Algorithmic Choices that Improve Hardware Utilization and Accuracy

Matthew Norman
Oak Ridge Leadership Computing Facility
https://mrnorman.github.io

The Challenge of Accelerated Computing

- Must reduce power consumption
  - Less cache
  - Slower memory clock
  - Wider memory bus
  - Compute power >> Bandwidth

- Nvidia V100 GPU
  - Capable of 15 teraflop/s (single precision)
  - Can only feed in 225 billion single floats per second
  - Most FP operations require two floats per operation
  - Bandwidth is **134x too slow**

The Challenge of Accelerated Computing

- The Cray-1 Vector Machine (1975)
  - 160 megaflop/s
  - 20 million single floats per second
  - Bandwidth only 16x too slow

- We’ve been here before, but not this extremely

What Do We Need From Algorithms?

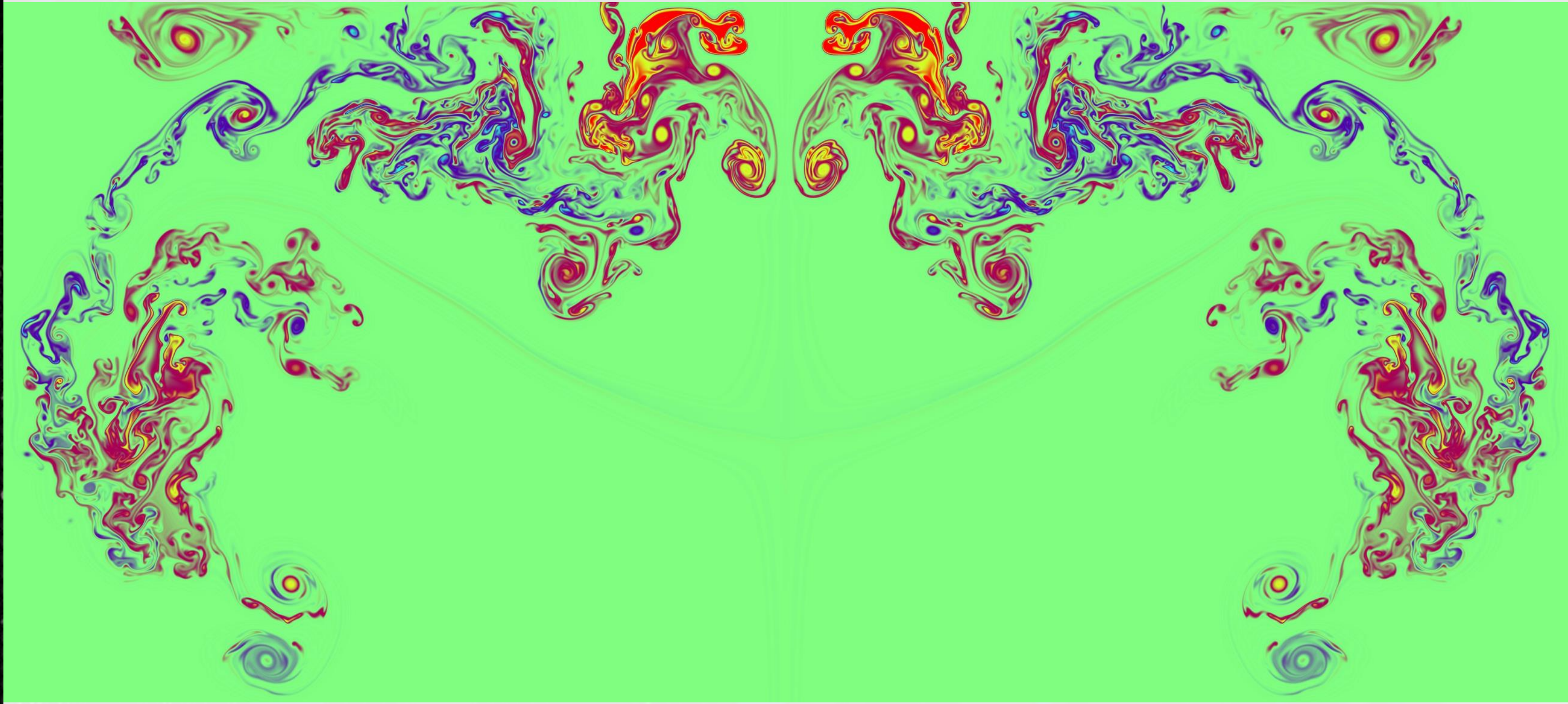
- We need more computations per data fetch (**Compute Intensity**)
  - GPUs have a small amount of fast on-chip cache
  - Load a small amount of data from main memory
  - Perform many computations within cache before writing back to memory
- We need less algorithmic dependence
  - Each global synchronization kicks your data out of cache
  - Each global loop through the data has a roughly fixed cost
    - You pay for out-of-cache data accesses, not computations
- We need less data movement over network
  - Network fabric is very slow compared to on-node memory
  - Want as few transfers as possible and as small as possible

The Euler Equations

- Euler equations govern atmospheric dynamics
  - Conservation of mass, momentum, & energy with gravity source term
  - Hyperbolic system of conservation laws
    - Waves travel at the speed of wind and the speed of sound

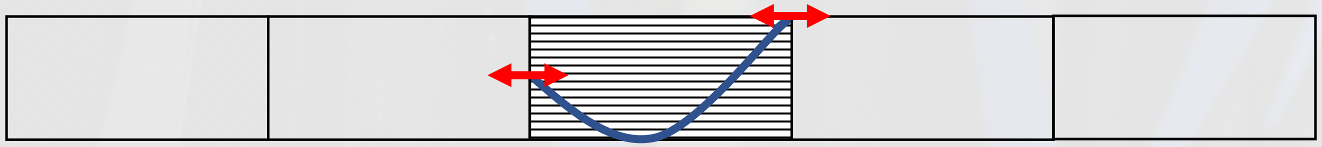
$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho u\\ \rho v\\ \rho w\\ \rho\theta\end{bmatrix}+\frac{\partial}{\partial x}\begin{bmatrix}\rho u\\ \rho u^2+p\\ \rho uv\\ \rho uw\\ \rho u\theta\end{bmatrix}+\frac{\partial}{\partial y}\begin{bmatrix}\rho v\\ \rho vu\\ \rho v^2+p\\ \rho vw\\ \rho v\theta\end{bmatrix}+\frac{\partial}{\partial z}\begin{bmatrix}\rho w\\ \rho wu\\ \rho wv\\ \rho w^2+p-p_H\\ \rho w\theta\end{bmatrix}=\begin{bmatrix}0\\ 0\\ 0\\ -\left(\rho-\rho_H\right)g\\ 0\end{bmatrix}$$

The Euler Equations



Upwind Finite-Volume Spatial Discretization

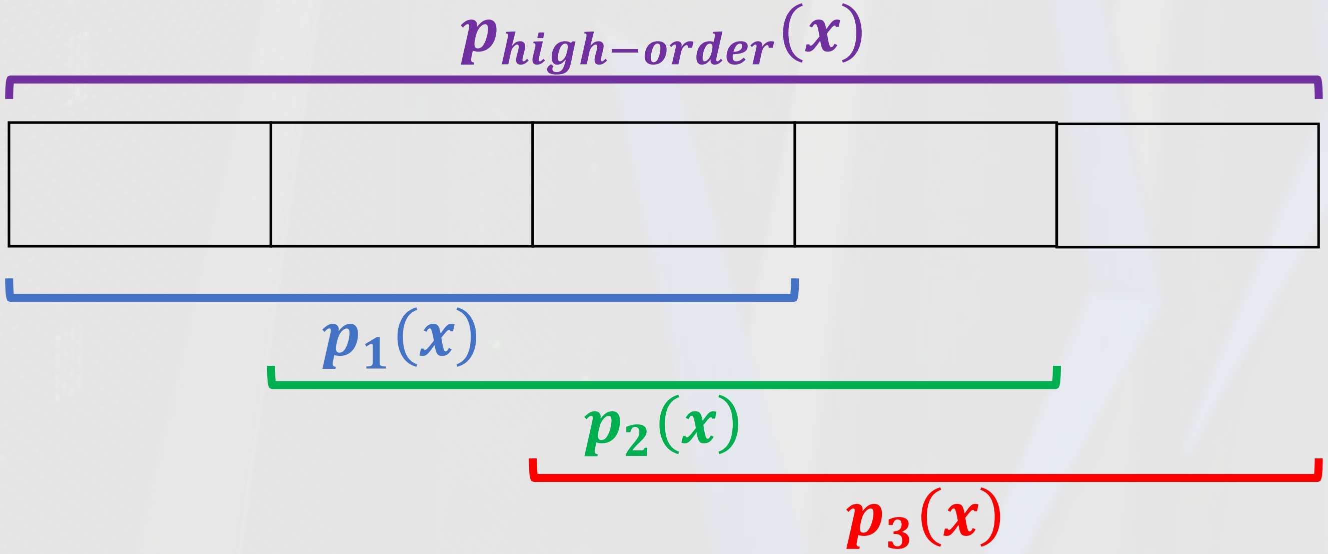
- Finite-Volume Algorithm
  - Solution is a set of non-overlapping cell averages
  - Cell average updates based on cell-edge fluxes
  - Use upwind Riemann solver to determine fluxes
  - Reconstruct intra-cell variation from surrounding “stencil” of cells



- Advantages
  - Conserves variables to machine precision
  - Large time step (CFL=1)
  - Treats each Degree Of Freedom individually (accuracy)
  - Stable for non-shock 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 (Norman & Nair, 2019, JAMES)



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

Arbitrary DERivatives (ADER) Time Discretization

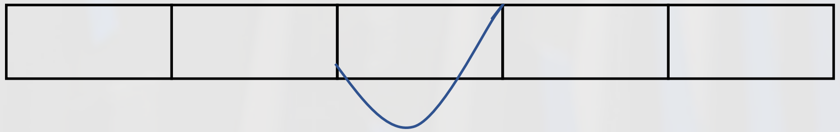
- ADER Algorithm
  - PDE itself translates spatial variation into temporal variation
  - $\frac{\partial q}{\partial t} = -\frac{\partial q}{\partial x}$ Differentiation gives higher-order time derivatives

$$\frac{\partial q}{\partial t} = -\frac{\partial q}{\partial x} \rightarrow \frac{\partial^2 q}{\partial t^2} = \frac{\partial^2 q}{\partial x^2} \rightarrow \frac{\partial^3 q}{\partial t^3} = -\frac{\partial^3 q}{\partial x^3}$$

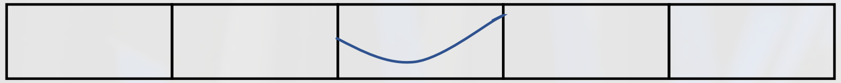
  - Use Differential Transforms for greater efficiency for non-linear PDEs
- Advantages
  - **Requires no additional data** for high-order time integration
  - Automatically propagates WENO limiting through time dimension
  - Allows larger time step than existing explicit ODE time integrators
    - Courant number of 1 for FV
  - More accurate than existing ODE time integrators

Algorithm Summary

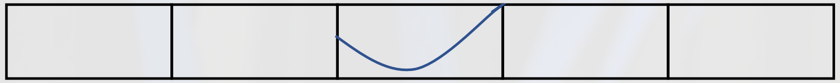
- **Reconstruct variation from stencil**



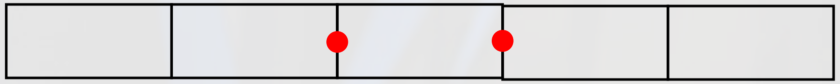
- **Apply WENO limiting**



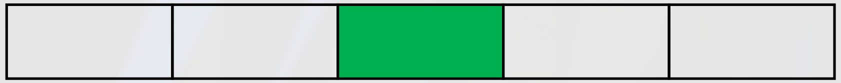
- **Compute high-order ADER time-average**



- Compute upwind fluxes

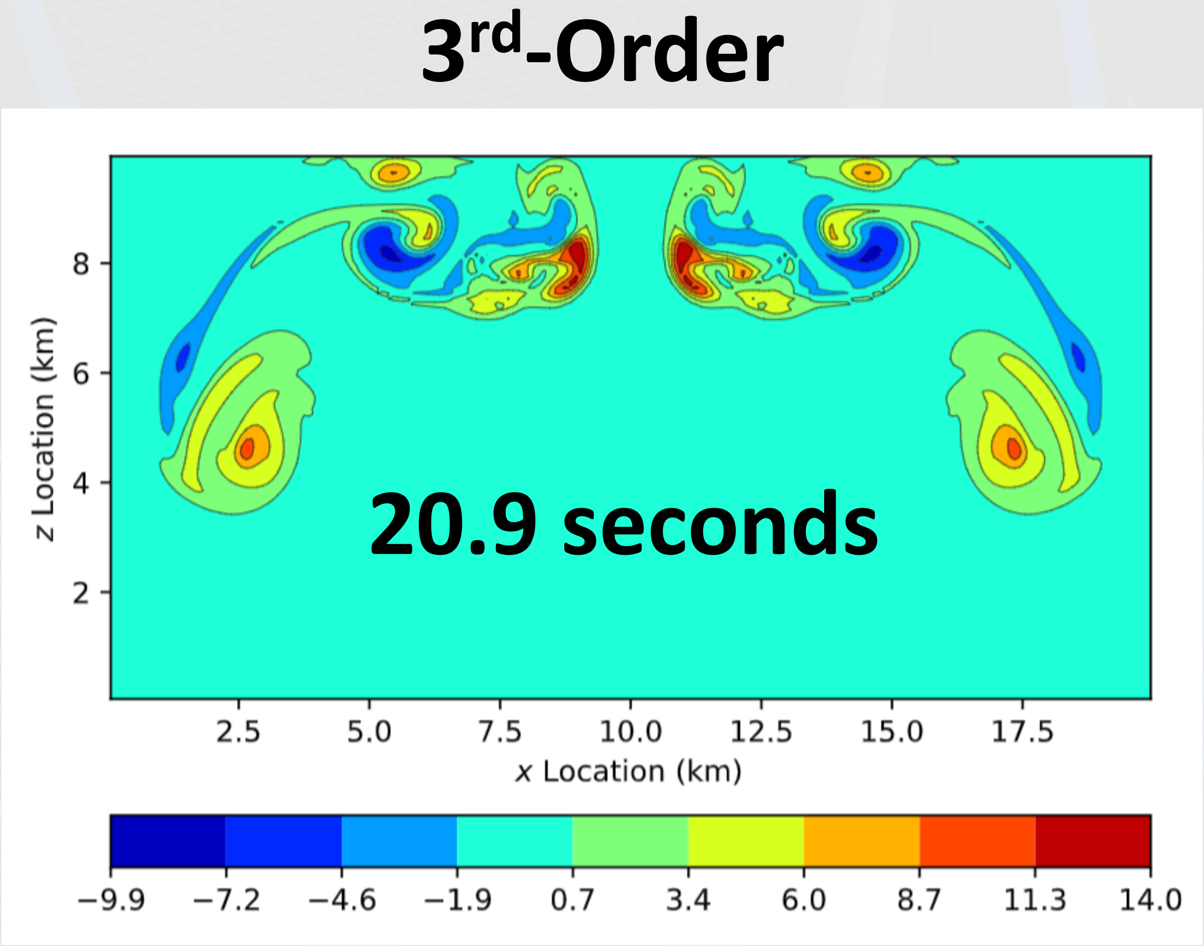


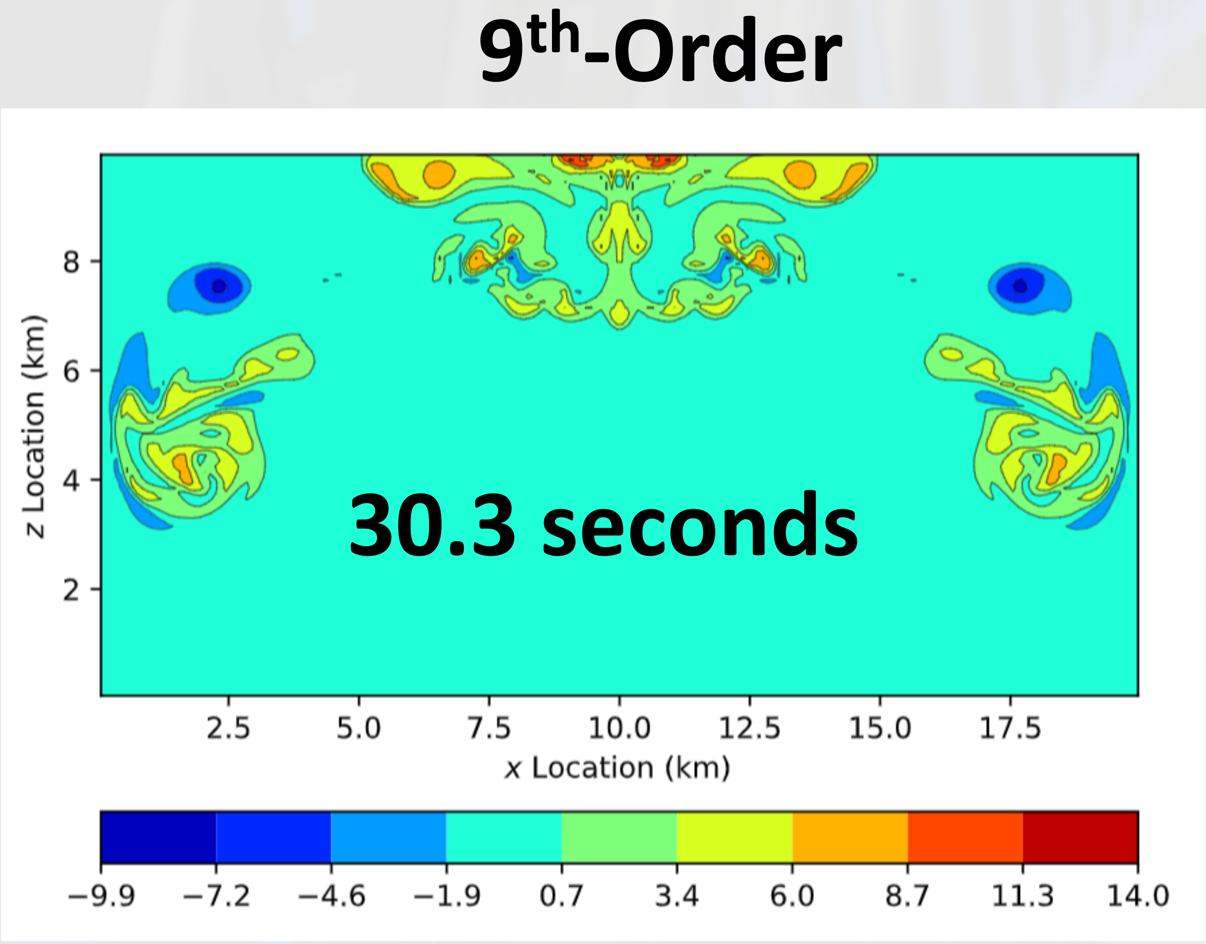
- Update the cell average from fluxes



- **Nearly all computations use only a small stencil of data**
- **Significant compute intensity**

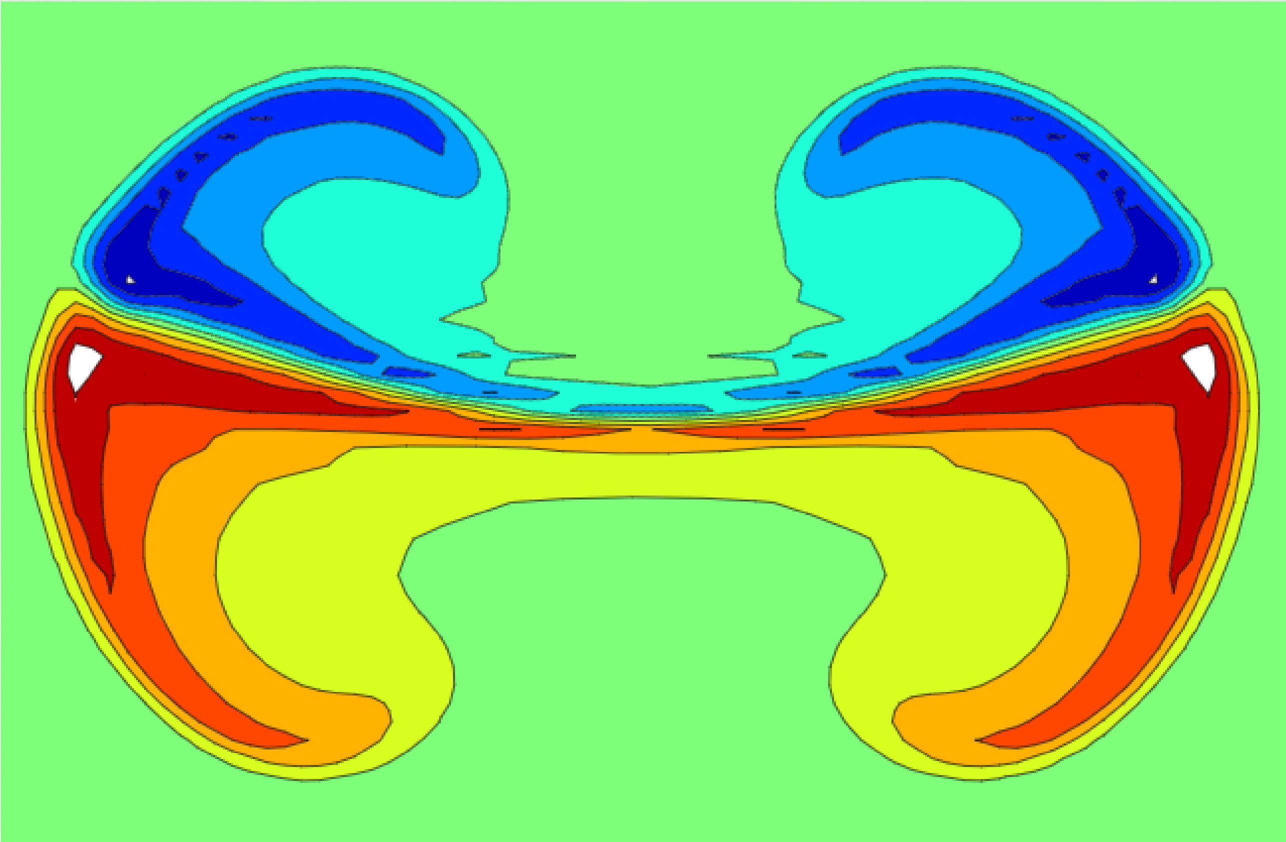
Accuracy



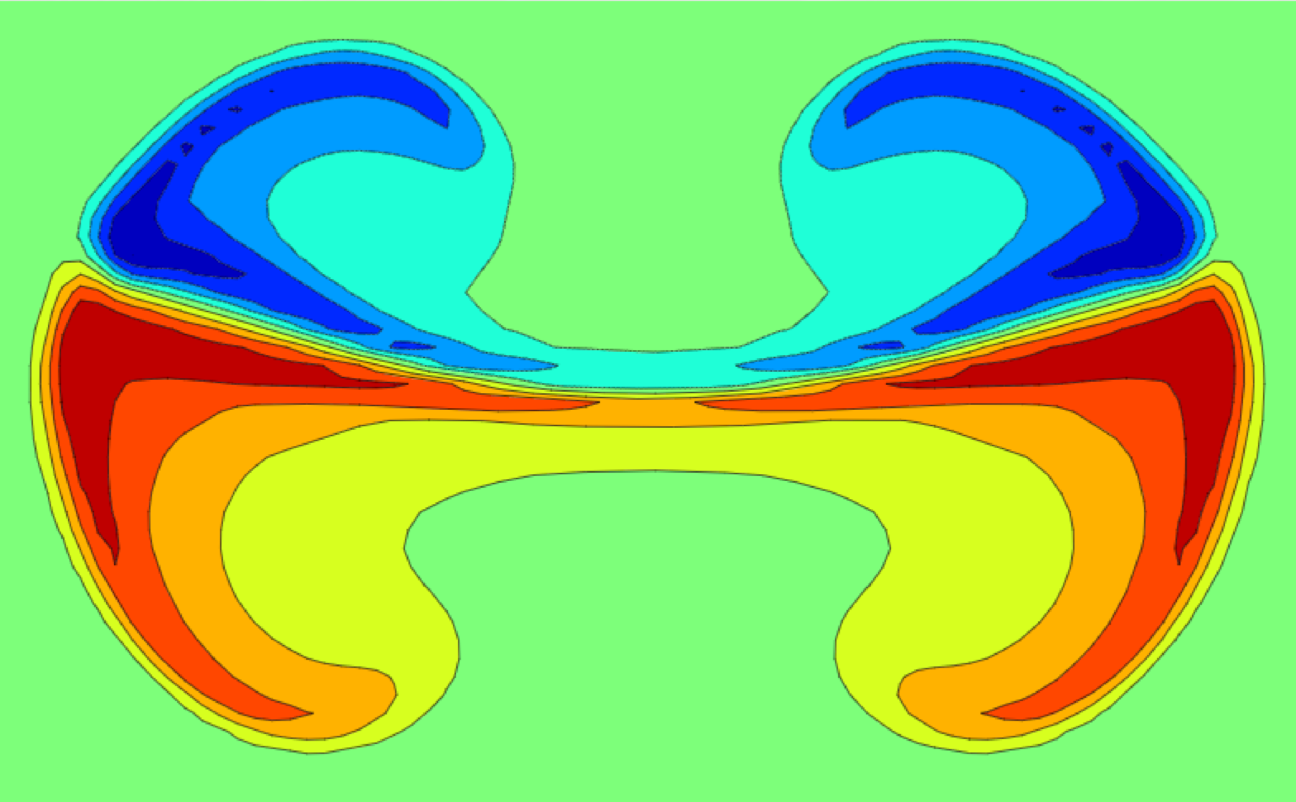


- 9th-order has **6x** more computations than 3rd-order (hardware counters)
- But it only costs 45% more on GPUs

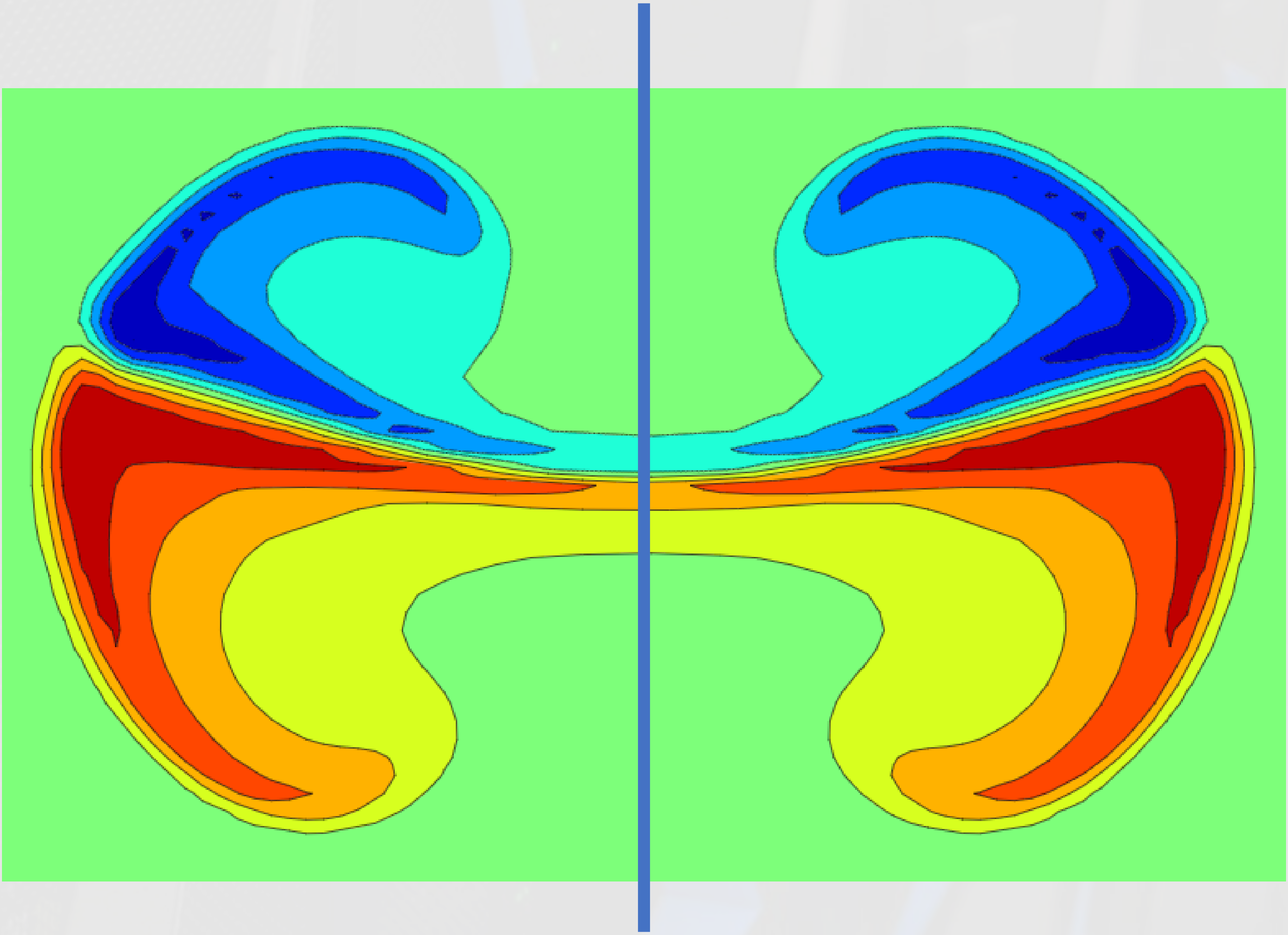
Robustness



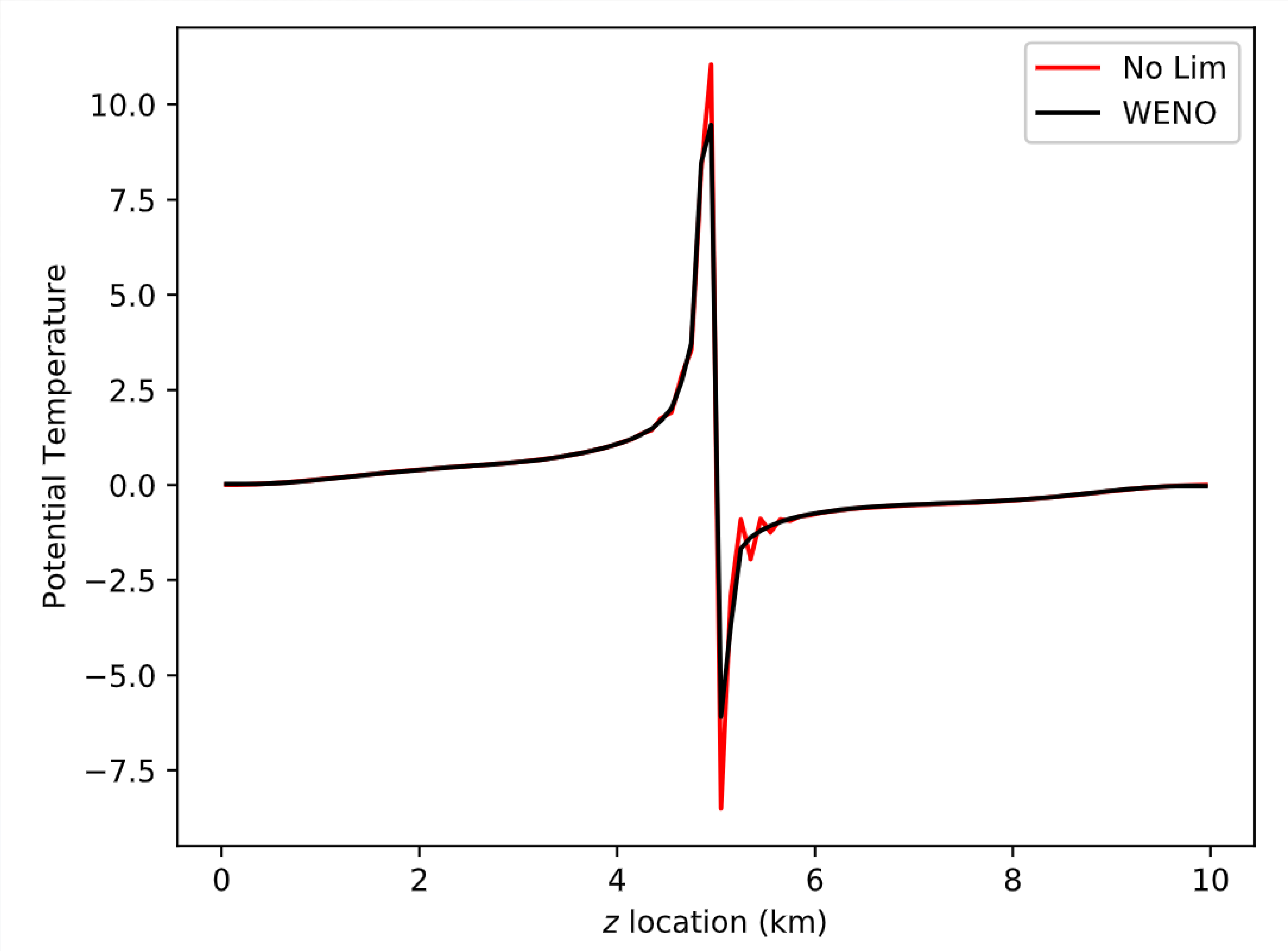
Robustness



Robustness



Robustness



No Lim
WENO
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
Potential Temperature
0
2
4
6
8
10
z location (km)

Robustness

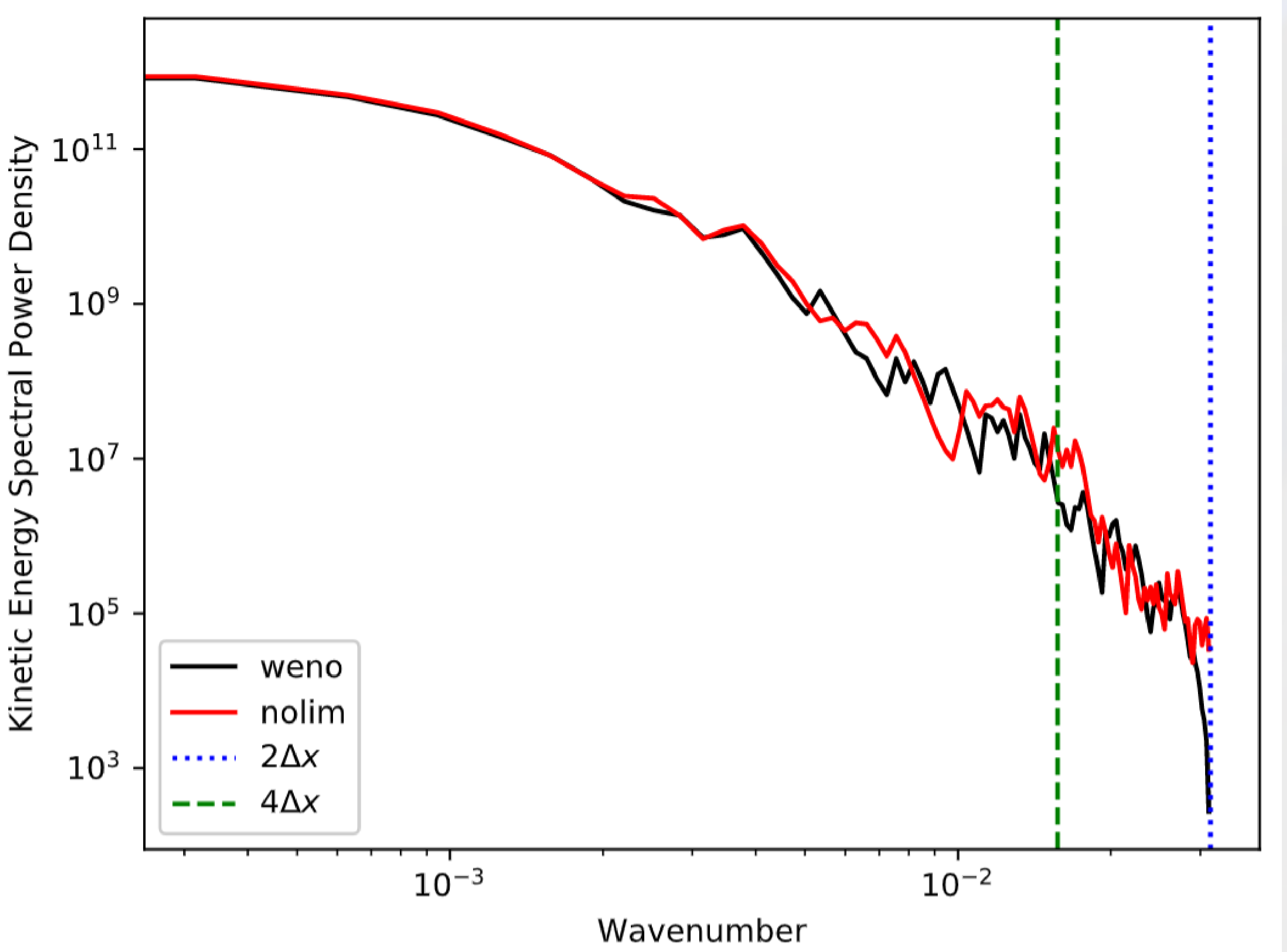
KE spectra

- 2-D simulation

NoLim: 26.2 sec
WENO: 30.3 sec

WENO has **16x** more computations than no limiting (HW counters)

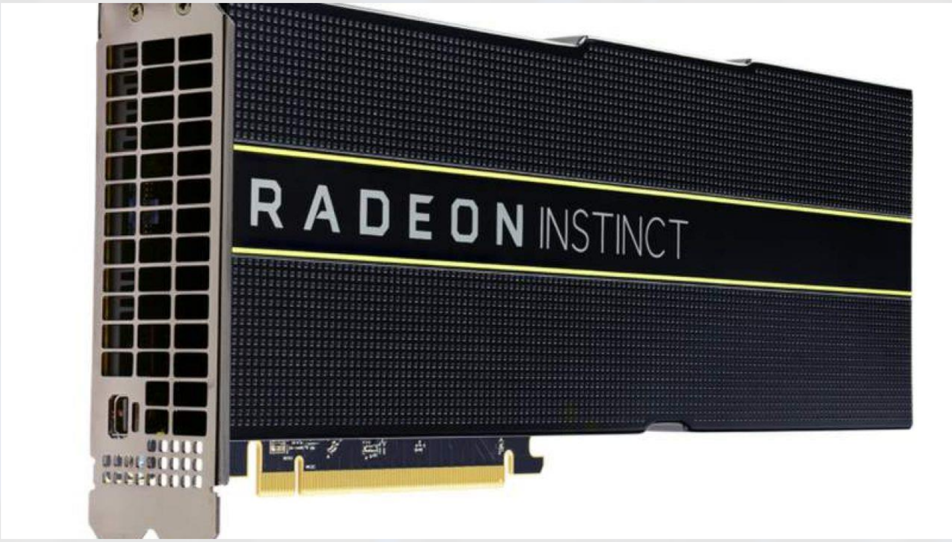
But it's only 15% more expensive on GPUs



Performance (Most Expensive GPU Kernel)

<u>Nvidia V100 GPU</u>

- 80% peak flop/s
- 11.9 trillion flop/s



<u>AMD MI60 GPU</u>

- 40% peak flop/s
- 5.9 trillion flop/s

C++ Performance Portability Approach

- Kernels specified as C++ Lambdas describing the work of one thread
    - Simply CUDA with different syntax
    - Burden of exposing parallelism is on the developer
    - Once exposed, parallelism is very portable across architectures
- Use multi-dimensional array classes for data
    - Object-bound dimension sizes → robust bounds checking
    - “Shallow copy” for easy GPU portability (allows Lambda capture-by-value)
- Launchers run the kernel with multiple backend options

C++ Performance Portability Approach

```
inline void applyTendencies(realArr &state2, real const c0, realArr const &state0,
                                              real const c1, realArr const &state1,
                                              real const ct, realArr const &tend,
                                              Domain const &dom) {
  for (int l=0; l<numState; l++) {
    for (int k=0; k<dom.nz; k++) {
      for (int j=0; j<dom.ny; j++) {
        for (int i=0; i<dom.nx; i++) {
          state2(l,hs+k,hs+j,hs+i) = c0 * state0(l,hs+k,hs+j,hs+i) +
                                     c1 * state1(l,hs+k,hs+j,hs+i) +
                                     ct * dom.dt * tend(l,k,j,i);
        }
      }
    }
  }
}
```

C++ Performance Portability Approach

```
inline void applyTendencies(realArr &state2, real const c0, realArr const &state0,
                                             real const c1, realArr const &state1,
                                             real const ct, realArr const &tend,
                                             Domain const &dom) {
  for (int l=0; l<numState; l++) {
    for (int k=0; k<dom.nz; k++) {
      for (int j=0; j<dom.ny; j++) {
        for (int i=0; i<dom.nx; i++) {
          state2(l,hs+k,hs+j,hs+i) = c0 * state0(l,hs+k,hs+j,hs+i) +
                                     c1 * state1(l,hs+k,hs+j,hs+i) +
                                     ct * dom.dt * tend(l,k,j,i);
        }
      }
    }
  }
}
```

Parallelism

Kernel

C++ Performance Portability Approach

```
inline void applyTendencies(realArr &state2, real const c0, realArr const &state0,
                                             real const c1, realArr const &state1,
                                             real const ct, realArr const &tend,
                                             Domain const &dom) {
  // for (int l=0; l<numState; l++) {
  //   for (int k=0; k<dom.nz; k++) {
  //     for (int j=0; j<dom.ny; j++) {
  //       for (int i=0; i<dom.nx; i++) {
  yakl::parallel_for( numState*dom.nz*dom.ny*dom.nx , YAKL_LAMBDA (int iGlob) {
    int l, k, j, i;
    unpackIndices(iGlob,numState,dom.nz,dom.ny,dom.nx,l,k,j,i);
    state2(l,hs+k,hs+j,hs+i) = c0 * state0(l,hs+k,hs+j,hs+i) +
                               c1 * state1(l,hs+k,hs+j,hs+i) +
                               ct * dom.dt * tend(l,k,j,i);
  });
}
```

C++ Performance Portability Approach

```
inline void applyTendencies(realArr &state2, real const c0, realArr const &state0,
                                              real const c1, realArr const &state1,
                                              real const ct, realArr const &tend,
                                              Domain const &dom) {
  // for (int l=0; l<numState; l++) {
  //   for (int k=0; k<dom.nz; k++) {
  //     for (int j=0; j<dom.ny; j++) {
  //       for (int i=0; i<dom.nx; i++) {
  yakl::parallel_for( numState*dom.nz*dom.ny*dom.nx , YAKL_LAMBDA (int iGlob) {
    int l, k, j, i;
    unpackIndices(iGlob,numState,dom.nz,dom.ny,dom.nx,l,k,j,i);
    state2(l,hs+k,hs+j,hs+i) = c0 * state0(l,hs+k,hs+j,hs+i) +
                               c1 * state1(l,hs+k,hs+j,hs+i) +
                               ct * dom.dt * tend(l,k,j,i);
  });
}
```

Parallelism

Kernel

C++ Performance Portability Approach

- CPU Backend

```
template <class F> void parallel_for( int const nIter , F &f ) {
  for (int i=0; i<nIter; i++) {
    f(i);
  }
}
```

C++ Performance Portability Approach

- Nvidia CUDA Backend

```
template <class F> __global__ void cudaKernel(int nThreads, F f) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < nThreads) { f( i ); }
}
int const vectorSize = 128;
template <class F> void parallel_for(int nThreads, F &f) {
  cudaKernel <<< (nIter-1)/vectorSize+1 , vectorSize >>> ( nThreads , f);
}
```

C++ Performance Portability Approach

- AMD HIP Backend

```
template <class F> __global__ void hipKernel(int nThreads, F f) {
  int i = hipBlockIdx_x*hipBlockDim_x + hipThreadIdx_x;
  if (i < nThreads) { f( i ); }
}
int const vectorSize = 128;
template<class F> void parallel_for( int const nThreads, F const &f ) {
  hipLaunchKernelGGL( hipKernel , dim3((nIter-1)/vectorSize+1) , dim3(vectorSize),
                      (std::uint32_t) 0 , (hipStream_t) 0 , nThreads , f );
}
```

AMD GPU Status

- Cloud dycore running efficiently on AMD MI60 GPUs using YAKL
  - github.com/mrnorman/awflCloud
  - github.com/mrnorman/YAKL ("Yet Another Kernel Launcher")
  - Eventual transition to Kokkos kernel launchers ("`parallel_for`")
- miniWeather Fortran code running on AMD GPUs with OpenMP 4.5
  - Using the Mentor Graphics gfortran compiler development
  - github.com/mrnorman/miniWeather
- SCREAM physics will use C++ & Kokkos
  - Kokkos HIP backend coming soon
- Sending kernels to AMD / Mentor Graphics to improve maturity
  - UKMO Psyclone generated Fortran kernels
  - RRTMGP OpenMP 4.5 port (coming soon)

Future Work: Handling Stiff Acoustics

- Vertical acoustic stiffness
  - 100:1 aspect ratio for horiz / vertical grid spacing at surface
  - Sound waves is 370 m/s, but wind at surface is order 1 m/s
- Approach 1: First-order upwind acoustics
  - Need accurate, large time step IMplicit-EXplicit (IMEX) Runge-Kutta
  - $\geq$ 4 tridiagonal solves per time step
- Approach 2: Infinite sound speed; Poisson pressure solve
  - Only 1 tridiagonal solve per time step for pressure
  - Diagnostic density advected with the other variables
- Approach 3: High-order coupled implicit vertical
  - Potentially better on GPU, but <u>much</u> more time consuming
  - Requires many loop iterations through data

Summary

- Download this presentation
  - tinyurl.com/norman-mc19