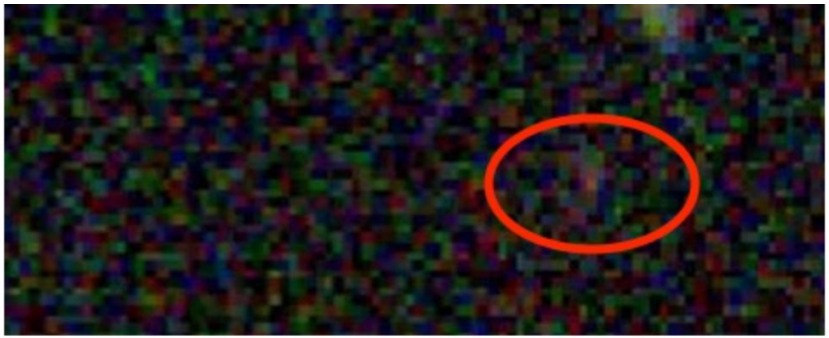
Scaling Up

2017

Cataloging the Visible Universe through Bayesian Inference at Petascale

Jeffrey Regier*, Kiran Pamnany†, Keno Fischer‡, Andreas Noack§, Maximilian Lam*, Jarrett Revels§, Steve Howard¶, Ryan Giordano¶, David Schlegel‖, Jon McAuliffe¶, Rollin Thomas‖, Prabhat‖

*Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
†Parallel Computing Lab, Intel Corporation
‡Julia Computing
§Computer Science and AI Laboratories, Massachusetts Institute of Technology
¶Department of Statistics, University of California, Berkeley
‖Lawrence Berkeley National Laboratory



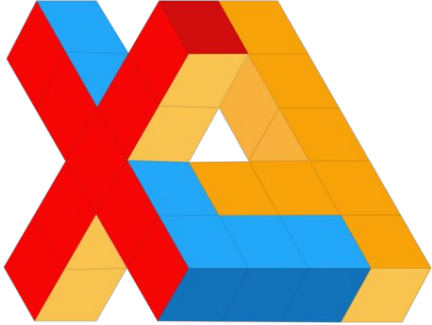
Most light sources are near the detection limit.

2019

Scaling GANs to ExaScale$_{16}$ on GPUs

(Work in Progress)

XLA.jl



```
struct XRTArray{T, Dims, N} <: AbstractArray{T, N}
        # ...
end
XRTScalar{T} = XRTArray{T, (), 0}

function +(a::XRTScalar{T},
           b::XRTScalar{T}) where T
    GenericHloOp{:add}(T, ())(a, b)
End

+(a::XRTScalar, b::XRTScalar) =
    +(promote(a, b)...)
```

Shape information represented in the type system

Declare primitives using multiple dispatch

Re-use vast amount of existing julia code (e.g. promotion, array abstractions, broadcast machinery)

Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs (arXiv:1810.09868)

Generic Code makes retargeting easy

In Julia Standard Library

```
function mapreduce(f, op, A; dims=:)
    ...
end

add_sum(x, y) = x + y
sum(f, a) = mapreduce(f, add_sum, a)
sum(a) = sum(identity, a)
```

Coverage of a large number of functions from a couple generic abstractions

Existing Julia array primitives map well to XLA primitives

In XLA.jl

```
function Base.mapreduce(f, op, A::XRTArray; dims=:)
   dt = dims_tuple(A, dims)
   res = HloReduce{Core.Typeof(op)}(dt)(op,
       HloMap{Core.Typeof(f)}()(f, A),
       XRTArray(zero(eltype(A)))
   )
   if dims != (:)
       # Put back the dimensions that HloReduce dropped;
       # Julia semantics require this.
       res = HloReshape(
             reduced_dimensions_collapes(size(A), dims))(res)
   end
   return res
end
```

Essentially: An embedding of HLO in Julia IR

dense(W, x, b) = W * x .+ b

Before Julia-level optimization

```
@code_typed_xla opt=false dense(W, x, b)
 %1 = Base.Broadcast.materialize::Const(
        materialize, false)
 %2 = Base.Broadcast.broadcasted::Const(
        broadcasted, false)
 %3 = (W * x)::XRTArray{Float32,(10,),1}
 %4 = (%2)(Main.:+, %3, b)::Broadcasted{
        XRTArrayStyle{1},Nothing,typeof(+),
        Tuple{XRTArray{Float32,(10,),1},
        XRTArray{Float32,(10,),1}}}
 %5 = (%1)(%4)::XRTArray{Float32,(10,),1}
      return %5
```

After Julia-level optimization

```
@code_typed_xla opt=true dense(W, x, b)
 %1 = invoke HloDot(XLA.DimNums{1,1,0,0}(
    (1,), (0,), (), ()))(_2, _3)::XRTArray
        Float32,(10,),1}
 %2 = invoke HloMap{typeof(+)}(+)(
    %1, _4)::XRTArray{Float32,(10,),1}
      return %2
```

Statements correspond 1:1 to HLO

Trivial to convert to HLO .pb from here

XLA.jl

Compilation

Full Julia Semantics (over XLA primitives)

Control Flow

Takes the place of LLVM

Re-use Inference/Julia Optimizer / AD

~1000 LOC

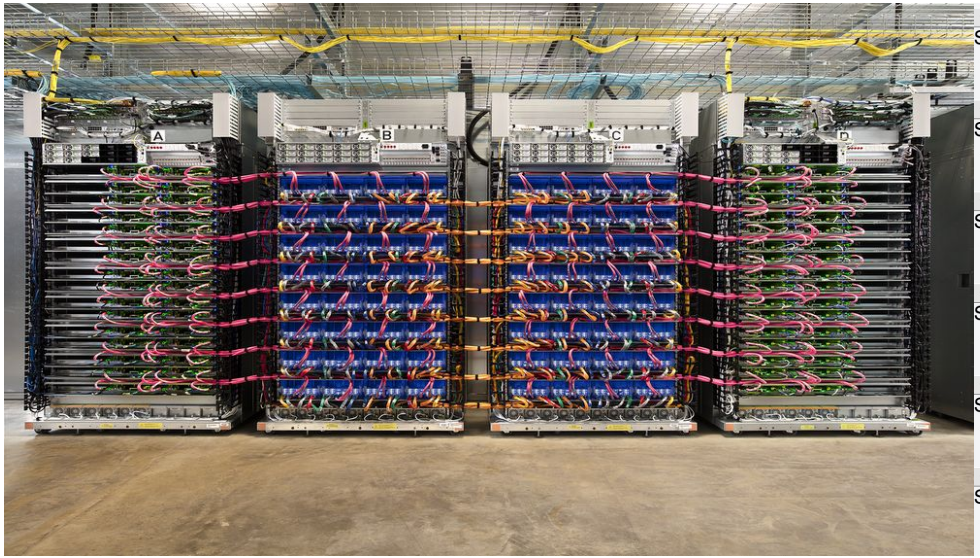
No XLA-specific changes to core julia

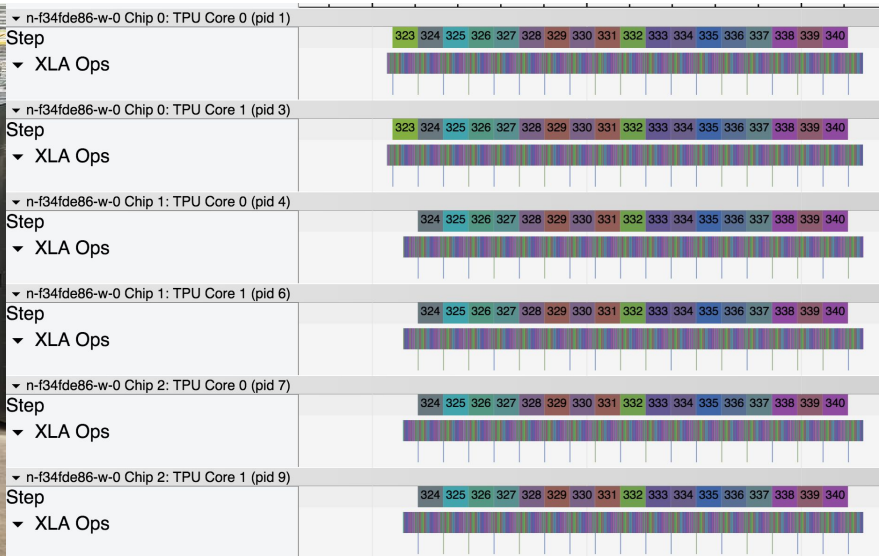
Template for Julia as a frontend to future/experimental IRs/Backends

Runs Well on TPU

Performance on par with TF

Scales to pods (512 TPU cores - 4.3 PF_{16}/s on ResNet50)





This leads me to claim that, from now on, a main goal in designing a language should be to plan for growth. The language must start small, and the language must grow as the set of users grows.

Guy Steele, “Growing a language”, 1998

Growing a language
(youtube:_ahvzDzKdB0)

www.cs.virginia.edu/~evans/cs655/readings/steele.pdf