An Approach for Indirectly Adopting a Performance Portability Layer in Large Legacy Codes

John K. Holmen, Brad Peterson, Martin Berzins
Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT 84112, USA

***Abstract*—Diversity among supported architectures in current and emerging high performance computing systems, including those for exascale, makes portable codebases desirable. Portability of a codebase can be improved using a performance portability layer to provide access to multiple underlying programming models through a single interface. Direct adoption of a performance portability layer, however, poses challenges for large pre-existing software frameworks that may need to preserve legacy code and/or adopt other programming models in the future. This paper describes an approach for indirect adoption that introduces a framework-specific portability layer between the application developer and the adopted performance portability layer to help improve legacy code support and long-term portability for future architectures and programming models. This intermediate layer uses loop-level, application-level, and build-level components to ease adoption of a performance portability layer in large legacy codebases. Results are shown for two challenging case studies using this approach to make portable use of OpenMP and CUDA via Kokkos in an asynchronous many-task runtime system, Uintah. These results show performance improvements up to 2.7x when refactoring for portability and 2.6x when more efficiently using a node. Good strong-scaling to 442,368 threads across 1,728 Knights Landing processors are also shown using MPI+Kokkos at scale.**

***Index Terms*—Frameworks, Parallel Architectures, Parallelism and Concurrency, Portability, Software Engineering**

I. Introduction

For large-scale simulation, the portability of a codebase is becoming more important due to the variety of architectures being introduced in current and emerging high performance computing (HPC) systems. Among current systems, the Top 10 of June 2019's Top500 list [1] includes heterogeneous IBM- and NVIDIA-based systems, Sunway-based systems, Intel Xeon-based systems, and Intel Xeon Phi-based systems. Forthcoming exascale systems continue this trend with systems such as the DOE Aurora [2] and DOE Frontier [3] to include Intel- and AMD-based GPUs, respectively. Such variety complicates programming model selection for codebases looking to maintain long-term portability across major HPC systems.

Programming model selection is simplified using a performance portability layer (PPL). Performance portability layers provide abstractions (e.g., parallel loop statements) that allow developers to use a single interface to interact with multiple underlying programming models (e.g., CUDA, OpenCL, OpenMP, etc) through PPL-specific back-ends. This approach eases adoption of multiple programming models by reducing the amount of duplicated code and the knowledge required of underlying programming models by offloading low-level implementation details to the performance portability layer.

This material 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 also comes from the Intel Parallel Computing Centers Program.

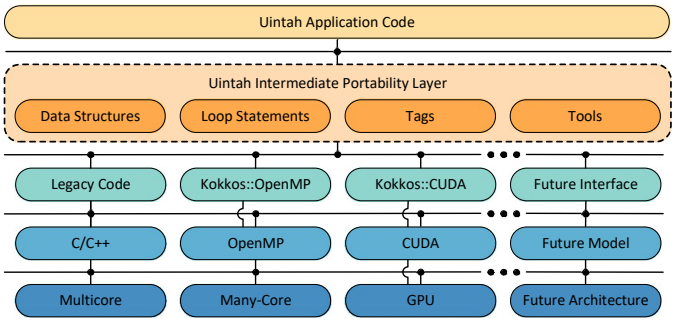

Fig. 1. Structure of Uintah's intermediate portability layer.

Direction adoption of a performance portability layer, however, poses challenges for large pre-existing software frameworks that may need to preserve legacy code and/or adopt other programming models in the future (e.g., a new programming model needed to support a novel architecture). This is a result of reliance on the performance portability layer to provide support for new underlying programming models. Such challenges are complicated in large legacy codebases where multiple refactors are not feasible and even one refactor may require a significant investment.

These challenges are addressed here using an indirect adoption approach that introduces a framework-specific portability layer between the application developer and the adopted performance portability layer. Figure 1 shows an example of such an intermediate layer in the context of this work. Much like how a performance portability layer eases investment in multiple programming models, a framework-specific intermediate portability layer is needed to ease investment in a performance portability layer.

The goal of this intermediate layer is for application developers to, hopefully, need only adopt the layer once to support current and future interfaces to underlying programming models. For application developers, this layer allows for easy adoption of underlying programming models without requiring knowledge of low-level implementation details. For infrastructure developers, this layer allows for easy addition, removal, and tuning of interfaces behind-the-scenes in a single location, reducing the need for far-reaching changes across application code.

This paper describes implementation of such a framework-

specific portability layer used to address performance portability layer limitations for legacy code. This intermediate layer consists of three components: (1) loop-level support providing application developers with framework-specific abstractions (e.g., generic parallel loop statements) that map to interface-specific abstractions (e.g., PPL-specific parallel loop statements), (2) application-level support that includes a tagging system to identify which interfaces are supported by a given loop, and (3) build-level support that includes selective compilation of loops to allow for incremental refactoring and simultaneous use of multiple underlying programming models for heterogeneous HPC systems.

This design is informed by a multi-year Kokkos C++ library [4] adoption effort adding portable support for OpenMP and CUDA in a complex real-world application and asynchronous many-task runtime system, the Uintah Computational Framework [5]. This ongoing effort has been non-trivial due to the codebase: (1) consisting of 1-2 million lines of complex code, (2) maintaining a divide between application code, where framework-specific abstractions are needed, and infrastructure code, where interface-specific abstractions are implemented, (3) having hundreds of pre-existing loops to port in application code, (4) being under active development with many contributors, and (5) having a pre-existing userbase to support. The resulting approach aims to ease performance portability layer adoption in similar codebases and help improve legacy code support and long-term portability for future architectures and programming models. Though adoption has been limited to Kokkos, high-level ideas associated with this approach are broad enough to apply to performance portability layers offering similar parallel loop statements such as RAJA, as will be discussed in Section V.

To demonstrate Kokkos capabilities, two case studies using this approach are examined for challenging calculations modeling the char oxidation of coal particles and radiative heat transfer in large-scale combustion simulations predicting performance of a next-generation, 1000 MWe ultra-supercritical clean coal boiler. These case studies show portable use of OpenMP and CUDA via Kokkos across multicore-, many-core-, and GPU-based nodes using a single implementation. The associated refactors have allowed for performance improvements up to 2.7x when refactoring for portability and 2.6x when more efficiently using a node to be achieved at the node-level. At scale, the use of MPI+Kokkos has allowed for good strong-scaling to 442,368 threads across 1,728 Knights Landing processors to be achieved.

The remainder of this paper is structured as follows. Section II provides an overview of the Uintah Computational Framework. Section III provides an overview of the Kokkos C++ library. Section IV outlines the history and current state of Uintah's support for Kokkos. Section V discusses direct adoption challenges and provides an overview of an indirect adoption approach in the context of Uintah and Kokkos. Sections VI through IX discuss loop-level, application-level, and build-level details for easing adoption of a portability layer. Section X presents single-node results for case studies using the char oxidation model. Section XI presents single-node and multi-node results for case studies using the radiation model. Section XII suggests potential challenges moving forward and Section XIII concludes this paper.

II. The Uintah Computational Framework

The Uintah Computational Framework is an open-source asynchronous many-task (AMT) runtime system and block-structured adaptive mesh refinement (SAMR) framework specializing in large-scale simulation of fluid-structure interaction problems. These problems are modeled by solving partial differential equations on structured adaptive mesh refinement grids. Uintah is based upon novel techniques for understanding a broad set of fluid-structure interaction problems. [5]

Through its lifetime, Uintah has been ported to a diverse set of major HPC systems. For multicore systems, good scaling has been shown to 96K, 262K, and 512K cores on the NSF Stampede, DOE Titan, and DOE Mira systems, respectively [5]–[7]. For many-core systems, good strong-scaling has been shown to 1,728 Knights Landing processors on the NSF Stampede 2 system [8] and 128 core groups on the NRCPC Sunway TaihuLight system [9]. For GPU-based systems, good strong-scaling has been shown to 16K GPUs [10] on the DOE Titan system.

Uintah is one of many AMT runtime systems and SAMR frameworks. Examples of similar AMT runtime systems include Charm++ [11], HPX [12], Legion [13], PaRSEC [14], and StarPU [15]. Examples of similar SAMR frameworks include BoxLib [16] (superseded by AMReX [17]) and Cactus [18]. An analysis of performance portability for representative AMT runtime systems, including Uintah, can be found in a recent technical report [19]. A review of representative SAMR frameworks, including Uintah, can be found in a recent survey [20].

A key idea maintained within Uintah is that application developers are isolated from infrastructure code. This is accomplished using an AMT-based approach to overdecompose application code into tasks and the computational domain into groups of individual cells, which tasks iterate over, to increase node-level parallelism. This approach is used to simplify development while easing use of the underlying hardware for application developers. For application developers, this divide allows them to focus on writing loop-based tasks rather than building an understanding of low-level execution details (e.g., data access patterns, load balancing, task scheduling). For infrastructure developers, this divide allows for fine-tuning of such details to be managed in a central location, reducing the need for far-reaching changes across application code.

The topmost layer of Uintah, application code, consists of simulation components such as ARCHES [21], which has been the focus of Kokkos porting efforts and is a Large Eddy Simulation (LES) code used to model heat, mass, and momentum transport in turbulent reacting flows with participating media radiation. Application code is decomposed into individual tasks that correspond to, for example, physics routines that are executed on either the host or device. The resulting collection

of tasks is compiled into a task graph and dynamically executed by the bottommost layer, infrastructure code, in an asynchronous out-of-order manner with implicit work stealing using the underlying runtime system. Execution is managed by the task scheduler, which interacts with per-MPI process task queues to select and execute ready tasks (e.g., tasks with satisfied data dependencies). Framework-specific abstractions (e.g., Uintah-specific parallel loop statements) are needed in individual tasks corresponding to portions of application code. Interface-specific abstractions (e.g., Kokkos views) are implemented in infrastructure immediately surrounding running tasks (e.g., the task scheduler and data warehouse).

III. The Kokkos C++ Library

The Kokkos C++ library [4] is an open-source C++ programming model developed at Sandia National Laboratories for writing portable, thread-scalable code optimized for a diverse set of architectures supported in major HPC systems. This programming model is part of the Kokkos C++ Performance Portability Programming EcoSystem, which additionally provides developers with Kokkos-aware algorithms, math kernels, and tools. Kokkos is one of many programming models offering a single interface to multiple underlying programming models (e.g., CUDA, OpenCL, OpenMP, etc). Examples of similar programming models include OCCA [22], RAJA [23], and SYCL [24].

A key idea among performance portability layers is use of back-ends to manage execution and memory in a portable manner. In the case of Kokkos, these back-ends are mapped to abstractions providing developers with portable parallel execution patterns (e.g. *parallel_for*, *parallel_reduce*, *parallel_scan*) and data structures (e.g., Kokkos Views). These fundamental abstractions allow Kokkos to manage both where and how: (1) patterns are executed and (2) data is stored and accessed. Note, Kokkos back-ends to OpenMP and CUDA are referred to throughout this paper as Kokkos::OpenMP and Kokkos::CUDA, respectively. More details on Kokkos usage can be found on the Kokkos GitHub [25], [26].

Uintah has adopted Kokkos to extend its codebase in a portable manner to multicore-, many-core-, and GPU-based systems. Specifically, to: (1) avoid code bifurcation when extending Uintah to accelerators and many-core devices with CUDA and OpenMP, respectively, (2) use a single interface to interact with multiple underlying programming models, and (3) offload low-level implementation details to Kokkos. This adoption has also allowed for a reduction in the gap between development time and our ability to run on newly introduced systems. For these advantages, Kokkos is believed to play a critical role in preparing Uintah for future HPC systems.

Uintah is an early adopter of Kokkos with Uintah developers collaborating directly with Kokkos developers as a part of the University of Utah's participation in the DOE/NNSA's Predictive Science Academic Alliance Program (PSAAP) II initiative. This collaboration has resulted in bi-directional development efforts with developers working in each other's codebases. At Sandia National Laboratories, Kokkos has been integrated in Trilinos [27] and used in codes such as Albany [28], GenTen [29], HOMMEXX [30], LAMMPS [31], and SPARTA [32]. Examples of other codes investigating and/or adopting Kokkos include BabelStream [33], K-Athena [34], KARFS [35], NekMesh [36], and TeaLeaf [37]. A list of applications using Kokkos can be found on the Kokkos GitHub [38].

IV. State of Uintah's Support for Kokkos

The non-trivial nature of Uintah's adoption of Kokkos has required a number of small-scale case studies and refactors [8], [39]–[42]. These individual efforts validate the use of Kokkos: (1) in simple representative settings outside of Uintah, (2) in simple isolated portions of Uintah, (3) in complex isolated portions of Uintah, (4) in complex far-reaching portions of Uintah, and (5) at scale. This incremental approach has been critical for ensuring continued feasibility and success of the effort given the high levels of investment required of Uintah when adopting a performance portability layer. Refactoring the most complex code early on has been key to identifying challenges quickly and refining best practices to simplify refactors moving forward. Such an approach is important for this and similar codebases where suitability must be evaluated before far-reaching adoption and significant investment in a performance portability layer.

To date, Uintah's Kokkos-related activities include:

- Single-node case studies exploring use of Kokkos parallel patterns with Kokkos::OpenMP in a simple standalone example outside of Uintah's simulation components [39]
- Single-node case studies exploring use of Kokkos parallel patterns and unmanaged Kokkos views with Kokkos::OpenMP and Kokkos::CUDA in a mock runtime system representative of Uintah and the ARCHES simulation component [40]
- Implementation of Uintah-specific abstractions to provide portable interfaces to Kokkos parallel patterns and unmanaged Kokkos views
- Refactoring of a challenging standalone radiative heat transfer calculation outside of Uintah's simulation components to support use of Kokkos::OpenMP [41]
- Multi-node case studies exploring use of Kokkos::OpenMP at scale on the NSF Stampede 2 system using the refactored radiative heat transfer calculation [41]
- Incremental refactoring of loops in ARCHES to support use of Kokkos::OpenMP *(ongoing)*
- Implementation of a portable random number generator adopting Kokkos_Random functionality [42]
- Refactoring of the radiative heat transfer calculation to additionally support use of Kokkos::CUDA [42]
- Extension of Kokkos itself to support asynchronous execution of Kokkos parallel patterns [42]
- Multi-node case studies exploring use of Kokkos::OpenMP at scale on the DOE Theta system and use of Kokkos::CUDA at scale on the DOE Titan

system using the refactored radiative heat transfer calculation [42]
- Implementation of a new task scheduler adopting Kokkos partitioning functionality to support use of nested Kokkos::OpenMP [8]
- Implementation of portable synchronization primitives based on Kokkos MasterLock functionality
- Implementation of a task tagging system to allow for selective compilation of loops across Kokkos back-ends [8]
- Refactoring of a challenging combustion loop modeling the char oxidation of coal particles in ARCHES to support use of Kokkos::OpenMP and Kokkos::CUDA [8]
- Single-node case studies exploring use of nested Kokkos::OpenMP and Kokkos::CUDA at the loop-level using both the refactored radiative heat transfer calculation and refactored char oxidation model [8]
- Multi-node case studies exploring use of nested Kokkos::OpenMP at scale on the NSF Stampede 2 system using the refactored radiative heat transfer calculation [8]
- Incremental refactoring of loops in ARCHES to additionally support use of Kokkos::CUDA *(ongoing)*
- Implementation of a new task scheduler to support simultaneous use of nested Kokkos::OpenMP and Kokkos::CUDA *(ongoing)*

This progress has been achieved using the following Kokkos functionality [43]:

- Kokkos::parallel_for
- Kokkos::parallel_reduce (min and sum reductions)
- Kokkos::View (unmanaged)
- Kokkos::OpenMP::partition_master
- Kokkos::Experimental::MasterLock
- Kokkos_Random

The indirect performance portability layer adoption approach that has been informed by this progress is discussed in Sections V through IX.

V. Uintah's Intermediate Portability Layer

A fundamental abstraction shared among several performance portability layers is the parallel loop statement. This abstraction is key to providing access to multiple underlying programming models through a single interface. Though exact syntax and implementation details vary, parallel loop statements generally rely upon an iteration range and a loop body defined as a C++ lambda or functor. An example of simplified syntax for Kokkos and RAJA parallel loop statements from Figure 5 in a recent evaluation [44] is shown below:

```
// Kokkos
parallel_for( n, KOKKOS_LAMBDA( int i )
BODY
);

// RAJA
forall<thread_exec>( 0, n, [=]( Index_type i )
BODY
);
```

TABLE I
Components of Uintah's Intermediate Portability Layer.

Level	Component
Loop[a]	Generic Loop Statements Mapped to Multiple Execution Schemes
	Generic Data Structures Mapped to Multiple Data Structures
Application[b]	Arbitrary Tags to Manage Interfaces to Programming Models
	Arbitrary Execution Spaces to Manage Execution Schemes
	Arbitrary Memory Spaces to Manage Data Structures
	Portable Tools (e.g., Locks, Random Number Generators)
Build[c]	Preprocessor Macros to Manage Multiple Build Configurations
	Build-Specific Tags to Manage Selective Compilation of Loops

[a]described further in Section VI.
[b]described further in Section VII.
[c]described further in Section VIII.

Note, more discussion on similarities and differences among modern C++ parallel programming models, including Kokkos, RAJA, and SYCL, can be found in a recent evaluation [44] and comparative analysis [45].

For OpenMP and CUDA themselves, the parallel loop statements and other abstractions offered by Kokkos have worked well in Uintah for the advantages discussed in Section III. The high levels of investment required of Uintah when adopting Kokkos, however, has discouraged direct adoption of these PPL-specific abstractions. Specifically, direct adoption of Kokkos throughout Uintah has been avoided to: (1) allow for legacy code to be preserved, (2) eliminate reliance on Kokkos to provide support for new underlying programming models, (3) simplify abstractions provided for application developers, and (4) ease re-work should implementation changes or a different performance portability layer be needed.

The approach taken to indirectly adopt Kokkos within Uintah uses an intermediate portability layer to provide Uintah-specific abstractions that interact with underlying programming models through various interfaces (e.g., implementing Kokkos-specific abstractions). These framework-specific interfaces allow for pre-existing code to be preserved when adopting Kokkos and, in theory, provide easy means of adopting other programming models should Kokkos not yet support one needed for a novel architecture. To date, Uintah's interfaces map Uintah-specific abstractions to: (1) legacy code, (2) Kokkos-specific abstractions for Kokkos::OpenMP, and (3) Kokkos-specific abstractions for Kokkos::CUDA. Note, individual interfaces are used for Kokkos::OpenMP and Kokkos::CUDA to ease selective compilation of loops and to provide more control over the implementation and execution of loops.

Table I shows the individual components that form Uintah's intermediate portability layer. Specifically, this intermediate layer consists of three components: (1) loop-level support providing application developers with framework-specific abstractions (e.g., generic parallel loop statements) that map to interface-specific abstractions (e.g., PPL-specific parallel loop statements), (2) application-level support that includes a tagging system to identify which interfaces are supported by a given loop, and (3) build-level support that includes selective compilation of loops to allow for incremental refactoring and simultaneous use of multiple underlying programming models for heterogeneous HPC systems.

An example of how these components are implemented is the *Uintah::parallel_for* shown below:

```
// Uintah
parallel_for( executionObject
            , iterationRange
            , LAMBDA( int i, int j, int k )
BODY
);
```

This is a framework-specific parallel loop statement modeled after the approach used by performance portability layers. Similar to performance portability layer goals, this abstraction aims to provide application developers with a single loop statement that allows for easy adoption of underlying programming models without requiring knowledge of low-level implementation details (e.g., for Kokkos). In practice, this approach has worked well for application developers used to writing serial loops in Uintah with little parallel programming experience.

The *Uintah::parallel_for* parameter list differs slightly from previous examples in that it includes an additional parameter and requires 3-dimensional indexing. The *executionObject* parameter is a templated object used to pass non-portable objects and additional parameters (e.g., CUDA streams, CUDA blocks per loop, template parameters used to manage paths of execution, etc) into portable loops for use behind-the-scenes in interfaces to underlying programming models. Three-dimensional indexing is used to ease legacy code support and is mapped to 1-dimensional indexing, as needed, behind-the-scenes. This is managed with the help of *iterationRange*, which is an object used to pass iteration range indices into portable loops. Note, *LAMBDA* is a generic macro for managing lambda capture clauses and CUDA annotation (e.g., *__device__*) in a manner similar to that used by Kokkos and RAJA (e.g., KOKKOS_LAMBDA, RAJA_DEVICE, RAJA_HOST_DEVICE).

Behind-the-scenes, preprocessor macros and template metaprogramming are used to manage paths of execution for Uintah's interfaces to underlying programming models in a single location. For example, a *Uintah::parallel_for* is executed using a *Kokkos::parallel_for* optimized for CUDA when Uintah is built with Kokkos::CUDA. This behind-the-scenes management is key to easily adding, removing, and tuning interfaces (e.g., to change how a *Kokkos::parallel_for* iterates over work items or, in theory, add support for another performance portability layer's parallel loop statement). Details on Uintah's intermediate portability layer, including code examples, can be found in a recent technical report [8].

VI. Loop-Level Details

A. Portable Code Inside of Portable Abstractions

A key benefit of performance portability layers is their ability to execute a single implementation in many different ways. This, however, is not guaranteed by adopting a portable abstraction itself. This is complicated by underlying programming models supporting code to different extents (e.g., convenience mechanisms). Understanding what can and cannot be done in portable loops is helpful for ensuring successful compilation and execution across multiple underlying programming models. This is eased by keeping code in portable loops as simple as possible.

Examples of changes needed in pre-existing loops to make portable use of OpenMP and CUDA via Kokkos in Uintah include: (1) eliminating use of C++ standard library classes and functions that do not have CUDA equivalents and (2) eliminating allocation of host memory in portable loops.

B. Implementation of Portable Loops

Portable loop abstractions bring with them implementation challenges independent of whether they are adopted directly or indirectly. This is a result of great flexibility in where and how execution and memory is managed. This is complicated by pre-existing serial loops where parallel execution and thread safety need not be accounted for. Thinking through implementation and execution of portable loops is important for improving loop-level performance and scalability.

Implementation and execution details found helpful when adopting Kokkos in Uintah include: (1) ensuring that portable loops are written in a thread-safe manner, (2) ensuring that portable loops are provided with enough work items to iterate over in parallel (e.g., at least as many work items as there are OpenMP threads), (3) using lambdas instead of functors (e.g., to avoid duplication of long parameter lists), (4) considering how to structure portable loops (e.g., 1D, 3D, etc), (5) considering how portable loops iterate over work items (e.g., individually or in groups), (6) considering how portable loops utilize underlying hardware (e.g., cores, caches, etc), (7) exploring configurability of underlying programming models (e.g., OpenMP loop scheduling parameters), (8) adding runtime parameters to manage execution (e.g., OpenMP threads per loop, CUDA blocks per loop, etc). For (1), tools such as Archer [46], Intel Inspector, and ThreadSanitizer [47] are helpful for identifying data races.

VII. Application-Level Details

A. Portable Code Outside of Portable Abstractions

A key benefit of performance portability layers is their ability to reduce the amount of duplicated code in an application. This, however, applies only to the portable abstractions adopted. In practice, application code extends beyond portable abstractions (e.g., in large legacy codebases). Looking for opportunities to apply portable techniques used by performance portability layers elsewhere in application code is important for improving long-term portability and code maintainability.

Examples encountered when adopting Kokkos in Uintah include using behind-the-scenes preprocessor macros and template metaprogramming to add portable support for: (1) arbitrary tags to manage interfaces to underlying programming models (e.g., for selective compilation of loops), (2) arbitrary execution spaces to manage loop execution schemes, (3) arbitrary memory spaces to manage data structures, and (4) an object to pass interface-specific needs into portable loops (e.g., CUDA streams).

B. Portable Tools for Application Code

Commonly used tools (e.g., C++ standard library convenience mechanisms) pose portability challenges when using multiple underlying programming models. This is a result of underlying programming models supporting such tools to different extents. This is complicated by pre-existing loops using non-portable tools (e.g., in large legacy codebases). Thinking through which portable tools to support before far-reaching adoption of a performance portability layer is important for avoiding unexpected refactors.

Portable tools found helpful when adopting Kokkos in Uintah include portable: (1) vector containers, (2) synchronization mechanisms, (3) random number generation, and (4) mechanisms for simultaneously executing portable loops. In Kokkos, portable options for (1), (3), and (4) are provided via *Kokkos::Vector*, *Kokkos_Random*, and *Kokkos::OpenMP::partition_master*, respectively. For (2), a Uintah-specific abstraction based on *Kokkos::Experimental::MasterLock* was implemented to avoid mixing use of *std::mutex* and *omp_lock_t*. Details on Uintah's use of *Kokkos::OpenMP::partition_master* can be found in a recent technical report [8].

VIII. Build-Level Details

A. Support for Multiple Build Configurations

Adoption of multiple underlying programming models requires careful consideration of new build configurations and paths of execution. This is complicated by heterogeneous HPC systems requiring simultaneous use of multiple underlying programming models (e.g., OpenMP and CUDA) to fully utilize a heterogeneous compute node. Thinking through how to manage current and future build configurations before far-reaching adoption of a performance portability layer is important for avoiding unexpected refactors.

Recurring paths of execution encountered when adopting Kokkos in Uintah include: (1) code needed for the underlying programming model independent of the performance portability layer (e.g., OpenMP locks), (2) code needed for the performance portability layer independent of the underlying programming model(s) (e.g., Kokkos Views), and (3) code needed for the performance portability layer dependent upon the underlying programming model(s) (e.g., Uintah-specific abstractions for Kokkos::OpenMP).

Consistent use of standardized preprocessor macros simplifies management of such paths. An example of a macro definition for (1) is *HAVE_<BACK-END>*, which is defined when the application picks up the underlying programming model. An example of a macro definition for (2) is *HAVE_<PPL>*, which is defined when the application picks up the performance portability layer. An example of a macro definition for (3) is *<APP>_ENABLE_<PPL>_<BACK-END(S)>*, which is defined when both the application and the performance portability layer pick up the underlying programming model(s). Note, preprocessor macros helpful for identifying when Kokkos itself picks up the underlying programming model(s) can be found in *kokkos/core/src/Kokkos_Macros.hpp*.

When using multiple underlying programming models, preprocessor macro logic to support (3) becomes complicated quickly. This posed unexpected challenges requiring additional refactors when adding support for Kokkos::OpenMP and Kokkos::CUDA in the same Uintah build. Use of preprocessor macros explicitly identifying code specific to such builds (e.g., *UINTAH_ENABLE_KOKKOS_OPENMP_CUDA*) is helpful for simplifying logic and readability. Note, heterogeneous builds can be simplified using the *nvcc_wrapper* Linux shell script found on the Kokkos GitHub [48].

B. Selective Compilation of Portable Loops

Use of portable abstractions across multiple loops poses challenges when adding support for additional underlying programming models. This is a result of every portable loop having to properly support the newly adopted programming model to avoid breaking builds. It is not feasible, however, to refactor all loops at once (e.g., to remove non-portable code) after portable abstractions have been widely adopted throughout a codebase.

This challenge is addressed here using a tagging system that allows application developers to individually identify the supported interfaces for each individual loop. These tags are used to ensure that loops are compiled for only the respective underlying programming models that are supported to avoid breaking builds. This allows for incremental refactoring on a loop-by-loop basis when adding support for additional programming models, eliminating the need to refactor all loops at once. This approach also simplifies the isolation of problematic code by allowing loops to be easily enabled/disabled across programming models when debugging.

Such a tagging system has been implemented in Uintah using preprocessor macros and template metaprogramming. At compile-time, a portable loop is compiled for all interfaces identified as being currently supported by application developers using macro-based tags (e.g., KOKKOS_OPENMP_TAG). Behind-the-scenes, provided tags are mapped to their respective underlying execution space and memory space (e.g., Kokkos::OpenMP and Kokkos::HostSpace). At run-time, the portable loop is executed by one of the supported underlying programming models based upon build-specific paths of execution and Uintah-specific template parameters (e.g., ExecSpace and MemSpace). Uintah-specific template parameters extend those used by Kokkos to allow for other non-Kokkos execution spaces and memory spaces to be supported (e.g., Uintah::Legacy and Uintah::HostSpace to preserve legacy code). This approach is used to improve long-term portability of execution spaces and memory spaces for future interfaces and underlying programming models.

IX. General Details

A. Far-Reaching Test Coverage and Regular Testing

Far-reaching test coverage and regular testing of a codebase is critical for easing adoption of a performance portability layer. This is a result of it being inherently easy to introduce unanticipated changes in portable code due to emphasis on

executing a single implementation in many different ways. This is complicated by combinatorially increasing scenarios to consider for newly introduced underlying programming models, build configurations, and run configurations. Test scenarios found helpful when adopting Kokkos in Uintah include testing: (1) each underlying programming model, (2) relevant combinations of underlying programming models, (3) each build configuration, (4) serial execution of portable loops, and (5) parallel execution of portable loops. Such testing is especially important for codebases where multiple tests execute different subsets of partially overlapping portable loops, which makes it is easier to introduce unanticipated changes.

B. Standardization of Adopted Portability Layers

Far-reaching adoption of a portability layer poses code maintainability and debugging challenges. This is a result of portable loops relying upon each other for successful compilation and execution. Standardization of newly adopted portability layers eases these challenges by improving searchability to simplify far-reaching changes (e.g., to add support for a new interface) and debugging (e.g., to quickly identify all code using a given interface). An example of standardization applied to Uintah's intermediate portability layer includes consistent formatting, naming conventions, and whitespace.

C. Living Best Practices

Pre-existing loops pose refactoring challenges when incrementally adopting a performance portability layer in a large legacy codebase. This is a result of not knowing both what and how much non-portable code is used in loops before adoption. This is complicated by loops having different barriers to portability. Maintaining a living document collecting best practices and portability barriers from past refactors helps simplify refactors moving forward.

D. Incremental Case Studies

Carefully selected case studies are helpful for evaluating a performance portability layer before far-reaching adoption and significant investment. Two types of case studies used to ease adoption of Kokkos in Uintah include those examining: (1) the most complex code and (2) simple representative code. Case studies using (1) identified challenges quickly and informed best practices. Examples of (1) include loops with complicated nesting hierarchies, extensive use of C/C++ functionality, complex data structures, etc. Case studies using (2) evaluated representative performance of more typical portable loops and refined best practices. Examples of (2) include loops with recurring patterns throughout the codebase, simple math, etc. Results from past Uintah case studies can be found in a recent technical report [8] and other publications [39]–[42].

X. Char Oxidation Modeling Case Studies

Uintah's char oxidation model is a large consumer of simulation time among models evaluated during the time integration of physics in ARCHES [21]. This complex loop has approximately 350 lines with multiple interior loops and Newton iterations. This algorithm has a theoretical arithmetic intensity of approximately 1.30 FLOPs per double precision number. Details on this model can be found in a recent technical report [8].

TABLE II
CharOx:CPU incremental refactor on Intel Sandy Bridge.[a]

Refactor Step*	$1 - 16^3$ Patch	$1 - 32^3$ Patch	$1 - 64^3$ Patch
0: Original serial loop	17.87 (-)	141.8 (-)	1132 (-)
1: Serial Uintah::parallel_for	19.19 (0.9x)	142.1 (1.0x)	1148 (1.0x)
2: No std::vector	11.74 (1.5x)	93.7 (1.5x)	753 (1.5x)
3: No temporary objects	10.96 (1.6x)	88.5 (1.6x)	710 (1.6x)
4: No virtual functions	9.75 (1.8x)	78.5 (1.8x)	634 (1.8x)
5: Portable data structures	10.18 (1.8x)	78.6 (1.8x)	633 (1.8x)
6: No std::string	9.16 (2.0x)	73.3 (1.9x)	591 (1.9x)
7: Improved memory access	6.73 (2.7x)	55.2 (2.6x)	444 (2.5x)

[a]per-loop timing reported in milliseconds (x speedup).
*refactor steps are cumulative.

A. Single-Node Studies

The results presented in this section used two implementations of Uintah's char oxidation model: (1) CharOx:CPU, an existing implementation with serial loops and (2) CharOx:Portable, a new portable implementation supporting legacy serial loops and Kokkos-based data parallel loops for Kokkos::OpenMP and Kokkos::CUDA.

SNB-based results were gathered on a node with two 2.7 GHz Intel Xeon E5-2680 Sandy Bridge processors with 8 cores (2 threads per core) per processor and 64 GB of RAM. SKX-based results were gathered on a node with one 2.7 GHz Intel Xeon Gold 6136 Skylake processor with 12 cores (2 threads per core) per processor and 256 GB of RAM. KNL-based results were gathered on a node with one 1.3 GHz Intel Xeon Phi 7210 Knights Landing processor configured for *Flat-Quadrant* mode with 64 cores (4 threads per core) and 96 GB of RAM. Maxwell-based results were gathered on a Maxwell-based NVIDIA GeForce GTX Titan X GPU with 12 GB of RAM. Volta-based results were gathered on a Volta-based NVIDIA Tesla V100 GPU with 16 GB of RAM.

Aside from Table II, simulations were launched using 1 MPI process per node with run configurations using the full node. SNB-, SKX-, and KNL-based problems used 1 patch per core. Maxwell- and Volta-based problems used 16 patches. Note, a patch is the collection of cells executed by a loop. Reported per-loop timings measure execution of the *Uintah::parallel_for*. Results have been averaged over 7 consecutive timesteps and 80-320 loops per timestep depending upon patch count. Over 10 identical runs, results gathered in this manner had at most 6.8% difference between two runs.

Table II shows incremental performance improvements achieved on Intel Sandy Bridge when refactoring the char oxidation model. These results were gathered using CharOx:CPU for three patch sizes (16^3, 32^3, and 64^3 cells) at various steps of the refactor. 16 MPI processes were used to simultaneously execute loops across 16 cores using 1 core and 1 thread per loop. Step 0 corresponds to serial execution of the original serial loop. Step 1 corresponds to serial execution of the

TABLE III
CHAROX:PORTABLE THREAD SCALABILITY ON INTEL SANDY BRIDGE.[a]

Cores per Loop	Threads per Loop	1 - 16^3 Patch	1 - 32^3 Patch	1- 64^3 Patch
1	2*	7.36 (0.9x)	50.38 (1.1x)	426.7 (1.1x)
1	1	6.90 (-)	55.88 (-)	469.2 (-)
2	2	4.38 (1.6x)	29.34 (1.9x)	239.8 (2.0x)
4	4	2.54 (2.7x)	15.13 (3.7x)	120.4 (3.9x)
8	8	1.54 (4.5x)	7.51 (7.4x)	60.3 (7.8x)
16**	16	0.48 (14.4x)	3.72 (15.0x)	30.6 (15.3x)
16**	32*	0.41 (16.8x)	3.24 (17.2x)	26.7 (17.6x)

[a]per-loop timing reported in milliseconds (x speedup).
*2 threads per core.
**2 sockets.

loop after refactoring to use Uintah's abstraction providing interfaces to legacy code and *Kokkos::parallel_for*. Step 2 corresponds to replacing use of *std::vector* inside of the loop with 1-dimensional arrays of doubles. Step 3 corresponds to replacing temporary object construction inside of the loop with 2-dimensional arrays of doubles. Step 4 corresponds to hard-coding short virtual functions inside of the loop. Step 5 corresponds to refactoring data warehouse variables to use Uintah's abstraction providing interfaces to legacy data structures and unmanaged Kokkos Views. Step 6 corresponds to replacing *std::string* comparisons inside of the loop with integer-based comparisons. Step 7 corresponds to restructuring the loop to improve data warehouse variable access patterns (e.g., algorithm transformations to avoid conditionals that operate on different variables). Note, Step 1 and Step 5 are not expected to impact performance as abstractions execute the underlying legacy code in the same manner as in Step 0. Refactoring for portability allowed for speedups up to 2.7x, 2.6x, and 2.5x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, over the original serial loop.

Table III shows Kokkos::OpenMP-based thread scalability on Intel Sandy Bridge for char oxidation modeling. These results were gathered using CharOx:Portable with Kokkos::OpenMP for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, 1 MPI process and 16 OpenMP threads were used to simultaneously execute loops across 16 cores using from 1 core and 1 thread per loop to 16 cores and 16 threads per loop. For 2 threads per core runs, 1 MPI process and 32 OpenMP threads were used to simultaneously execute loops across 16 cores using both 1 core and 2 threads per loop and 16 cores and 32 threads per loop. Adding loop-level parallelism via more threads allowed for speedups up to 16.8x, 17.2x, and 17.6x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 16 cores with 2 threads per core over use of 1 core and 1 thread per loop.

Table IV shows Kokkos::OpenMP-based thread scalability on Intel Skylake for char oxidation modeling. These results were gathered using CharOx:Portable with Kokkos::OpenMP for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, 1 MPI process and 12 OpenMP threads were used to simultaneously execute loops across 12 cores using from 1 core and 1 thread per loop to 12 cores and 12 threads per loop. For 2 threads per core runs, 1 MPI process and 24 OpenMP threads were used to simultaneously execute loops

TABLE IV
CHAROX:PORTABLE THREAD SCALABILITY ON INTEL SKYLAKE.[a]

Cores per Loop	Threads per Loop	1 - 16^3 Patch	1 - 32^3 Patch	1- 64^3 Patch
1	2*	3.32 (0.8x)	22.78 (1.1x)	164.0 (1.2x)
1	1	2.81 (-)	25.60 (-)	196.9 (-)
2	2	2.03 (1.4x)	15.06 (1.7x)	103.8 (1.9x)
3	3	1.40 (2.0x)	10.13 (2.5x)	74.1 (2.7x)
4	4	1.01 (2.8x)	7.77 (3.3x)	54.9 (3.6x)
6	6	0.74 (3.8x)	5.20 (4.9x)	37.5 (5.3x)
12	12	0.29 (9.7x)	2.24 (11.4x)	18.6 (10.6x)
12	24*	0.22 (12.8x)	1.72 (14.9x)	14.1 (14.0x)

[a]per-loop timing reported in milliseconds (x speedup).
*2 threads per core.

TABLE V
CHAROX:PORTABLE THREAD SCALABILITY ON INTEL KNIGHTS LANDING.[a]

Cores per Loop	Threads per Loop	1 - 16^3 Patch	1 - 32^3 Patch	1- 64^3 Patch
1	4**	23.44 (1.2x)	150.83 (1.4x)	1232.8 (1.5x)
1	2*	23.48 (1.2x)	160.96 (1.3x)	1343.6 (1.3x)
1	1	27.36 (-)	216.79 (-)	1812.6 (-)
2	2	18.02 (1.5x)	114.52 (1.9x)	903.2 (2.0x)
4	4	9.55 (2.9x)	59.26 (3.7x)	459.4 (3.9x)
8	8	4.84 (5.7x)	31.18 (7.0x)	232.4 (7.8x)
16	16	2.62 (10.4x)	17.76 (12.2x)	122.8 (14.8x)
32	32	1.64 (16.7x)	10.55 (20.5x)	63.0 (28.8x)
64	64	0.63 (43.4x)	4.63 (46.8x)	30.8 (58.9x)
64	128*	0.59 (46.4x)	3.31 (65.5x)	23.6 (76.8x)
64	256**	1.59 (17.2x)	5.08 (42.7x)	27.2 (66.6x)

[a]per-loop timing reported in milliseconds (x speedup).
*2 threads per core.
**4 threads per core.

across 12 cores using both 1 core and 2 threads per loop and 12 cores and 24 threads per loop. Adding loop-level parallelism via more threads allowed for speedups up to 12.8x, 14.9x, and 14.0x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 12 cores with 2 threads per core over use of 1 core and 1 thread per loop.

Table V shows Kokkos::OpenMP-based thread scalability on Intel Knights Landing for char oxidation modeling. These results were gathered using CharOx:Portable with Kokkos::OpenMP for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, 1 MPI process and 64 OpenMP threads were used to simultaneously execute loops across 64 cores using from 1 core and 1 thread per loop to 64 cores and 64 threads per loop. For 2 threads per core runs, 1 MPI process and 128 OpenMP threads were used to simultaneously execute loops across 64 cores using both 1 core and 2 threads per loop and 64 cores and 128 threads per loop. For 4 threads per core runs, 1 MPI process and 256 OpenMP threads were used to simultaneously execute loops across 64 cores using both 1 core and 4 threads per loop and 64 cores and 256 threads per loop. Adding loop-level parallelism via more threads allowed for speedups up to 46.4x, 65.5x, and 76.8x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 64 cores with 2 threads per core over use of 1 core and 1 thread per loop.

Table VI shows Kokkos::CUDA-based block scalability using 256 CUDA threads per block on NVIDIA Maxwell for char oxidation modeling. These results were gathered using CharOx:Portable with Kokkos::CUDA for three patch sizes (16^3, 32^3, and 64^3 cells). 1 MPI process and 16 threads were

TABLE VI
CHAROX:PORTABLE BLOCK SCALABILITY ON NVIDIA MAXWELL.[a]

CUDA Blocks per Loop*	1 - 16^3 Patch	1 - 32^3 Patch	1- 64^3 Patch
1	2.80 (-)	18.57 (-)	147.6 (-)
2	1.47 (1.9x)	9.59 (1.9x)	77.6 (1.9x)
4	0.80 (3.5x)	5.50 (3.4x)	44.0 (3.4x)
8	0.48 (5.8x)	3.17 (5.9x)	25.6 (5.8x)
16	0.36 (7.8x)	2.29 (8.1x)	19.0 (7.8x)
24	0.28 (10.0x)	1.92 (9.7x)	13.9 (10.6x)

[a]per-loop timing reported in milliseconds (x speedup).
*256 CUDA threads per block.

TABLE VII
CHAROX:PORTABLE BLOCK SCALABILITY ON NVIDIA VOLTA.[a]

CUDA Blocks per Loop*	1 - 16^3 Patch	1 - 32^3 Patch	1- 64^3 Patch
1	0.66 (-)	4.65 (-)	36.49 (-)
2	0.36 (1.8x)	2.74 (1.7x)	19.43 (1.9x)
4	0.21 (3.1x)	1.48 (3.1x)	10.63 (3.4x)
8	0.13 (5.1x)	0.88 (5.3x)	6.82 (5.4x)
16	0.09 (7.3x)	0.55 (8.5x)	4.46 (8.2x)
32	0.09 (7.3x)	0.41 (11.3x)	3.17 (11.5x)
64	0.09 (7.3x)	0.33 (14.1x)	2.67 (13.7x)
80	0.09 (7.3x)	0.33 (14.1x)	2.52 (14.5x)

[a]per-loop timing reported in milliseconds (x speedup).
*256 CUDA threads per block.

used to simultaneously execute loops using 1 CUDA stream and from 1 to 24 CUDA block(s) per loop with 256 CUDA threads per block and 255 registers per thread. Adding loop-level parallelism via more blocks allowed for speedups up to 10.0x, 9.7x, and 10.6x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 24 blocks per loop over use of 1 block per loop and 256 threads per block.

Table VII shows Kokkos::CUDA-based block scalability using 256 CUDA threads per block on NVIDIA Volta for char oxidation modeling. These results were gathered using CharOx:Portable with Kokkos::CUDA for three patch sizes (16^3, 32^3, and 64^3 cells). 1 MPI process and 16 threads were used to simultaneously execute loops using 1 CUDA stream and from 1 to 80 CUDA block(s) per loop with 256 CUDA threads per block and 255 registers per thread. Adding loop-level parallelism via more blocks allowed for speedups up to 7.3x, 14.1x, and 14.5x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 80 blocks per loop over use of 1 block per loop and 256 threads per block.

Results presented in Table II show that performance is a by-product of refactoring for portability. Comparing 1 core per loop, 1 thread per loop results in Table III to Step 7 results in Table II suggests that little to no performance has been lost when moving to CharOx:Portable. This is encouraging as this single implementation of CharOx now supports execution with Kokkos::OpenMP and Kokkos::CUDA in addition to legacy execution.

Results presented in Tables III through IV show that it is possible to achieve good loop-level scalability across multicore-based nodes. Results presented in Tables V through VII show that it can be difficult to achieve good loop-level scalability across many-core- and GPU-based nodes. This is not unexpected given the increasing amounts of parallelism offered by such nodes. As will be shown in Section XI, executing more, yet smaller, loops at the same time (e.g., using hierarchical parallelism) can be used to improve node utilization for loops that scale poorly. Looking more closely at Tables III through V, these results suggest that care must be taken when using multiple threads per core on multicore- and many-core-based nodes. Though modest performance improvements are achievable, performance reductions are also possible when using multiple threads per core with insufficient per-core work.

XI. RADIATION MODELING CASE STUDIES

Uintah's 2-level reverse Monte-Carlo ray tracing (RMCRT) radiation model plays a key role in CCMSC boiler simulations, where radiation is the dominant mode of heat transfer. This complex loop has approximately 500 lines with multiple interior loops and far-reaching irregular memory access patterns. This algorithm has a theoretical arithmetic intensity of approximately 0.66 FLOPs per double precision number. Details on this model can be found in a recent technical report [8].

A. Single-Node Studies

The results presented in this section used three implementations of Uintah's 2-Level RMCRT radiation model: (1) 2-Level RMCRT:CPU, an existing implementation with serial loops, (2) 2-Level RMCRT:GPU, an existing implementation with CUDA-based data parallel loops, and (3) 2-Level RMCRT:Portable, a new portable implementation supporting legacy serial loops and Kokkos-based data parallel loops for Kokkos::OpenMP and Kokkos::CUDA.

Aside from patch count, results were gathered on the same nodes as described in Section X-A and using 1 MPI process per node with run configurations using the full node. Here, patch counts were based on quantities constructing a 128^3 fine mesh. Reported per-timestep timings measure execution of the timestep and, thus, simultaneous execution of all loops in a given timestep. Results have been averaged over 7 consecutive timesteps. Over 10 identical runs, results gathered in this manner had at most 3.0% difference between two runs.

Table VIII shows performance comparisons across Intel Sandy Bridge, Intel Skylake, Intel Knights Landing, and NVIDIA Maxwell for radiation modeling. These results were gathered using 2-Level RMCRT:CPU, 2-Level RMCRT:Portable with Kokkos::OpenMP, 2-Level RMCRT:GPU, and 2-Level RMCRT:Portable with Kokkos:CUDA for a problem with 128^3 cells on the fine mesh and 32^3 cells on the coarse mesh for three fine mesh configurations (512, 64, and 8 patches with 16^3, 32^3, and 64^3 cells per patch, respectively). For each architecture, all supported run configurations were explored (e.g., from 1 to many loops executing at the same time with many to 1 threads per loop). For SNB, best run configurations for 2-Level RMCRT:Portable allowed for speedups up to 1.5x and 1.7x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported best run configurations for 2-Level RMCRT:CPU. For SKX, best run configurations for 2-Level RMCRT:Portable allowed for speedups up to 1.4x and 1.7x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported best

TABLE VIII
2-LEVEL RMCRT FULL-NODE PERFORMANCE COMPARISONS.[a]

Architecture[b]	2-Level RMCRT Implementation	512 - 16^3 Patches	64 - 32^3 Patches	8 - 64^3 Patches
Dual SNB	CPU	51.6* (-)	71.7 (-)	-[c] (-)
Same Config.	Portable	36.3* (1.4x)	55.5 (1.3x)	-[c] (-)
Best Config.	Portable	35.0* (1.5x)	42.0* (1.7x)	60.5* (-)
SKX	CPU	40.2* (-)	61.9 (-)	-[c] (-)
Same Config.	Portable	32.4* (1.2x)	46.9 (1.3x)	-[c] (-)
Best Config.	Portable	28.0* (1.4x)	37.0* (1.7x)	59.7* (-)
KNL	CPU	57.9** (-)	102.1 (-)	-[c] (-)
Same Config.	Portable	43.8** (1.3x)	81.0 (1.3x)	-[c] (-)
Best Config.	Portable	29.2** (2.0x)	38.8** (2.6x)	60.4** (-)
Maxwell	GPU	32.1 (-)	46.6 (-)	-[c] (-)
Same Config.	Portable	25.9 (1.2x)	36.7 (1.3x)	-[c] (-)
Best Config.	Portable	20.0 (1.6x)	25.6 (1.8x)	43.6 (-)

[a]per-timestep timing reported in milliseconds (x speedup).
[b]same configuration as the best non-Kokkos; best configuration using Kokkos.
[c]impractical full-node patch count.
*2 threads per core.
**4 threads per core.

run configurations for 2-Level RMCRT:CPU. For KNL, best run configurations for 2-Level RMCRT:Portable allowed for speedups up to 2.0x and 2.6x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported best run configurations for 2-Level RMCRT:CPU. For Maxwell, best run configurations for 2-Level RMCRT:Portable allowed for speedups up to 1.6x and 1.8x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported best run configurations for 2-Level RMCRT:GPU.

These results suggest that performance is a by-product of refactoring for portability. This is encouraging as this single implementation of 2-Level RMCRT now supports execution with Kokkos::OpenMP and Kokkos::CUDA in addition to legacy execution. Best run configurations differing from previously supported best run configurations suggest that node utilization has been improved. This is attributed to new flexibility in run configuration that allows for cooperative use of compute resources (e.g., cores, caches, etc) and optimization of the balance between the number of loops executing at the same time and the number of resources available to each loop.

B. Strong-Scaling Studies

While single-node studies help understand how to efficiently use a node, it is also important to ensure that results apply at scale. To demonstrate MPI+Kokkos scalability, multi-node studies were performed on the Knights Landing portion of the NSF Stampede 2 system. This portion of Stampede 2 features the Intel Xeon Phi 7250 Knights Landing processor and offers a variety of memory and cluster mode configurations. These studies explored varying numbers and sizes of loops executing at the same time across nodes configured for *Cache-Quadrant* mode with a problem that fit in the 16 GB memory footprint of MCDRAM.

Figure 2 shows strong-scaling across Intel Knights Landing nodes for radiation modeling. These results were gathered using 2-Level RMCRT:Portable with Kokkos::OpenMP for a problem featuring 768^3 cells on the fine mesh (decomposed in 16^3 cells per patch) and 192^3 cells on the coarse mesh for three run configurations (1, 4, and 32 loop(s) executing at the same time with 256, 64, and 8 threads per loop, respectively, via 1 MPI process and 256 OpenMP threads).

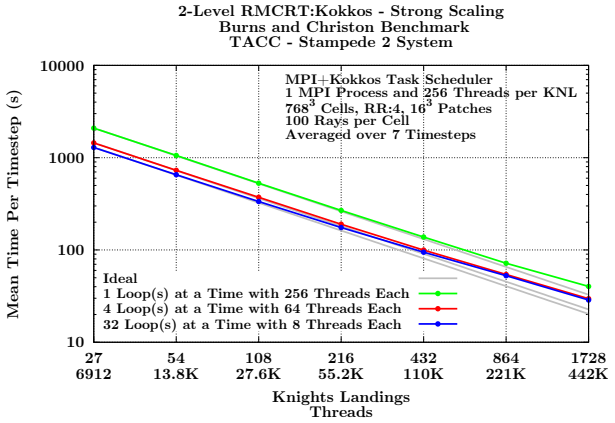

Fig. 2. 2-Level RMCRT:Portable strong-scaling to 1,728 KNL nodes.

These results show that as more, yet smaller, loops are executed at the same time, node-level performance increases at the expense of reductions in strong-scaling efficiency. This is attributed to thread scalability in individual loops. For 16^3 patches, compute resources are used more efficiently when individual loops use fewer threads, resulting in faster execution of loops as a whole. This leads to quicker breakdown of scalability as computation no longer hides communication sooner. Executing more, yet smaller, loops at the same time has allowed for speedups up to 1.6x and 1.4x to be achieved at 27 and 1,728 nodes, respectively, over execution of 1 loop at a time with 256 threads per loop.

Further, these results show that it is possible to achieve good strong-scaling to 442,368 threads across 1,728 Knights Landing processors using MPI+Kokkos. This is encouraging as it suggests a potential for reducing the number of per-node MPI processes by a factor of up to the number of cores/threads per node in comparison to an MPI-only approach. This is advantageous for many-core systems where the number of MPI processes required to utilize increasingly larger per-node core/thread counts becomes intractable.

XII. FORESEEABLE CHALLENGES

The approach presented here is a starting point for easing adoption of a performance portability layer in large legacy codebases. Foreseeable challenges include understanding how to: (1) use third party libraries using a performance portability layer in a codebase using a performance portability layer (e.g., using hypre in Uintah while using Kokkos in both hypre and Uintah), (2) manage increasing configurability (e.g., multiple tuneable run-time parameters across host and device), (3) make informed use of underlying programming models (e.g., using the device only when advantageous for a given loop), and (4) efficiently manage parallel execution and memory across multiple underlying programming models.

XIII. CONCLUSIONS AND FUTURE WORK

This work has helped improve Uintah's portability to current and future architectures and programming models while also

preserving support for pre-existing code. Specifically, it has shown an approach for indirectly adopting a performance portability layer to help improve legacy code support and long-term portability in a large legacy codebase. Kokkos capabilities have been shown when using this approach to make portable use of Kokkos::OpenMP and Kokkos::CUDA across multicore-, many-core-, and GPU-based nodes using a single implementation for the case studies examined. At the node-level, performance improvements up to 2.7x when refactoring for portability and 2.6x when more efficiently using a node have been achieved. At scale, good strong-scaling to 442,368 threads across 1,728 Knights Landing processors has been achieved using MPI+Kokkos.

These advancements have been made possible by the introduction of a framework-specific portability layer between Uintah's application code and Kokkos. This intermediate layer consists of three components: (1) loop-level support providing application developers with framework-specific abstractions (e.g., generic parallel loop statements) that map to interface-specific abstractions (e.g., PPL-specific parallel loop statements), (2) application-level support that includes a tagging system to identify which interfaces are supported by a given loop, and (3) build-level support that includes selective compilation of loops to allow for incremental refactoring and simultaneous use of multiple underlying programming models for heterogeneous HPC systems. This layer provides application developers with easy to use portable abstractions while allowing maintaining developers to easily add, remove, and tune interfaces to underlying programming models in a single location with fewer far-reaching changes across application code.

The portability and performance improvements shown here offer encouragement as we extend more of Uintah to heterogeneous HPC systems using Kokkos::OpenMP and Kokkos::CUDA. Next steps include furthering our understanding of Kokkos use across host and device simultaneously on heterogeneous IBM- and NVIDIA-based systems with multiple sockets and devices per node. As a part of this, emphasis will again be placed on long-term portability and managing simultaneous use of host and device in a portable manner with upcoming systems such as the Intel-based DOE Aurora and AMD-based DOE Frontier in mind.

Acknowledgments

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. An award of computing time was provided by the NSF Extreme Science and Engineering Discovery Environment (XSEDE) program. This research used resources of the Texas Advanced Computing Center, under Award Number(s) MCA08X004 - "Resilience and Scalability of the Uintah Software". This research also used resources donated to the University of Utah Intel Parallel Computing Center (IPCC) at the SCI Institute. Support for J. K. Holmen also comes from the Intel Parallel Computing Centers Program. Additionally, we would like to thank all of those involved with the CCMSC and Uintah past and present.

References

[1] (2019) June 2019 - TOP500 Supercomputer Sites. Https://www.top500.org/lists/2019/06/.

[2] (2019) Aurora. Https://aurora.alcf.anl.gov/.

[3] (2019) Frontier. Https://www.olcf.ornl.gov/frontier/.

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014.

[5] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, , and C. Wight, "Extending the uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. 101–122, 2016.

[6] A. Humphrey, T. Harman, M. Berzins, and P. Smith, "A scalable algorithm for radiative heat transfer using reverse monte carlo ray tracing," in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel and T. Ludwig, Eds. Springer International Publishing, 2015, vol. 9137, pp. 212–230.

[7] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the uintah DAG-based runtime system on PetaScale supercomputers," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 96:1–96:12.

[8] J. K. Holmen, B. Peterson, A. Humphrey, D. Sunderland, O. H. Diaz-Ibarra, J. N. Thornock, and M. Berzins, "Portably improving uintah's readiness for exascale systems through the use of kokkos," SCI Institute, Tech. Rep. UUSCI-2019-001, 2019.

[9] Z. Yang, D. Sahasrabudhe, A. Humphrey, and M. Berzins, "A preliminary port and evaluation of the uintah amt runtime on sunway taihulight," in *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2018)*. IEEE, May 2018.

[10] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins, "Radiative heat transfer calculation on 16384 gpus using a reverse monte carlo ray tracing approach with adaptive mesh refinement," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1222–1231.

[11] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108.

[12] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11.

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 66.

[14] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov 2013.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[16] W. Zhang, A. S. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An AMR software framework," *CoRR*, vol. abs/1604.03570, 2016.

[17] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "AMReX: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, p. 1370, May 2019.

[18] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, *The Cactus Framework and Toolkit: Design and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 197–227.

[19] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer, and T. Gamblin, "ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms," Sandia National Laboratories, Tech. Rep., 2015.

[20] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. OShea, E. Schnetter, B. V. Straalen, and K. Weide, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, 2014.

[21] P. J. Smith, R.Rawat, J. Spinti, S. Kumar, S. Borodai, and A. Violi, "Large eddy simulations of accidental fires using massively parallel computers," in *16th AIAA Computational Fluid Dynamics Conference*, 2003, p. 3697.

[22] D. S. Medina, A. St-Cyr, and T. Warburton, "Occa: A unified approach to multi-threading languages," *arXiv preprint arXiv:1403.0968*, 2014.

[23] R. D. Hornung and J. A. Keasler, "The raja portability layer: overview and status," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2014.

[24] R. Keryell, M. Rovatsou, and L. Howes. (2019) Khronos Group SYCL 1.2.1 Specification. Https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf.

[25] (2019) Kokkos: The c++ performance portability programming model wiki. Https://github.com/kokkos/kokkos/wiki.

[26] (2019) Tutorials for the kokkos c++ performance portability programming ecosystem. Https://github.com/kokkos/kokkos-tutorials.

[27] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[28] I. Demeshko, J. Watkins, I. K. Tezaur, O. Guba, W. F. Spotz, A. G. Salinger, R. P. Pawlowski, and M. A. Heroux, "Toward performance portability of the albany finite element analysis code using the kokkos library," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 332–352, 2019.

[29] E. Phipps and T. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, 2019.

[30] L. Bertagna, M. Deakin, O. Guba, D. Sunderland, A. M. Bradley, I. K. Tezaur, M. A. Taylor, and A. G. Salinger, "Hommexx 1.0: a performance-portable atmospheric dynamical core for the energy exascale earth system model," *Geoscientific Model Development*, vol. 12, no. 4, pp. 1423–1441, 2019.

[31] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995.

[32] M. A. Gallis, J. R. Torczynski, S. J. Plimpton, D. J. Rader, and T. Koehler, "Direct simulation monte carlo: The quest for speed," *AIP Conference Proceedings*, vol. 1628, no. 1, pp. 27–36, 2014.

[33] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via babelstream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018.

[34] P. Grete, F. W. Glines, and B. W. O'Shea, "K-athena: a performance portable structured grid finite volume magnetohydrodynamics code," *CoRR*, vol. abs/1905.04341, 2019.

[35] F. E. H. Prez, N. Mukhadiyev, X. Xu, A. Sow, B. J. Lee, R. Sankaran, and H. G. Im, "Direct numerical simulations of reacting flows with detailed chemistry using many-core/gpu acceleration," *Computers & Fluids*, vol. 173, pp. 73 – 79, 2018.

[36] J. Eichstdt, M. Green, M. Turner, J. Peir, and D. Moxey, "Accelerating high-order mesh optimisation with an architecture-independent programming model," *Computer Physics Communications*, vol. 229, pp. 36 – 53, 2018.

[37] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the performance portability of modern parallel programming models using tealeaf," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, p. e4117, 2017, e4117 cpe.4117.

[38] C. Trott. (2018) Apps Using Kokkos. Https://github.com/kokkos/kokkos/issues/1950.

[39] B. Peterson, N. Xiao, J. K. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. Humphrey, and M. Berzins, "Developing uintah's runtime system for forthcoming architectures," SCI Institute, Tech. Rep., 2015.

[40] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, "An overview of performance portability in the uintah runtime system through the use of kokkos," in *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2. Piscataway, NJ, USA: IEEE Press, 2016, pp. 44–47.

[41] J. K. Holmen, A. Humphrey, D. Sunderland, and M. Berzins, "Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, ser. PEARC17. New York, NY, USA: ACM, 2017, pp. 27:1–27:8.

[42] B. Peterson, A. Humphrey, J. K. Holmen, T. Harman, M. Berzins, D. Sunderland, and H. C. Edwards, "Demonstrating gpu code portability and scalability for radiative heat transfer computations," *Journal of Computational Science*, vol. 27, pp. 303 – 319, 2018.

[43] (2019) Kokkos c++ performance portability programming ecosystem: The programming model - parallel execution and memory abstraction. Https://github.com/kokkos/kokkos.

[44] J. R. Hammond and T. G. Mattson, "Evaluating data parallelism in c++ using the parallel research kernels," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL'19. New York, NY, USA: ACM, 2019, pp. 14:1–14:6.

[45] J. R. Hammond, M. Kinsner, and J. Brodman, "A comparative analysis of kokkos and sycl as heterogeneous, parallel programming models for c++ applications," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL'19. New York, NY, USA: ACM, 2019, pp. 15:1–15:2.

[46] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Mller, "Archer: Effectively spotting data races in large openmp applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 53–62.

[47] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71.

[48] (2019) Wrapper shell script for nvidia nvcc to better conform to host compiler command line arguments. Https://github.com/kokkos/nvcc_wrapper.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: A FRAMEWORK-SPECIFIC APPROACH FOR PERFORMANCE PORTABILITY LAYER ADOPTION IN LARGE LEGACY CODES

A. Abstract

This description contains information describing how to download, install, and run Uintah for the case studies in Section X and Section XI. This information includes example build scripts and run scripts.

B. Description

1) Check-list (artifact meta information):

- **Algorithm:** Char Oxidation, Reverse Monte-Carlo Ray Tracing Radiation.
- **Program:** Uintah.
- **Compilation:** C++11 compliant compiler.
- **Binary:** sus.
- **Data set:** Uintah-provided benchmark.
- **Hardware:** See Sections X-A and XI-B.
- **Output:** Script-generated output.
- **Experiment workflow:** See below.
- **Experiment customization:** See below.
- **Publicly available?:** Yes.

2) How software can be obtained: Uintah's latest support for Kokkos can be checked out using:

```
svn co https://gforge.sci.utah.edu/svn/uintah/branches/kokkos_dev
username: anonymous
password: anonymous
```

Source for this paper can be checked out using:

```
svn co https://gforge.sci.utah.edu/svn/uintah/branches/sc19
username: anonymous
password: anonymous
```

Source for Steps 0 through 6 in Table II can be found in:

```
/sc19/src-char-ox-step-0-6
```

Files to reproduce Steps 0 through 6 in Table II can be found in:

```
/sc19/src-char-ox-step-0-6/CCA/Components/Arches/ParticleModels/
```

Source for Tables III through VII and Step 7 in Table II can be found in:

```
/sc19/src-char-ox
```

Source for Table VIII and Figure 2 can be found in:

```
/sc19/src-rmcrt
```

Modified source for Kokkos can be obtained using the below:

```
git clone https://github.com/kokkos/kokkos.git src
cd src
git checkout 2.7.00
git apply /sc19/build-scripts/kokkos_brad_oct122018.patch
```

3) Hardware dependencies: Uintah runs on Linux-based platforms supporting recent C++11 compliant compilers.

4) Software dependencies: Uintah require use of a recent C++11 compliant compiler (e.g., ICC 18.0.1 and GCC 7.3.1 used for most results here). Uintah builds using Kokkos::CUDA require use of the Kokkos patch applied in Section A-B2 to add support for asynchronous execution of Kokkos parallel patterns. A list of general third party dependencies can be found in:

```
/sc19/doc/InstallationGuide/
```

5) Datasets: Inputs for Steps 0 through 6 in Table II include:

```
/sc19/tmp/char-ox-step-0-6-0016-16.ups
/sc19/tmp/char-ox-step-0-6-0016-32.ups
/sc19/tmp/char-ox-step-0-6-0016-64.ups
```

Inputs for Tables III through VII and Step 7 in Table II include:

```
/sc19/tmp/char-ox-0012-16.ups
/sc19/tmp/char-ox-0012-32.ups
/sc19/tmp/char-ox-0012-64.ups
/sc19/tmp/char-ox-0016-16.ups
/sc19/tmp/char-ox-0016-32.ups
/sc19/tmp/char-ox-0016-64.ups
/sc19/tmp/char-ox-0064-16.ups
/sc19/tmp/char-ox-0064-32.ups
/sc19/tmp/char-ox-0064-64.ups
```

Inputs for Table VIII include:

```
/sc19/tmp/rmcrt-128-16.ups
/sc19/tmp/rmcrt-128-32.ups
/sc19/tmp/rmcrt-128-64.ups
```

Inputs for Figure 2 include:

```
/sc19/tmp/rmcrt-stampede-2.ups
```

C. Installation

Example scripts for building Uintah can be found in:

```
/sc19/build-scripts/
```

General Uintah installation instructions including third party library installation can be found in:

```
/sc19/doc/InstallationGuide/
```

General instructions for using Uintah on major HPC systems can be found at:

```
http://uintah-build.sci.utah.edu/trac/wiki/MachineSpecificInfo
```

D. Experiment workflow

Below is an example command for running *sus*:

```
mpirun -np <#_of_processes> ./sus <run_configuration_parameters> <input.ups> | tee out.txt
```

Run configuration parameters are discussed in Section A-F.

Example scripts for generating results in Section X and XI can be found in:

```
/sc19/tmp/
```

After running *sus*, output saved to text files can be parsed to compute per-loop and per-timestep timings using:

```
/sc19/tmp/extractScalingData
```

Below is an example of using *extractScalingData*:

```
$ mpirun -np 1 ./sus -npartitions 16 -nthreadsperpartition 1 input-1.ups | tee out-1.txt
$ mpirun -np 1 ./sus -npartitions 16 -nthreadsperpartition 2 input-2.ups | tee out-2.txt
...
$ ./extractScalingData.sh out-1.txt out-2.txt ...
```

E. Evaluation and expected result

Running *extractScalingData* saves output to *avgComponentTimes* and *scalingData*. Per-loop timings in Tables II through V correspond to *avgLoop* timings reported in *avgComponentTimes* as milliseconds. Per-loop timings in Tables VI and VII correspond to average kernel times reported by nvprof. Per-timestep timings in Table VIII and Figure 2 correspond to *avgMean* timings reported in *scalingData* as seconds.

F. Experiment customization

Results in Sections X and XI experiment with varying amounts of per-loop work and different run configurations.

Per-loop work is managed by the *resolution* and *patches* fields in the *Level* block of an input file.

Below is an example of constructing a 64^3 domain subdivided into 64 patches with 16^3 cells each:

```
<resolution>[64,64,64]</resolution>
<patches>[4,4,4]</patches>
```

Run configuration is managed by run-time parameters specified between the executable and input file, e.g.,:

```
mpirun -np <#_of_processes> ./sus <run_configuration_parameters> <input.ups> | tee out.txt
```

Uintah's latest parameters can be listed using the below:

```
./sus --help
```

Parameters for non-Kokkos builds include:

```
-nthreads <#> : Number of threads per MPI process
```

Parameters for Kokkos::OpenMP builds include:

```
-npartitions <#>          : Number of loops executing at the same time
-nthreadsperpartition <#> : Number of threads per loop
```

Parameters for Kokkos::CUDA builds include:

```
-nthreads <#>              : Number of threads per MPI process
-cuda_threads_per_sm <#>   : Number of threads per streaming multiprocessor (SM)
-cuda_sms_per_loop <#>     : Number of streaming multiprocessors (SMs) per loop
-cuda_streams_per_task <#> : Number of CUDA streams per task
```

Per-loop work and run configuration can be verified in the topmost output generated by *sus*, e.g.,:

```
$ Parallel: 1 MPI process
$ Parallel: 64 OMP thread partitions per MPI process
$ Parallel: 1 OMP thread per partition
...
$ Patch layout:           (4,4,4)
$ Grid statistics:
$ Number of levels:               1
$ Level 0:
$   Periodic boundaries:          [int 1, 1, 1]
$   Number of patches:            64
$   Total number of cells:        262144 (4096 avg. per patch)
$ Total patches in grid:          64
$ Total cells in grid:            262144 (4096 avg. per patch)
```

G. Notes

Uintah is in the process of moving to GitHub. An equivalently name branch will be maintained there.