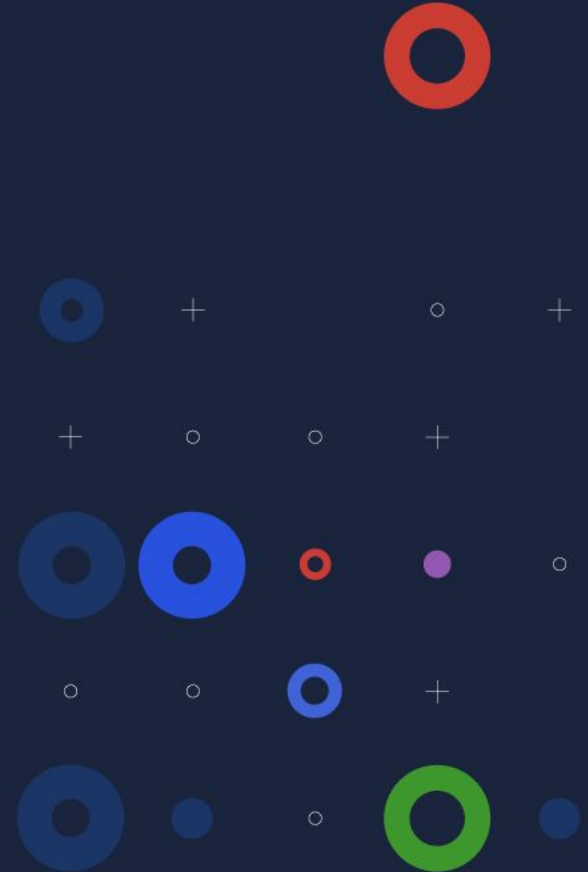
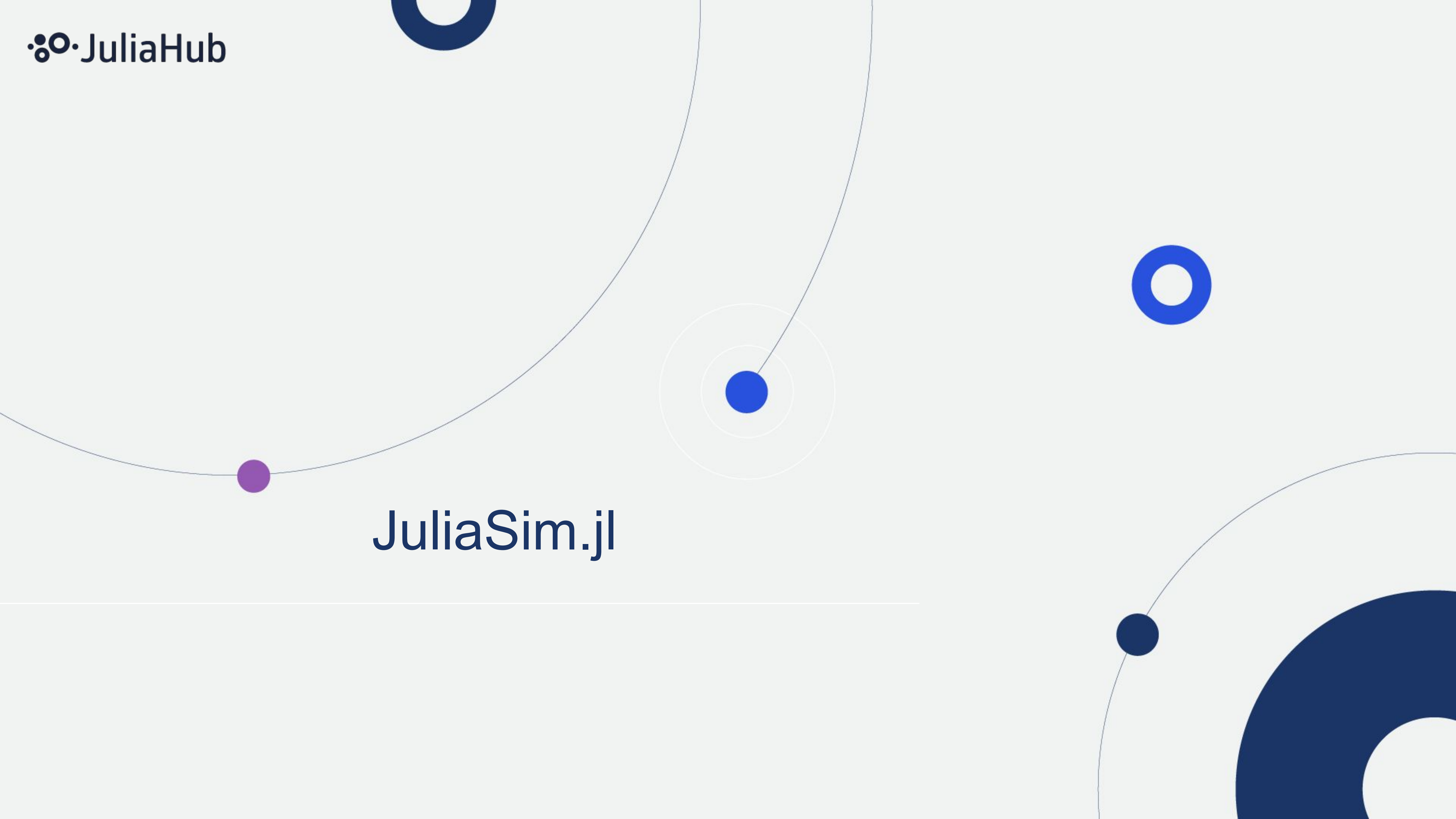
JuliaHub



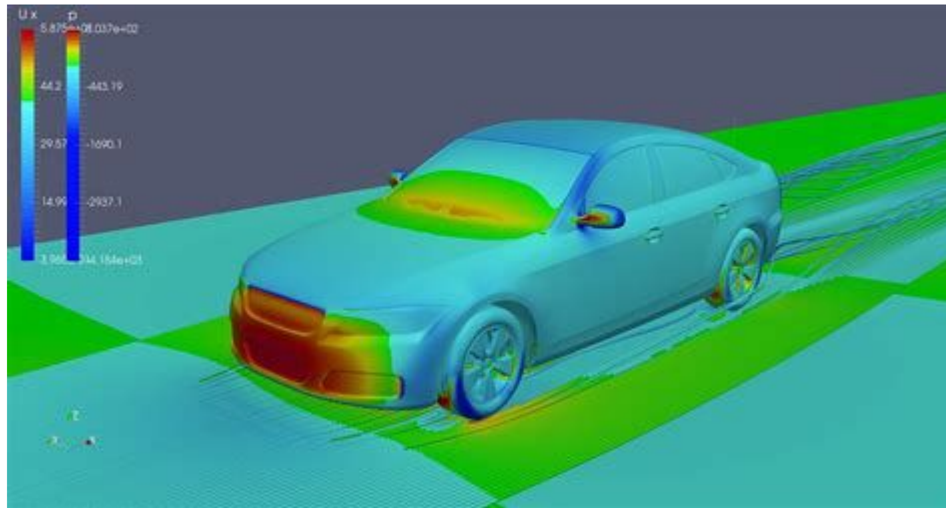
Advancing Scientific Discovery Across Architectures

Jacob Vaverka
Tim Besard


JuliaHub
JuliaSim.jl

Accelerating Scientific Computing

Julia SIM



Accelerating Scientific Computing: Gameplan

1. Facilitate the modeling process with better tooling
2. Generate performant simulation code by default
3. Integrate into all existing workflows

Empowering Modelers

- Formulate complex models in a declarative manner
- Structurally simplify systems of equations
- Automatically lower high-level models into efficient code
- Compose with pre-built models from JuliaSim Model Store

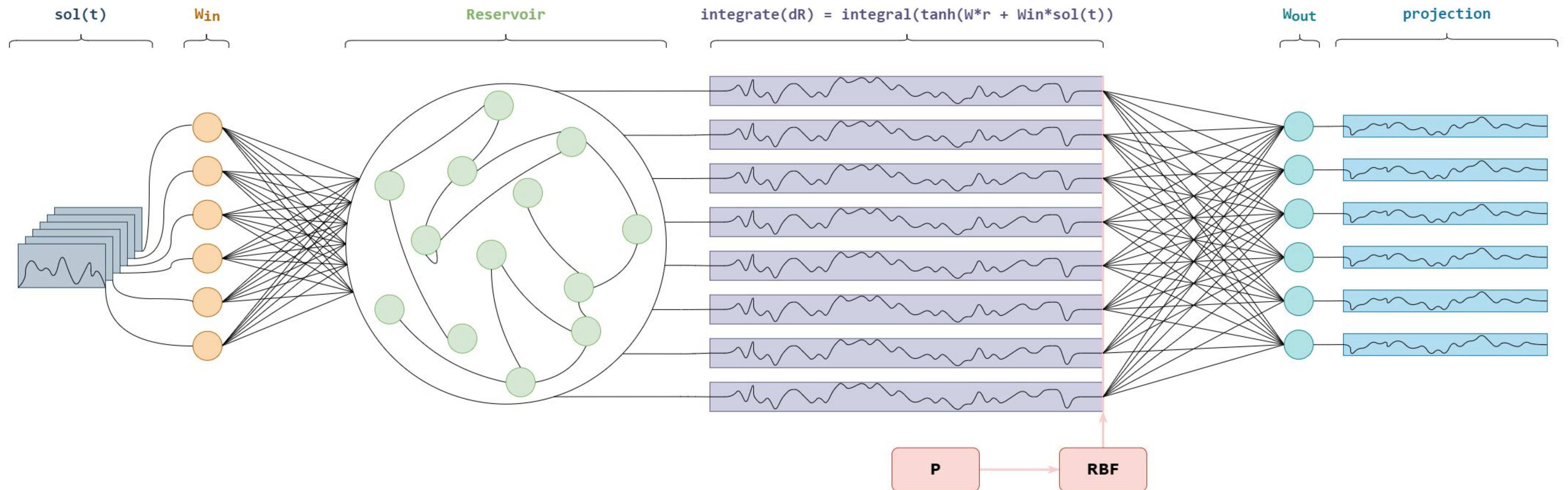
Accelerating Models

JuliaSIM

- Achieve 500x speedup with Surrogates
- High performance approximate models
- Latest techniques in scientific machine learning
- Latest techniques in model order reduction
- Continuous Time Echo-State Networks

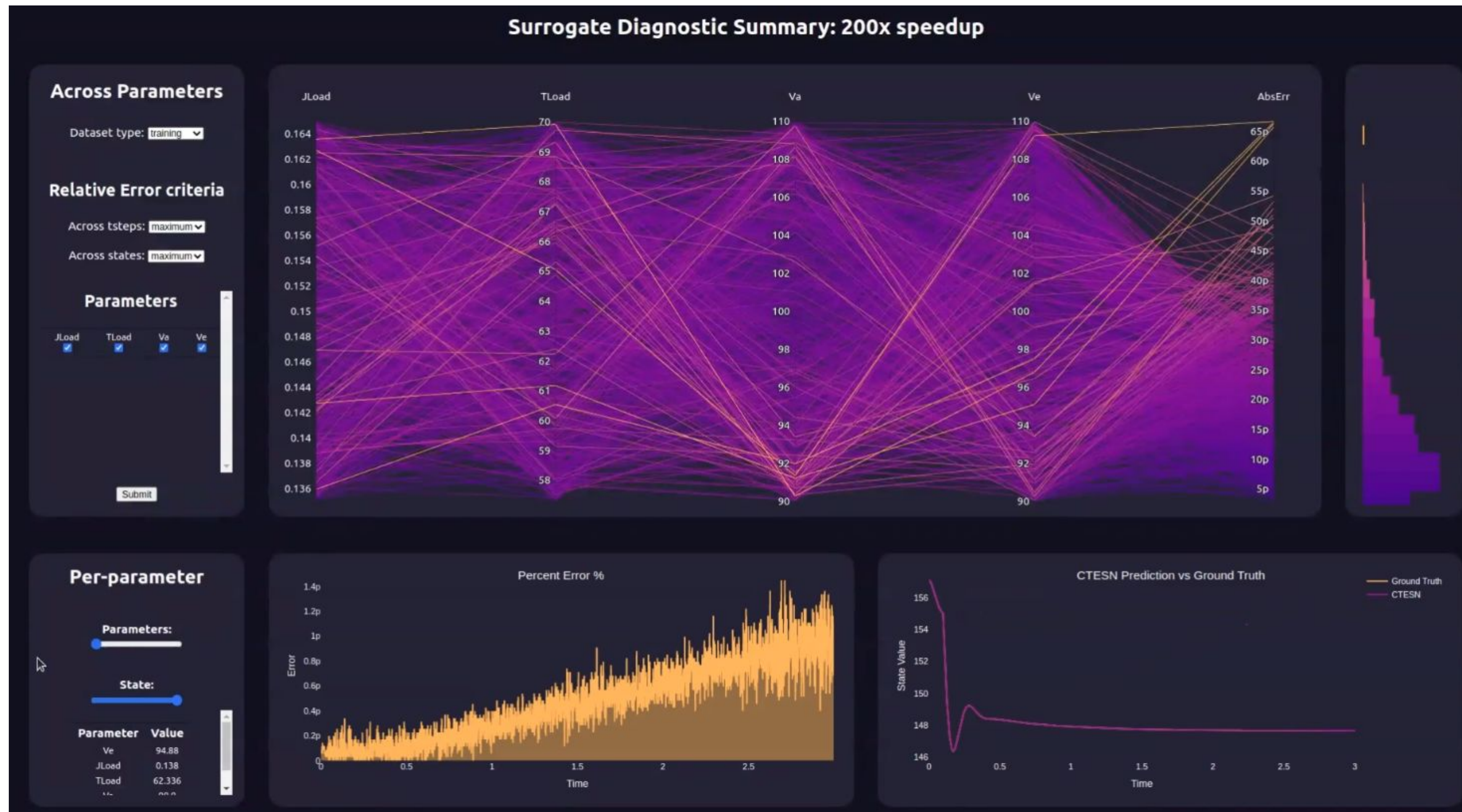
CTESNs

Julia SIM



Diagnose Surrogate Models

Julia SIM



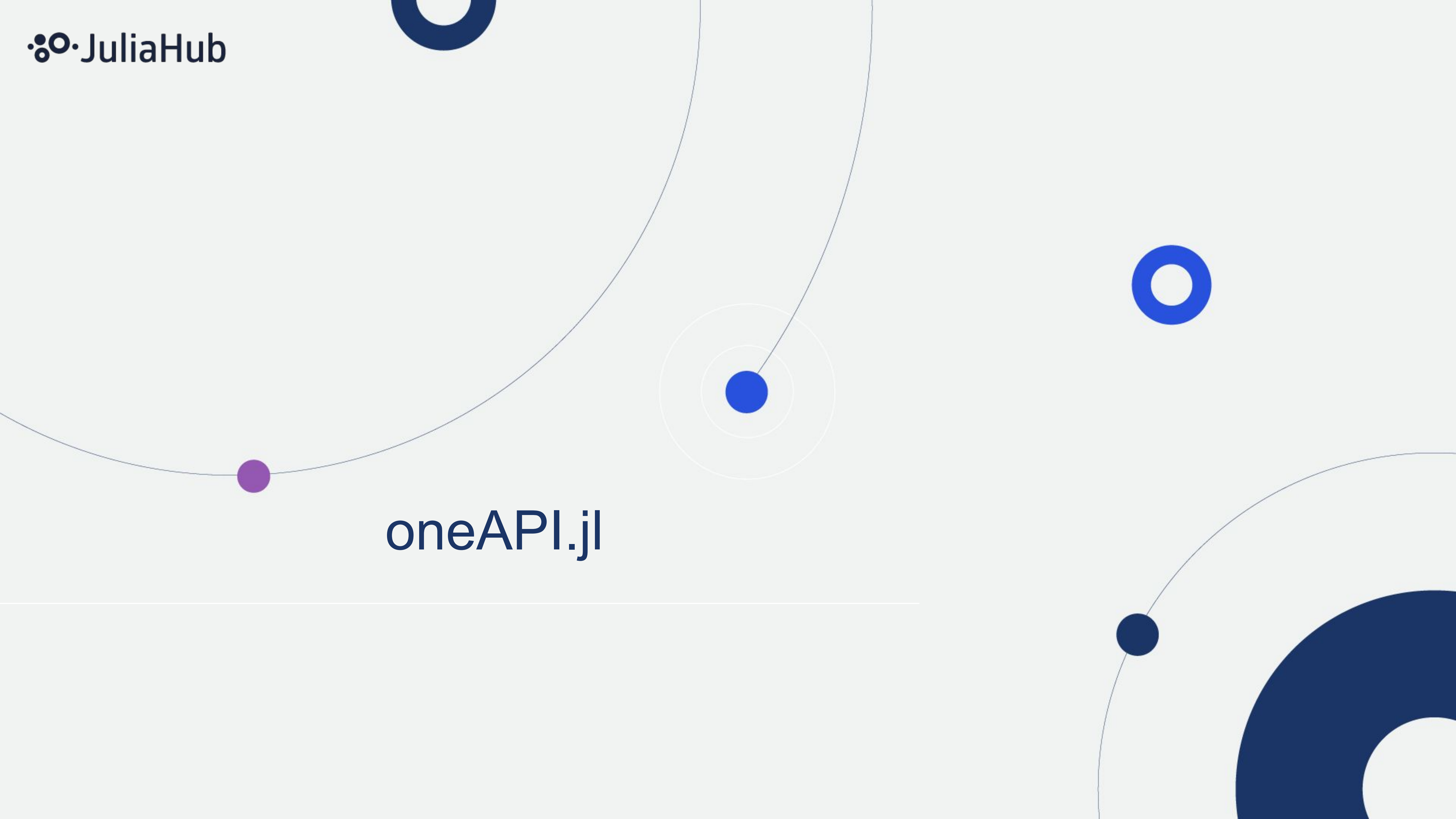
Keeping Compatibility

- Functional Mock-up Interface (FMI)
- Interface to exchange dynamic models
- Supported in +170 modeling tools
- Generate compatible binaries
  - Functional Mock-up Units (FMUs)

Accelerating Scientific Computing: Gameplan

Julia SIM

1. Facilitate the modeling process with better tooling →
   build more complex systems & iterate faster
2. Generate performant simulation code by default →
   surrogates & digital twins + diagnostics
3. Integrate into any & all existing workflows →
   maintaining compatibility with standard tooling


JuliaHub
oneAPI.jl

Why Julia?

High-level programming language designed for performance

Why Julia?

<u>High-level</u> programming language designed for performance

```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829
```

Why Julia?

High-level programming language
designed for performance

```
julia> data = (1, rand())
(1, 0.5326182923218289)

julia> sum(data)
1.532618292321829

julia> @code_llvm sum(data)
define double @julia_sum({ i64, double }* nocapture nonnull readonly align 8 dereferenceable(16) %0) #0 {
top:
  %1 = getelementptr inbounds { i64, double }, { i64, double }* %0, i64 0, i32 0
  %2 = getelementptr inbounds { i64, double }, { i64, double }* %0, i64 0, i32 1
  %3 = load i64, i64* %1, align 8
  %4 = sitofp i64 %3 to double
  %5 = load double, double* %2, align 8
  %6 = fadd double %5, %4
  ret double %6
}
```

Why Julia?

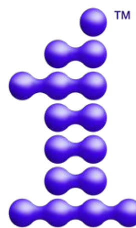
High-level programming language
designed for performance

```
julia> data = (1, rand())
(1, 0.532618292321829)

julia> sum(data)
1.532618292321829

julia> @code_native debuginfo=:none sum(data)
_julia_sum:                                    ; @julia_sum
; %bb.0:                                       ; %top
     ldp  d0, d1, [x0]
     scvtf     d0, d0
     fadd d0, d1, d0
     ldr  d1, [x0, #16]
     fadd d0, d0, d1
     ret
```

GPU support in Julia

GPU-enabled applications

- Flux.jl (deep learning)
- CLiMA (ocean modeling)
- DifferentialEquations.jl
- Yao.jl (quantum information)
- …



oneAPI.jl

Metal.jl

AMDGPU.jl

CUDA.jl

Shared infrastructure

- GPUCompiler.jl
- GPUArrays.jl
- KernelAbstractions.jl
- …

Easy to get started

1. Download and unpack Julia 1.7: https://julialang.org/downloads/
2. Launch Julia and enter the the package manager
```
pkg> add oneAPI
```
3. Import and verify the oneAPI.jl package
```
julia> using oneAPI
Downloading artifacts: ...

julia> oneAPI.versioninfo()
Binary dependencies:
- NEO_jll: 22.17.23034+0
- libigc_jll: 1.0.11061+0
- …

1 device:
- Intel(R) Iris(R) Xe Graphics [0x9a49]
```

Automatic download of binary dependencies

Key components of oneAPI.jl

1. oneAPI Level Zero wrappers
2. Kernel programming
3. Array abstraction

Level Zero wrappers

```
julia> using oneAPI.oneL0

julia> drv = first(drivers())
ZeDriver(00000000-0000-0000-16eb-40b501030000, version 1.3.0)

julia> dev = first(devices(drv))
ZeDevice(GPU, vendor 0x8086, device 0x9a49): Intel(R) Iris(R) Xe Graphics [0x9a49]

julia> properties(dev)
(type = oneAPI.oneL0.ZE_DEVICE_TYPE_GPU,
 vendorId = 0x8086, deviceId = 0x9a49, subdeviceId = nothing,
 coreClockRate = 1300, maxMemAllocSize = 4294959104, maxHardwareContexts = 65536, …)
```

Level Zero wrappers

```
julia> queue = global_queue(context(), device())
ZeCommandQueue(...)

julia> execute!(queue) do list
           append_barrier!(list)
       end

julia> fence = ZeFence(queue)
       Base.isdone(fence)
false

julia> execute!(queue, fence) do list
           # signal the fence on completion
       end
       Base.isdone(fence)
true
```

Global (task-bound) device contexts

Array abstraction

```
julia> vec = oneArray([1])
1-element oneVector{Int64, oneAPI.oneL0.DeviceBuffer}:
 1
```

`oneArray` serves multiple purposes:

1. container for device memory
2. abstraction for data-parallel programming

```
julia> vec .+ 1
1-element oneVector{Int64, oneAPI.oneL0.DeviceBuffer}:
 2
```

Array abstraction

Higher-order functions:

```
julia> map(vec) do val
           val + 1
       end

julia> reduce(+, vec)
```

Linear algebra:

```
julia> using LinearAlgebra

julia> mat = oneArray(rand(Float32, 2, 2))
       mat * mat

julia> norm(mat)
```

Statistics:

```
julia> using Statistics

julia> mean(mat)

julia> std(mat)
```

Implemented using native Julia kernels!

Kernel programming

```julia
function vadd(a, b, c)
   function kernel(d_a, d_b, d_c)
       i = get_global_id()
       d_c[i] = d_a[i] + d_b[i]
       return
   end

   d_a = oneArray(a)
   d_b = oneArray(b)
   d_c = oneArray(c)

   len = prod(size(a))
   @oneapi items=len kernel(d_a, d_b, d_c)
   c .= Array(d_c)
end
```

Similar to CUDA.jl, AMDGPU.jl, ...
Differences with DPC++/SYCL

- OpenCL intrinsics
- Global semantics

Kernel programming

```julia
function vadd(a, b, c)
   function kernel(d_a, d_b, d_c)
       i = get_global_id()
       d_c[i] = d_a[i] + d_b[i]
       return
   end

   d_a = oneArray(a)
   d_b = oneArray(b)
   d_c = oneArray(c)

   len = prod(size(a))
   @oneapi items=len kernel(d_a, d_b, d_c)
   c .= Array(d_c)
end
```

```
julia> @device_code_llvm vadd([1], [2], [0])

define spir_kernel void @kernel(
    { { [1 x i64], i8 addrspace(1)* } }* byval,
    { { [1 x i64], i8 addrspace(1)* } }* byval,
    { { [1 x i64], i8 addrspace(1)* } }* byval
   ) local_unnamed_addr {
entry:
 %3 = call i64 @_Z13get_global_idj(i32 0)
 ...
 store i64 %14, i64 addrspace(1)* %18, align 8
 ret void
}
```

Kernel programming

```julia
function vadd(a, b, c)
   function kernel(d_a, d_b, d_c)
       i = get_global_id()
       d_c[i] = d_a[i] + d_b[i]
       return
   end

   d_a = oneArray(a)
   d_b = oneArray(b)
   d_c = oneArray(c)

   len = prod(size(a))
   @oneapi items=len kernel(d_a, d_b, d_c)
   c .= Array(d_c)
end
```

```
julia> @device_code_spirv vadd([1], [2], [0])

; SPIR-V
; Version: 1.0
; Bound: 45
; Schema: 0
             OpCapability Addresses
             OpCapability Linkage
             OpCapability Kernel
             ...
             OpStore %43 %39 Aligned 8
             OpReturn
             OpFunctionEnd
```

Array programming is often sufficient

```julia
function vadd(a, b, c)




    d_a = oneArray(a)
    d_b = oneArray(b)
    d_c = oneArray(c)


    d_c = d_a .+ d_b
    c .= Array(d_c)
end
```

Array programming is powerful

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

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

```julia
x = oneArray(x)
y = oneArray(y)
w = oneArray(w)
```

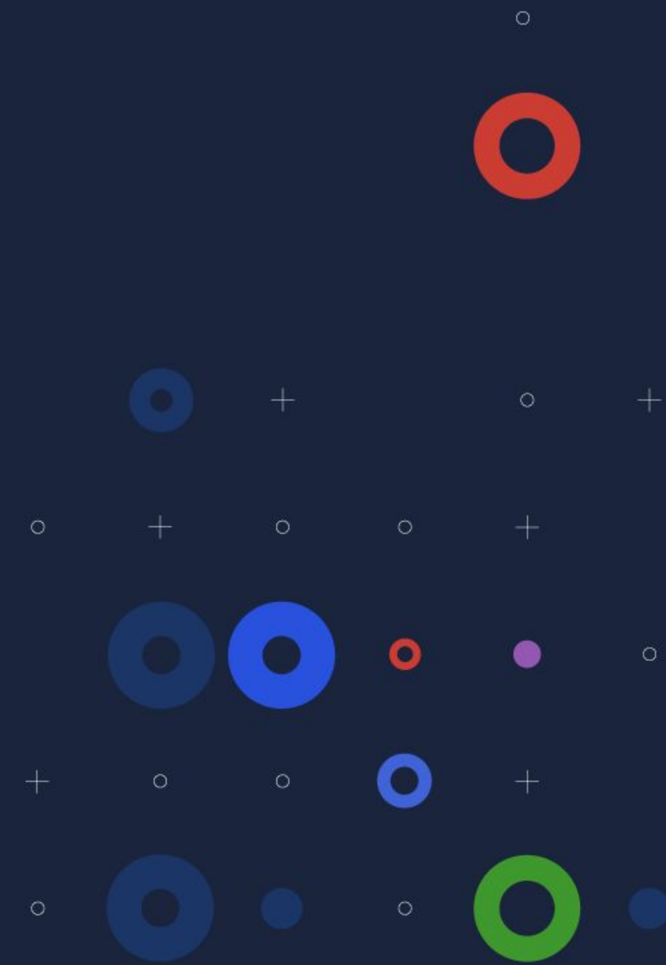
Generic programming:

- Easy: develop on the CPU, deploy on a GPU
- Reusable
- Composable

Work in progress

- Performance: optimization for Intel hardware
- Integration with math libraries (oneMKL, oneDNN, …)
- Integration with performance tools (VTune)
- Platform support: Linux & Intel GPUs only

JuliaHub



Advancing Scientific Discovery Across Architectures

Jacob Vaverka
Tim Besard