Julia

A Fresh Approach to GPU Computing
What is Julia?

Technical computing language

High-level like Python

Performance of C

```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 for GPU programming
What is GPU programming?

High-level

TensorFlow, Keras
ArrayFire, Thrust
cuBLAS, cuDNN
CUB, MGPU

Low-level

CUDA C
Flux.jl, Knet.jl
GPUArrays.jl
CuArrays.jl
CUDAnative.jl
Programming with libraries

- cuBLAS.jl
- cuFFT.jl
- cuRAND.jl
- cuSPARSE.jl
- cuDNN.jl
- cuSOLVER.jl

CuArrays.jl

```julia
a = CuArray(Float32, 2)
b = curand(Float32, 2)
a*a
fft(a)
qrfact(a)
softmax(a)
```
Programming with kernels

Much harder!
function foo(x)
    if isa(x, Int64)
        ...
    elseif isa(x, Float64)
        ...
    end
end

foo(x::Int64) = ...
foo(x::Float64) = ...
function sigmoid(x)
    temp = exp(-x)
    return (1 / (1+temp))
end
function `sigmoid`(x::Int)
    temp = exp(-x)::Float64
    return (1 / (1+temp))::Float64
end
Designed for performance

Type inference

function sigmoid(x::Float32)
    temp = exp(-x)::Float32
    return (1 / (1+temp))::Float32
end

Machine-native types
Designed for performance

- Multiple dispatch
- Type inference
- Machine-native types

Specializing JIT compiler

- High-quality stand-alone machine code
How does it look?

No DSL, no subset, just Julia

CUDA abstraction level

Performance parity

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

a = CuArray(randn(2,2))
b = CuArray(randn(2,2))
c = similar(a)

@cuda threads=4 vadd(a,b,c)
```
How does it run?
How does it work?

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

a = CuArray(randn(2,2))
b = CuArray(randn(2,2))
c = similar(a)

@cuda threads=4 vadd(a,b,c)
```
How does it work?

Run time JIT compiler

Fully transparent

No overhead!
```julia
using CUDA

function hello()
    @cuprintf("Hello, World!\n")
    return
end

hello (generic function with 1 method)

@cuda hello(); synchronize()
Hello, World!

function hello()
    @cuprintf("Hello, World! from thread %u\n", threadIdx().x)
    return
end

hello (generic function with 1 method)

@cuda threads=2 hello(); synchronize()
Hello, World! from thread 1
Hello, World! from thread 2
```
In [1]: using CUDAnative, CUDAdrv

In [2]: function hello()
    @cuprintf("Hello, World!\n")
    return
end

Out[2]: hello (generic function with 1 method)

In [3]: @cuda hello()
synchronize()

Hello, World!
function vadd(a, b, c)
    i = (blockIdx().x-UInt32(1)) * blockDim().x + threadIdx().x
    c[i] = a[i] + b[i]
end

a, b, c = CuArray{Float32,1}(..., ...)

@device_code_lowered  @cuda vadd(a,b,c)          %0 = (blockIdx)()
                         %1 = (getproperty)(%0, :x)

@device_code_typed
%0 = $(Expr(:foreign, %2 = (blockDim)())
    UInt32, %3 = (getproperty)(%2, :x)

@device_code_llvm
%1 = $(Expr(:foreign, %4 = %1 * %3)
    UInt32, %5 = (threadIdx)())

@device_code_ptx
%2 = (mul_int)(%0, %6 = (getproperty)(%5, :x))

@device_code_sass
S2R R0, SR_CTAIDX.X;
    :llvmcall), 0))
S2R R1, SR_TID.X;
IMAD R2, R0, c[0x0][0x28], R1;
add.s64   %rd5, %rd1, %rd4;
# ordinary scalar function

sigmoid(x) = 1/(1+exp(-x))

a = CuArray([1.])

# typical GPU application

function elwise_kernel(op, a)
    i = (blockIdx().x-1) * blockDim().x + threadIdx().x
    a[i] = op(a[i])
    return
end
cuda elwise_kernel(sigmoid, a)

map(sigmoid, a)
sigmoid.(a)

1 ./ (1 .+ exp.(-a))  # fused into a *single* function!
High-level GPU programming

Great performance

Clean & concise

Generic code

\[(\frac{a + b}{d}) - e\]
# ordinary scalar function

```plaintext
sigmoid(x) = 1/(1+exp(-x))
```

```julia
julia> a = CuArray(1.0:5.)
5-element CuArray{Float64,1}:
  1.0
  2.0
  3.0
  4.0
  5.0

julia> b = CuArray(Dual.(1.0:5., 1))
5-element CuArray{Dual{Float64,1},1}:
  Dual(1.0,1.0)
  Dual(2.0,1.0)
  Dual(3.0,1.0)
  Dual(4.0,1.0)
  Dual(5.0,1.0)
```

```julia
julia> sigmoid.(a)
5-element CuArray{Float64,1}:
  0.731059
  0.880797
  0.952574
  0.982014
  0.993307
```

```julia
julia> sigmoid.(b)
5-element CuArray{Dual{Float64,1},1}:
  Dual(0.731059,0.196612)
  Dual(0.880797,0.104994)
  Dual(0.952574,0.0451767)
  Dual(0.982014,0.0176627)
  Dual(0.993307,0.00664806)
```

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

W = randn(2, 10)
b = randn(2)
f(x) = softmax(W * x .+ b)

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

From GPU kernels, to differentiable algorithms, to high-level layer stacking. All on one platform.
Dz, Dh = 5, 500 # Latent dimensionality, # hidden units.

# Components of recognition model / "encoder" MLP.
A, μ, logσ = Dense(28^2, Dh, tanh), Dense(Dh, Dz), Dense(Dh, Dz)
g(X) = (h = A(X); (μ(h), logσ(h)))

z(μ, logσ) = μ + exp(logσ) * randn()

# Generative model / "decoder" MLP.
f = Chain(Dense(Dz, Dh, tanh), Dense(Dh, 28^2, σ))

# KL-divergence between approximation posterior and N(0, 1) prior.
kl_q_p(μ, logσ) = 0.5 * sum(exp.(2 .* logσ) + μ.^2 - 1 .+ logσ.^2)

# logp(x|z) - conditional probability of data given latents.
logp_x_z(x, z) = sum(logpdf.(Bernoulli.(f(z)), x))

# Monte Carlo estimator of mean ELBO using M samples.
L(X) = ((μ, logσ) = g(X); (logp_x_z(X, z.(μ, logσ)) - kl_q_p(μ, logσ)) / M)

loss(X) = -L(X) + 0.01 * sum(x->sum(x.^2), params(f))

# Sample from the learned model.
modelsample() = rand.(Bernoulli.(f(z.(zeros(Dz), zeros(Dz))))))
Pkg.add("Flux")
The Julia Magic

Everything just works with everything else!

Differential Equations  →  Everything You Build  →  Machine Learning

CUDA  →  Everything You Build  →  Automatic Differentiation
\[
\frac{dx}{dt} = \alpha x - \beta xy \\
\frac{dy}{dt} = \delta xy - \gamma y
\]
All the HPC Tooling

Differential Equations

Operations Research

Deep Learning

StructOfArrays.jl

DistributedArrays.jl

JuliaDB.jl & DataFrames.jl

Generic programming is extremely powerful.
a screenplay ingeniously constructed more than "Memento"
function model(tree)
    if isleaf(tree)
        tree.value
    else
        model(tree.left) + model(tree.right)
    end
Flux Experiment: MNIST Classifier

Input: 7
julia> m = Chain(Dense(10,5,relu),Dense(5,2),softmax)

julia> @code_js m(x)
let model = (function () {
  let math = dl.ENV.math;
  function badger(eland) {
    return math.add(math.matrixTimesVector(model.weights[0], eland), model.weights[1]);
  }
  function chimpanzee(mongoose) {
    return math.relu(math.add(math.matrixTimesVector(model.weights[2], mongoose), model.weights[3]));
  }
  function model(shark) {
    return math.softmax(badger(chimpanzee(shark)));
  }
  model.weights = [];
  return model;
})();
flux.fetchWeights("model.bson").then((function (ws) {
  return model.weights = ws;
}));
Case Studies

New York Federal Reserve Bank

Nobel Laureate Thomas J. Sargent
Next-generation macroeconomic models require high-performance computing: enter Julia.

Safer Skies
The Federal Aviation Administration is using Julia to develop the Next-Generation Airborne Collision Avoidance System.

Sports and High Frequency Trading
Julia finds its way into statistical computing engines focused on sports betting and high frequency trading.

Deep Learning for Medical Diagnosis
Deep learning used to diagnose diabetic retinopathy.

Autonomous Race Cars
UC Berkeley researchers use Julia to optimize model predictive control for the Berkeley Autonomous Race Car (BARC).

Optimizing the Electrical Grid
Invenia Technical Computing is scaling up its energy intelligence system using Julia.

Modeling Cancer Evolution
UK cancer researchers use Julia to model tumor growth, informing interpretation of cancer genomes.

Real Time Energy Forecasting
Tangent Works uses Julia to provide a comprehensive analytics solution, eliminating the barrier between prototyping and production.

Augmented Reality Gives Surgeons 'X-Ray Vision'
Augmedics uses CT scans to give surgeons a 3D image of patient anatomy.
JuliaCon – 300 Attendees, 150 Talks
Julia

https://github.com/JuliaGPU/
NVIDIA Parallel Forall blog

https://github.com/FluxML/