Just compile it: High-level programming on the GPU with Julia

Tim Besard (@maleadt)

julia

Yet another high-level language?

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end

julia> mandel(complex(.3, -.6))
14
```

Yet another high-level language?

Typical features

Dynamically typed, high-level syntax

Open-source, permissive license

Built-in package manager

Interactive development

Unusual features

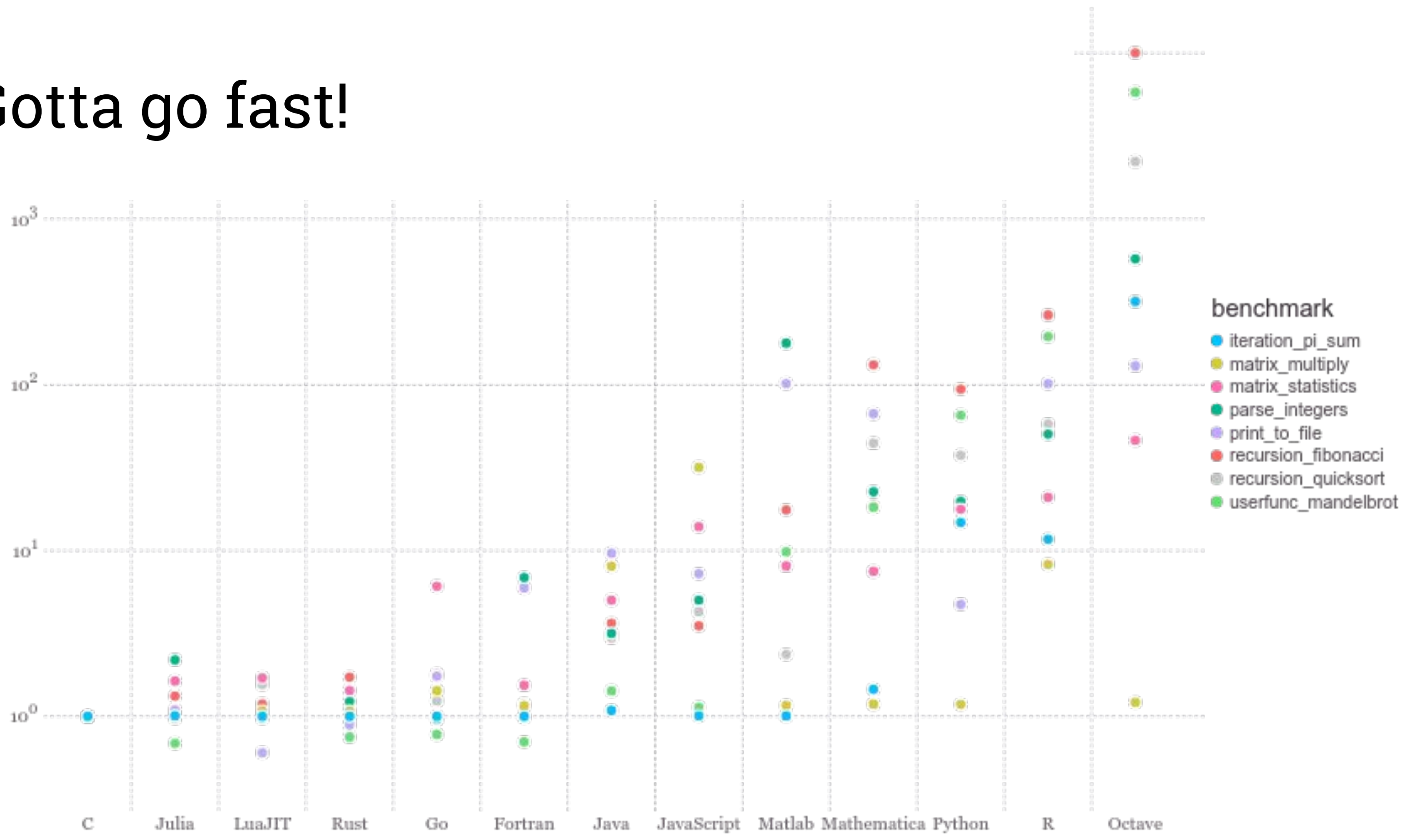
Great performance!

JIT AOT-style compilation

Most of Julia is written in Julia

Reflection and metaprogramming

Gotta go fast!



Avoid runtime uncertainty

1. Sophisticated type system
2. Type inference
3. Multiple dispatch
4. Specialization
5. JIT compilation

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

Avoid runtime uncertainty

1. Sophisticated type system
2. Type inference
3. Multiple dispatch
4. **Specialization**
5. **JIT compilation**

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

Dynamic semantics + Static analysis

```
julia> function mandel(z)
           c = z
           maxiter = 80
           for n = 1:maxiter
               if abs(z) > 2
                   return n-1
               end
               z = z^2 + c
           end
           return maxiter
       end

julia> mandel(UInt32(1))
2
```

```
julia> methods(abs)
# 13 methods for generic function "abs":
[1] abs(x::Float64) in Base at float.jl:522
[2] abs(x::Float32) in Base at float.jl:521
[3] abs(x::Float16) in Base at float.jl:520
…
[13] abs(z::Complex) in Base at complex.jl:260
```

Everything is a virtual function call?

Dynamic semantics + Static analysis

```
julia> function mandel(z::UInt32)
          c::UInt32 = z
          maxiter::Int = 80
          for n::Int = 1:maxiter
              if abs(z)::UInt32 > 2
                  return (n-1)::Int
              end
              z = (z^2 + c)::UInt32
          end
          return maxiter::Int
      end::Int

julia> @code_typed  mandel(UInt32(1))
```

Devirtualized!

Dynamic semantics + Static analysis

```
julia> function mandel(z::UInt32)
           c::UInt32 = z
           maxiter::Int = 80
           for n::Int = 1:maxiter
               if abs(z)::UInt32 > 2
                   return (n-1)::Int
               end
               z = (z^2 + c)::UInt32
           end
           return maxiter::Int
       end::Int

julia> @code_llvm   mandel(UInt32(1))
```

```
define i64 @julia_mandel_1(i32) {
top:
  %1 = icmp ult i32 %0, 3
  br i1 %1, label %L11.lr.ph, label %L9
L11.lr.ph:
  br label %L11
L9:
  %value_phi.lcssa =
    phi i64 [ 0, %top ], [ %value_phi7, %L23 ], [ 80, %L11 ]
  ret i64 %value_phi.lcssa
L11:
  %value_phi28 = phi i32 [ %0, %L11.lr.ph ], [ %5, %L23 ]
  %value_phi7 = phi i64 [ 1, %L11.lr.ph ], [ %3, %L23 ]
  %2 = icmp eq i64 %value_phi7, 80
  br i1 %2, label %L9, label %L23
L23:
  %3 = add nuw nsw i64 %value_phi7, 1
  %4 = mul i32 %value_phi28, %value_phi28
  %5 = add i32 %4, %0
  %6 = icmp ult i32 %5, 3
  br i1 %6, label %L11, label %L9
}
```

Dynamic semantics + Static analysis

```
julia> function mandel(z::UInt32)
           c::UInt32 = z
           maxiter::Int = 80
           for n::Int = 1:maxiter
               if abs(z)::UInt32 > 2
                   return (n-1)::Int
               end
               z = (z^2 + c)::UInt32
           end
           return maxiter::Int
       end::Int

julia> @code_native mandel(UInt32(1))
```

```
.text
        xorl    %eax, %eax
        cmpl    $2, %edi
        ja      L36
        movl    %edi, %ecx
        nopl    (%rax)
L16:
        cmpq    $79, %rax
        je      L37
        imull   %ecx, %ecx
        addl    %edi, %ecx
        addq    $1, %rax
        cmpl    $3, %ecx
        jb      L16
L36:
        retq
L37:
        movl    $80, %eax
        retq
        nopl    (%rax,%rax)
```

NVIDIA
CUDA®

julia

WA

RO
Cm
ON

Retargeting the language

1. Powerful dispatch
2. Small runtime library
3. Staged metaprogramming
4. Built on LLVM

Retargeting the language

1. **Powerful dispatch**
2. Small runtime library
3. Staged metaprogramming
4. Built on LLVM

```
lmul!(n::Number, A::GPUArray{Float64}) =
  ccall(:cublasDscal, ...)

sin(x::Float32) =
  ccall((:sinf, :libm), Cfloat, (Cfloat,) x)

@context GPU
@contextual(GPU) sin(x::Float32) =
  ccall((:__nv_sinf, :libdevice), Cfloat, (Cfloat,) x)
```

Retargeting the language

1. Powerful dispatch

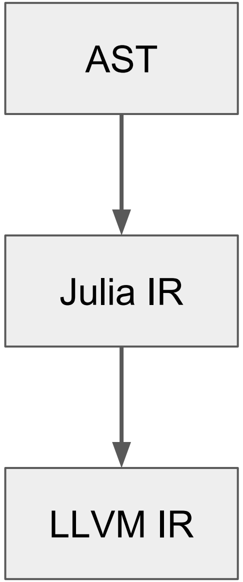
2. **Small runtime library**

3. Staged metaprogramming

4. Built on LLVM

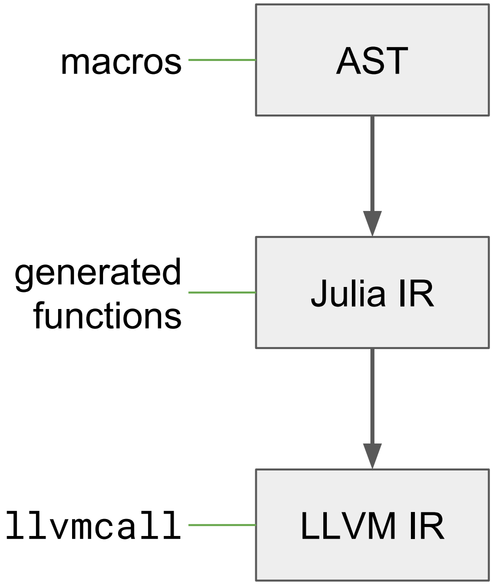
Retargeting the language

1. Powerful dispatch
2. Small runtime library
3. **Staged metaprogramming**
4. Built on LLVM



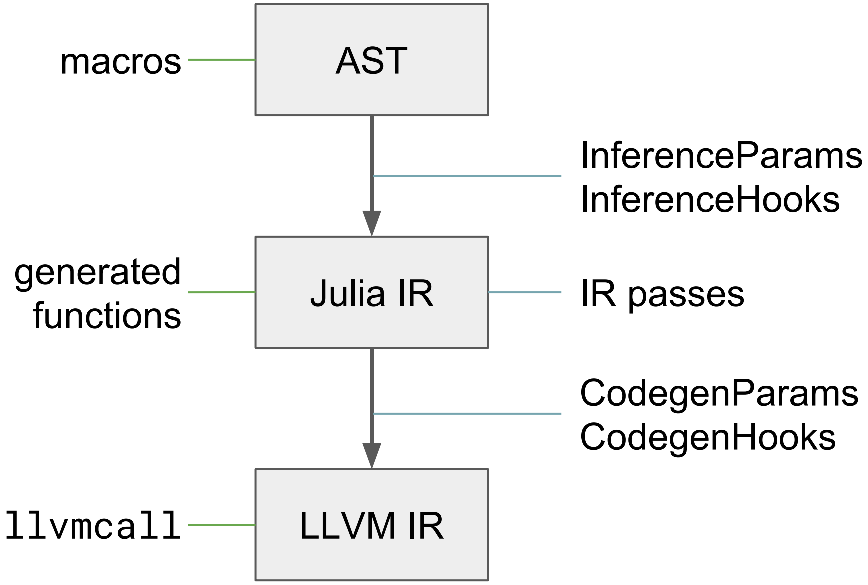
Retargeting the language

1. Powerful dispatch
2. Small runtime library
3. **Staged metaprogramming**
4. Built on LLVM



Retargeting the language

1. Powerful dispatch
2. Small runtime library
3. **Staged metaprogramming**
4. Built on LLVM



Retargeting the language

1. Powerful dispatch
2. Small runtime library
3. Staged metaprogramming
4. **Built on LLVM**

maleadt / **LLVM.jl**

Unwatch 4 | Unstar 40 | Fork 18

Code | Issues 5 | Pull requests 0 | Insights | Settings

Julia wrapper for the LLVM C API

Edit

julia | julia-library | llvm-bindings | llvm | Manage topics

578 commits | **10** branches | **38** releases | **1** environment | **11** contributors | View license

High Level LLVM Wrapper

```julia
using LLVM

mod = LLVM.Module("my_module")

param_types = [LLVM.Int32Type(), LLVM.Int32Type()]
ret_type = LLVM.Int32Type()
fun_type = LLVM.FunctionType(ret_type, param_types)
sum = LLVM.Function(mod, "sum", fun_type)

Builder() do builder
   entry = BasicBlock(sum, "entry")
   position!(builder, entry)

   tmp = add!(builder, parameters(sum)[1],
              parameters(sum)[2], "tmp")
   ret!(builder, tmp)

   println(mod)
   verify(mod)
end
```

```julia
julia> mod = LLVM.Module("test")
  ; ModuleID = 'test'
  source_filename = "test"

julia> test = LLVM.Function(mod, "test",
                LLVM.FunctionType(LLVM.VoidType()))
  declare void @test()

julia> bb = BasicBlock(test, "entry")
  entry:

julia> builder = Builder();
       position!(builder, bb)

julia> ret!(builder)
  ret void
```

High Level LLVM Wrapper

```julia
function runOnModule(mod::LLVM.Module)
   # ...
   return changed
end

pass = ModulePass("SomeModulePass", runOnModule)
ModulePassManager() do pm
   add!(pm, pass)
   run!(pm, mod)
end
```

High Level LLVM Wrapper

```
julia> using LLVM
julia> include("Kaleidoscope.jl")

julia> program = """def fib(x) {
                        if x < 3 then
                            1
                        else
                            fib(x-1) + fib(x-2)
                    }
                    def entry() {
                        fib(10)
                    }""";

julia> LLVM.Context() do ctx
           m = Kaleidoscope.generate_IR(program, ctx)
           Kaleidoscope.optimize!(m)
           Kaleidoscope.run(m, "entry")
       end
55.0
```

Descent into madness

```julia
function add(x::T, y::T) where {T <: Integer}
    return x + y
end

@test add(1, 2) == 3
```

Descent into madness

```julia
@generated function add(x::T, y::T) where {T <: Integer}
    return quote
        x + y
    end
end

@test add(1, 2) == 3
```

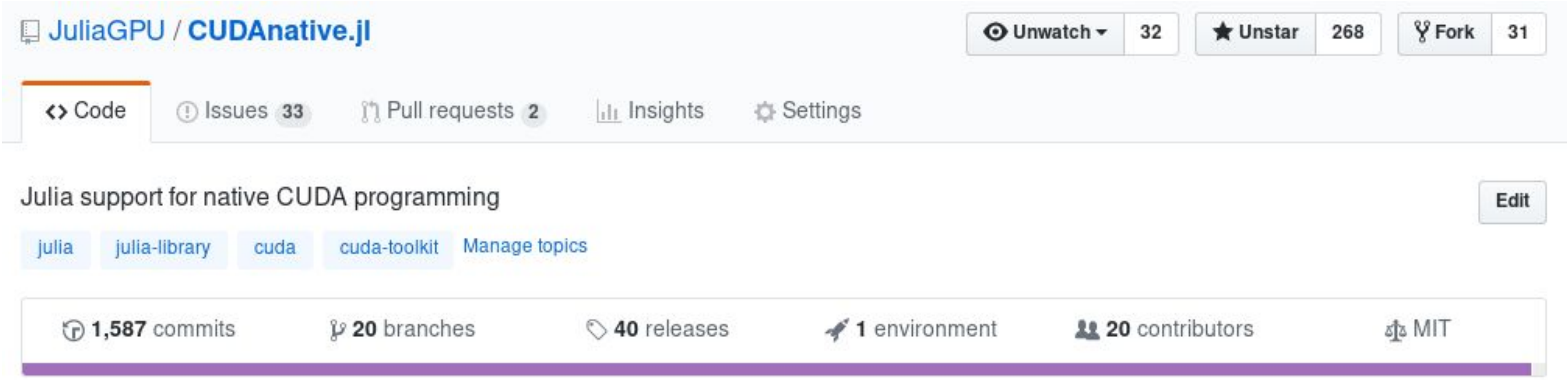
Descent into madness

```julia
@generated function add(x::T, y::T) where {T <: Integer}
    T_int = "i$(8*sizeof(T))"

    return quote
        Base.llvmcall($"""%rv = add $T_int %0, %1
                          ret $T_int %rv""", T,
                      Tuple{T, T}, x, y)
    end
end

@test add(1, 2) == 3
```

Descent into madness

```julia
@generated function add(x::T, y::T) where {T <: Integer}
    T_int = convert(LLVMType, T)

    param_types = LLVMType[T_int, T_int]
    llvm_f, _ = create_function(T_int, [T_int, T_int])
    mod = LLVM.parent(llvm_f)

    Builder() do builder
        entry = BasicBlock(llvm_f, "top")
        position!(builder, entry)
        rv = add!(builder, parameters(llvm_f)...)
        ret!(builder, rv)
    end

    call_function(llvm_f, T, Tuple{T, T}, :((x, y)))
end

@test add(1, 2) == 3
```

```
julia> @code_llvm add(UInt128(1),
                      UInt128(2))

define void @julia_add(i128* sret,
                       i128, i128) {
top:
 %3 = add i128 %2, %1
 store i128 %3, i128* %0, align 8
 ret void
}
```



- Just another package
  No special version of Julia
- 3000 LOC, 100% pure Julia

Extending the compiler

```
Ptr{T} → Base.unsafe_load → Core.Intrinsics.pointerref
```

```
primitive type DevicePtr{T,A}

@generated function Base.unsafe_load(p::DevicePtr{T,A}) where {T,A}
   T_ptr_with_as = LLVM.PointerType(eltyp, convert(Int, A))

   Builder(JuliaContext()) do builder
       # ...

       ptr_with_as = addrspacecast!(builder, ptr, T_ptr_with_as)
       ld = load!(builder, ptr_with_as)

       # ...
   end
end
```

Address Space	Memory Space
0	Generic
1	Global
2	Internal Use
3	Shared
4	Constant
5	Local

Show me what you got

```
pkg> add CUDAnative CuArrays

julia> using CUDAnative, CuArrays

julia> a = CuArray{Int}(undef, (2,2))
2×2 CuArray{Int64,2}:
0  0
0  0

julia> function memset(arr, val)
           arr[threadIdx().x] = val
           return
       end

julia> @cuda threads=4 memset(a, 1)

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

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

Show me what you got

```
pkg> add CUDAnative CuArrays

julia> using CUDAnative, CuArrays

julia> a = CuArray{Int}(undef, (2,2))
2×2 CuArray{Int64,2}:
0  0
0  0

julia> function memset(arr, val)
           arr[threadIdx().x] = val
           return
       end

julia> @cuda threads=4 memset(a, 1)

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

```
julia> @device_code_typed @cuda memset(a, 1)
...
2 ─ %10 = (Core.tuple)(%4)::Tuple{Int64}
│   %11 = (Base.getfield)(arr,
│           :shape)::Tuple{Int64,Int64}
│   %12 = (getfield)(%11, 1)::Int64
│   %13 = (getfield)(%11, 2)::Int64
│   %14 = (Base.mul_int)(%12, %13)::Int64
│   %15 = (Base.slt_int)(%14, 0)::Bool
│   %16 = (Base.ifelse)(%15, 0, %14)::Int64
│   %17 = (Base.sle_int)(1, %4)::Bool
│   %18 = (Base.sle_int)(%4, %16)::Bool
│   %19 = (Base.and_int)(%17, %18)::Bool
└──       goto #4 if not %19
...
) => Nothing
```

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

Show me what you got

```
pkg> add CUDAnative CuArrays

julia> using CUDAnative, CuArrays

julia> a = CuArray{Int}(undef, (2,2))
2×2 CuArray{Int64,2}:
0  0
0  0

julia> function memset(arr, val)
           arr[threadIdx().x] = val
           return
       end

julia> @cuda threads=4 memset(a, 1)

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

```
julia> @device_code_llvm  @cuda memset(a, 1)

define void @memset({ [2 x i64], { i64 } }, i64) {
entry:
 %7 = extractvalue { [2 x i64], { i64 } } %0, 1, 0
 %2 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
 %3 = zext i32 %2 to i64
 %4 = inttoptr i64 %7 to i64*
 %5 = getelementptr i64, i64* %4, i64 %3
 %6 = addrspacecast i64* %5 to i64 addrspace(1)*
 store i64 %1, i64 addrspace(1)* %6, align 8
 ret void
}
```

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

Show me what you got

```
pkg> add CUDAnative CuArrays

julia> using CUDAnative, CuArrays

julia> a = CuArray{Int}(undef, (2,2))
2×2 CuArray{Int64,2}:
0  0
0  0

julia> function memset(arr, val)
           arr[threadIdx().x] = val
           return
       end

julia> @cuda threads=4 memset(a, 1)

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

```
julia> @device_code_ptx  @cuda memset(a, 1)

.visible .entry memset(
    .param .align 8 .b8 a[24],
    .param .u64 val)
{
    .reg .b32       %r<2>;
    .reg .b64       %rd<6>;

    ld.param.u64    %rd1, [a+16];
    ld.param.u64    %rd2, [val];
    mov.u32         %r1, %tid.x;
    mul.wide.u32    %rd3, %r1, 8;
    add.s64         %rd4, %rd1, %rd3;
    cvta.to.global.u64      %rd5, %rd4;
    st.global.u64   [%rd5], %rd2;
    ret;
}
```

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

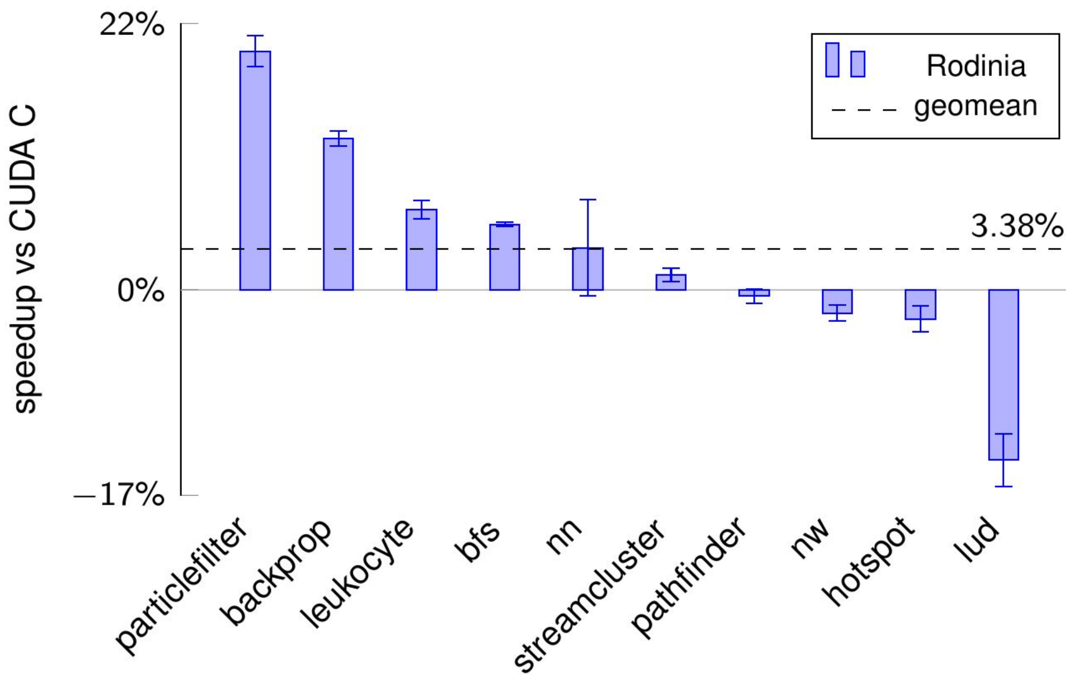
Show me what you got

```
pkg> add CUDAnative CuArrays

julia> using CUDAnative, CuArrays

julia> a = CuArray{Int}(undef, (2,2))
2×2 CuArray{Int64,2}:
0  0
0  0

julia> function memset(arr, val)
           arr[threadIdx().x] = val
           return
       end

julia> @cuda threads=4 memset(a, 1)

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

```
julia> @device_code_sass  @cuda memset(a, 1)

.text.memset:
    MOV R1, c[0x0][0x44];
    S2R R0, SR_TID.X;
    MOV32I R3, 0x8;
    MOV R4, c[0x0][0x158];
    MOV R5, c[0x0][0x15c];
    ISCADD R2.CC, R0, c[0x0][0x150], 0x3;
    IMAD.U32.U32.HI.X R3, R0, R3, c[0x0][0x154];
    ST.E.64 [R2], R4;
    EXIT;
```

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

It's fast!



22%
0%
−17%
speedup vs CUDA C
Rodinia
geomean
3.38%
particlefilter
backprop
leukocyte
bfs
nn
streamcluster
pathfinder
nw
hotspot
lud

It’s high-level!

```
julia> a = CuArray([1., 2., 3.])
3-element CuArray{Float64,1}:
1.0
2.0
3.0

julia> function square(a)
         i = threadIdx().x
         a[i] = a[i] ^ 2
         return
       end

julia> @cuda threads=length(a) square(a)

julia> a
3-element CuArray{Float64,1}:
1.0
4.0
9.0
```

It’s high-level!

```
julia> a = CuArray([1., 2., 3.])
3-element CuArray{Float64,1}:
1.0
2.0
3.0

julia> function apply(op, a)
         i = threadIdx().x
         a[i] = op(a[i])
         return
       end

julia> @cuda threads=length(a) apply(x->x^2, a)

julia> a
3-element CuArray{Float64,1}:
1.0
4.0
9.0
```

```
julia> @device_code_ptx @cuda apply(x->x^2, a)
apply(.param .b8 a[16])
{
        ld.param.u64    %rd1, [a+8];
        mov.u32         %r1, %tid.x;

        // index calculation
        mul.wide.u32    %rd2, %r1, 4;
        add.s64         %rd3, %rd1, %rd2;
        cvta.to.global.u64      %rd4, %rd3;

        ld.global.f32   %f1, [%rd4];
        mul.f32         %f2, %f1, %f1;
        st.global.f32   [%rd4], %f2;

        ret;
}
```

21st century array abstractions

```
julia> a = CuArray([1., 2., 3.])
3-element CuArray{Float64,1}:
1.0
2.0
3.0
```

```
julia> map!(x->x^2, a)
```

```
julia> a
3-element CuArray{Float64,1}:
1.0
4.0
9.0
```

21st century array abstractions

```
julia> a = CuArray([1., 2., 3.])
3-element CuArray{Float64,1}:
1.0
2.0
3.0
```

```
julia> a = a.^2
```

dot syntax

```
julia> a
3-element CuArray{Float64,1}:
1.0
4.0
9.0
```

21st century array abstractions

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

julia> f(x) = 3x^2 + 5x + 2
       f.(2 .* a .- 3)
3-element CuArray{Float64,1}:
  0.0
 10.0
 44.0


julia> using DualNumbers
julia> wrt(x) = Dual(x, typeof(x)(1))    # helper function, derive "with respect to"

julia> a = wrt.(a)
       f.(2 .* a .- 3)
3-element CuArray{Dual{Float64},1}:
  0.0 - 2.0ɛ
 10.0 + 22.0ɛ
 44.0 + 46.0ɛ
```

Fused broadcast!

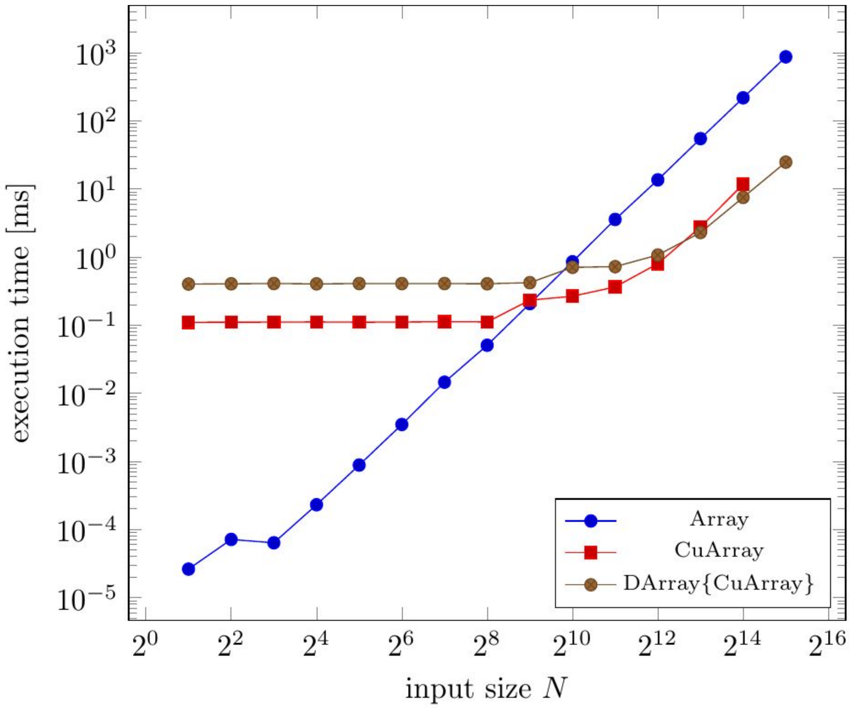
Dynamic Automatic Differentiation of GPU Broadcast Kernels ([arXiv:1810.08297](https://arxiv.org/abs/1810.08297))

Composability

```
julia> A = rand(4096,4096)
4096×4096 Array{Float64,2}
```

JuliaParallel / DistributedArrays.jl

```
julia> using DistributedArrays
julia> dA = distribute(A)
4096×4096 DArray{Float64,2,Array{Float64,2}}

julia> using CuArrays
julia> dgA = map_localparts(CuArray, dA)
4096×4096 DArray{Float64,2,CuArray{Float64,2}}

julia> dgA * dgA

julia> DistributedArrays.transfer(::CuArray)
```



Rapid Software Prototyping for Heterogeneous and Distributed Platforms

Composability → Separation of concerns

```
julia> map(x->x^2, CuArray([1 2 3]))
```

- *what* is computed
- *where* does it happen
- *how* is it computed

CUDAnative.jl	3000 LOC
GPUArrays.jl	1500 LOC
CuArrays.jl	1000 LOC (without libraries)

Wrapping up

- Julia: highly-dynamic language
  - Design → JIT AOT-style compilation
  - Accelerator programming
- Retargetable compiler
- High-level, high-performance (GPU) programming

Just compile it:

High-level programming on the GPU with Julia

Tim Besard (@maleadt)

Thanks to: James Bradbury, Valentin Churavy, Simon Danisch, Keno Fischer, Katharine Hyatt, Mike Innes, Jameson Nash, Andreas Noack, Jarrett Revels, and many others.

MIT
GHENT UNIVERSITY
Julia computing