Julia for Machine Learning
– Case Study using miniWeather –

Hyun Kang, Youngsung Kim, and Sarat Sreepathi

Computational Sciences and Engineering Division,
Oak Ridge National Laboratory

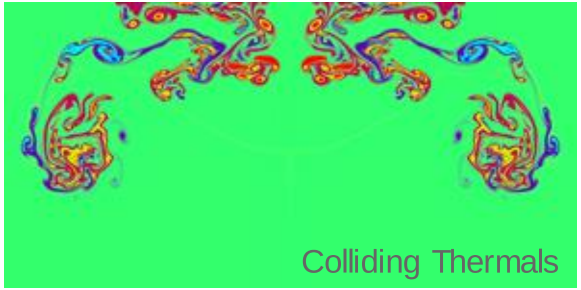
2023. 02. 07.
ECPAM
Julia programming for Exascale

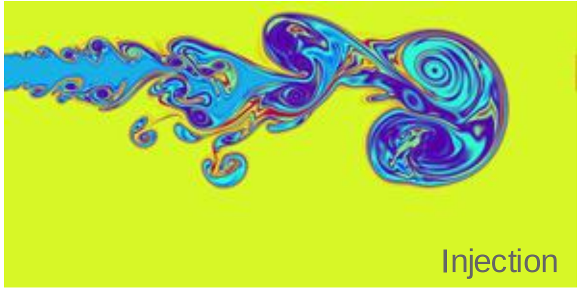
Machine learning packages in Julia

- Machine learning packages in Julia
  - A variety of packages are ready in use.
    - Flux
    - FluxMPI
    - Merlin
    - Mocha
    - Knet
    - MLBase
    - ScikitLearn - Julia wrapper for ‘ScikitLearn’ library
    - TensorFlow - Julia wrapper for ‘TensorFlow’ library
    - Etc …
  - Flux.jl
    - A library for machine learning geared towards high-performance production pipelines
    - Many useful tools built in
    - Fully support Julia language
    - NVIDIA GPU support via CUDA.jl

Porting miniWeather-Fortran to Julia

- miniWeather (github.com/mrnorman/miniWeather)
  - Accelerant app mainly developed by Dr. Matthew Norman (ORNL)
  - Simulating weather-like flows for training in parallelizing accelerated HPC architectures
  - Various versions ported to several HPC programming frameworks.
    - MPI (C, Fortran, and C++), OpenACC Offload (C and Fortran), OpenMP Threading (C and Fortran) …
  - Decent code size (~ 600 SLOC) and well-documented
  - Finite volume spatial discretization (x-z, 2D) and Runge-Kutta time integration
- We used MPI-Fortran version of miniWeather and ported it to Julia.





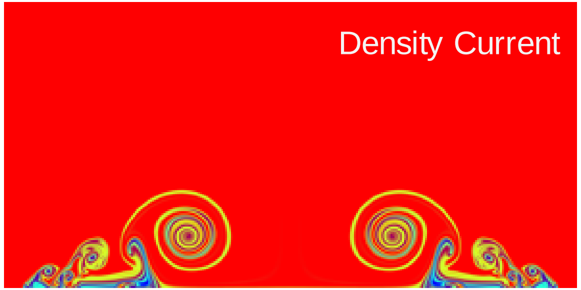


Image source: https://github.com/mrnorman/miniWeather

Porting miniWeather-Fortran to Julia - Porting details

- Porting details
  - Comparison with Fortran codes
    - Array allocation and indices

Fortran

```
allocate(state(1-hs:nx+hs,1-hs:nz+hs,  NUM_VARS))
```

Julia (using ‘OffsetArray’ module)

```
state = OffsetArray(_state,1-HS:NX+HS,1-HS:NZ+HS,  NUM_VARS )
```

Column-major, array index starts arbitrary integer

Porting miniWeather-Fortran to Julia - Porting details

- Porting details
  - Comparison with Fortran codes
    - Loops & IF statements

Fortran

```fortran
!Use the fluxes to compute tendencies for each cell
do ll = 1 , NUM_VARS
  do k = 1 , nz
    do i = 1 , nx
      tend(i,k,ll) = -(  flux(i,k+1,ll)       &
                       - flux(i,k,ll) ) / dz
      if (ll == ID_WMOM) then
        tend(i,k,ID_WMOM) = tend(i,k,ID_WMOM) &
                          - state(i,k,ID_DENS)*grav
      endif
    enddo
  enddo
enddo
```

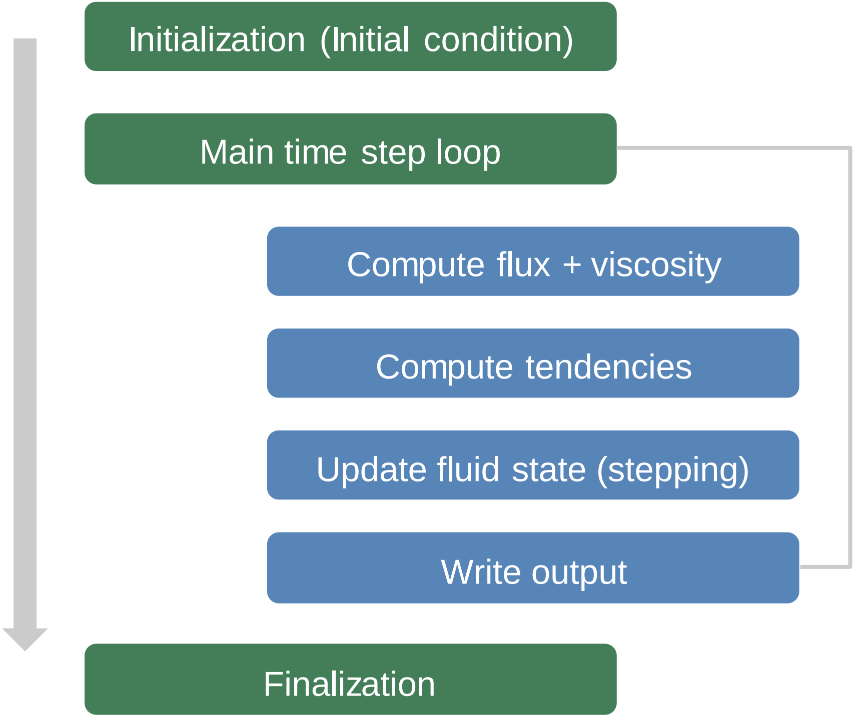
Julia

```julia
#Use the fluxes to compute tendencies for each cell
for ll in 1:NUM_VARS
  for k in 1:NZ
      for i in 1:NX
         tend[i,k,ll] = -( flux[i,k+1,ll] -
                           flux[i,k,ll] ) / DZ
         if (ll == ID_WMOM)
            tend[i,k,ID_WMOM] = tend[i,k,ID_WMOM] -
                                state[i,k,ID_DENS]*GRAV
         end
      end
    end
end
```

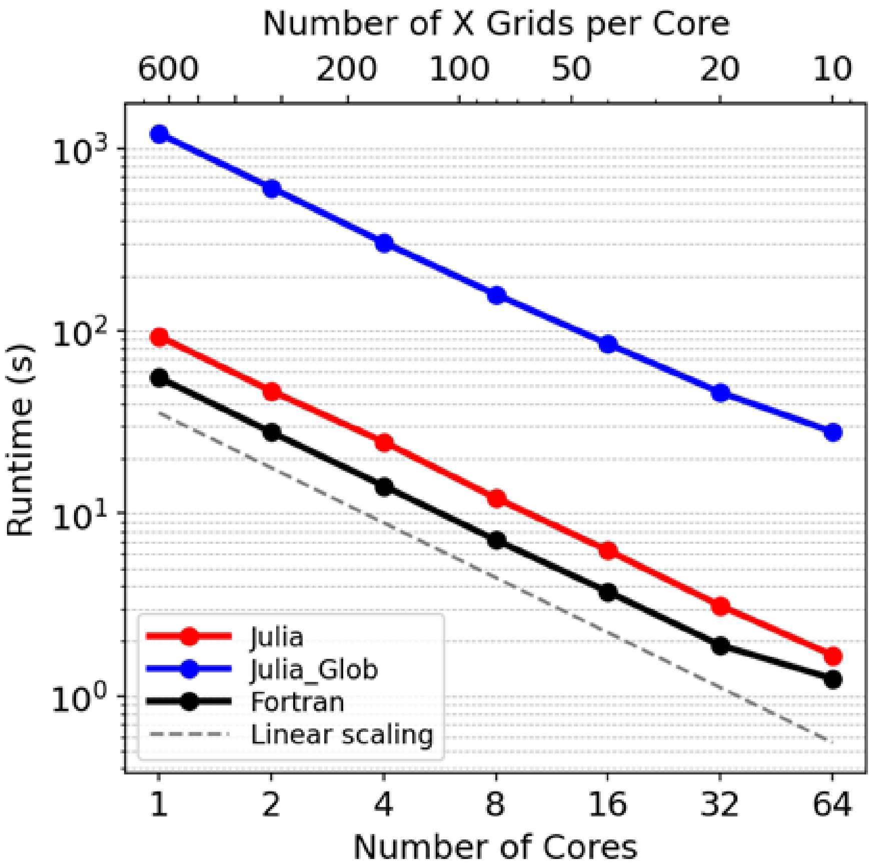
- Julia seems to be Fortran-friendly
  - Ported in 2 days thanks to column-major array and Fortran-style array indexing (OffsetArrays)

Porting miniWeather-Fortran to Julia

- miniWeather model structure



Porting miniWeather-Fortran to Julia - Performance Results



- Fortran is about 1.6x faster than Julia. Wow! Julia is faster than we thought!
- Julia with naively using global variable (Julia_Glob) is about ~13x slower.
  - Be careful when using global variables
- Tested on Cori (Intel KNL) at NERSC
  - On Crusher (a testbed for Frontier), Fortran is about 1.8x faster than Julia.

https://github.com/grnydawn/jlweather

Porting miniWeather-Fortran to Julia

- Promising speed of Julia in a simple weather model
  ⇒ We saw possibilities of Julia for use in HPC area

- Unique benefits as a dynamic language?
  - Fast data processing with convenient use of packages
    - Ex) Huge NetCDF file + custom diagnoses
    - Ex) Packages for data sciences (DataFrames.jl, Pandas.jl (Julia interface for Pandas) …)
    - Ex) Fast pre+post data processing for ML

  - Seamless integration of machine learning
    - Ex) Replacement of a conventional ML process :
      - Fortran (modeling & data generation & processing) + Python (ML)
        → 2 languages, different coding layer & execution
    - Ex) **Online learning benefitting from fast Julia modeling + convenient ML package use**
      - A common technique used in areas of machine learning where it is computationally infeasible to train over the entire dataset

Machine learning in Julia

- Machine learning packages in Julia
  - A variety of packages are ready in use.
    - Flux
    - FluxMPI
    - Merlin
    - Mocha
    - Knet
    - MLBase
    - ScikitLearn - Julia wrapper for ‘ScikitLearn’ library
    - TensorFlow - Julia wrapper for ‘TensorFlow’ library
    - Etc …
  - Flux.jl
    - A library for machine learning geared towards high-performance production pipelines
    - Many useful tools built in
    - Fully support Julia language
    - NVIDIA GPU support via CUDA.jl

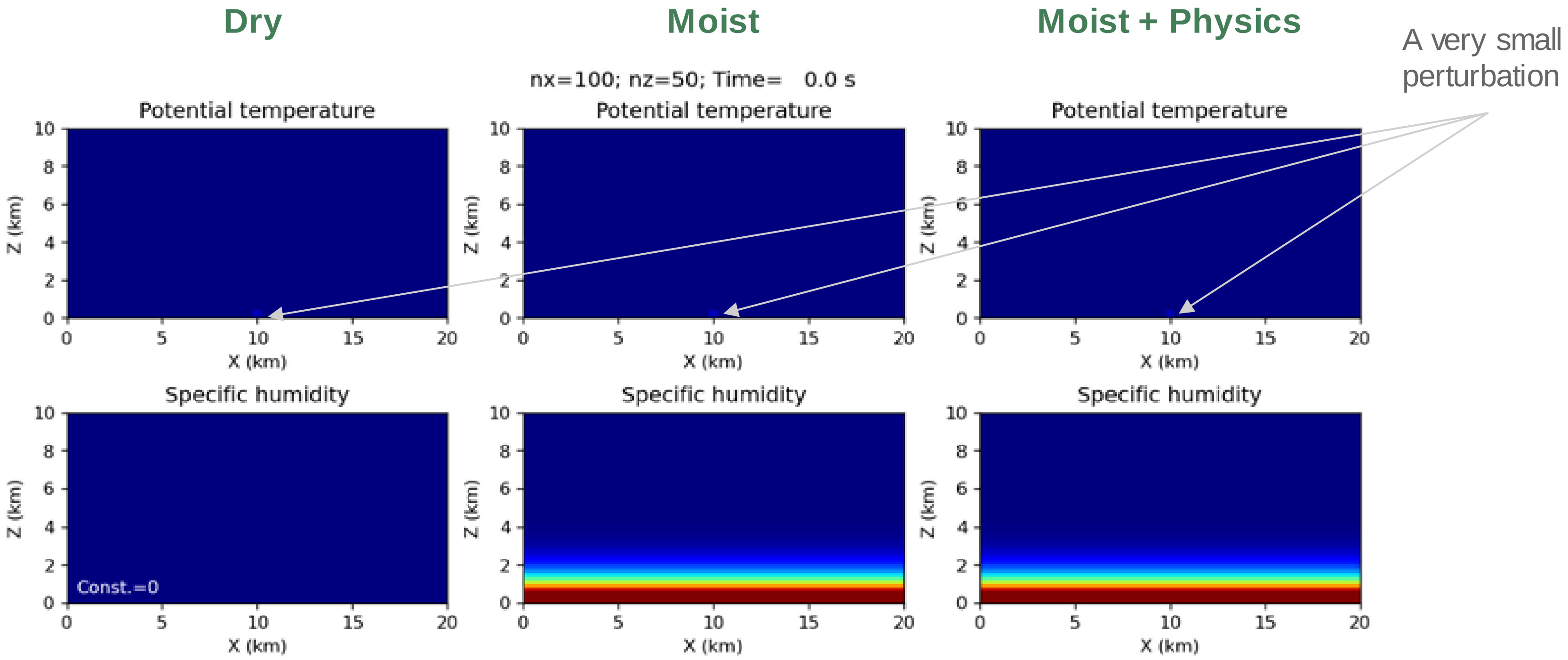
Machine learning with miniWeather

- Learning simple physics during simulations
  - Physics parameterizations in NWP models usually occupies the second largest portion of a whole runtime.
    - Physics parameterization mimics physical process + sub-grid scale phenomena in numerical weather prediction models
      - E.g., turbulence, precipitation, radiation, etc
  - Emulate an existing physics parameterization to enable faster computation
    - Note: In our demonstration, however, it's a very simple physics schemes. *There is no computational advantage.*

- Coupling parameterized physical processes to miniWeather
  - Simple physics package (Reed and Jablonowski, 2012)
    - Code from DCMIP 2016 (Dynamical Core Model Intercomparison Project (Ullrich et al., 2017))
    - Condensation & Latent heat release (large-scale condensation, no cloud stage)
    - Bulk aerodynamic surface fluxes
    - Boundary-layer mixing

Machine learning using miniWeather

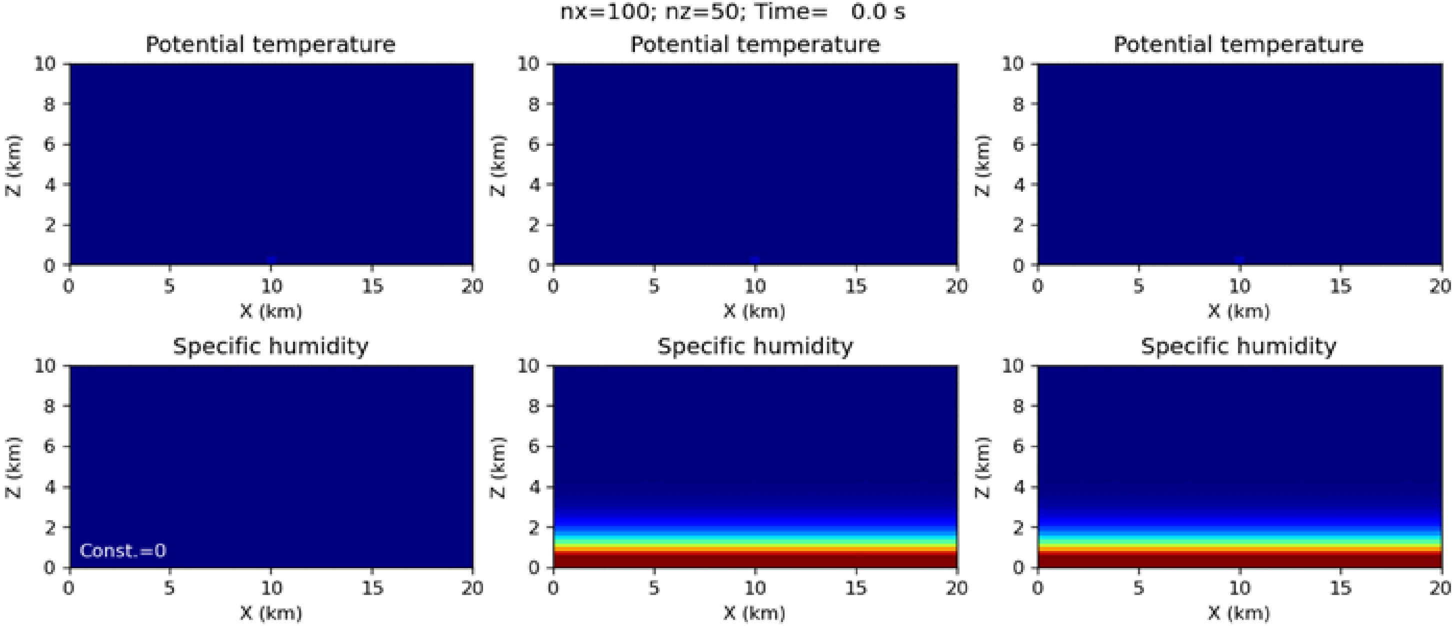
- Simulation results (Dry VS Moist VS Moist + Physics)
  - A very small perturbation in the initial potential temperature
  - Stable atmosphere ($d\theta / dz > 0$)



Machine learning using miniWeather

- Simulation results (Dry VS Moist VS Moist + Physics)
  - A very small perturbation in the initial potential temperature
  - Stable atmosphere ($d\theta / dz > 0$)
  - Key process = **Condensation & Latent heat release → Learn this process**

Dry **Moist** **Moist + Physics**



Machine learning using miniWeather

- Brief introduction of learning process
  - Train data
    - Input (x train data)
      - Normalized T (temperature), Q (specific humidity)
    - Output (y train data)
      - Tendencies (dt/dt, dq/dt) after large-scale condensation process
  - Neural Network using two layers

```
model = Chain(Dense(Qin => Q1  , relu),
              Dense(Q1  => Qout, relu))
```

  - Loss function (MSE)

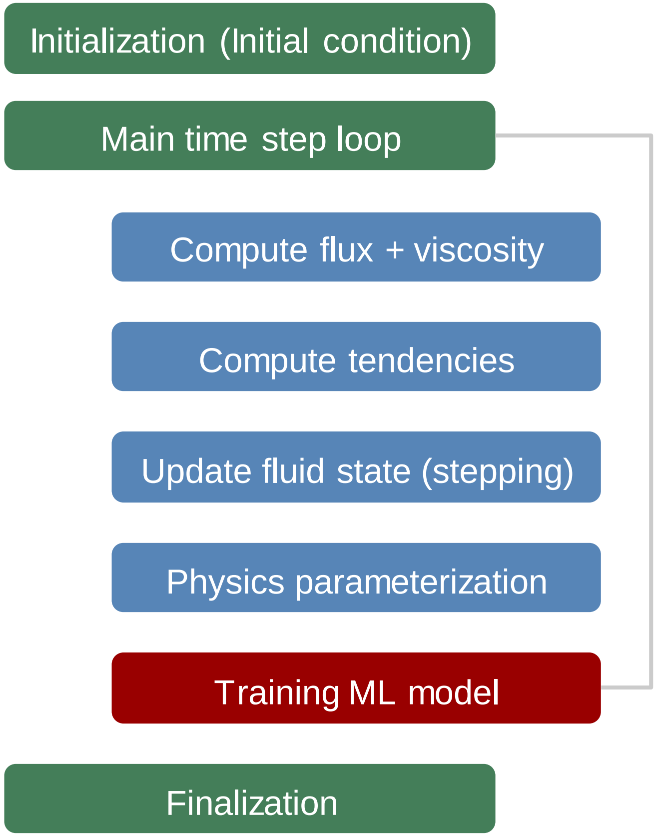
```
loss(x,y) = Flux.Losses.mse(model(collect(x)), y)
```

  - Optimizer

```
opt = Flux.Descent(lr)
```

  - Training

```
Flux.train!(loss, ps, trainingData, opt)
```



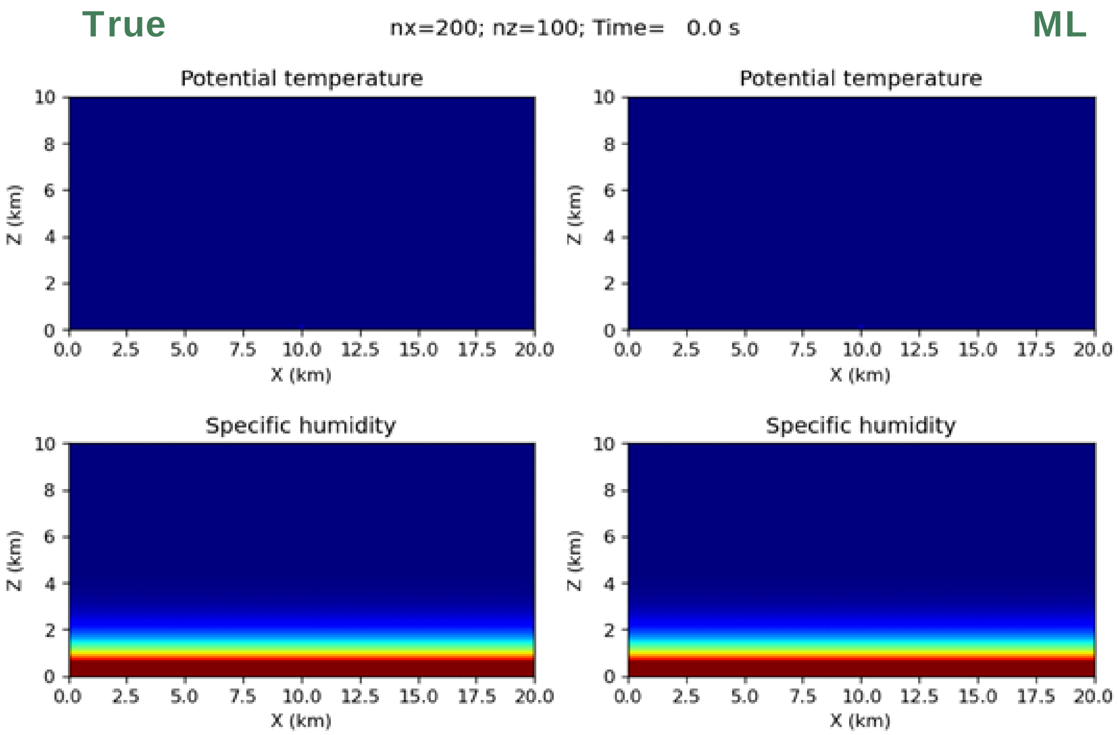
Machine learning using miniWeather

```julia
function train_model(x,y,etime)

    # Parameter sizes
    Qin  = 2
    Q1   = Qin*4
    Qout = 2

    batchsize = 1
    # DataLoader : Performant data loading for deep learning
    loader_XY = Flux.Data.DataLoader(
            (x,y),
            batchsize = batchsize,
            shuffle = true)

    # Initial time
    if etime == 0.0

       # Neural network using two layers
       model = Chain(Dense(Qin , Q1, relu),
                     Dense(Q1 , Qout,relu))

    # Activate learning process after 1,500 s model time when flow is active
    elseif etime > 1500.0

       # Save the model to re-train in next time step
       @load "mymodel.bson" model

    end
```

```julia
    # Activate learning process after 1,500 s model time when flow is active
    if etime == 0.0 || etime > 1500.0

    println("Model time = ",etime)

    loss(x, y) = Flux.mse(model(x), y) # Our loos function to minimize

    lr = 5e-4 # Learning rate
    opt = Flux.Descent(lr) # Gradient descent optimizer

    epochs = 500 # Number of epochs
    trainingLosses = zeros(epochs)

    ps = Flux.params(model)

    p = Progress(epochs; desc = "Training in progress");

    showProgress = true # Display progress bar

    # Training loop
    @time for ii in 1:epochs
        Flux.train!(loss, ps, loader_XY, opt) # Training the model
        if showProgress
            trainingLosses[ii] = Flux.mean([loss(x,y) for (x,y) in loader_XY])
            next!(p; showvalues = [(:loss, trainingLosses[ii]),
                   (:logloss, log10.(trainingLosses[ii]))], valuecolor = :grey)
        end
    end

    # Save the model to re-train in next time step
    @save "mymodel.bson" model

    end
end
```

Machine learning using miniWeather

- Results (ML model applied to a 2x higher resolution simulation)
  - ML version mimics the large-scale condensation process.
    - Note: This is a very simple ML model for a demonstration.



Thank you

Machine learning with miniWeather

- Learning simple physics during simulations
  - Physics parameterizations in NWP models usually occupies the second largest portion of a whole runtime.
    - Physics parameterization mimics sub-grid scale phenomena in numerical weather prediction models
      - E.g., cloud, precipitation,
  - Emulate an existing physics parameterization to enable faster computation
    - Note: In our demonstration, however, it's a very simple physics schemes. *There is no computational advantage.*

- Change from dry atmosphere to moist atmosphere in miniWeather
  - Add specific humidity (q) as a tracer
  - Specific humidity affects pressure profile

$$p = C_0 (R T_v)^\gamma \qquad T_v = (1 + 0.61q)\, T$$

- Coupling parameterized physical processes to the dynamical core (miniWeather)
  - Simple physics package (Reed and Jablonowski, 2012)
    - Code from DCMIP 2016 (Dynamical Core Model Intercomparison Project (Ullrich et al., 2017))
    - Condensation & Latent heat release (large-scale condensation, no cloud stage)
    - Bulk aerodynamic surface fluxes
    - Boundary-layer mixing

Porting miniWeather-Fortran to Julia - Validation

- Result validations - digit comparison with Fortran results
  - Direct printout from NetCDF output (u-wind comp. at 150 time step)

Fortran
```
5.17481312383945, 5.27095857741744, 4.85877713965263,
```

Julia
```
5.17481312383922, 5.27095857741761, 4.85877713965261,
```

  - Globally reduced values (MPI_ALLREDUCE) over 4 cores at 150 time step

Fortran
```
d_te  :   -5.299941425218241e-04
```

Julia
```
d_te:     -5.2999414252196178E-004
```

<u>Around machine precision differences</u>

Porting miniWeather-Fortran to Julia - Porting details

- Porting details
  - Comparison with Fortran codes
    - Subroutine & MPI use

Fortran

```fortran
subroutine reductions( mass , te )
  implicit none
  real(rp), intent(out) :: mass, te
  integer :: i, k, ierr
  real(rp) :: r,u,w,th,p,t,ke,ie
  real(rp) :: glob(2)
  mass = 0
  te   = 0
  do k = 1 , nz
    do i = 1 , nx
      r  =   state(i,k,ID_DENS) + hy_dens_cell(k)             ! Density
      u  =   state(i,k,ID_UMOM) / r                           ! U-wind
      w  =   state(i,k,ID_WMOM) / r                           ! W-wind
      th = ( state(i,k,ID_RHOT) + hy_dens_theta_cell(k) ) / r ! Potential Temperature
      p  = C0*(r*th)**gamma       ! Pressure
      t  = th / (p0/p)**(rd/cp)   ! Temperature
      ke = r*(u*u+w*w)            ! Kinetic Energy
      ie = r*cv*t                 ! Internal Energy
      mass = mass + r             *dx*dz ! Accumulate domain mass
      te   = te   + (ke + r*cv*t)*dx*dz ! Accumulate domain total energy
    enddo
  enddo

  call mpi_allreduce((/mass,te/),glob,2,mpi_type,MPI_SUM,MPI_COMM_WORLD,ierr)

  mass = glob(1)
  te   = glob(2)
end subroutine reductions
```

Julia

```julia
function reductions(state::OffsetArray{Float64, 3, Array{Float64, 3}},
                    hy_dens_cell::OffsetVector{Float64, Vector{Float64}},
                    hy_dens_theta_cell::OffsetVector{Float64, Vector{Float64}})

    local mass, te, r, u, w, th, p, t, ke, ie = [zero(Float64) for _ in 1:10]
    glob = Array{Float64}(undef, 2)

    for k in 1:NZ
        for i in 1:NX
            r  =   state[i,k,ID_DENS] + hy_dens_cell[k]             # Density
            u  =   state[i,k,ID_UMOM] / r                           # U-wind
            w  =   state[i,k,ID_WMOM] / r                           # W-wind
            th = ( state[i,k,ID_RHOT] + hy_dens_theta_cell[k] ) / r # Potential Temperature
            p  = C0*(r*th)^GAMMA      # Pressure
            t  = th / (P0/p)^(RD/CP)  # Temperature
            ke = r*(u*u+w*w)          # Kinetic Energy
            ie = r*CV*t               # Internal Energy
            mass = mass + r            *DX*DZ # Accumulate domain mass
            te   = te   + (ke + r*CV*t)*DX*DZ # Accumulate domain total energy
        end
    end

    MPI.Allreduce!(Array{Float64}([mass,te]), glob, +, COMM)

    return glob
end # function
```

Oak Ridge National Laboratory