Building a High-Performance Earth System Model in Julia

Maciej Waruszewski[1], Lucas Wilcox[1], Jeremy Kozdon[1], Frank Giraldo[1],
Tapio Schneider[2]

[1]Department of Applied Mathematics
Naval Postgraduate School

[2]California Institute of Technology

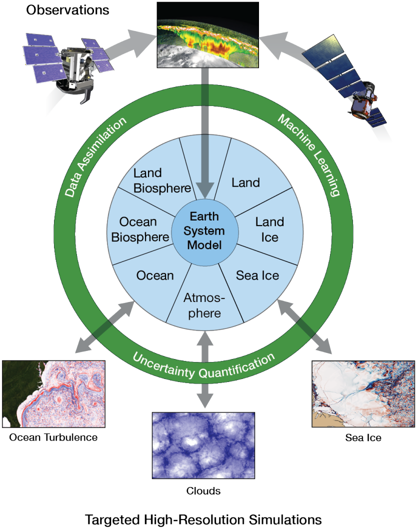
MultiCore 9, NCAR, September 26 2019

CLIMATE MODELING ALLIANCE

The CliMA Project

- collaboration between Caltech, MIT, NPS, and JPL to build a new climate model
- model will learn from observational data and targetted high-resolution simulations
- NPS responsible for the DG-based dynamical core
- development from scratch in Julia
- open-source under a permissive license (Apache 2.0)

https://github.com/climate-machine



Julia

- dynamic high-level language designed for technical computing (MIT, 2009)
- aims to solve the two-language problem
- based on LLVM
- most of Julia is written in Julia
- can be used interactively via REPL
- has a package manager
- achieves high performance by JIT compilation and aggressive specialization
- has powerful metaprogramming and reflection capabilities

Example Julia code

(CLIMA GMRES loop)

```
for outer j = 1:M
  # Arnoldi using Modified Gram Schmidt
  linearoperator!(krylov_basis[j + 1], krylov_basis[j])
  for i = 1:j
    H[i, j] = dot(krylov_basis[j + 1], krylov_basis[i])
    krylov_basis[j + 1] .-= H[i, j] .* krylov_basis[i]
  end
  H[j + 1, j] = norm(krylov_basis[j + 1])
  krylov_basis[j + 1] ./= H[j + 1, j]

  # apply the previous Givens rotations
  # to the new column of H
  @views H[1:j, j:j] .= Ω * H[1:j, j:j]

  # compute a new Givens rotation to zero out H[j + 1, j]
  G, _ = givens(H, j, j + 1, j)

  # apply the new rotation to H and the rhs
  H .= G * H
  g0 .= G * g0

  # compose the new rotation with the others
  Ω = lmul!(G, Ω)

  residual_norm = abs(g0[j + 1])

  if residual_norm < threshold
    converged = true
    break
  end
end
```

Julia: example of specialization

Julia

```
julia> f(x, y) = x * y
f (generic function with 1 method)
julia> x = 1; # Int64
julia> y = 1; # Int64
julia> @code_native f(x, y)
```

Assembly

```
; ┌ @ REPL[1]:1 within `f'
; │┌ @ REPL[1]:1 within `*'
        imulq   %rsi, %rdi
; │└
        movq    %rdi, %rax
        retq
        nopl    (%rax,%rax)
; └
```

Julia: example of specialization

Julia

```
julia> f(x, y) = x * y
f (generic function with 1 method)
julia> x = 1.0; # Float64
julia> y = 1.0; # Float64
julia> @code_native f(x, y)
```

Assembly

```
; ┌ @ REPL[1]:1 within `f'
; │┌ @ REPL[1]:1 within `*'
        vmulsd  %xmm1, %xmm0, %xmm0
; │└
        retq
        nopw    %cs:(%rax,%rax)
; └
```

Julia: example of specialization

Julia

```
julia> f(x, y) = x * y
f (generic function with 1 method)
julia> x = 1.0; # Float64
julia> y = 1  ; # Int64
julia> @code_native f(x, y)
```

Assembly

```
; ┌ @ REPL[1]:1 within `f'
; │┌ @ promotion.jl:314 within `*'
; ││┌ @ promotion.jl:284 within `promote'
; │││┌ @ promotion.jl:261 within `_promot
; ││││┌ @ number.jl:7 within `convert'
; │││││┌ @ REPL[1]:1 within `Type'
        vcvtsi2sdq  %rdi, %xmm1, %xmm1
; │└└└└└
; │┌ @ float.jl:399 within `*'
        vmulsd      %xmm0, %xmm1, %xmm0
; │└
        retq
        nopw        (%rax,%rax)
; └
```

Julia: example of specialization

Julia

```julia
julia> f(x, y) = x * y
julia> using StaticArrays
julia> x = @SMatrix rand(4, 4)
julia> y = @SVector rand(4)
julia> @code_native f(x, y)
```

Assembly

```
      ; ┌ @ REPL[1]:1 within `f'
; │┌ @ matrix_multiply.jl:8 within `*'
; ││┌ @ matrix_multiply.jl:45 within `_mul'
; │││┌ @ matrix_multiply.jl:58 within `macro expansion'
; ││││┌ @ REPL[1]:1 within `*'
        vbroadcastsd  (%rdx), %ymm0
        vmulpd        (%rsi), %ymm0, %ymm0
        vbroadcastsd  8(%rdx), %ymm1
        vmulpd        32(%rsi), %ymm1, %ymm1
; │││└└
; │││┌ @ float.jl:395 within `macro expansion'
        vaddpd        %ymm1, %ymm0, %ymm0
; │││└
; │││┌ @ matrix_multiply.jl:58 within `macro expansion'
; ││││┌ @ float.jl:399 within `*'
        vbroadcastsd  16(%rdx), %ymm1
        vmulpd        64(%rsi), %ymm1, %ymm1
; ││││└
; ││││┌ @ float.jl:395 within `+'
        vaddpd        %ymm1, %ymm0, %ymm0
; ││││└
; ││││┌ @ float.jl:399 within `*'
        vbroadcastsd  24(%rdx), %ymm1
        vmulpd        96(%rsi), %ymm1, %ymm1
; ││││└
; ││││┌ @ float.jl:395 within `+'
        vaddpd        %ymm1, %ymm0, %ymm0
; │└└└└
        vmovupd       %ymm0, (%rdi)
        movq          %rdi, %rax
        vzeroupper
        retq
        nopw          %cs:(%rax,%rax)
; └
```

Julia benefits for CliMA

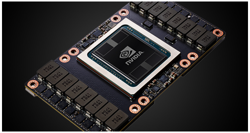
In addition to being performant Julia

- is a good common language for domain experts from the Earth sciences and uncertainty quantification/machine-learning communities
- enables rapid development and refactoring
- makes coupling independently developed components easy

We also get special support from the MIT Julia Lab.

A new climate model needs to fully embrace accelerators

Modern supercomputers are increasingly becoming accelerator-based with hardware evolving at a rapid pace



Julia support for programming accelerators is another of its strong points.

Julia GPU ecosystem

Pioneering work by Tim Besard (@maleadt, Julia Computing)

Low level - `CUDAnative`

- "write CUDA in Julia"
- Julia GPU compiler implemented as a library with maximal reuse of the Julia compiler infrastructure ($\sim$ 4.5K lines of code, backend provided by LLVM)
- the same approach already inspired efforts for AMD GPUs and Google TPUs

High level - `CuArrays`

- provides arrays that live in the GPU memory and data transfer primitives
- can program both CPUs and GPUs using element wise operations and (map)reduce functions

More on `CUDAnative`

- leverages Julia ability to generate static code
- accepts mostly undocumented subset of Julia in kernels ("if it works it works")
- integrates well with CUDA tools (nvprof, nvvp, etc.)
- performance for simple code is often as good as CUDA compiled with clang
- performance for more abstract code can be hard to predict
- debugging is tricky

Example CUDAnative code
(matrix transpose using shared memory)

```
const TDIM = 32
const BLOCK_ROWS = 8

function cudanative_transpose!(a_transposed, a)
  T = eltype(a)
  tile = @cuStaticSharedMem T (TDIM + 1, TDIM)

  by = blockIdx().y
  bx = blockIdx().x

  ty = threadIdx().y
  tx = threadIdx().x

  i = (bx - 1) * TDIM + tx
  j = (by - 1) * TDIM + ty

  for k = 0:BLOCK_ROWS:TDIM-1
    @inbounds tile[ty + k, tx] = a[i, j + k]
  end

  sync_threads()

  i = (by - 1) * TDIM + tx
  j = (bx - 1) * TDIM + ty

  for k = 0:BLOCK_ROWS:TDIM-1
    @inbounds a_transposed[i, j + k] = tile[tx, ty + k]
  end

  nothing
end
```

CLIMA abstraction for platform portability - `GPUifyLoops`

GPUifyLoops transpose

```julia
function gpuifyloops_transpose!(a_transposed, a)
  T = eltype(a)
  tile = @shmem T (TDIM + 1, TDIM)

  @loop for by in (1:size(input, 2) ÷ TDIM; blockIdx().y)
  @loop for bx in (1:size(input, 1) ÷ TDIM; blockIdx().x)

  @loop for ty in (1:BLOCK_ROWS; threadIdx().y)
  @loop for tx in (1:TDIM; threadIdx().x)

  i = (bx - 1) * TDIM + tx
  j = (by - 1) * TDIM + ty

  for k = 0:BLOCK_ROWS:TDIM-1
    @inbounds tile[ty + k, tx] = a[i, j + k]
  end

  end # tx
  end # ty

  @synchronize

  @loop for ty in (1:BLOCK_ROWS; threadIdx().y)
  @loop for tx in (1:TDIM; threadIdx().x)

  i = (by - 1) * TDIM + tx
  j = (bx - 1) * TDIM + ty

  for k = 0:BLOCK_ROWS:TDIM-1
    @inbounds a_transposed[i, j + k] = tile[tx, ty + k]
  end

  end # tx
  end # ty
  end # bx
  end # by
end
```

CUDAnative transpose

```julia
function cudanative_transpose!(a_transposed, a)
  T = eltype(a)
  tile = @cuStaticSharedMem T (TDIM + 1, TDIM)

  by = blockIdx().y
  bx = blockIdx().x

  ty = threadIdx().y
  tx = threadIdx().x

  i = (bx - 1) * TDIM + tx
  j = (by - 1) * TDIM + ty

  for k = 0:BLOCK_ROWS:TDIM-1
    @inbounds tile[ty + k, tx] = a[i, j + k]
  end




  sync_threads()




  i = (by - 1) * TDIM + tx
  j = (bx - 1) * TDIM + ty

  for k = 0:BLOCK_ROWS:TDIM-1
    @inbounds a_transposed[i, j + k] = tile[tx, ty + k]
  end

  nothing
end
```

CLIMA abstraction for platform portability - `GPUifyLoops`

- developed by Valentin Churavy (@vchuravy, MIT) motivated by CLIMA needs
- inspired by OCCA
- handles lowering of math functions to CUDA intrinsics on the GPU (e.g. translates `sin` to `CUDAnative.sin`)
- provides a loop unrolling macro
- performs additional optimization passes on the GPU (inlining, FMA generation)
- helps with GPU debugging since you can try running on the CPU first

CLIMA abstraction for platform portability - `GPUifyLoops`

- developed by Valentin Churavy (@vchuravy, MIT) motivated by CLIMA needs
- inspired by OCCA
- handles lowering of math functions to CUDA intrinsics on the GPU (e.g. translates `sin` to `CUDAnative.sin`)
- provides a loop unrolling macro
- performs additional optimization passes on the GPU (inlining, FMA generation)
- helps with GPU debugging since you can try running on the CPU first
- **does all of this in less than 500 lines of code !**

Example of abstractions inside kernels - balance laws

CLIMA assumes equations of the form

$$\frac{\partial \boldsymbol{q}}{\partial t} + \nabla \cdot \boldsymbol{F} = \boldsymbol{S}$$

which can be specified inside kernels using vector notation. For example, the shallow water equations can be written in code as

```
@inline function flux!(m::SWModel,
                       F::Grad,
                       q::Vars,
                       α::Vars,
                       t::Real)
  U = q.U
  η = q.η
  H = m.problem.H

  F.η += U
  F.U += grav * H * η * I
  F.U += 1 / H * U * U'

  return nothing
end
```

```
@inline function source!(m::SWModel,
                         S::Vars,
                         q::Vars,
                         α::Vars,
                         t::Real)
  τ = α.τ
  f = α.f
  U = q.U
  S.U += τ - f × U

  linear_drag!(m.turbulence, S, q, α, t)

  return nothing
end
```

CLIMA approach to distributed computing

Julia wrapper for MPI - `MPI.jl`

- started by Lucas Wilcox (@lcw, NPS) in 2012, under active development with many contributors since
- recently gained support for CUDA-aware MPI

Distributed arrays abstraction - `CLIMA.MPIStateArrays`

- an array with support for MPI holding extra ghost elements
- has methods for communicating neighbours etc.
- backed by either a CPU-resident `Array` or a GPU-resident `CuArray`
- supports distributed broadcasting and global reductions

CLIMA performance on CPUs: single CPU run time

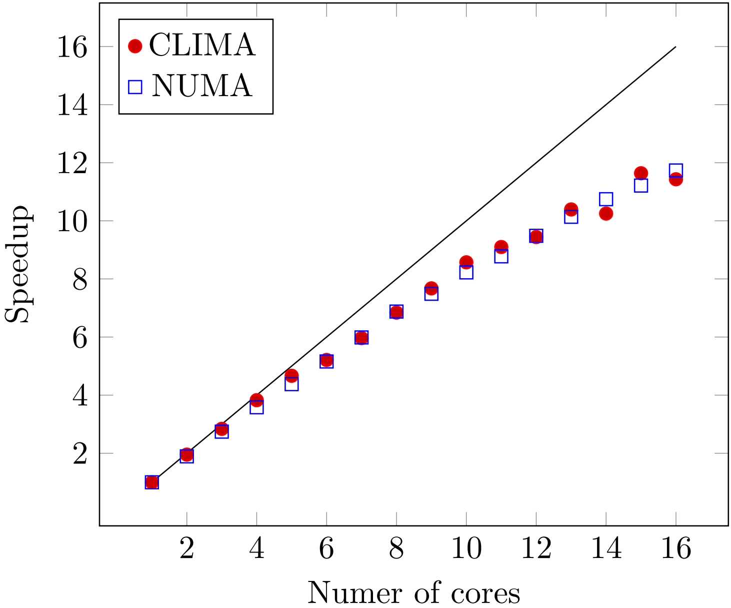
Direct run time comparison to NUMA – another DG code from NPS written in Fortran. Single core run with 10^3 elements and polynomial order 4 (rising thermal bubble test).

Timings

Kernel	CLIMA	NUMA
Volume	601.3 s	773 s
Face	297.5 s	310.5 s
LSRK	13.4 s	120.8 s
Total	912.8 s	1289.5 s

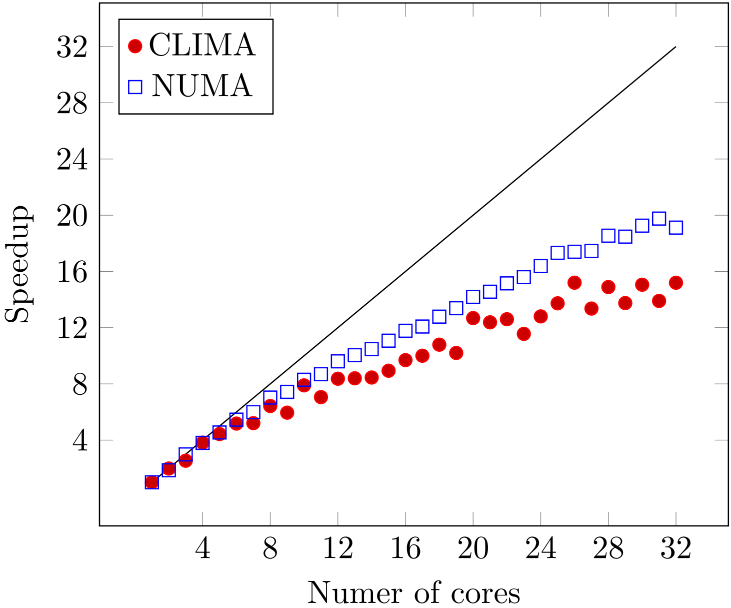
CLIMA performance on CPUs: strong scaling (1)

Scaling comparison to NUMA

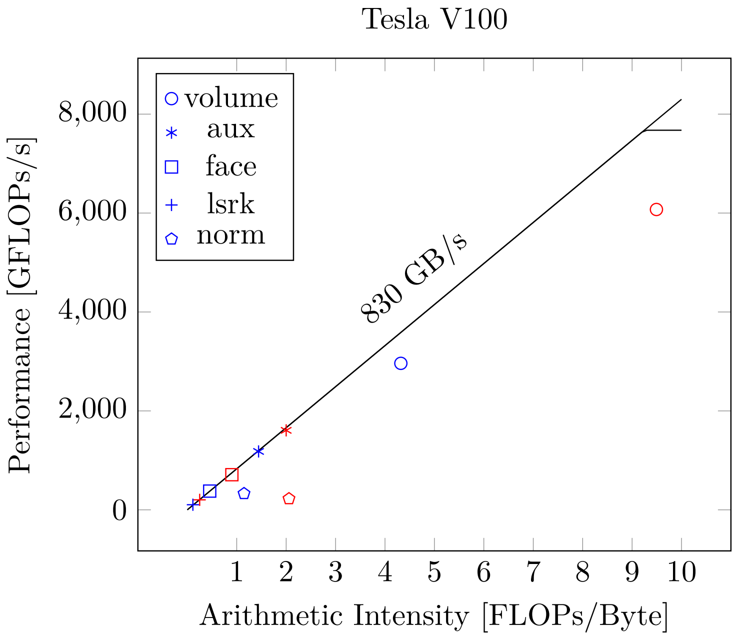


CLIMA performance on CPUs: strong scaling (2)

Scaling comparison to NUMA



CLIMA performance on GPUs: roofline



Conclusions and outlook

Conclusions

- Julia delivers on its promises, enabling high-performance while keeping productivity and abstraction level high
- macros and other code transformation tools enable platform independent programming in Julia using custom kernels
- CLIMA is faster than NUMA on the CPU and our kernels get fairly close to machine limits on the GPU

Outlook and future work

- performance CI
- more `GPUifyLoops` backends
- benchmarks using multiple GPUs and multiple nodes

CLiMA is funded by private and public funders

ERIC AND WENDY SCHMIDT

SCHMIDT **FUTURES**

EARTHRISE
ALLIANCE

CHARLES TRIMBLE

**RONALD AND MAXINE LINDE
CLIMATE CHALLENGE**

PAUL G. ALLEN
FAMILY FOUNDATION

Google Cloud

NSF