Using MPI+Kokkos to Portably Extend Uintah to Many-Core Systems

John K. Holmen¹, Brad Peterson¹, Alan Humphrey¹, Daniel Sunderland², Martin Berzins¹
University of Utah¹
Sandia National Laboratories²
Uintah

- Open source computational framework
  - Worldwide distribution
- Application code programming model
  - App devs isolated from infrastructure
- Auto-generated abstract C++ task graph
- Adaptive execution of tasks by runtime
  - Asynchronous out-of-order execution
  - Work stealing
  - Overlapping of communication & computation

---

**Simulation Controller**
**Runtime System**
**Load Balancer**

**Scheduler**
**Task**
**Data Warehouse**

**PIDX**
**VisIT**

**Hypre Linear Solver**

**GPUs**
**CPUs**
**Xeon Phis**
Kokkos

- C++ library enabling portable, thread-scalable code optimized for CPU, GPU, and MIC architectures

- Provides abstractions to control:
  - how/where kernels are executed,
  - where data is allocated, and
  - how data is mapped to memory

- Enables performance portability
  - Developers remain responsible for writing performant code

Kokkos Calls (Hidden from Users)

- Application developers write high-level calls for loops and data structures
- High-level calls mapped to low-level Kokkos calls behind-the-scenes
- Calls compiled for target Kokkos back-ends via C++ template metaprogramming

Kokkos Calls

- `Uintah::parallel_<pattern>`
- `getKokkosView()`

Kokkos Libraries

- `Kokkos::Cuda`
- `Kokkos::OpenMP`
Exascale Target Problem

- Work motivated by participation in the DOE/NNSA’s PSAAP II initiative
- Supporting the design and evaluation of an existing ultra-supercritical clean coal boiler
  - 1000 MWe (power for ~1M people)
- Large-scale simulations using Arches, Uintah’s Large Eddy Simulation (LES) code
- Significantly larger than our largest problems solved today
  - 1 mm grid resolution target (9 x 10^{12} cells)

50-92 meters
Case Studies

- Arches LES code has ~500 loops
  - Range from 1 line of code to ~800 lines of code
  - 20 lines of code on average

- Case studies used to ease future refactoring

- Early Kokkos case studies have targeted Uintah’s most complex loops
  - Chosen to identify refactoring challenges

- A more recent Kokkos case study has targeted simpler loops
  - Chosen to understand more typical performance
Case Studies: Complex Loop

• Complex target loop has \(~350\) lines of code and models char oxidation of a coal particle:

• Loop structure has multiple interior loops and Newton iterations:

```plaintext
for all mesh patches do
    for all Gaussian quadrature nodes do
        for all cells in a mesh patch do
            loop over reactions with inner loop over reactions or species
            multiple loops over reactions
            loop over species
            for all Newton iterations do
                multiple loops over reactions
                multiple loops over reactions with inner loops over reactions
            end for
            loop over reactions
        end for
    end for
end for
```
Case Studies: Simple Loop

- Simple target loop has 3 lines of code and computes radiative properties:

```plaintext
for all mesh patches do
    for all cells in a mesh patch do
        apply a weight to a particle’s absorption coefficient
        store the weighted coefficient for flow cells
        store a zero for non-flow cells
    end for
end for
```

- Weighted properties are then used to compute global radiative heat flux
  - e.g. to understand heat flux profiles in large coal boilers
Refactoring Lessons

- Straightforward to refactor legacy loops for the Kokkos::OpenMP back-end
  - Loops must be thread-safe

- Challenging to refactor legacy loops for the Kokkos::Cuda back-end
  - Loops must be thread-safe
  - Loops may not use C/C++ features that are not supported in CUDA
  - Care must be taken when allocating memory

- Use lambdas, instead of functors, for parallel patterns (see next slides)
  - Few changes needed with lambdas
  - Lambdas handle functor class setup behind-the-scenes
Enabling Kokkos Support

// LEGACY APPROACH WITHOUT KOKKOS SUPPORT
for ( CellIterator iter = patch->getCellIterator(); !iter.done(); iter++ ) {
    IntVector c = *iter;

    double particle_absorption = abs_scat_coeff[abs_coef][c] * weightQuad[ix][c] * portable_absorption_modifier;
    abskpQuad[ix][c] = ( vol_fraction[c] > 1e-16 ) ? particle_absorption : 0.0;
    abskp[0][c] += abskpQuad[ix][c];
}

// LAMBDA-BASED APPROACH WITH KOKKOS SUPPORT
Uintah::BlockRange range( patch->getCellLowIndex(), patch->getCellHighIndex() );
Uintah::parallel_for( executionObject, range, KOKKOS_LAMBDA(int i, int j, int k) {
    double particle_absorption = abs_scat_coeff[abs_coef](i,j,k) * weightQuad[ix](i,j,k) * portable_absorption_modifier;
    abskpQuad[ix](i,j,k) = ( vol_fraction[i,j,k] > 1e-16 ) ? particle_absorption : 0.0;
    abskp[0][i,j,k] += abskpQuad[ix](i,j,k);
});
namespace {

struct eval_functor {

KokkosView3<double, Kokkos::HostSpace> abs_scat_coeff;
KokkosView3<const double, Kokkos::HostSpace> weightQuad;
const double portable_absorption_modifier;
KokkosView3<double, Kokkos::HostSpace> abskpQuad;
KokkosView3<const double, Kokkos::HostSpace> vol_fraction;
KokkosView3<double, Kokkos::HostSpace> abskp;

    eval_functor( KokkosView3<double, Kokkos::HostSpace> & m_abs_scat_coeff,
                 KokkosView3<const double, Kokkos::HostSpace> & m_weightQuad,
                 const double & m_portable_absorption_modifier,
                 KokkosView3<double, Kokkos::HostSpace> & m_abskpQuad,
                 KokkosView3<const double, Kokkos::HostSpace> & m_vol_fraction,
                 KokkosView3<double, Kokkos::HostSpace> & m_abskp )
    : abs_scat_coeff ( m_abs_scat_coeff ), weightQuad ( m_weightQuad ),
      portable_absorption_modifier ( m_portable_absorption_modifier ),
      abskpQuad ( m_abskpQuad ),
      vol_fraction ( m_vol_fraction ),
      abskp ( m_abskp )
    {};

    void operator() ( int i, int j, int k ) const {
        double particle_absorption = abs_scat_coeff(i,j,k) * weightQuad(i,j,k) *
                                    portable_absorption_modifier;

        abskpQuad(i,j,k) = ( vol_fraction(i,j,k) > 1e-16 ) ? particle_absorption : 0.0;
        abskp(i,j,k) += abskpQuad(i,j,k);
    }
};

Uintah::BlockRange range( patch->getCellLowIndex(), patch->getCellHighIndex() );
eval_functor functor( abs_scat_coeff[abs_coef], weightQuad[ix], portable_absorption_modifier,
                        abskpQuad[ix], vol_fraction, abskp[0] );
Uintah::parallel_for( executionObject, range, functor );
Results

- Three nodes used to understand loop-level performance with Kokkos:
  
  - **CPU**: Two Intel Xeon E5-2680 Sandy Bridge processors
    - 2.7 GHz; 16 cores; 2 threads per core
    - 64 GB of RAM
  
  - **GPU**: One Maxwell-based NVIDIA GeForce GTX Titan X GPU
    - 12 GB of RAM
  
  - **KNL**: One Intel Xeon Phi 7210 Knights Landing processor
    - 1.3 GHz; 64 cores; 4 threads per core
    - 96 GB of RAM
Results: Refactoring for Portability

- Results on the next slide show performance achieved when refactoring loops for Kokkos::OpenMP and Kokkos::Cuda
  - e.g. Changes to build successfully and execute correctly

- Speedups compare performance between:
  - The original serial loop run across 1 thread on a Xeon core, and
  - The refactored loop run across 1 thread on a Xeon core

- In both cases, the loop was executed without Kokkos
Results: Refactoring for Portability

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Loop</th>
<th>16$^3$ Cells</th>
<th>32$^3$ Cells</th>
<th>64$^3$ Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Complex</td>
<td>2.66x</td>
<td>2.57x</td>
<td>2.55x</td>
</tr>
<tr>
<td></td>
<td>Simple</td>
<td>4.42x</td>
<td>4.93x</td>
<td>4.70x</td>
</tr>
</tbody>
</table>

- Changes needed for portability:
  - Replace legacy loop statement with Uintah::parallel_for
  - Replace legacy data structures with Uintah::KokkosView3
  - Remove C/C++ features not supported in CUDA (e.g. std::vector)
Results: Adding Loop-Level Parallelism

- Results on the next slide show performance achieved when using multiple cores/threads to execute a loop.

- Speedups compare performance between:
  - The original serial loop run across 1 thread on a Xeon core, and
  - The refactored loop run across the full node.

- The original serial loop was executed without Kokkos.

- The refactored loop was executed with Kokkos::OpenMP or Kokkos::Cuda.
## Results: Adding Loop-Level Parallelism

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Loop</th>
<th>16³ Cells</th>
<th>32³ Cells</th>
<th>64³ Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Complex</td>
<td>43.15x</td>
<td>43.83x</td>
<td>42.48x</td>
</tr>
<tr>
<td></td>
<td>Simple</td>
<td>4.14x</td>
<td>10.00x</td>
<td>12.83x</td>
</tr>
<tr>
<td>GPU</td>
<td>Complex</td>
<td>129.53x</td>
<td>170.84x</td>
<td>162.48x</td>
</tr>
<tr>
<td></td>
<td>Simple</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>KNL</td>
<td>Complex</td>
<td>30.11x</td>
<td>42.84x</td>
<td>48.04x</td>
</tr>
<tr>
<td></td>
<td>Simple</td>
<td>2.27x</td>
<td>4.03x</td>
<td>6.04x</td>
</tr>
</tbody>
</table>

- Small loops scale poorly across a node
- GPU performance achieved at the expense of being limited to ~8x smaller problems
Results: Using More Threads per Core

- Results on the next slide show performance achieved when using multiple threads per core to execute a loop.

- Speedups compare performance between:
  - The **refactored loop** run across 1 thread on a Xeon/Xeon Phi core, and
  - The **refactored loop** run across 2-4 threads on a Xeon/Xeon Phi core

- In both cases, the loop was executed **with Kokkos::OpenMP**
Results: Using More Threads per Core

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Loop</th>
<th>16^3 Cells</th>
<th>32^3 Cells</th>
<th>64^3 Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Complex</td>
<td>0.94x</td>
<td>1.11x</td>
<td>1.10x</td>
</tr>
<tr>
<td>2 Threads</td>
<td>Simple</td>
<td>0.46x</td>
<td>1.01x</td>
<td>1.19x</td>
</tr>
<tr>
<td>KNL</td>
<td>Complex</td>
<td>1.16x</td>
<td>1.35x</td>
<td>1.35x</td>
</tr>
<tr>
<td>2 Threads</td>
<td>Simple</td>
<td>0.66x</td>
<td>1.13x</td>
<td>1.32x</td>
</tr>
<tr>
<td>KNL</td>
<td>Complex</td>
<td>1.17x</td>
<td>1.44x</td>
<td>1.47x</td>
</tr>
<tr>
<td>4 Threads</td>
<td>Simple</td>
<td>0.48x</td>
<td>1.16x</td>
<td>1.45x</td>
</tr>
</tbody>
</table>

- Possible to improve performance by using multiple threads per core
- Care must be taken to avoid slowdowns
Uintah’s Runtime System

- MPI+Kokkos using:
  - 1 MPI Process per Node
  - n Running Tasks per Process
  - n Threads per Task

- Kokkos partitioning used for per-process task parallelism
Results: More Efficiently Using a Node

• Results on the next slide show performance achieved when more efficiently using a node
  • e.g. Using threads within a core to execute the same loop instead of multiple serial loops

• Speedups compare performance between:
  • Executing multiple original serial loops across the full node, and
  • Executing multiple refactored loops across the full node

• The original serial loops were executed in parallel without Kokkos

• The refactored loops were executed in parallel with Kokkos::OpenMP
Results: More Efficiently Using a Node

- Possible to improve performance by cooperatively using threads within a core

- *RMCRT* is another complex loop that has ~500 lines of code and models radiative heat transfer using reverse Monte-Carlo ray tracing
  - Also used for strong-scaling results to follow

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Loop</th>
<th>512 - 16$^3$ Cells</th>
<th>64 - 32$^3$ Cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>RMCRT *</td>
<td>1.48x</td>
<td>1.71x</td>
</tr>
<tr>
<td>KNL</td>
<td>RMCRT *</td>
<td>1.99x</td>
<td>2.63x</td>
</tr>
</tbody>
</table>
Results: Stampede 2 Strong-Scaling*

- Good strong-scaling with MPI+Kokkos on Intel Omni-Path
- Up to $1.62x$ performance improvement with optimal run configurations
- Possible to use Kokkos at scale
Summary

- MPI+Kokkos offers a flexible MPI+X solution that:
  - Eases ports to future HPC systems, and
  - Scales well across Intel Xeon Phi-based nodes

- Performance improvements:
  - Tend to be a by-product of refactoring for portability, and
  - Have been achieved with little added overhead (< 0.2% overheads)

- When there is enough per core work:
  - Loops scale well across multiple cores/sockets, and
  - Performance can be improved using multiple threads per core
Questions?

This materials is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375

Support for J.K. Holmen comes from the Intel Parallel Computing Centers Program

This research used resources donated to the University of Utah’s Intel Parallel Computing Center at the SCI Institute

Computing time provided by the NSF Extreme Science and Engineering Discovery Environment (XSEDE) program

This research used resource of the Texas Advanced Computing Center under Award Number(s) MCA08X004 - “Resilience and Scalability of the Uintah Software”

John K. Holmen
jholmen@sci.utah.edu
http://www.cs.utah.edu/~jholmen/

Uintah Download: http://www.uintah.utah.edu
Kokkos Download: https://github.com/kokkos/kokkos