OAK RIDGE National Laboratory

E3SM-MMF Experiences with AMD Hardware

Matt Norman (ORNL); Mark Taylor (SNL);
Walter Hannah (LLNL); Benjamin Hillman (SNL);
Kyle Pressel (PNNL); Chris Eldred (SNL);
Isaac Lyngaas (ORNL);
Murali Gopalakrishnan Meena (ORNL)

OLCF User Meeting, 2021

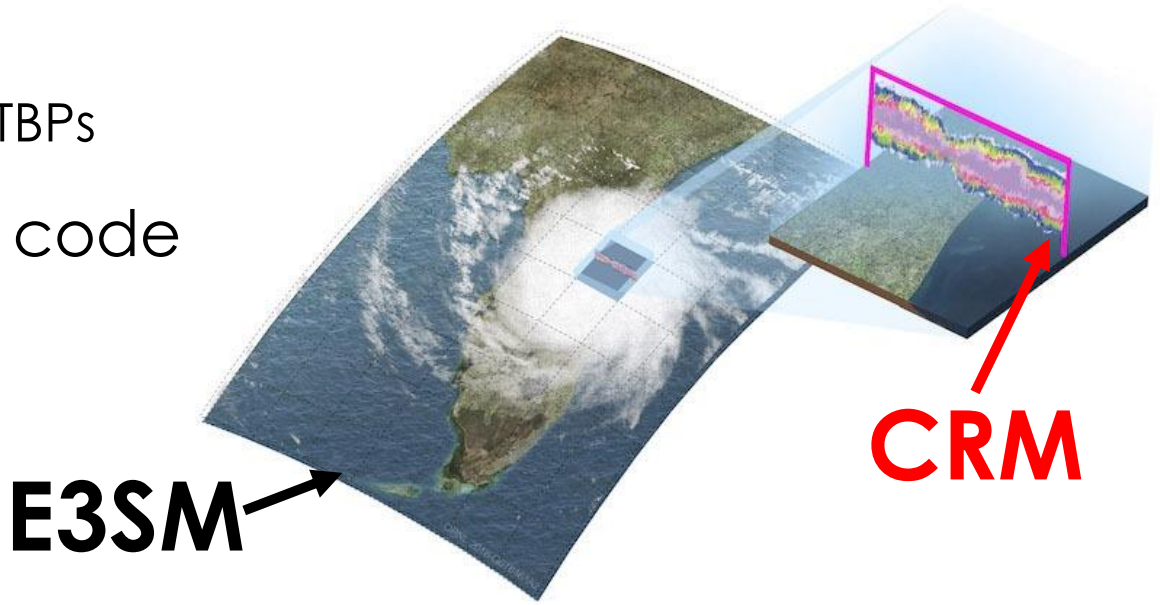
U.S. DEPARTMENT OF ENERGY

Acknowledgements

- This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.
- This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

E3SM Multiscale Modeling Framework (E3SM-MMF)

- Energy Exascale Earth System Model: U.S. DOE hi-res climate model
  - Collection of atmosphere, ocean, land surface, sea ice, and land ice
- Multi-scale Modeling Framework (E3SM-MMF)
  - Embed hi-res (1km) Cloud Resolving Model (CRM) at each global model grid point
  - Explicit simulate moist convection on reduced domain
  - Improves upon heuristic "parameterization" for moist convection
  - 2 million lines of mostly Fortran 77, 90, 2003+
  - Threading is poorly exposed
  - Heavily reliant on Fortran classes, pointers, TBPs
- >90% of runtime concentrated in CRM code
  - Only 10-20K lines of code
  - Natural target for acceleration



Approach to AMD Hardware and Portability

- We chose to use C++ portability over Fortran + Directives

1. C++ can conveniently describe any code as an object (Lambdas)
   - We can then launch this code on CUDA, HIP, SYCL, OpenMP, etc. ourselves
   - Fortran cannot do this at this time
2. C is typically the first implementation a vendor develops
   - This is critical for efforts that must work on "day 1" for new architectures
3. C is the more reliable implementation
   - Fortran is less popular overall: fewer dedicated developers
   - C++ classes/templates better defined by standards than modern Fortran constructs

- However: Porting from Fortran to C++ is time consuming
  - Also, significant effort is needed to keep C++ user-level code **readable**
  - C++ Lambda capture semantics can be confusing to domain scientists
  - Have to put guardrails on C++ experts (reliance on std::, virtual inheritance, etc.)

What is Portable C++?

- A C++ library, not a separate language or a language extension
- Based on the "kernel" paradigm (e.g., CUDA, HIP, SYCL, …):
  - A kernel performs work on a single thread
  - Let the launcher know how many threads to launch
  - Requires no more work or information than you're already used to providing

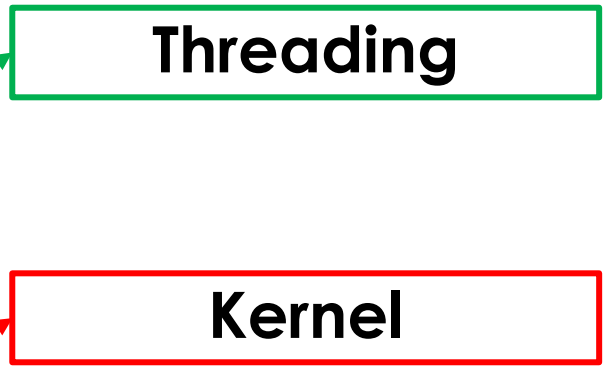
```
!$acc parallel loop collapse(4)
do l = 1 , numState
  do k = 1 , nz
    do j = 1 , ny
      do i = 1 , nx
        stateTend(i,j,k,l) = - ( stateFluxLimits(i+1,j,k,l) -
                                 stateFluxLimits(i  ,j,k,l) ) / dx;
      enddo
    enddo
  enddo
enddo
```

Loops define the threading

Kernel is the loop **body**

The Core of Portable C++

```
// for (int l=0; l < numState; l++) {
//   for (int k=0; k < nz; k++) {
//     for (int j=0; j < ny; j++) {
//       for (int i=0; i < nx; i++) {
parallel_for( Bounds<4>(numState,nz,ny,nx) ,
              YAKL_LAMBDA(int l, int k, int j, int i) {
  stateTend(l,k,j,i) = - ( stateFluxLimits(l,k,j,i+1) -
                           stateFluxLimits(l,k,j,i  ) ) / dx;
});
```



The Core of Portable C++

- C++ can pass code as an object

```
// for (int l=0; l < numState; l++) {
//   for (int k=0; k < nz; k++) {
//     for (int j=0; j < ny; j++) {
//       for (int i=0; i < nx; i++) {
parallel_for( Bounds<4>(numState,nz,ny,nx) ,
              YAKL_LAMBDA(int l, int k, int j, int i) {
  stateTend(l,k,j,i) = - ( stateFluxLimits(l,k,j,i+1) -
                           stateFluxLimits(l,k,j,i  ) ) / dx;
});
```

- C++ "lambdas" convert code into a class object for you
- You can then pass the code to whatever backend you want
  - "parallel_for" can launch with CUDA, HIP, OpenMP, OpenMP 4.5+, SYCL, etc.
- Just as flexible and generic as directives

Yet Another Kernel Launcher (YAKL)

- C++ Performance Portability Library for Fortran, simplicity, & readability
  - https://github.com/mrnorman/YAKL
- Syntax patterned after Kokkos & interoperates easily with Kokkos
- Focus on simplicity
  - Only one level of parallelism
    - Two levels of parallelism: already making statements about hardware
  - Only two memory spaces: host and device
- Allows Fortran-like behavior in multi-dimensional arrays
  - One-based indexing, column-major index ordering, basic slicing
  - Limited Fortran intrinsic library (size, shape, maxval, minloc, sum, etc.)
- Interacts with Fortran
  - Wrap existing Fortran allocations in YAKL Fortran-like Arrays
  - Pool allocator with CUDA Managed Memory hooks and Fortran hooks
  - Enables an incremental porting path with relatively little code change
- Convenient NetCDF, FFT, tridiagonal utilities for YAKL Arrays

YAKL Fortran-Like Multi-Dimensional Arrays

```
real stateTend      (nx  ,ny,nz,numState);
real stateFluxLimits(nx+1,ny,nz,numState);

!$acc parallel loop collapse(4)
do l = 1 , numState
  do k = 1 , nz
    do j = 1 , ny
      do i = 1 , nx
        stateTend(i,j,k,l) = - ( stateFluxLimits(i+1,j,k,l) -
                                 stateFluxLimits(i  ,j,k,l) ) / dx;
      enddo
    enddo
  enddo
enddo
```

Fortran code: Finite-Volume update from fluxes

Equivalent Portable C++

```
real4d stateTend      ("stateTend"      ,nx  ,ny,nz,numState);
real4d stateFluxLimits("stateFluxLimits",nx+1,ny,nz,numState);

// do l = 1 , numState
//   do k = 1 , nz
//     do j = 1 , ny
//       do i = 1 , nx
parallel_for( Bounds<4>(numState,nz,ny,nx) ,
              YAKL_LAMBDA(int l, int k, int j, int i) {
  stateTend(i,j,k,l) = - ( stateFluxLimits(i+1,j,k,l) -
                           stateFluxLimits(i  ,j,k,l) ) / dx;
});
```

YAKL Fortran-Like Multi-Dimensional Arrays

- YAKL Arrays inherently behave like Fortran assumed-shape arrays
  - They carry all metadata with them:

```
real3d transpose(real3d &a) {
  int nz = size(arr,3);
  int ny = size(arr,2);
  int nx = size(arr,1);
  real3d a_t("a_t",nz,ny,nx);
  parallel_for( Bounds<3>(nz,ny,nx) , YAKL_LAMBDA(int k, int j, int i) {
    a_t(k,j,i) = a(i,j,k);
  });
  return a_t;
}
```

- In practice, you would want to tile this loop

YAKL Reduction & Limited Fortran Intrinsics Library

- Computing the maximum stable time step for Shallow Water (**Portable C++**)

```
real max_stable_dt(real2d const &height, real2d const &u,
                   real2d const &v, real grav, real cfl,
                   real dx, real dy) {
  real2d dt2d("dt2d",nx,ny);
  parallel_for( Bounds<2>(ny,nx) , YAKL_LAMBDA (int j, int i) {
    real g_w = sqrt(grav * height(i,j));
    real dt_x = cfl * dx / ( abs(u(i,j)) + g_w );
    real dt_y = cfl * dy / ( abs(v(i,j)) + g_w );
    dt2d(i,j) = min( dt_x , dt_y );
  });
  return minval( dt2d );  // This is a device-wide reduction
}
```

Future Plans for YAKL

- YAKL currently works on CPU (serial), Nvidia GPUs, and AMD GPUs
- YAKL works on Intel GPUs with OpenMP Offload backend
  - Except for atomic min and max instructions (waiting on OpenMP 5.1 implementations)
- SYCL backend (Intel) is waiting on "is_memcpyable" implementation
- Coming soon to a YAKL near you:
  - CPU threading (OpenMP first, then pthreads for more efficiency if needed)
  - An approach to cache blocking and SIMD vectorizations
- Long term plan: create a version of YAKL that wraps Kokkos for as many components as possible

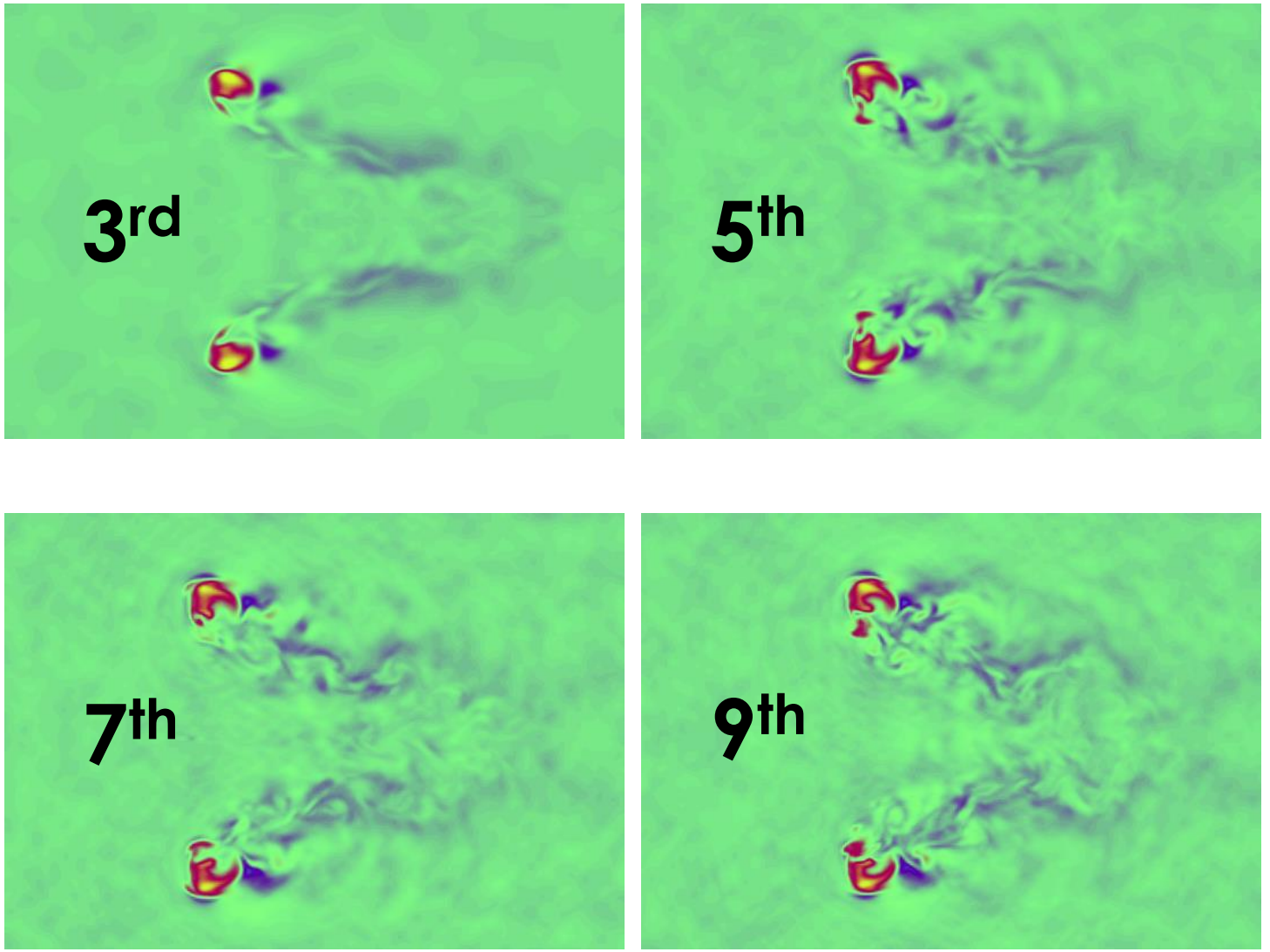
Algorithmic Advances for GPU Hardware

- Important to consider algorithms better suited for GPUs
- Arithmetically intense algorithms perform well on all architectures
- Increase computations while decreasing data movement

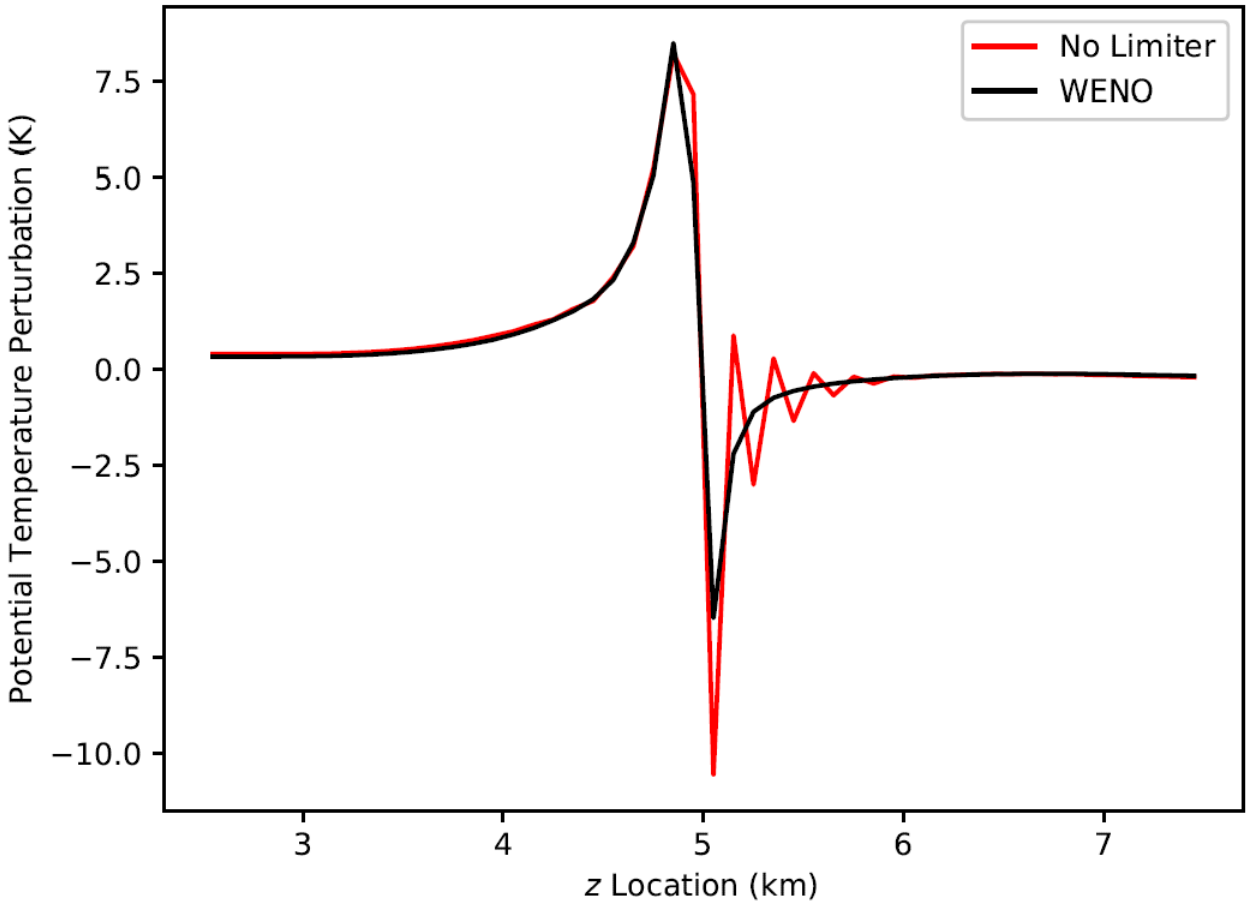
- E3SM-MMF Algorithmic Improvement Examples:
  - "A high-order WENO-limited finite-volume algorithm for atmospheric flow using the ADER-differential transform time discretization" https://rmets.onlinelibrary.wiley.com/doi/full/10.1002/qj.3989
  - "A Holistic Algorithmic Approach to Improving Accuracy, Robustness, and Computational Efficiency for Atmospheric Dynamics" https://epubs.siam.org/doi/abs/10.1137/19M128435X

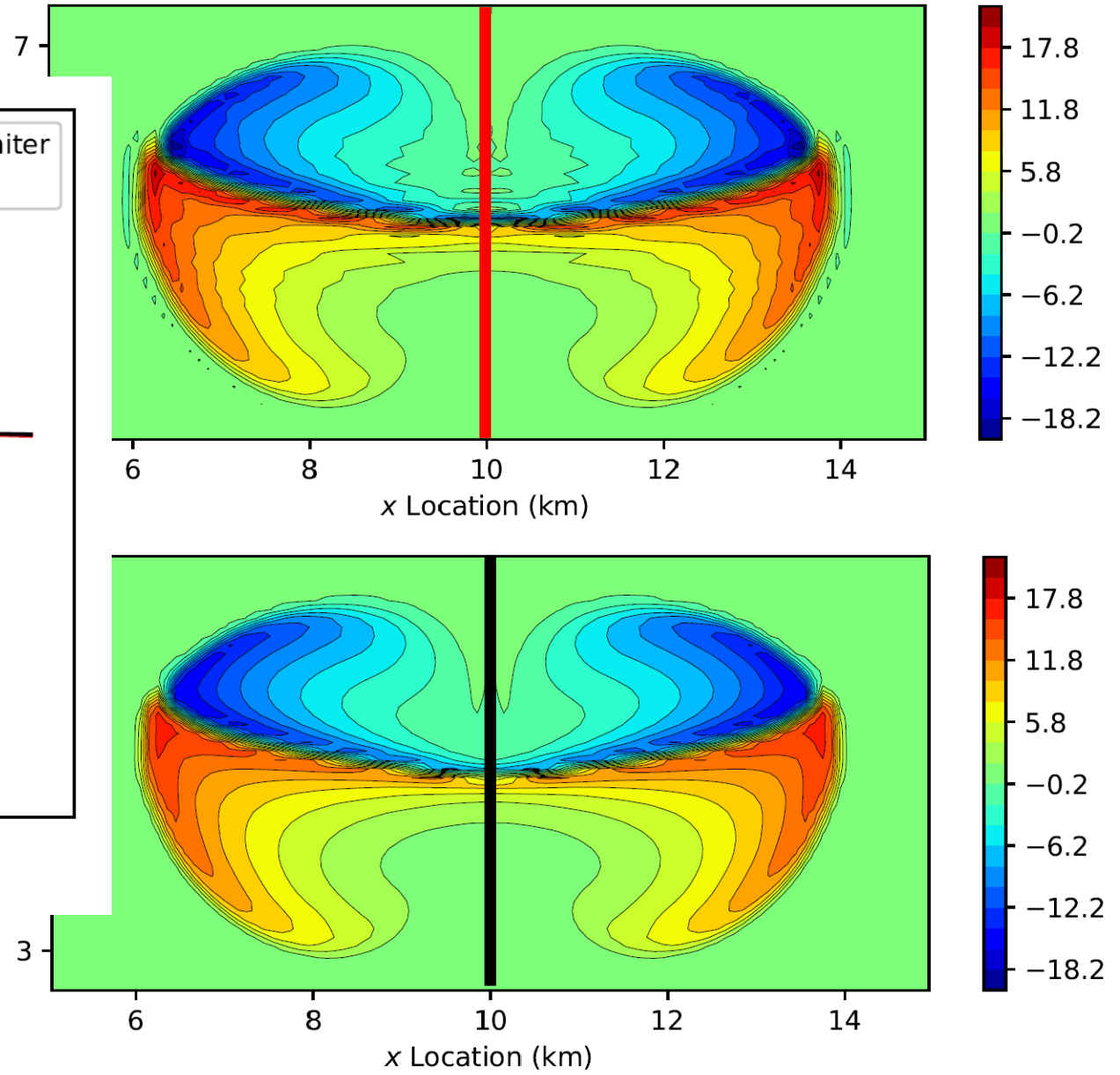
Algorithmic Advances for GPU Hardware: High-Order

- Splitting supercell test case
  - Cartesian grid
  - High CAPE, initial thermal
  - Splits into two cells
  - Doesn't use added viscosity from test case
- AWFL + P3 microphysics
- 20 second physics time step
  - Dycore sub-cycled
- 5km vertical velocity after 2 hrs plotted on right



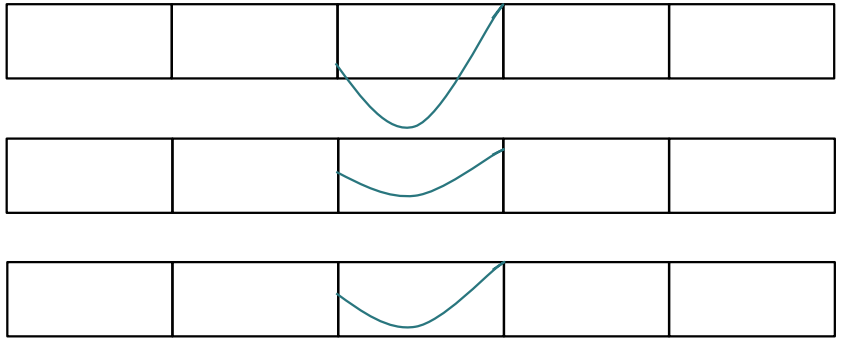
Algorithmic Advances for GPU Hardware: WENO Limiting





New Algorithm Increases Arithmetic Intensity

- Reconstruct variation from stencil
- Apply WENO limiting
- Compute high-order ADER time-average



- Compute upwind fluxes

- Update the cell average from fluxes

- Vast majority of computations use only a small stencil of data from DRAM
  - Significant compute intensity
  - Time stepping happens in the same loop as reconstruction / limiting

- Most expensive kernels: 80% peak flop/s on V100; "Quite good" on MI-60
- Currently working on MI-100 results

Further Resources & Questions

- YAKL Documentation: https://github.com/mrnorman/YAKL
- miniWeather parallel programming training app: https://github.com/mrnorman/miniWeather
- Readable C++: https://tinyurl.com/readable-cplusplus
- C++ Portability for Directives Folks: https://tinyurl.com/cplusplus-portability
- Other (more functional) C++ Portability Frameworks:
  - Kokkos: https://github.com/kokkos/kokkos/
  - RAJA: https://github.com/LLNL/RAJA
    - UMPIRE: https://github.com/LLNL/Umpire
  - SYCL: https://github.com/KhronosGroup/SYCL-Docs

Challenges with Portable C++

- Vectorization
  - C++ has less information than Fortran
  - C++ pointers need "restrict" keyword to allow the compiler to optimize more
- Staying away from non-portable C++ features
  - Most of the standard template library ("std::") is either invalid or non-portable for GPUs
  - Virtual functions in class inheritance is difficult on GPUs
- Unified shared memory helps with "std::" , **but**…
  - Unified memory often performs poorly
  - Unified memory isn't portable to all compilers and hardware
- C++ Lambdas behave strangely
  - They only "capture" needed data automatically if it's in the local scope
  - Global scope and class data are **not** captured automatically (more on this later)
- Portable C++ requires a rewrite of Fortran
  - However, YAKL significantly reduces this burden by allowing Fortran-like behavior

YAKL Scalar Live-Out

- Scalars written to in a device kernel and read from outside the kernel
  - Scalars are normally thread-private; here they must be allocated on the device
  - "Scalar Live-Out" is most common in error-checking routines

YAKL Scalar Live-Out

Fortran

```
logical :: data_is_bad
data_is_bad = .false.
!$acc parallel for collapse(3) copy(data_is_bad) async(id)
do k = 1 , nz
  do j = 1 , ny
    do i = 1 , nz
      if (density(i,j,k) <= 0) data_is_bad = .true.
    enddo
  enddo
enddo
!$acc wait(id)
if (data_is_bad) then
  ...
```

C++ Portability

```
ScalarLiveOut<bool> data_is_bad(false);
parallel_for( Bounds<3>(nz,ny,nx) , YAKL_LAMBDA (int k, int j, int i) {
  if (density(i,j,k) <= 0) data_is_bad = .true.
});
if (data_is_bad.hostRead()) { ...
```

YAKL Pool Allocator (Gator)

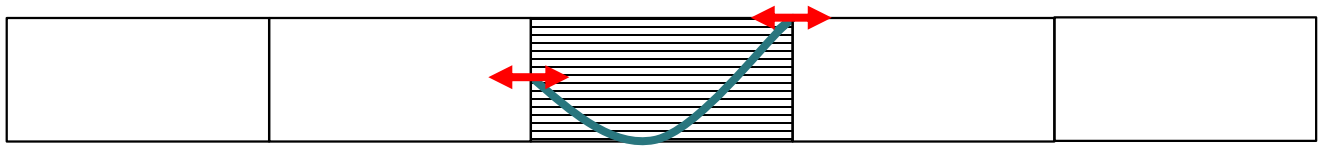
- The YAKL "pool alligator" (speech-to-text error)
- Allows cheap frequent allocations & free's
  - Optimized for "stack-like" allocation patterns
  - Most Fortran apps allocate in a stack-like fashion
  - Little to no segmentation and wasted memory
  - Does not behave well for random allocation patterns
- Fortran hooks for up to 7-D arrays
- Allows CUDA & HIP Managed Memory
  - Automatic hooks into OpenACC and OpenMP runtimes so data is handled by CUDA & HIP
- Controlled by environment variables

Readability

- Able to find the code
  - Need a **very** good reason for > 1 level of abstraction
  - Some code duplication is OK, it's a balance
  - Loops should live together, one after the other
- Comfortable syntax
  - The syntax should be familiar, Fortran-like, and clear
  - No "figuring out" what the code is doing (nothing "clever" or any such adjective)
  - Hide metaprogramming, SFINAE, complex template expressions from the user
  - Developer profanity should be minimized
- Looping should be clear
  - Typically no need for C++ iterators
- Data structures should be clear (Multi-dimensional Arrays)

Upwind Finite-Volume Spatial Discretization

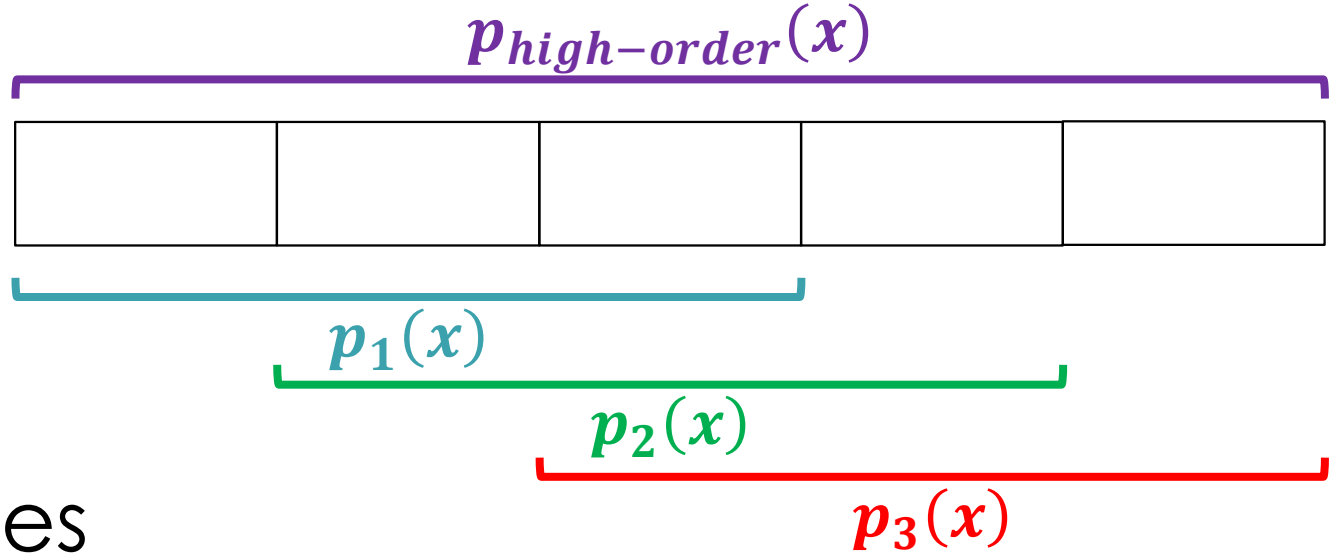
- Finite-Volume Algorithm
  - Solution is a set of non-overlapping cell averages
  - Cell average updates based on cell-edge fluxes
  - Use upwind Riemann solver to determine fluxes
  - Reconstruct intra-cell variation from surrounding "stencil" of cells



- Advantages
  - Conserves variables to machine precision
  - Large time step
  - Treats each Degree Of Freedom individually (accuracy)
  - Unconditionally stable for Euler eqns without added dissipation

Weighted Essentially Non-Oscillatory Limiting (WENO)

- WENO Algorithm
  - Compute multiple polynomials using multiple stencils
  - **Weight the most oscillatory polynomials the lowest**
  - Custom low-dissipation implementation



- Advantages
  - **Requires no additional data** when used with Finite-Volume
  - Very accurate and effective at limiting oscillations

Other Advantages of Portable C++

- Debugging
  - Fortran array metadata can change in routine interfaces (# of dimensions & sizes)
    - Thus, Fortran index checking can never fully be trusted
  - C++ Array class metadata is tied to the Array object and never changes
    - C++ index checking is far more reliable because of this
  - C++ exception throwing makes stack tracing more direct as well (e.g., in gdb)
- C++ compilers inline much better than Fortran compilers typically do
  - Rarely any need for an equivalent of "!$acc routine" or "!$omp declare target"
  - "!$acc routine" and OpenMP equivalent are typically quite buggy
- Simple C++ templating can significantly reduce code duplication
  - No more complex Fortran interface blocks for multiple data types

YAKL Fortran-Like Multi-Dimensional Arrays

- Use C++ “typedef” and “using” in a header to hide unsightly C++ namespace and template expressions

```
typedef yakl::Array<float,1,yakl::memDevice,yakl::styleFortran> real1d;
typedef yakl::Array<float,2,yakl::memDevice,yakl::styleFortran> real2d;
...
using yakl::fortran::parallel_for;
using yakl::fortran::Bounds;
```

YAKL Atomic

- Compute a column average:

Fortran

```
function col_average(var) rslt(avg)
  real, intent(in) :: var(:,:,:)
  real             :: avg(:)
  real tmp, factor
  factor = 1./(nx*ny)
  !$acc parallel loop
  do k = 1 , size(var,3)
    avg(k) = 0
  enddo
  !$acc parallel loop collapse(3)
  do k = 1 , size(var,3)
    do j = 1 , size(var,2)
      do i = 1 , size(var,1)
        tmp = var(i,j,k) / factor
        !$acc atomic update
        avg(k) = avg(k) + tmp
      enddo
    enddo
  enddo
end function max_stable_dt
```

C++ Portability

```
real1d col_average(real3d const &var) {
  real factor = 1./(nx*ny);
  real1d avg("avg",size(var,3));
  memset( avg , 0. );
  parallel_for( Bounds<3> (size(var,3),size(var,2),size(var,1)) ,
                YAKL_LAMBDA (int k, int j, int i) {
    // avg(k) = avg(k) + var(i,j,k) / factor
    yakl::atomicAdd( avg(k) , var(i,j,k) / factor );
  });
  return avg;
}
```