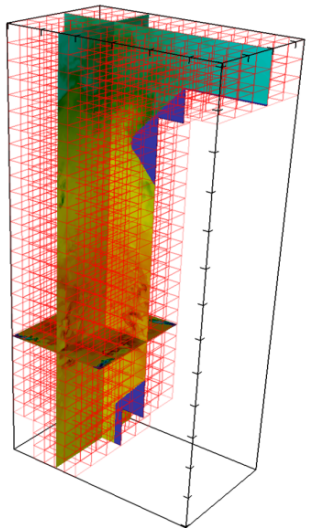
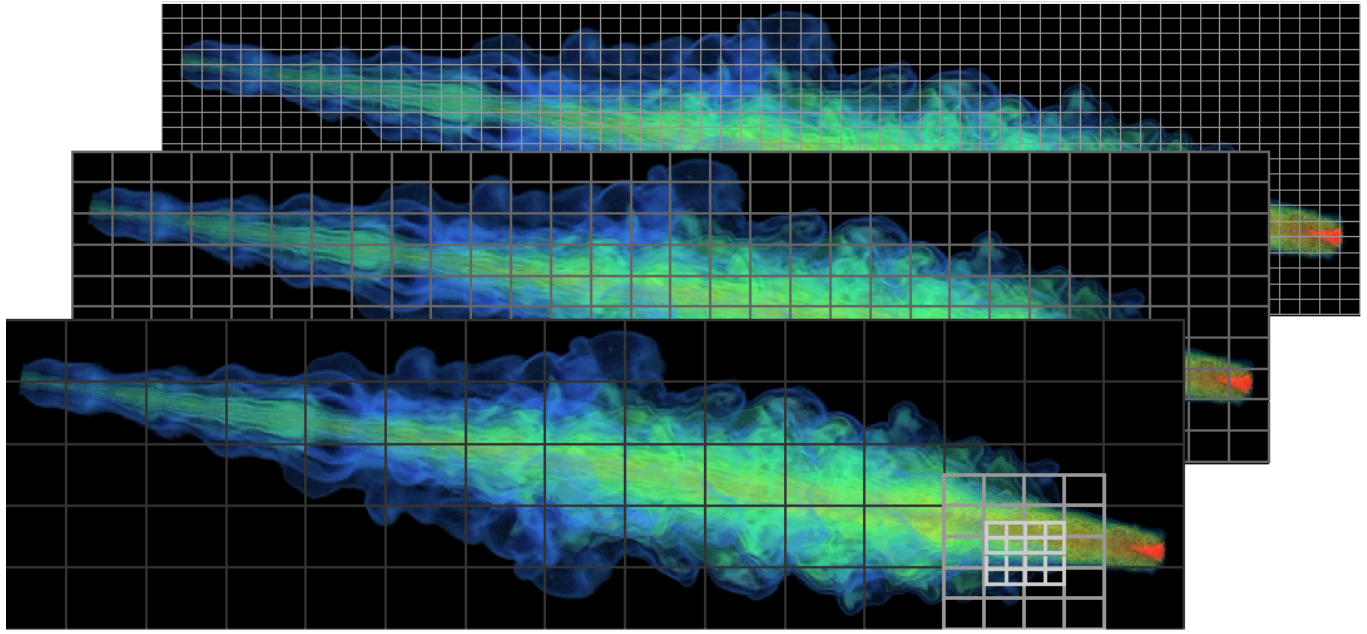
SAND2016-11774C

An Overview of the Performance Portability in the Uintah Runtime System Through the Use of Kokkos





CARBON CAPTURE
MULTIDISCIPLINARY
SIMULATION CENTER

Dan Sunderland[1], Brad Peterson[2], John Schmidt[2], Alan Humphrey[2], Jeremy Thornock[2], Martin Berzins[2]

Sandia National Laboratories[1]

Scientific Computing and Imaging Institute, University of Utah[2]

Sandia National Laboratories

SCI INSTITUTE

THE UNIVERSITY OF UTAH

Outline

I. **Uintah Overview**
II. **Kokkos Overview**
III. **Modifying Uintah to Use Kokkos**
IV. **Results**
V. **Summary and Questions**

Central Theme

Incremental refactoring of complex applications for performance portability

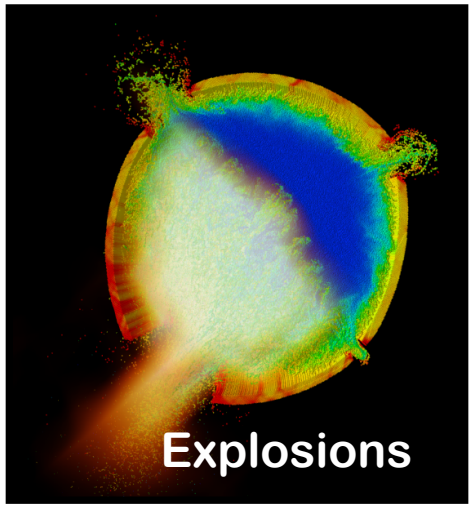
Thanks to:

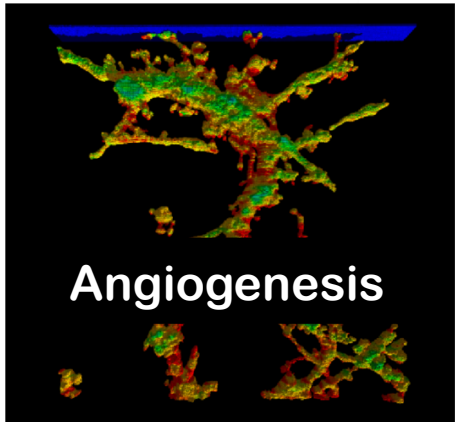
- The Uintah team, past and present
- The Kokkos team, Sandia National Laboratories
- Justin Luitjens and Steve Parker, NVIDIA
- University of Utah GPU Center of Excellence
- DOE NETL, DOE NNSA, INCITE, ALCC
- DOE NNSA PSAPP II (from March 2014)

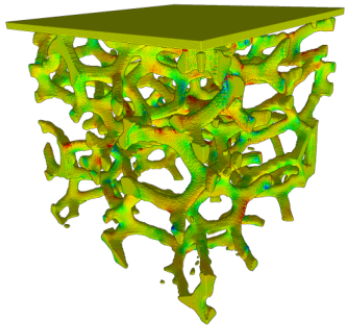
Uintah Overview

- Parallel, adaptive multi-physics framework
- Asynchronous task-based runtime system
- Fluid-structure interaction problems
- Structured grids - patch-based AMR
- Particle system and mesh-based fluid solve

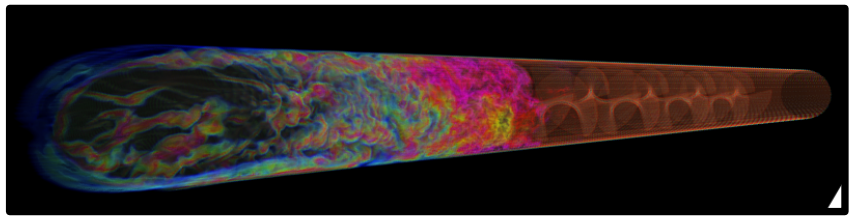
Plume Fires



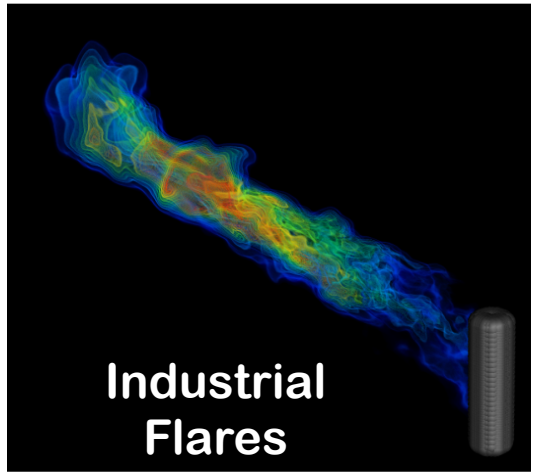


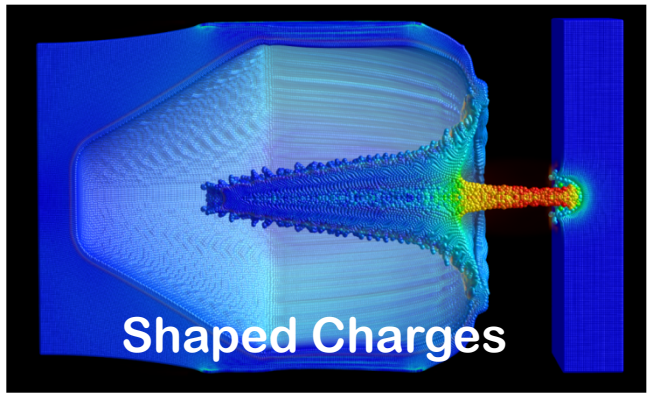


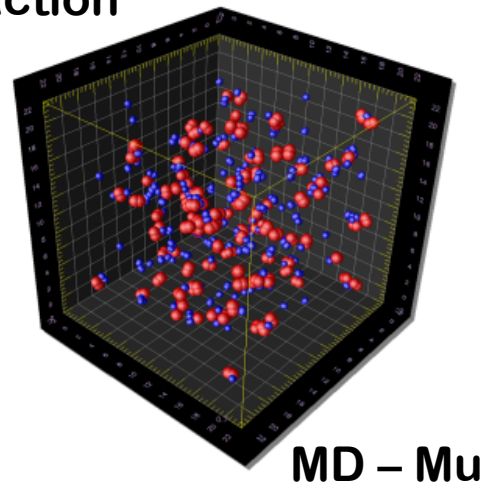
Foam Compaction

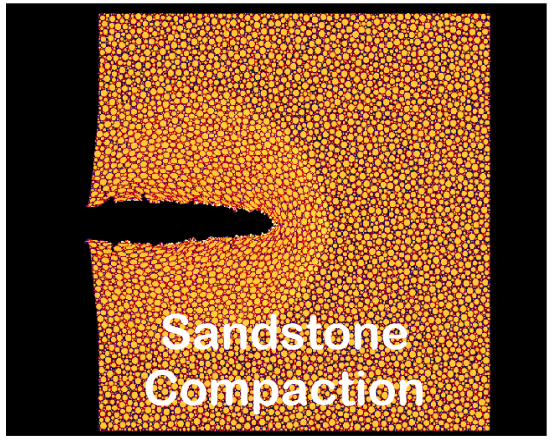


Chemical/Gas Mixing





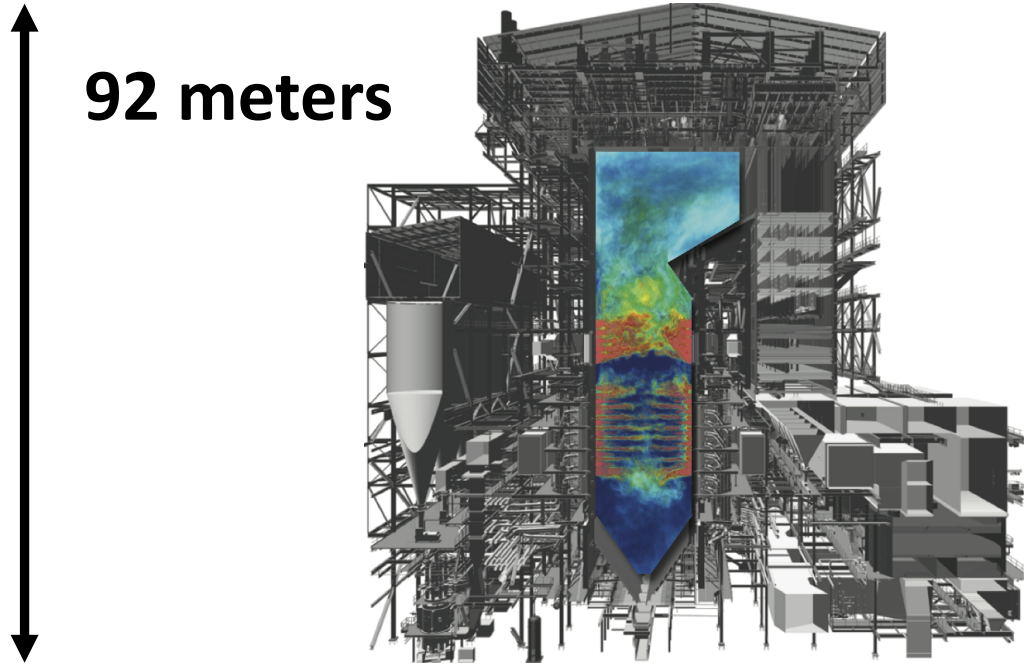




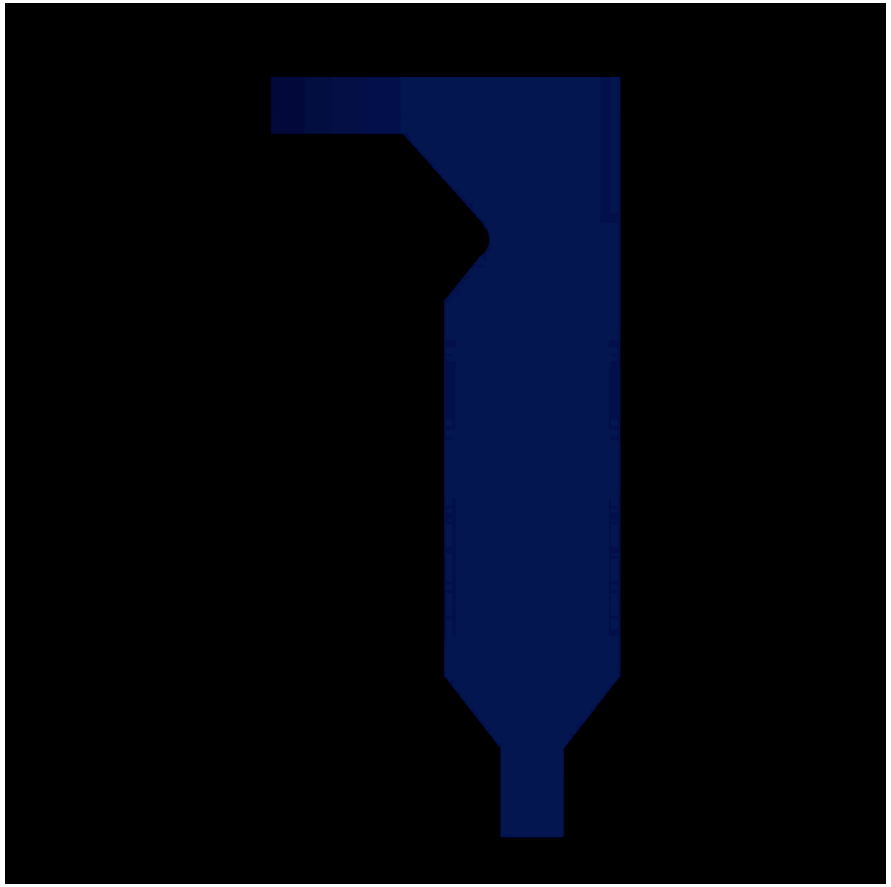
MD – Multiscale Materials Design

Exascale Target Problem

DOE NNSA PSAAP II Center – U. Utah



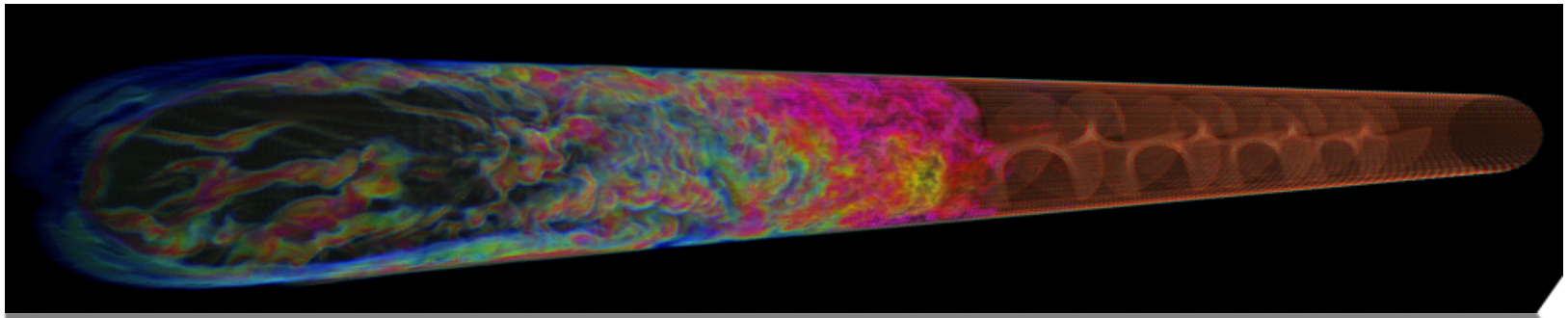
- Alstom Power 1000MWe “Twin Fireball” boiler
- Supply power for 1M people
- 1mm grid resolution = 9 x 10^{12} cells
- 1000 times larger than largest problems solved today



O_2 concentrations
Clean coal boiler simulation

ARCHES Combustion Component

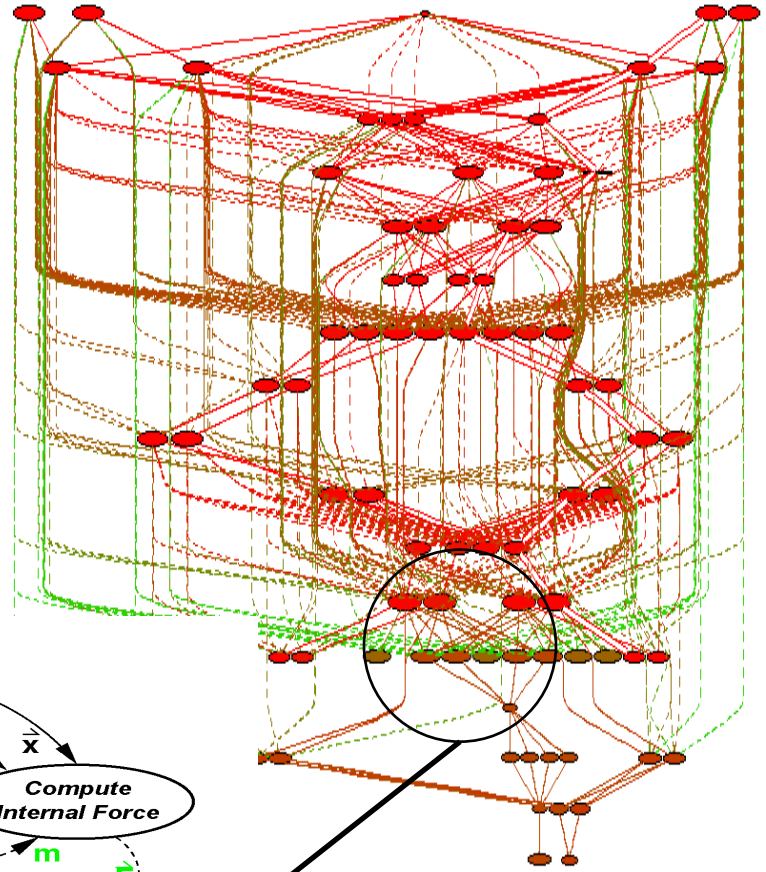
- **Designed for simulating turbulent reacting flows with participating media radiation**
  - *Heat, mass, and momentum transport*

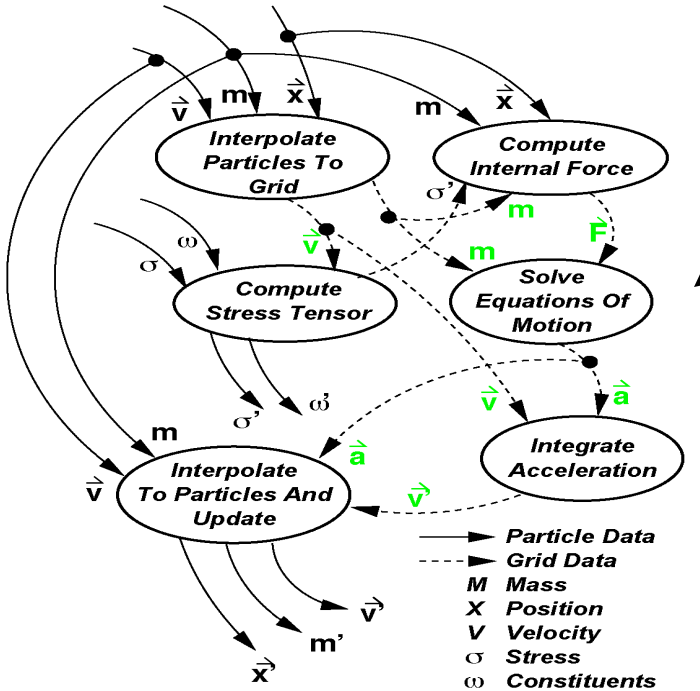


- **3D Large Eddy Simulation** (LES) code
  - Evaluate large clean coal boilers that alleviate CO_2 concerns
- **ARCHES** is massively parallel & highly scalable through its integration with Uintah

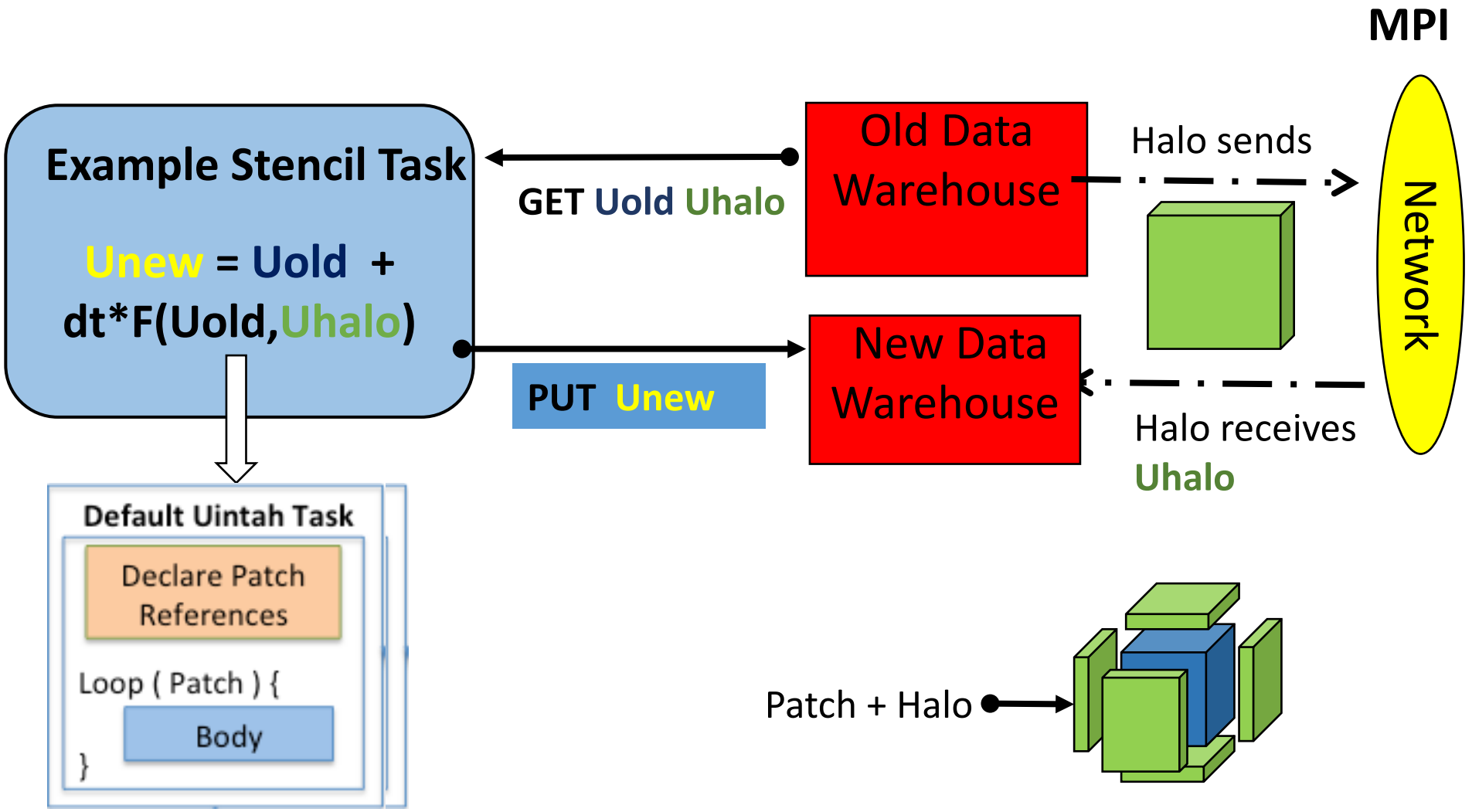
Uintah: Task-Graph Approach

- **Task Graph: Directed Acyclic Graph (DAG)**
- **Task** – basic unit of work
  - C++ method with computation (user written callback)
- Asynchronous, dynamic execution of tasks
  - **key idea, execute when ready**
- Overlap communication & computation

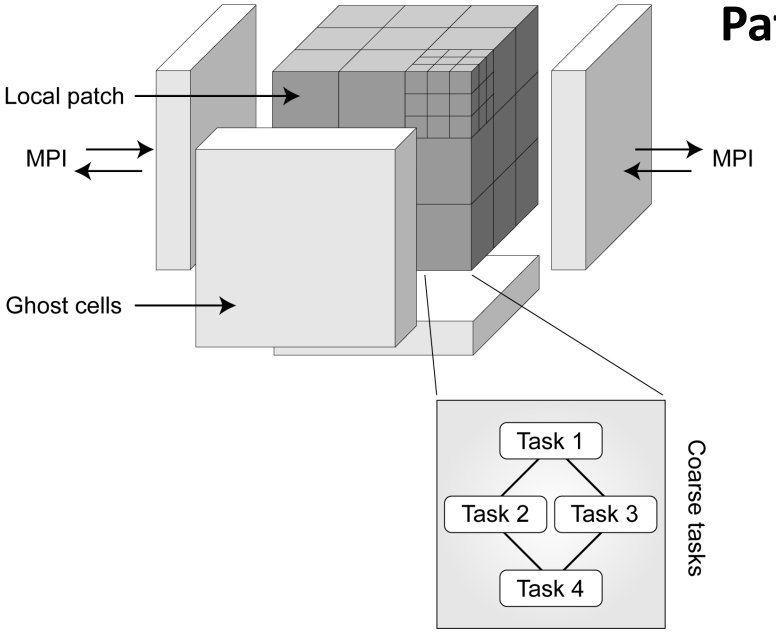




Uintah Programing Model for per Patch Stencil Task

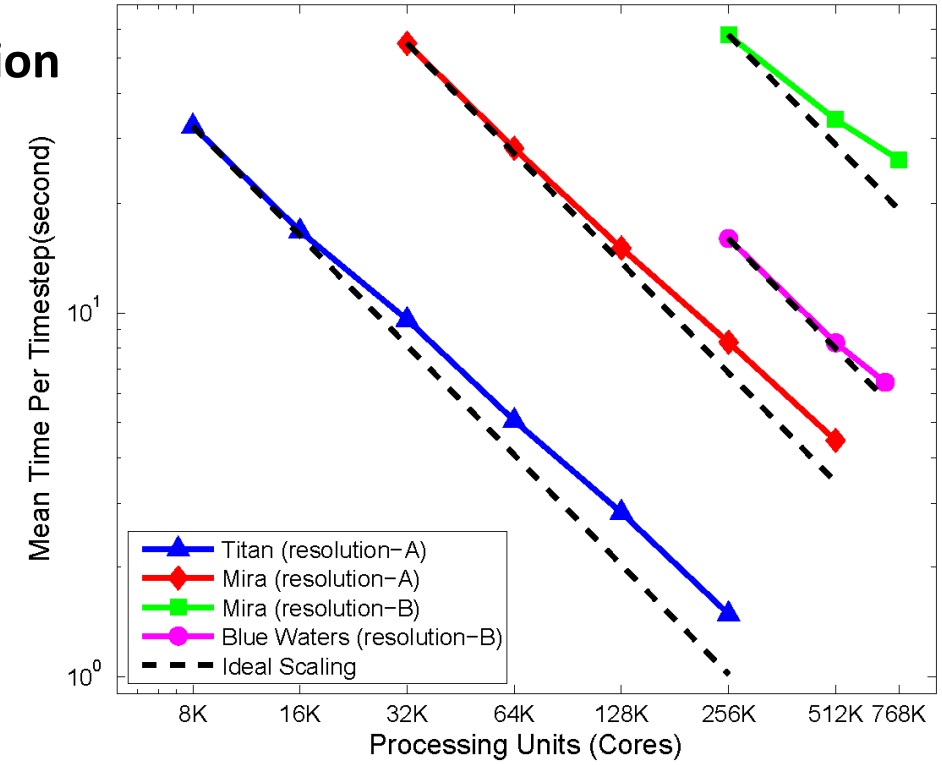


Uintah: Parallelism & Scaling



Patch-based domain decomposition

Asynchronous task-based paradigm



Strong Scaling:
Fluid-structure interaction problem

- **Clear separation of user code and runtime system**
- Task - serial code on generic "patch"
- Task specifies data dependencies & desired halo region
- Uintah manages MPI and load balancing

Existing components scale from 7k to 700K cores w/o changing any user code

Kokkos Overview

- C++11 Library for implementing portable thread-parallel codes
- Application identifies parallelizable grains of ***computation*** and ***data***
- Kokkos ***maps*** those computations onto cores and that data onto memory

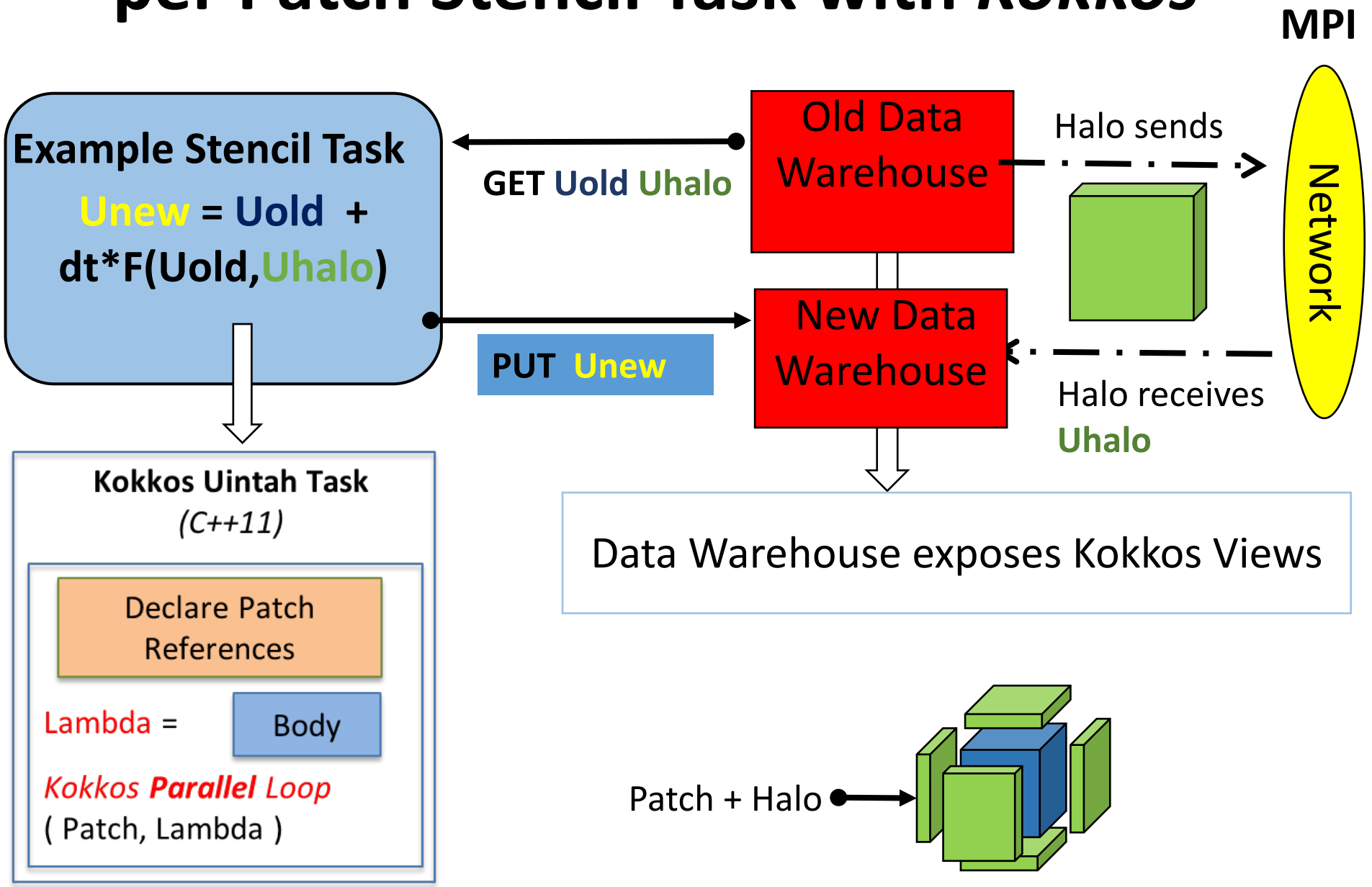
- The user is responsible for writing thread-scalable, high-performance kernels
- Carefully written kernels can obtain portable SIMD auto vectorization

source at https://github.com/kokkos/kokkos

Kokkos Abstractions

- **Parallel Pattern** user's computations (kernel)
  parallel_for, parallel_reduce, parallel_scan, task_graph, …
- **Execution Policy** how the kernel should be executed
  static scheduling, dynamic scheduling, thread-teams, …
- **Execution Space** where the kernel will execute
- **Memory Space** where the data is allocated
- **Layout** how the data is mapped to memory
  row-major, column-major, tiled, …
- **View** multiple dimensional array
  resident in the given *memory space* with the given *layout*

Uintah Programing Model for per Patch Stencil Task with *Kokkos*



Incremental refactor to Kokkos

Replace patch grid iterator loops

Refactored grid variables to expose Kokkos views

Removes many levels of indirection in existing implementation

Example:

Previous:

```
for (auto itr = patch.begin(); itr != patch.end(); ++itr) {
  A[*itr] = B[*itr] + C[*itr];
}
```

New:

```
Kokkos::parallel_for(patch.range(), KOKKOS_LAMBDA(int i, int j, int k) {
  A(i,j,k) = B(i,j,k) + C(i,j,k)
});
```

Arches Diffusion Kernel

*Convolution of 1D stencils for 3 face center variables **X, Y, Z** with 3D stencils of 2 cell center variables **D, phi***

```
for( auto itr = patch.begin(); itr != patch.end(); ++itr) {
  IntVector c = *itr;
  IntVector xp1 = c + IntVector(1,0,0);
  ... // declare xm1, yp1, ym1, zp1, zm1

  rhs[c] +=
      ax * ( X[xp1] * ( D[xp1] + D[c  ]) * (phi[xp1] – phi[c  ])
           -X[c  ] * ( D[c  ] + D[xm1]) * (phi[c  ] – phi[xm1]) )

    + ay * ( Y[yp1] * ( D[yp1] + D[c  ]) * (phi[yp1] – phi[c  ])
           -Y[c  ] * ( D[c  ] + D[ym1]) * (phi[c  ] – phi[ym1]) )

    + az * ( Z[zp1] * ( D[zp1] + D[c  ]) * (phi[zp1] – phi[c  ])
           -Z[c  ] * ( D[c  ] + D[zm1]) * (phi[c  ] – phi[zm1]) );

});
```

Naïve Kokkos Diffusion Kernel

```
//1D stencil for X, Y, Z convoluted with 3D stencil of D and phi
parallel_for( patch.range(), KOKKOS_LAMBDA(int i, int j, int k) {

  rhs(i,j,k) +=

      ax * ( X(i+1,j,k) * ( D(i+1,j,k) + D(i  ,j,k)) * (phi(i+1,j,k) - phi(i  ,j,k))
           -X(i  ,j,k) * ( D(i  ,j,k) + D(i-1,j,k)) * (phi(i  ,j,k) - phi(i-1,j,k)) )

    + ay * ( Y(i,j+1,k) * ( D(i,j+1,k) + D(i,j  ,k)) * (phi(i,j+1,k) - phi(i,j  ,k))
           -Y(i,j  ,k) * ( D(i,j  ,k) + D(i,j-1,k)) * (phi(i,j  ,k) - phi(i,j-1,k)) )

    + az * ( Z(i,j,k+1) * ( D(i,j,k+1) + D(i,j,k  )) * (phi(i,j,k+1) - phi(i,j,k  ))
           -Z(i,j,k  ) * ( D(i,j,k  ) + D(i,j,k-1)) * (phi(i,j,k  ) - phi(i,j,k-1)) );

});
```

Vectorizable Diffusion kernel

```
parallel_for( range, KOKKOS_LAMBDA(int i, int j) {

  auto r   = subview( rhs, i, j, ALL() );
  auto x_0 = subview( X, i  , j, ALL() );
  auto x_p = subview( X, i+1, j, ALL() );
  auto y_0 = subview( Y, i, j  , ALL() );
  auto y_p = subview( Y, i, j+1, ALL() );
  auto z_  = subview( Z, i, j  , ALL() );

  auto d_00 = subview( D, i  , j  , ALL() );
  auto d_m0 = subview( D, i-1, j  , ALL() );
  auto d_p0 = subview( D, i+1, j  , ALL() );
  auto d_0m = subview( D, i  , j-1, ALL() );
  auto d_0p = subview( D, i  , j+1, ALL() );

  auto p_00 = subview( phi, i  , j  , ALL() );
  auto p_m0 = subview( phi, i-1, j  , ALL() );
  auto p_p0 = subview( phi, i+1, j  , ALL() );
  auto p_0m = subview( phi, i  , j-1, ALL() );
  auto p_0p = subview( phi, i  , j+1, ALL() );

  ...
```

*Parallel for over **i** & **j***

Extract 1D subviews to aid compiler in auto vectorizing

continue on next slide

Vectorizable Diffusion kernel

... *continued from previous slide*

inner loop only depends on k

```
  for (int k = k_begin; k < k_end; ++k) {
      r(k) +=
                ax * (  x_p(k)  * ( d_p0(k)   + d_00(k))   * (p_p0(k)   - p_00(k))
                     - x_0(k)  * ( d_00(k)   + d_m0(k))   * (p_00(k)   - p_m0(k)) )

              + ay * (  y_p(k)  * ( d_0p(k)   + d_00(k))   * (p_0p(k)   - p_00(k))
                     - y_0(k)  * ( d_00(k)   + d_0m(k))   * (p_00(k)   - p_0m(k)) )

              + az * (  z_(k+1) * ( d_00(k+1) + d_00(k  )) * (p_00(k+1) - p_00(k  ))
                     - z_(k  ) * ( d_00(k  ) + d_00(k-1)) * (p_00(k  ) - p_00(k-1)) );

  }

});
```

end of parallel for

Speedups *(with respect to base Uintah)*

2 sockets X 8 cores X 2 hyperthreads, avx, INTEL Xeon E5-2660 0

	Serial		Thread-2		Thread-4	
Patch Size	Naïve	Vector	Naïve	Vector	Naïve	Vector
16^3	1.8	3.4	3.5	6.5	6.8	12.1
32^3	1.6	3.4	3.2	6.8	6.4	13.2
64^3	1.9	3.3	3.6	7.0	6.9	13.8
128^3	1.8	3.2	3.7	6.5	7.3	12.3

Speedups *(with respect to base Uintah)*

2 sockets X 8 cores X 2 hyperthreads, avx, INTEL Xeon E5-2660 0

	Thread-8		Thread-16		Thread-32	
Patch Size	Naïve	Vector	Naïve	Vector	Naïve	Vector
16^3	12.7	20.3	15.3	19.8	22.3	26.9
32^3	12.7	24.2	15.5	24.3	28.7	42.5
64^3	12.9	25.6	14.8	25.7	28.9	49.2
128^3	13.9	22.2	14.4	25.5	18.4	19.0

Speedups *(with respect to base Uintah)*

GTX TITAN X capability 5.2, Total Global Memory: 12 G,

	Cuda	
Patch Size	Naïve	Vector
16^3	3.0	3.2
32^3	11.8	11.9
64^3	38.4	37.8
128^3	105.7	103.4

Summary

- **Uintah Framework - DAG Approach**
  - Powerful abstraction for solving challenging engineering problems
  - Extended with relative ease to efficiently leverage GPUs
  - Provides convenient separation of problem structure from data and communication – application code vs. runtime
  - Shields applications developer from complexities of parallel programming involved with heterogeneous HPC systems
  - Allows scheduling algorithms to optimize for scalability and performance
  - Enabling challenging exascale candidate problems on current and emerging heterogeneous architectures

Questions?