

CAAR Porting Experiences: RAPTOR

Ramanan Sankaran

Oak Ridge National Laboratory

CAAR Team

Joseph C. Oefelein, Georgia Tech

Ramanan Sankaran, K. C. Gottiparthi, ORNL

Brian Rogers, Lixiang Luo, IBM

Levi Barnes, NVIDIA

Acknowledgements

This research used resources of the Oak Ridge Leadership Computing Facility at ORNL, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

RAPTOR – Software for Large Eddy Simulation (LES)

- Theoretical framework (Comprehensive)
 - Fully-coupled, compressible conservation equations
 - Real-fluid equation(s) of state for non-ideal gas/liquid systems (high-pressure phenomena)
 - Detailed thermodynamics, transport, and chemistry
 - Multiphase flow, sprays (Lagrangian-Eulerian formulation)
 - Generalized subfilter model framework for treatment of turbulence
 - Dynamic subfilter modeling (no tuned constants)
- Numerical framework (High-quality)
 - Structured multiblock topology in generalized curvilinear coordinates with unstructured connectivity between blocks
 - Staggered finite-volume differencing (non-dissipative, discretely conservative)
 - Dual-time stepping with generalized preconditioning (all-Mach-number)
 - Detailed treatment of geometry, wall phenomena, transient BC's

Mesh discretization

- Structured curvilinear mesh
- Multi-block mesh decomposition
 - Point-to-point full face match at block interfaces
 - Arbitrary number of blocks can merge at block boundaries
- Multi-block decomposition naturally leads to distributed memory parallelism
 - Data exchange on block interfaces

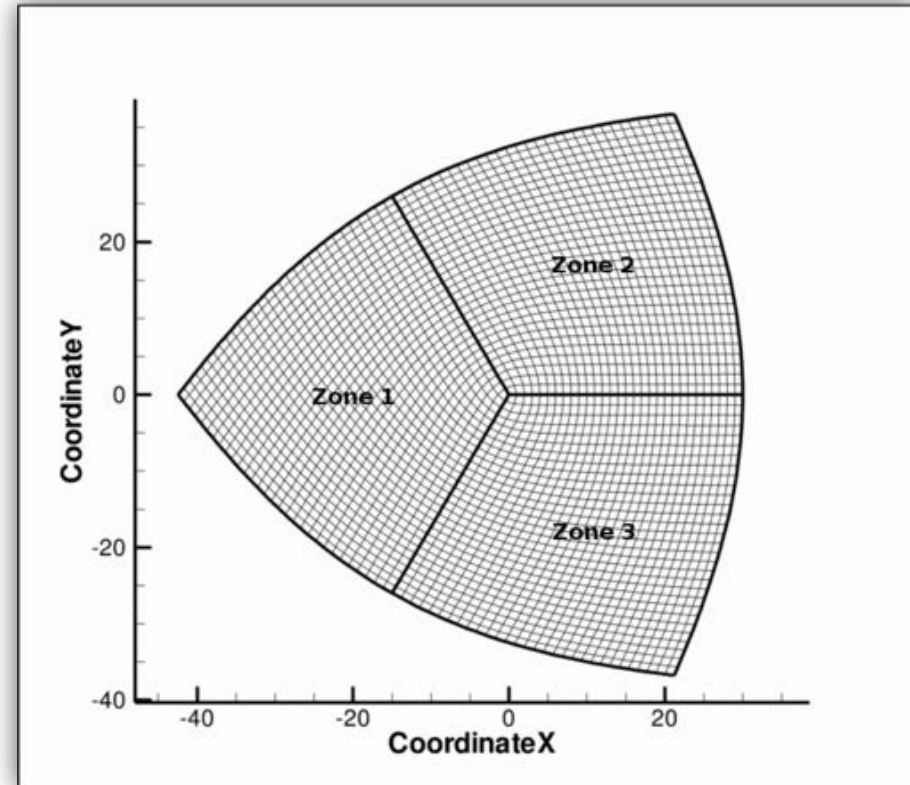


Image source: <http://www2.le.ac.uk/>

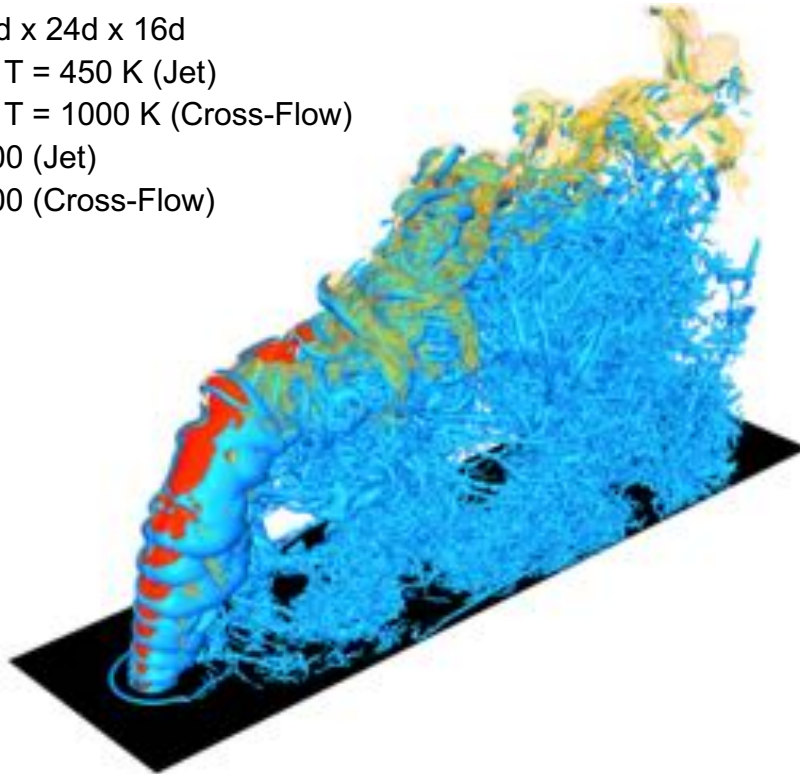
Objectives

- Setup a canonical simulation problem
 - Representative of performance characteristics of high pressure reacting flow calculations
 - Scientifically interesting
 - Convenient to generate test configurations for strong/weak scaling studies
- Analyze performance and arrive at strategy for acceleration
- Prepare code for Summit
- Establish path forward for performance improvement

LES of high pressure fuel injection

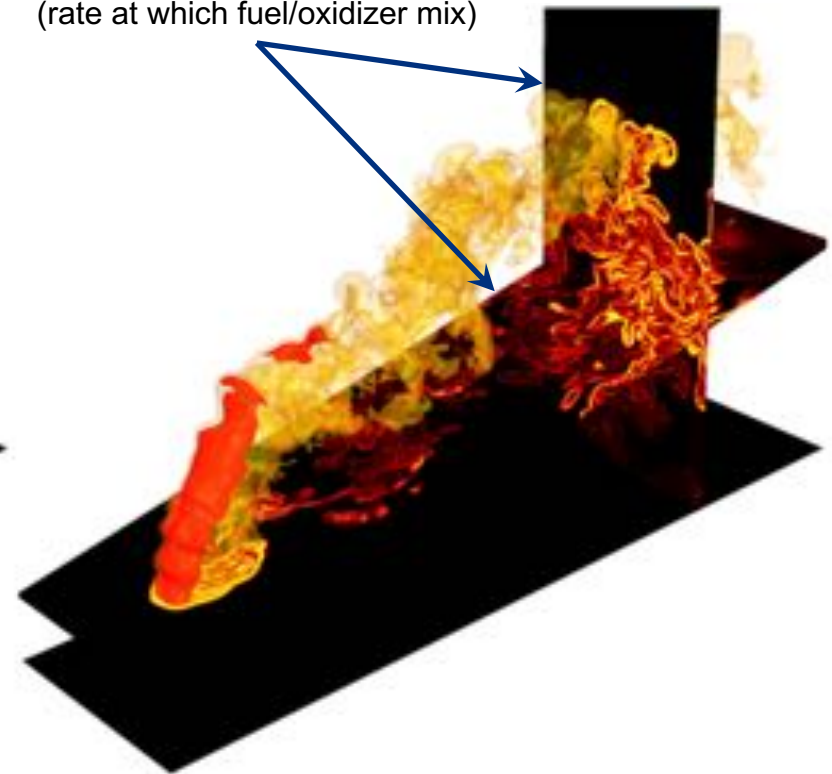
Liquid n-Decane-air jet-in-cross-flow:

- Pressure: 40 bar (supercritical)
- Jet Diameter: $d = 1$ mm
- Domain: $32d \times 24d \times 16d$
- $U = 40$ m/s, $T = 450$ K (Jet)
- $U = 80$ m/s, $T = 1000$ K (Cross-Flow)
- $Re = 125,000$ (Jet)
- $Re = 277,000$ (Cross-Flow)
- $J = 10$



- Mixture fraction isosurface (high)
- Mixture fraction isosurface (low)
- Vorticity (reveals structure)

Cut planes show
scalar-dissipation field
(rate at which fuel/oxidizer mix)



1. A. Ruiz, G. Lacaze and J. C. Oefelein. Flow topologies and turbulence scales in a jet-in-cross-flow. *Physics of Fluids*, **27**, 045101, 2015.
2. K. C. Gottiparthi, R. Sankaran, A. Ruiz, G. Lacaze and J. C. Oefelein, *AIAA-2016-1939*, 2016
3. K. C. Gottiparthi, R. Sankaran and J. C. Oefelein, *AIAA-2017-1960*, 2017

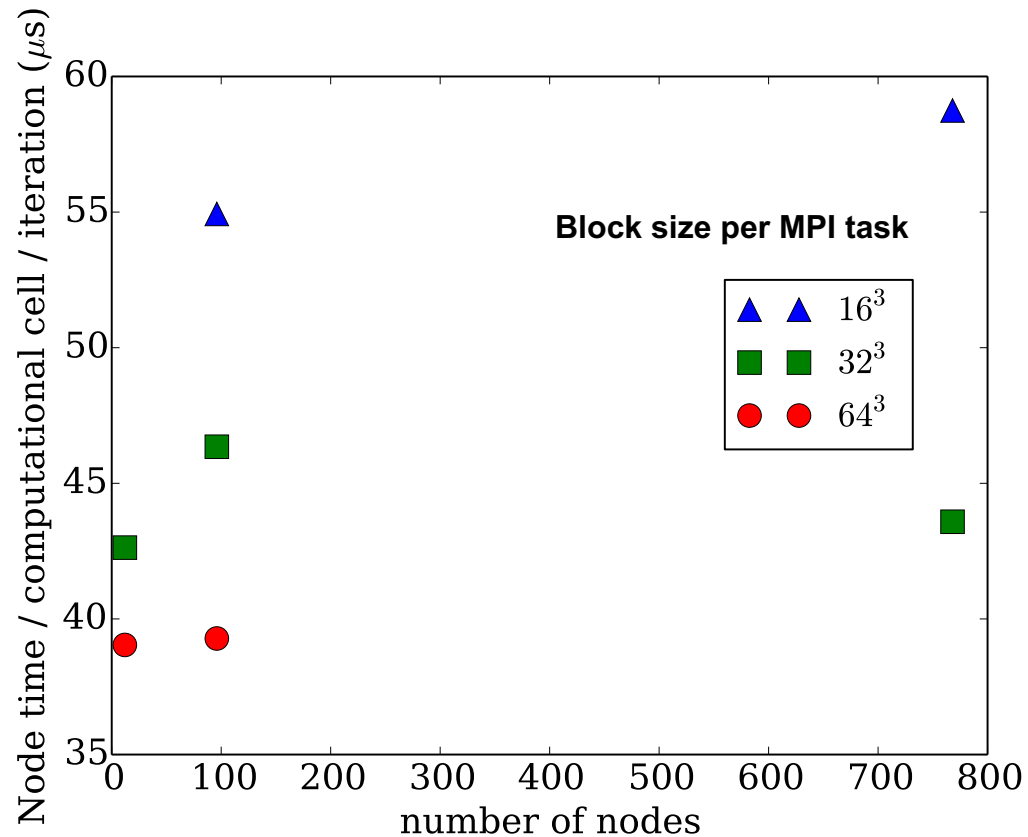
Parametric scaling studies ...

- Grids G1 to G4 for resolution tests
- Range of grid topology (T1-T5) for performance analysis

Grid	No. of cells across d_j	Grid size (in million cells)
G1	8	6.29
G2	16	50.33
G3	32	402.65
G4	64	3221.23

Topology	No. of blocks	G1	G2	G3	G4
T1	24	64^3	128^3	256^3	512^3
T2	192	32^3	64^3	128^3	256^3
T3	1536	16^3	32^3	64^3	128^3
T4	12288	8^3	16^3	32^3	64^3
T5	98304		8^3	16^3	32^3

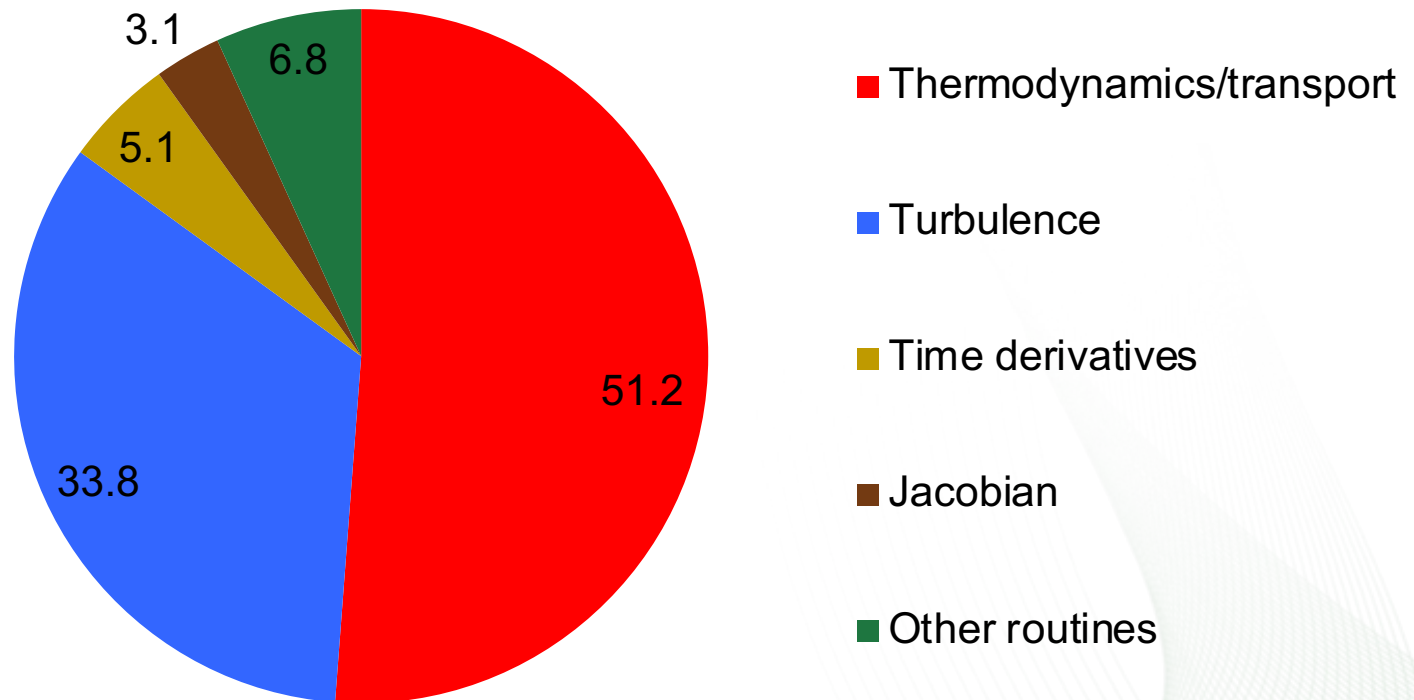
RAPTOR weak scaling on Titan



- Weak scaling of original code was measured on Titan
- Pure MPI code, with 16 MPI ranks per node

Performance profile on Titan

- Highest share of cost - EOS, thermodynamics and transport property evaluation
- Turbulence closure model follows
- Test problem uses a simple chemistry. Complex chemistry could make this another major kernel



Summit Readiness Strategy

- Rewrite kernels in C++
 - Use Kokkos performance portable programming model
 - Provide interfaces for coupling with the Fortran solver
- Hybridize the Fortran solver
 - Refactor from pure MPI to hybrid MPI+OpenMP
 - Port to GPUs using OpenMP 4.5

Vectorizing a serial fortran routine

```
subroutine convertYtoX(ys, xs, ws, ws_bar)
real, dimension(ns) :: ys, xs
real, dimension(ns) :: ws
real :: ws_bar

do m=1, ns
  xs(m)=ys(m)/ws(m)/ws_bar;
end do
```

Vectorizing a serial fortran routine

```
subroutine convertYtoX(ys, xs, ws, ws_bar)
real, dimension(ns) :: ys, xs
real, dimension(ns) :: ws
real :: ws_bar
```

```
do m=1, ns
  xs(m)=ys(m)/ws(m)/ws_bar;
end do
```

```
subroutine convertYtoX(ys, xs, ws, ws_bar)
real, dimension(nc, ns) :: ys, xs
real, dimension(ns) :: ws
real, dimension(nc) :: ws_bar
```

```
do i=1, nc
  do m=1, ns
    xs(i,m)=ys(i,m)/ws(m)/ws_bar(i);
  end do
end do
```

Rewritten as a Kokkos kernel

```
struct convertYtoX {
    VectorFieldType ys;
    VectorFieldType xs;
    ScalarFieldType ws_bar;

    // constructor
    convertYtoX(
        VectorFieldType ys_, VectorFieldType xs_,
        ScalarFieldType ws_bar_)
        : ys(ys_), xs(xs_), ws_bar(ws_bar_) {};

    KOKKOS_INLINE_FUNCTION
    void operator()(const size_type i) const {
        for(int m=0; m<ns; m++){
            xs(i,m)=ys(i,m)/ws(m)/ws_bar(i);
        }
    }
};

Kokkos::parallel_for(nCells, convertYtoX(ys, xs, ws_bar));
```

Rewritten as a Kokkos kernel

```
for(int m=0; m<ns; m++){  
    xs(i,m)=ys(i,m)/ws(m)/ws_bar(i);  
}
```

- Fortran-like array access operator
- However, the data layout in memory is specified separately
 - LayoutLeft (GPU) or LayoutRight (CPU)

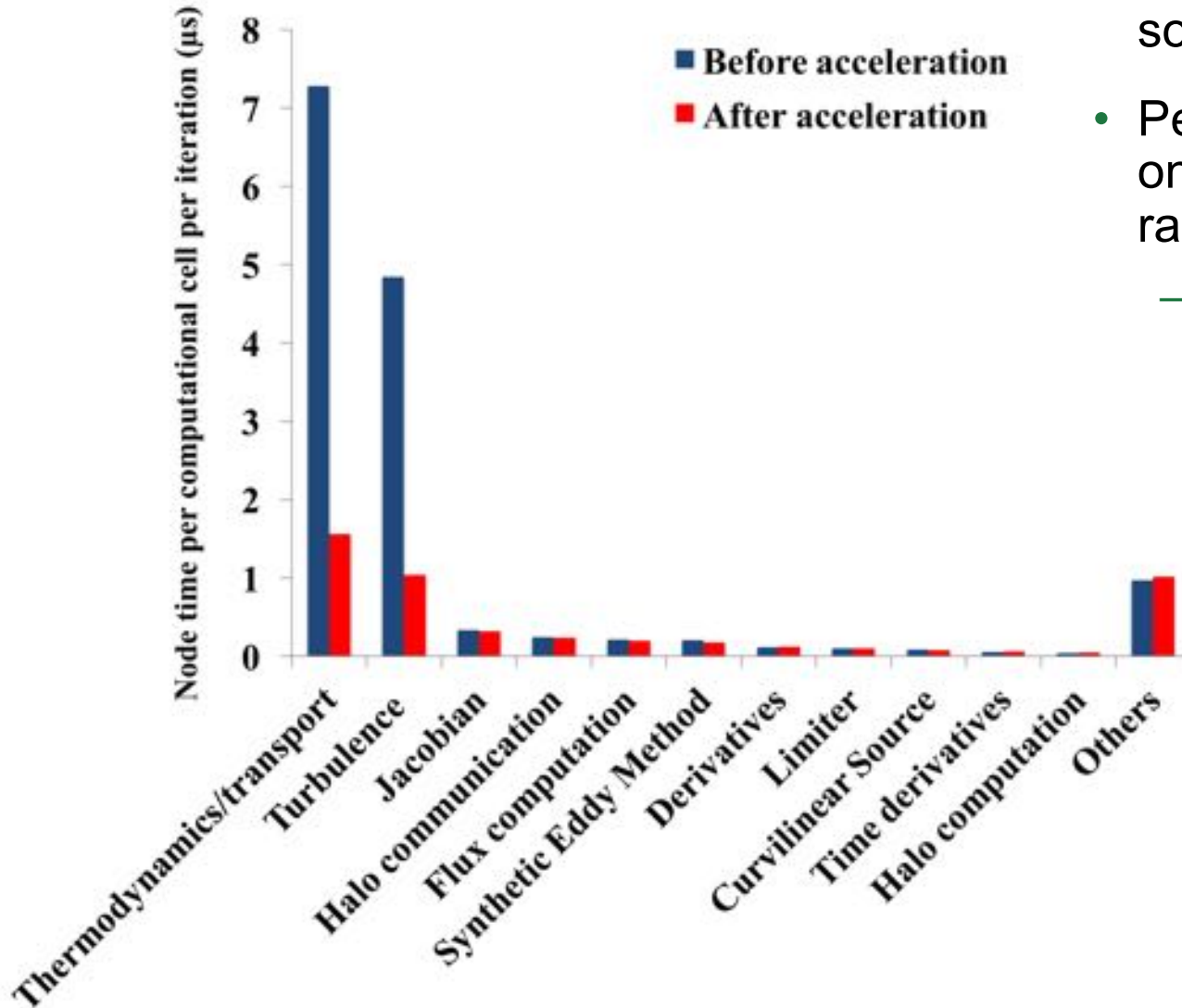
```
Kokkos::parallel_for(nCells, convertYtoX(ys, xs, ws_bar));
```

- Code generated for target execution space without device specific programming model

Special Sauce

- Data interfaces to copy data to and from the fortran arrays of the main solver
 - Transposes and deep_copy only when needed
- Custom layout and indexing operators
 - Kokkos does not support non-zero start index
 - Custom indexing needed to facilitate rewrite of fortran code and to keep the code readable
- Consider a multidimensional array
`Array(NX, NY, NZ, NV)`
 - LayoutLeft vs. LayoutRight matters for NV
 - However NX/NY/NZ are equivalent. There are benefits in keeping the layout of grid index constant

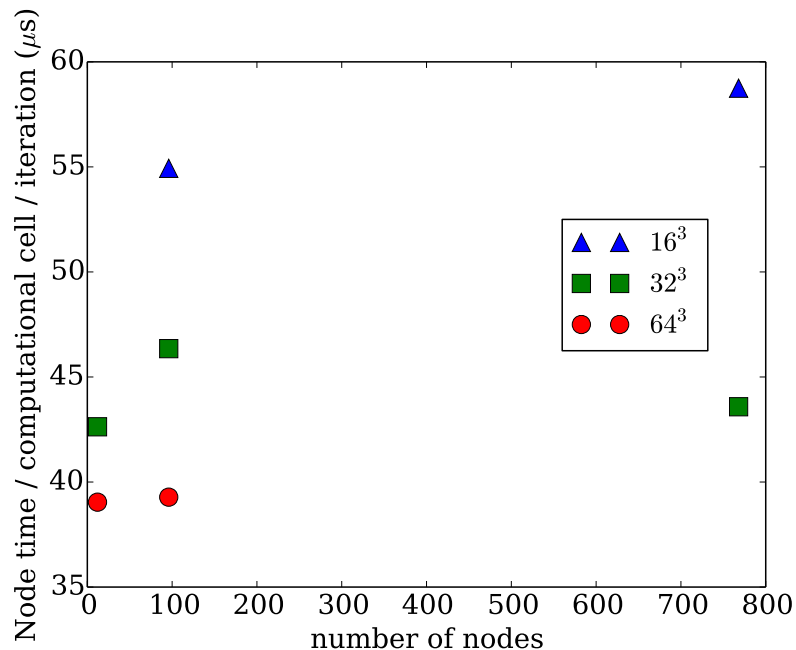
Performance profile after acceleration



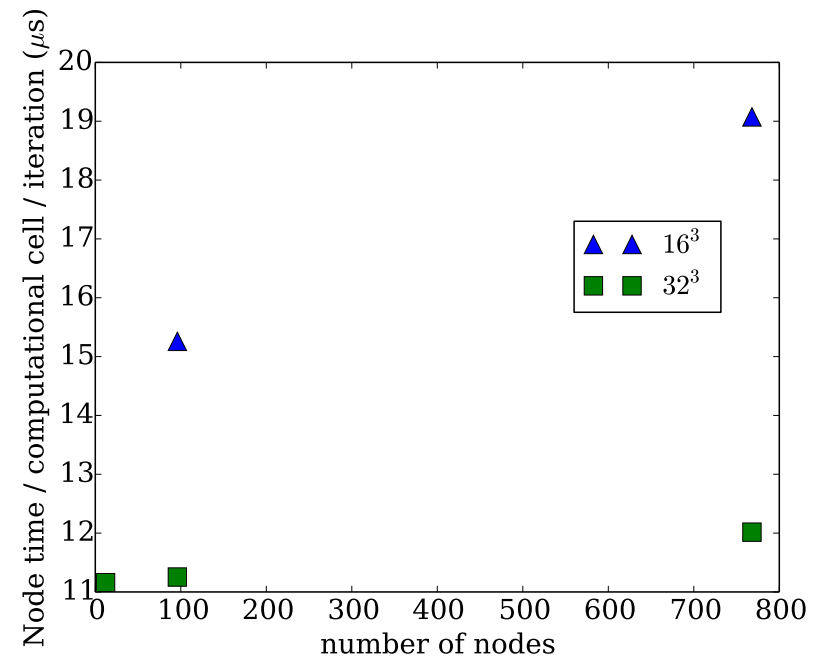
- Kokkos kernels are coupled to the Fortran solver
- Performance profile on Titan with 16 MPI ranks per node
 - K20x shared through MPS

Scalability of the Accelerated Code

- Performance of RAPTOR with the Kokkos kernels tested on Titan with 16 MPI per node
- Post-acceleration code is ~3.8X faster on 12 to 800 nodes of Titan
- 64^3 could not fit in GPU memory



Before



After

Optimization of the Kokkos Kernels

- Turbulence closure kernel was profiled using visual profiler
 - This kernel involves a filter operator applied on a 3D 27 point stencil on multiple field variables
- Significant time was spent in allocation and deallocation of memory for temporary arrays
 - A single Kokkos view allocation and creating Kokkos subviews improves performance by 2X.
- High register usage was degrading performance
 - Limiting maxregcount through compiler flags yields 12% improvement
- Addressed non-coalesced memory access
- Using shared memory through Kokkos thread teams

RAPTOR - flow solver

- Beyond the two big kernels, remaining computational cost is spread across several functions and loops
- An explicit/dual-time integrator advances the state
- Computing the RHS of the governing equations involves
 - Computing face values of cell center quantities and vice versa
 - Computing fluxes on the faces
 - Determining flux balance in a cell
 - Derivative operators
- The RHS function is coded in modern Fortran
 - Long development history with code verification
- Difficult to rewrite and verify

Hybridization of RAPTOR

- RAPTOR was hybridized from a pure MPI code to a hybrid MPI+OpenMP model
 - MPI between nodes, OpenMP on the node and vectorizable loops
- As a first step, we added OpenMP for a significant portion of RHS targeted at CPU threading
 - Scoping of private and shared variables
 - Loop analysis for potential data races
 - Created high level OMP regions with multiple parallel loops within
 - Identification of wait and nowait based on data dependencies
 - Rewriting the loops and functions to remove false sharing
- The hybridization of RAPTOR went better than expected
 - Very few instances of data races
 - Most of the RHS evaluation was safe from data races

Data Offloading with OpenMP 4.5

- OpenMP 4.5 target directives are used to map the host memory to device memory
- Two categories of data structures
 - Allocated and initialized once at the start and do not change for the duration of the program (such as grid metrics)

```
!$omp target enter data &
```

```
!$omp map(to: sxu, syv, szw)
```

- Allocated at the start and need to be updated on the host/device periodically

```
!$omp target enter data &
```

```
!$omp map(alloc: qh, qv, dq)
```

```
.....
```

```
!$omp target update to (qh)
```

```
.....
```

```
!$omp target update from (dq)
```

Offloading loops to device (1)

```
!$omp target
!$omp map      (      UF, VF, WF, dq) &
!$omp depend (in:  UF, VF, WF      ) &
!$omp depend (inout:      dq) &
!$omp nowait
!$omp teams distribute parallel &
!$omp do collapse(4) default(none) &
!$omp shared (      UF, VF, WF, dq) &
!$omp shared (nx, ny, nz, ne)
!$omp teams distribute parallel do collapse(4)
do l=1,ne
do k=1,nz-1
do j=1,ny-1
do i=1,nx-1
```

Arrays map'ed to target device
Already allocated at start using
`!$omp map (alloc: ...)`

```
    dq(i,j,k,l) = dq(i,j,k,l) - ( UF(i+1,j,k,l) - UF(i,j,k,l) &
                                +  VF(i,j+1,k,l) - VF(i,j,k,l) &
                                +  WF(i,j,k+1,l) - WF(i,j,k,l) )
```

```
end do; end do; end do; end do
!$omp end target
```


Offloading loops to device (2)

```
!$omp target
!$omp map      (      UF, VF, WF, dq) &
!$omp depend (in:  UF, VF, WF      ) &
!$omp depend (inout:      dq) &
!$omp nowait
!$omp teams distribute parallel &
!$omp do collapse(4) default(none) &
!$omp shared (      UF, VF, WF, dq) &
!$omp shared (nx, ny, nz, ne)
!$omp teams distribute parallel do collapse(4)
do l=1,ne
do k=1,nz-1
do j=1,ny-1
do i=1,nx-1

    dq(i,j,k,l) = dq(i,j,k,l) - ( UF(i+1,j,k,l) - UF(i,j,k,l) &
                                +   VF(i,j+1,k,l) - VF(i,j,k,l) &
                                +   WF(i,j,k+1,l) - WF(i,j,k,l) )

end do; end do; end do; end do
!$omp end target
```

Dependencies on data are marked.
Data updates to/from target are
similarly marked

!\$omp target update to(dq)&
!\$omp depend (out:dq)

Offloading loops to device (3)

```
!$omp target
!$omp map      (      UF, VF, WF, dq) &
!$omp depend (in:   UF, VF, WF      ) &
!$omp depend (inout:      dq) &
!$omp nowait
!$omp teams distribute parallel &
!$omp do collapse(4) default(none) &
!$omp shared (      UF, VF, WF, dq) &
!$omp shared (nx, ny, nz, ne)
!$omp teams distribute parallel do collapse(4)
do l=1,ne
do k=1,nz-1
do j=1,ny-1
do i=1,nx-1
```

Most kernels are marked “nowait”
Dependencies are marked
explicitly

```
    dq(i,j,k,l) = dq(i,j,k,l) - ( UF(i+1,j,k,l) - UF(i,j,k,l) &
                                +  VF(i,j+1,k,l) - VF(i,j,k,l) &
                                +  WF(i,j,k+1,l) - WF(i,j,k,l) )
```

```
end do; end do; end do; end do
!$omp end target
```

Offloading loops to device (4)

```
!$omp target
!$omp map      (      UF, VF, WF, dq) &
!$omp depend (in:  UF, VF, WF      ) &
!$omp depend (inout:      dq) &
!$omp nowait
!$omp teams distribute parallel &
!$omp do collapse(4) default(none) &
!$omp shared (      UF, VF, WF, dq) &
!$omp shared (nx, ny, nz, ne)
!$omp teams distribute parallel do collapse(4)
do l=1,ne
do k=1,nz-1
do j=1,ny-1
do i=1,nx-1
```


Loops are collapsed and parallelized
Raptor loops are fine grained
parallel with no data races or
dependencies across iterations

```
    dq(i,j,k,l) = dq(i,j,k,l) - ( UF(i+1,j,k,l) - UF(i,j,k,l) &
                                +  VF(i,j+1,k,l) - VF(i,j,k,l) &
                                +  WF(i,j,k+1,l) - WF(i,j,k,l) )
```

```
end do; end do; end do; end do
!$omp end target
```

Offloading loops to device (5)

```
!$omp target
!$omp map      (      UF, VF, WF, dq) &
!$omp depend (in:   UF, VF, WF      ) &
!$omp depend (inout:      dq) &
!$omp nowait
!$omp teams distribute parallel &
!$omp do collapse(4) default(none) &
!$omp shared (      UF, VF, WF, dq) &
!$omp shared (nx, ny, nz, ne)
do l=1,ne
do k=1,nz-1
do j=1,ny-1
do i=1,nx-1
```



Explicit data sharing clauses.
Default set to none.
Loop indices are auto private.

```
    dq(i,j,k,l) = dq(i,j,k,l) - ( UF(i+1,j,k,l) - UF(i,j,k,l) &
                                +  VF(i,j+1,k,l) - VF(i,j,k,l) &
                                +  WF(i,j,k+1,l) - WF(i,j,k,l) )
```

```
end do; end do; end do; end do
!$omp end target
```

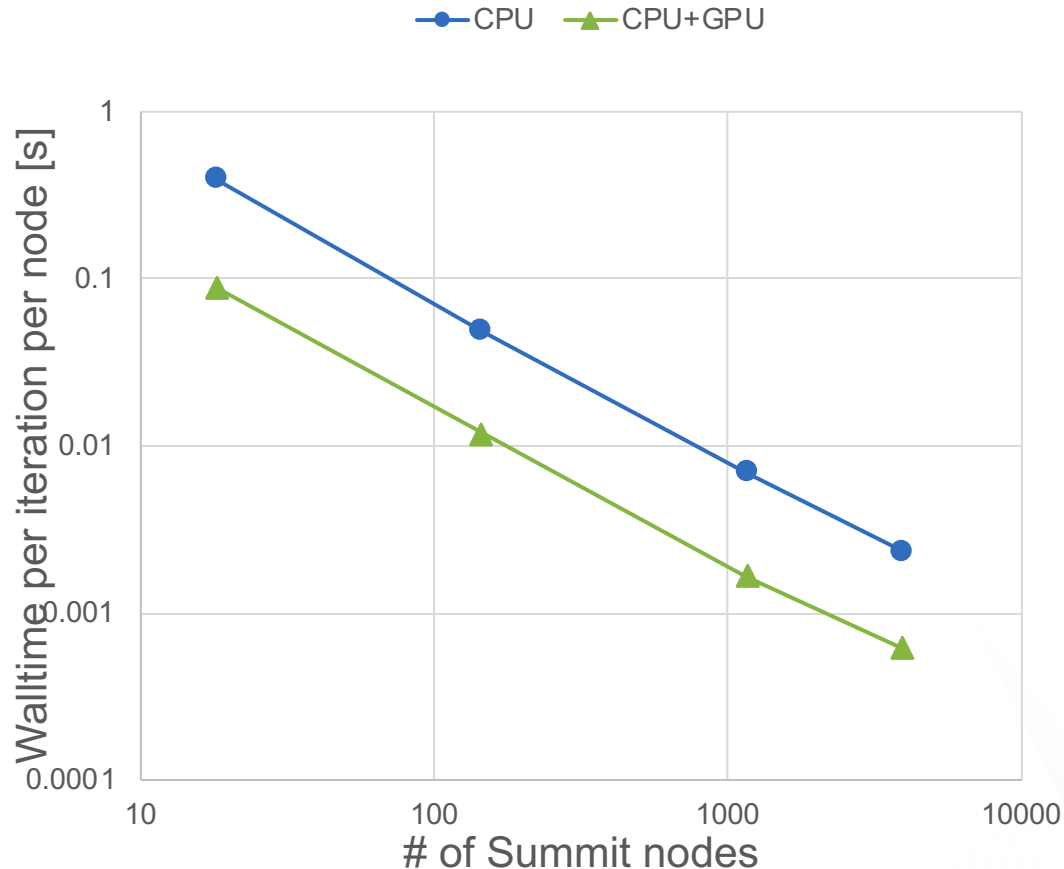
Data update to/from host

- MPI exchange of halo requires special handling of the halos for packing/unpacking buffers
- Halo packing code better suited on host

```
!$omp target update if(lacceldev) &  
!$omp to      (      dq) &  
!$omp depend (out: dq) &  
!$omp nowait  
        call flxQ_C  
        call flxQ(1); call srcQ_C  
!$omp target update if(lacceldev) &  
!$omp from      (      dq) &  
!$omp depend (out: dq)
```

RAPTOR Performance on Summit

RAPTOR weak scaling performance for a n-decane air jet in cross flow (JICF).



- GPU speedup = 4.3X on 1152 nodes
- GPU speedup = 3.8x on 3888 nodes

Resource-aware Task Placement

- Heterogeneous systems are a network of connected resources
- Tasks need to be assigned to the resources
- Tasks talk to each other - application topology
- System resources are interconnected – machine topology
- What is the optimal placement of tasks on the machine resources to maximize performance?

Formulation (QAP)

- Let a parallel application have M tasks
 - matrix F is the application topology matrix ($M \times M$)
 - $f_{i,j}$ is the volume of communication from task i to task j
 - F is a sparse matrix with bandwidth “ b ”
 - Some algorithms require all-to-all communication
 - More predominantly, communication occurs between subsets
- Network characteristics given by matrix D
 - $d_{i,j}$ is the communication “distance” between resource i, j
 - D is symmetric and dense
- A placement function $\rho(i)$ that gives the node on which task i is placed
 - the communication overhead or cost is $C = \sum_{i,j} f_{ij} d_{\rho(i)\rho(j)}$

GAMPI – Massively parallel genetic algorithm for the task placement problem

- Identifies the resources allocated by the batch job scheduler
- Spends the first few minutes of the job to determine suitable task placement
 - Solves the quadratic assignment problem using the machine network topology and task communication topology
- Open source (LGPL) software library
 - <https://github.com/ramanan/gampi>
- WIP: Rank reordering for RAPTOR's communication topology with jsrun on Summit

Sankaran, R., Angel, J., & Brown, W. M. (2015). Concurrency and Computation: Practice and Experience, <http://dx.doi.org/10.1002/cpe.3457>

Summary

- New C++ implementation of Real gas thermophysics kernels using Kokkos
- New C++ implementation of dynamic subfilter turbulence model using Kokkos
- Hybridization of MPI code to MPI+OpenMP
- Addition of OpenMP 4.5 target directives for offloading flow solver to GPU accelerator
- Topology aware MPI rank placement to improve scalability