Demonstrating GPU Code Portability and Scalability for Radiative Heat Transfer Computations

Brad Peterson, Alan Humphrey, John Holmen
Todd Harman, Martin Berzins

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

Dan Sunderland

Sandia National Laboratories
PO Box 5800 / MS 1418
Albuquerque, NM 87175 USA

H. Carter Edwards

NVIDIA
2788 San Tomas Expressway
Santa Clara, CA 95051 USA

Abstract

High performance computing frameworks utilizing CPUs, Nvidia GPUs, and/or Intel Xeon Phis necessitate portable and scalable solutions for application developers. Nvidia GPUs in particular present numerous portability challenges with a different programming model, additional memory hierarchies, and partitioned execution units among streaming multiprocessors. This work presents modifications to the Uintah asynchronous many-task runtime and the Kokkos portability library to enable one single codebase for complex multiphysics applications to run across different architectures. Scalability and performance results are shown on multiple architectures for a globally coupled radiation heat transfer simulation, ranging from a single node to 16384 Titan compute nodes.

Key words: Asynchronous many-task runtime; GPU; Scalability; Portability; Radiative Heat Transfer

1 Introduction

The need to solve larger and more complex simulation problems while at the same time not incurring additional power costs has led to an increasing focus on GPU and Intel Xeon Phi-based architectures. Many current and future high performance computing (HPC) systems rely on such architectures. In the case of the DOE Titan system, with a theoretical peak performance of 27 petaflops, over 90% of the computational power come from its 18,688 GPUs. These heterogeneous systems pose significant challenges in terms of programmability due to deep memory hierarchies, vendor-specific language extensions and memory constraints, e.g. less device-side memory compared to host memory per node. This paper focuses on scalability, portability, and programmability of multiphysics applications. This work covers 1.) scalability improvements necessary to compute a radiation transport problem on 16,384 GPUs on Titan using the Uintah asynchronous many-task runtime, and 2.) portability improvements necessary to utilize a single codebase capable of execution on nodes containing CPUs, GPUs, and/or Intel Xeon Phi processors. Nvidia GPUs receive particular emphasis as they introduce four challenges distinct from the CPU execution model: 1.) task asynchrony, 2.) multiple memory spaces, 3.) an additional programming model (e.g. CUDA), and 4.) another level of parallelism through partitioned execution units among streaming multiprocessors.

Uintah's emphasis on scalability across a diverse set of HPC architectures is currently driven by the target problem of the University of Utah Carbon Capture Multidisciplinary Simulation Center (CCMSC), funded by the NNSA Predictive Science Academic Alliance Program (PSAAP) II. The CCMSC aims to simulate, using petascale/exascale computing, a 1000MWe oxy-fired clean coal boiler being developed by Alstom Power to deliver high efficiency electric power generation with carbon capture. A primary CCMSC focus is on using extreme-scale computing for reacting, large eddy simulation (LES)-based codes within the Uintah open source framework, using machines like Titan and the upcoming Summit system in a scalable manner. The physical size of the CCMSC target boiler simulations and the resolution required to resolve the dominant physical processes necessitates the use of systems like DOE Titan at near-capacity.

Radiation is the dominant mode of heat transfer in these boiler simulations. A principal challenge in modeling radiative heat transfer is the nonlocal nature of it. Thermal energy propagates across the entire computational domain from any point in space. Our radiation model, a reverse Monte Carlo ray tracing (RMCRT) technique [1], described further in Section 3 requires an all-to-all communication to replicate the radiative properties and boiler geometry on each node to facilitate local ray tracing. This challenge is addressed by leveraging Uintah's adaptive mesh refinement (AMR) capabilities, using Cartesian mesh patches to generate a fine mesh that is only used locally (close to each grid point) and a successively coarser mesh is used further away, via a level-upon-level approach. This approach

is fundamental to the CCMSC target problem, where the entire computational domain needs to be resolved to adequately model the radiative heat flux. Using this approach enabled excellent strong scaling to over 256K CPU cores on the DOE Titan system for problem sizes that were previously intractable with a single fine mesh (single-level) RMCRT approach due to on-node memory constraints [1]. This scaling was consistent with the communication and computation model [1].

The challenges in moving from a CPU to a GPU-based multi-level RMCRT algorithm using this mesh refinement approach have extended well beyond what a typical GPU port of a CPU codebase might entail. A core Uintah design focuses on insulating the application developer from the underlying runtime, which requires more automation of runtime features. Uintah's runtime requires that all host-to-device and device-to-host data copies for computational task dependencies (inputs and outputs), as well as device context management must be handled automatically in the same way MPI messages are generated by the Uintah runtime system [2–4]. Meeting these challenges required numerous runtime changes to support the RMCRT problem on Titan's GPUs.

This paper is an extended form of the workshop paper [5], which addressed scalability and runtime improvements necessary to run this difficult globally-coupled, all-to-all problem to 16,384 GPUs on the DOE Titan system. This extended paper addresses the portability challenges of implementing RMCRT into one single portable codebase using the Kokkos portability library and executing this code on CPUs, GPUs, and Intel Xeon Phi Knights Landing (KNL) architectures.

Prior to this work, Uintah's use of Kokkos has been limited to support for CPU and Xeon Phi processors [6]. This work extends portability support to the GPU. Special focus is given to GPU portability enabling Kokkos to now efficiently execute Uintah's fine-grained tasks on GPUs. In particular, modifications are made to Kokkos itself to enable GPU asynchronous and performant execution of parallel work loops with fewer iterations (i.e. an iteration range the low hundreds). This work also describes modifications for GPU portability that affected Xeon Phi performance and describes how it was addressed to enable one portable codebase across three architectures. Intel Xeon Phi portable performance has been addressed in prior work [6], and does not need to be extended here.

The contributions from the original paper [5] are:

(i) Leveraging Uintah's AMR infrastructure in a novel way to reduce the volume of communication sufficiently so as to allow scalability. Uintah's AMR capabilities are introduced in Section 2, along with an overview of Uintah.

(ii) Changing the way that AMR meshes are stored on the GPU to overcome the limited available GPU global memory. This has entailed a significant extension of the Uintah `GPU DataWarehouse` system [7] to support a mesh-level database,

a repository for shared, per-mesh-level variables such as global radiative properties. This has allowed multiple mesh patches, each with associated GPU tasks, to run concurrently on the GPU while sharing coarse, radiation mesh data. This extension of the `GPU DataWarehouse` is discussed in Section 3, which also gives an overview on radiation transport and describes a GPU-based multi-level RMCRT model.

(iii) The introduction of novel non-blocking, thread-scalable data structures for managing asynchronous MPI communication requests, replacing previously problematic mutex-protected vectors of MPI communication records. To be non-blocking a wait, failure, or resource allocation by one thread cannot block progress on any other thread. Non-blocking data-structures are lock-free if at all steps at least one thread is guaranteed to make progress, and are wait-free if at any step all threads are guaranteed to make progress [8]. Section 4 describes these changes and their motivation, and also shows speedups in local MPI communication times made possible through these infrastructure improvements.

(iv) A vastly improved memory allocation strategy to reduce heap fragmentation is covered in Section 4. This strategy allows running simulations at the edge of the nodal memory footprint on machines like Titan.

(v) Determining optimal fine mesh patch sizes to yield GPU performance while maintaining over-decomposition of the computational domain to hide latency. This is covered in Section 5 with strong scaling results over a wide range of GPU counts up to 16,384 GPUs, and also show the results of differing over-decomposition configurations across this range of GPUs.

The four major extensions to this paper from the prior paper [5] are:

(vi) Using and modifying Kokkos to improve performance portability on GPUs. Kokkos's current GPU execution model is bulk synchronous, where a parallel loop is partitioned into many CUDA blocks and the GPU distributes those blocks throughout the GPU device. However, the Uintah asynchronous many-task runtime is designed to asynchronously execute many overlapping finer-grained tasks, many of which require only one CUDA block each. Section 6 describes modifications to Kokkos's GPU execution model so that it is no longer bulk synchronous and can instead overlap many smaller asynchronous execution units.

(vii) Section 7 reviews prior Intel Xeon Phi performance portability work [6] for Uintah and describes portability challenges relating to architecture specific iteration patterns.

(viii) The integration of GPU portability into Kokkos and Uintah is given in Section 8. The challenges here include using Uintah's own memory management sys-

tem within Kokkos, using Kokkos's portable random number library, and supplying task execution parameters for many architectures.

(ix) Results of running portable code on multiple architectures is given in Section 9. In particular, the results compare three codebases: 1.) prior CPU code, 2.) prior GPU code, and 3.) Kokkos-enabled code. Portability is demonstrated on CPUs, GPUs, and Intel Xeon Phi KNLs. GPU portability is shown before and after Kokkos modifications from Section 6.

An overview of related work is given in Section 10, and the paper concludes in Section 11 with future work in this area.

2 The Uintah Code

The Uintah asynchronous many-task (AMT) runtime [2, 9] is open-source (MIT License) software that has been widely ported and used for many different types of problems involving fluids, solids, and fluid-structure interaction problems [9], with the latest release in September 2017 [10]. Uintah consists of a set of parallel software components and libraries that facilitate the solution of partial differential equations on structured AMR grids. Uintah presently contains four main simulation components: 1.) the multi-material ICE [11] code for compressible flows; 2.) the particle-based code MPM [12] for structural mechanics; 3.) the combined fluid-structure interaction (FSI) algorithm MPM-ICE [13], and 4.) the ARCHES turbulent reacting CFD component [14] that was designed for simulating turbulent reacting flows with participating media radiation. Uintah is highly scalable [15], and solves a broad class of PDE problems on many National Science Foundation (NSF), Department of Energy (DOE) and Department of Defense (DOD) parallel computers.

Uintah has a unique set of methods and uses a directed acyclic graph (DAG) approach as part of a production-strength code in a way that is coupled to a runtime system. Uintah's design maintains a clear partition between applications code and its runtime system, making it possible to achieve great increases in scalability through changes to the runtime system *without changes to the applications themselves*. An application developer creates tasks by indicating all needed simulation variables for the task's code, and then later writes task code using C++ or CUDA. Within that task code the application developer requests all simulation variables from Uintah's data stores known as a `DataWarehouse`, which will have all simulation variables ready in the correct memory space, including all needed halo data. The tasks themselves are assigned to a particular `patch`, which is a cuboid region of cells on a structured grid. Uintah's runtime is responsible for the proper data preparation, scheduling, and execution of these tasks.

Particular advances made in Uintah include highly scalable AMR using Cartesian mesh patches [15]. A key factor in improving performance has been the reduction in MPI wait time through the dynamic and even out-of-order execution of task-graphs [16]. The need to reduce memory use in Uintah has led to the adoption of a nodal shared memory model in which there is only one MPI process per multi-core node, and tasks are executed serially on individual cores using Pthreads [17]. As a result, Uintah has demonstrated scalability to 768K cores on complex fluid-structure interactions with AMR. Uintah's thread-based runtime system [17] uses decentralized execution of the task-graph, implemented by each CPU core requesting work itself and performing its own MPI. A lock-free shared memory abstraction through Uintah's `DataWarehouse` approach [17] was implemented using atomic operations, allowing efficient access by all cores to the shared data on a node. Finally, the nodal architecture of Uintah has been extended to run tasks on one or more on-node accelerators [18] by using a multi-stage queue architecture to organize work for CPU cores and GPUs in a dynamic way, and is the starting point for this paper.

2.1 The ARCHES Combustion Simulation Component

ARCHES is the primary CCMSC simulation component within Uintah and was designed for the simulation of turbulent reacting flows with participating media radiation. It is a three-dimensional, Large Eddy Simulation (LES) code [19], which solves the coupled mass, momentum, and energy conservation equations on a staggered finite-volume mesh for the gas and solid phase with combustion [14, 20]. It uses a low-Mach number ($M < 0.3$), variable density formulation to model heat, mass, and momentum transport in reacting flows.

The discretized equations are integrated in time using an explicit, strong-stability preserving second or third-order Runge-Kutta method [21]. Spatial discretization is handled with central differencing where appropriate for energy conservation or flux limiters (eg, scalar mixture fractions) to maintain numerical accuracy. The low-mach, pressure projection formulation requires a solution of sparse linear system at each timestep using the *Hypre* linear solver package [22]. The turbulent subgrid velocity and species fluctuations [23] are modeled with the dynamic Smagorinsky closure model. The Discrete Ordinates Method for solving the radiation heat transfer equation uses over a discrete set of ordinates and, like the pressure equation, is formulated as a linear system that is solved using *Hypre*. Research using ARCHES has been done on radiative heat transfer using the parallel discrete ordinates method [24] (DOM, a modeling method developed at Los Alamos National Laboratory for neutron transport) and the P1 approximation to the radiative transport equation [25]. Work done by Sun [26] and Hunsaker [27] has shown that Monte Carlo ray tracing methods are potentially more efficient and offer an alternative to DOM.

3 RMCRT Model

Scalable radiation modeling plays a key computational role in applications such as heat transfer in combustion simulations [19], neutron transport modeling [28] in nuclear reactors, and astrophysics modeling. Generally, radiation modeling is considered one of the most challenging problems in large-scale computational science and engineering due to the global nature of radiation. For heat transfer problems such as the CCMSC boiler simulations, coupling combustion and radiation poses several numerical challenges. The fluid mechanics of combustion are an inherently local phenomena, wherein conservation laws may be applied over a finite volume. Radiation however, is a long-distance phenomenon due to strong nonlocal effects. Because of these nonlocal effects, conservation laws cannot be applied over an infinitesimal volume, but must be applied over the entire computational domain. This global interdependency creates difficulties for domain decomposition due to the need for nonlocal data.

3.1 Radiation Transport Models

A critical quantity of interest for all boiler simulations is the heat flux to the surrounding walls, as the major mode of heat transfer in the coal-fired boiler is radiation. In the context of the CCMSC, the design of new boiler facilities utilizing ultra super critical air-combustion technology will require accurate radiative heat flux estimates in environments with increased CO_2 concentrations, higher temperatures and different radiative properties for new metal alloys. Thermal radiation in the target boiler simulations is loosely coupled to the computational fluid dynamics (CFD) due to time-scale separation. This section briefly describes three other common approaches for solving radiation heat transfer, namely the Discrete Ordinances Method (DOM) [24], spatial transport sweeps [29], and forward Monte Carlo ray tracing (MCRT) [26, 27], and then describes the reverse Monte Carlo ray tracing (RMCRT) model in more detail. All of these approaches to radiative heat transfer are currently implemented and available within Uintah.

ARCHES is designed to solve the mass, momentum, mixture fraction, and thermal energy governing equations inherent to coupled turbulent reacting flows. ARCHES has relied primarily on a DOM solver [24] to compute the radiative source term in the energy equation shown by:

$$c_v \frac{dT}{dt} = -\nabla \cdot (\kappa \nabla T) - p\nabla \cdot v + \Phi + Q''' - \nabla \cdot q_r \tag{1}$$

where c_v is the specific heat, T is the temperature field, p is the pressure, κ is the thermal conductivity, v is the velocity vector, Φ is the dissipation function,

Q''' is the heat generated within the medium, e.g. chemical reaction, and $\nabla \cdot q_r$ is the net radiative source [1]. A radiatively participating medium can emit, absorb and scatter thermal radiation. The energy equation is then conventionally solved by ARCHES (finite volume) and the temperature field, T is used to compute the net radiative source term. This net radiative source term is then fed back into the energy equation (for the ongoing CFD calculation) which is solved to update the temperature field, T [1].

A particular limitation of DOM is false scattering. This is a due to spatial discretization error, similar to numerical diffusion in CFD calculations. A ray that is traced through the enclosure by DOM will gradually widen as it moves farther away from its point of origin. False scattering can be addressed by using a finer mesh of control volumes, but at greater computational cost [30].

Temporal sweeps [29] utilizes a four point stencil and a matrix back substitution computation. The algorithm at its fundamental level is serial, where computations directly affect the values in subsequent iterations of its loops. The resulting algorithm is computationally lightweight, and some parallelization is possible if independent ordinate directions and independent spectral frequencies are used. An implementation of temporal sweeps into Uintah [31] using CPU cores demonstrated fast wall time execution but is currently limited to scaling up to 128K cores due to its high memory footprint.

Recent work has shown that Monte Carlo ray tracing (MCRT) methods are potentially more efficient [26, 27]. Traditional forward MCRT approaches are inefficient though, in that large numbers of traced rays may not reach the subdomain of interest. Both DOM and MCRT methods aim to approximate the radiative transfer equation (2), the equation describing the interaction of absorption, emission and scattering for radiative heat transfer, which is an integro-differential equation with three spatial variables and two angles that determine the direction of $\hat{s}$ [32]. For MCRT methods, a statistically significant number of rays (photon bundles) are traced from a computational cell to the point of extinction, that is, until their radiative intensity falls below a specified threshold. This method is then able to calculate energy gains and losses for every element in the computational domain.

Reverse Monte Carlo ray tracing (RMCRT), the focus of this work, is an emission-based reciprocity method, where rays are traced backwards from the detector, thus eliminating the need to track ray bundles that never reach the detector [33]. Rather than integrating the energy lost as a ray traverses the domain as in forward MCRT approaches, RMCRT integrates the incoming intensity absorbed at the origin, where the ray was emitted. RMCRT is more amenable to domain decomposition, and thus Uintah's parallelization scheme due to the backward nature of the process [26] and the mutual exclusivity of the rays themselves. The process is considered reverse through the Helmholtz Reciprocity Principle, e.g. incoming and outgoing intensity can be considered as reversals of each other [34].

$$\begin{aligned}\frac{dI(\hat{s})}{ds} &= \hat{s}\nabla I(\hat{s}) \\ &= k_\eta I - \beta I(\hat{s}) \\ &+ \frac{\sigma_s}{4\pi}\int_{4\pi} I(\hat{s})\Phi(\hat{s_i},\hat{s})d\Omega_i,\end{aligned} \tag{2}$$

In equation 2, k_η is the absorption coefficient, σ_s is the scattering coefficient, dependent on the incoming direction s. β is the extinction coefficient that describes total loss in radiative intensity, I is the change in intensity of incoming radiation from point s to point $s + ds$ and is determined by summing the contributions from emission, absorption and scattering from direction $\hat{s}$ and scattering into the same direction $\hat{s}$. $\Phi(\hat{s_i},\hat{s})$ is the phase function that describes the probability that a ray coming from direction s_i will scatter into direction $\hat{s}$ and integration is performed over the entire solid angle Ω_i [32, 33]. The currently implemented RMCRT algorithm uses a mean absorption coefficient approximation σ_s and hence not resolving spectral frequencies, e.g. η for wavelength. Adding spectral frequencies to RMCRT would entail adding a loop over wavelengths, η and is part of future work.

3.2 RMCRT and Ray Tracing Overview

The principal motivation for the development of a GPU-based RMCRT radiation calculation arises from the computational intensity of the radiation solve in the CCMSC production runs, which consumes as much as 50% of the overall CPU time per timestep when using DOM. Additionally this work is motivated by access to large-scale GPU-based machines like DOE Titan, where over 90% of the available FLOPS are on the GPUs. Many of the CCMSC target simulations will run on Titan over its life span. Beyond this, utilization of the upcoming DOE Summit system is planned.

RMCRT uses rays more efficiently than forward MCRT, but it is still an *all-to-all* method, for which all of the geometric information and radiative properties for the entire computational domain must be accessible by every ray [26]. These radiative properties consist of; κ, the absorption coefficient, a property of the medium the ray is traveling through, σT^4, a physical constant $\sigma\cdot$ temperature field, T^4 and, $cellType$ (boundary or flow cell), a property of each computational cell in the domain. For RMCRT, the boiler geometry is replicated on each node and ray tracing takes place without the need to pass ray information across nodal boundaries (via MPI) as rays traverse the computational domain. The RMCRT approach is afforded the choice of replication due to the relative simplicity of the boiler geometry.

In order to address these communication challenges, a multi-level AMR approach was developed for both CPU [1] and now GPU, in which a fine mesh is only used

close to each grid point and a successively coarser mesh is used further away, significantly reducing MPI message volume and nodal memory footprint. This algorithm allows for the radiation computation to be performed with an appropriate mesh resolution while still being coupled with other physics components. The LES CFD, particle transport and particle reactions are solved on a different mesh resolution appropriate to their physics and models. This balanced approach to coupling multiphysics is made possible by Uintah's AMR design. This design significantly reduces the amount of data stored on every computational patch, and significantly reduces computational overhead for successively finer computation.

3.3 Multi-Level GPU Implementation

Following the original proof-of-concept GPU task scheduler [18], a single-level CPU and GPU RMCRT approach was initially considered. This approach was to begin comparisons against the current DOM solver within the Uintah ARCHES component, using the benchmark problem described by Burns and Christen [35]. Accuracy studies of this single-level RMCRT approach exist [27] for this benchmark, which examines the accuracy of the computed divergence of the heat flux and shows expected Monte Carlo convergence when compared to the published data [35]. In this approach, the quantity of interest, the divergence of the heat flux, ∇q is calculated for every cell in the computational domain. The entire domain was replicated on every node (with all-to-all communication) for the radiative properties. This replication occurred on the single fine mesh, which for N_{total} mesh cells, the amount of data communicated is $\mathcal{O}(N_{total}^2)$.

Though this single, fine mesh approach was highly accurate and effective at lower core and GPU counts, problem sizes beyond 256^3 were intractable for highly resolved domains, especially on machines with less than 2GB of memory per core. GPU scalability results were demonstrated up to 64 GPUs [18] to achieve basic accelerator task scheduling and execution. Using a problem size of 128^3, the volume of communication coupled with the PCIe transfers begins to dominate, and the GPUs were starved for work with only a single patch per GPU. These difficulties led to the use of an AMR approach that uses a mesh hierarchy to limit the amount of communication on CPU architectures [1].

Figure 1 best illustrates this approach, depicting how a ray from a fine-level patch (right) might be traced across a coarsened domain (left). In general, the data required by the multi-level RMCRT algorithm from the fine CFD mesh, is projected to all coarse levels subject to a user-defined refinement ratio (typically 2 or 4), where each coarse level spans the entire domain. The general multi-level RMCRT ray marching process is described in detail in prior work [1], which includes a precise model of communication and computation.

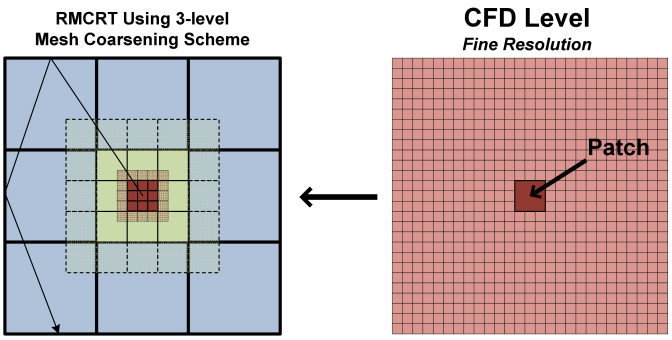

Fig. 1. RMCRT - 2D diagram of three-level mesh refinement scheme.

A significant challenge in moving to a GPU-based, multi-level RMCRT algorithm is the limited amount of global memory available on the current generation of GPUs found on Titan. These Nvidia K20X models have 6GB compared to 32GB CPU host-side. Previously, Uintah automatically generates MPI messages and kept multiple versions of simulation variables for out-of-order scheduling and execution [16]. This allows different tasks to require the same variable on the same neighboring patch multiple times for differing halo cell requirements. In the context of the multi-level RMCRT radiation model, this is a global halo requirement on all coarse levels. Having multiple versions of simulation variables with global halos presents problems for a limited memory footprint as on Titan's K20X GPUs.

A solution to this problem has been to short-circuit the creation of these redundant global copies of the radiative properties on the host and their subsequent transfer across the PCIe bus to the GPU. This has been achieved by a significant extension of the Uintah `GPU DataWarehouse` system [7] to support a level database that stores a single copy of shared global radiative properties (per-mesh level based on Uintah's level-upon-level approach to AMR). This solution has minimized PCIe transfers and allowed multiple mesh patches, each with GPU tasks, to run concurrently on the GPU while sharing data from the coarse radiation mesh. This design leverages the two copy engines available on the K20X GPUs and also makes use of support for running multiple, concurrent kernels. Using these features, Uintah can copy data for multiple fine-mesh patches to the GPU, each sharing a global copy of the coarsened radiative properties.

Data for these GPU tasks can be simultaneously copied to-and-from the device as multiple RMCRT kernels run simultaneously. `CUDA Streams`, managed by the Uintah infrastructure provide additional concurrency, as operations from different streams can be interleaved.

3.4 Complexity Analysis

Previous work [1] provided a complexity analysis for CPU-based single-level and multi-level RMCRT implementations. The GPU-based RMCRT analysis adds two additional terms for data copy cost between host memory and GPU memory. The single-level RMCRT cost for a single patch is given by

$$T_{rmcrt}^{global} = (t_{h2d} + t_{d2h})n_{mesh}^{3} + C^{*}n_{rays}{n_{mesh}^{3}}^{4/3}, \tag{3}$$

where t_{d2h} and t_{h2d} are the costs to copy one cell host-to-device or device-to-host, n_{mesh}^{3} is the total number of mesh cells, C^{*} is a constant, and n_{rays} is the number of rays emitted per cell. If all tasks in a timestep compute on GPUs in a single-node implementation, there is no need to copy data back into host memory, and thus the t_{h2d} and t_{d2h} terms are zero. Complexity costs for a fine and coarse mesh level implementation, including communication costs, are also given in previous work [1].

4 Uintah Runtime Improvements

Uintah uses an "MPI + *X*" parallelism approach, using a combination of MPI + (Pthreads + Nvidia CUDA). This mixed concurrency model has the potential for problematic race conditions and deadlock scenarios, some of which only manifest at larger scale in our experience. Significant infrastructure changes were necessary to improve nodal throughput and to expose more concurrency while maintaining correctness within this complex environment. In particular, choosing optimal data structures and algorithms for management of all MPI communications was required to efficiently expose concurrency, as well as to maintain critical sections around legacy serial data structures. Furthermore, it was vital for Uintah to manage limited resources such as nodal memory through the use of custom allocators that allow frameworks like Uintah to choose more optimal allocation policies for different objects to better utilize available resources and improve nodal throughput.

4.1 Multi-Threaded Processing of Asynchronous MPI

Uintah currently utilizes MPI_THREAD_MULTIPLE, which allows individual threads to perform their own MPI sends and receives, as well as collectives. In our experience, MPI_THREAD_MULTIPLE is rarely adopted by MPI users. Additionally, Uintah previously utilized `MPI_Testsome()` to process groups of `MPI_Request` objects. Initial attempts to run at large scale with accelerators in

this environment exposed a subtle race condition which created memory leaks due to unused MPI buffers.

Uintah uses a process where any arbitrary thread can post MPI receives for any task prior to execution. The `MPI_request` objects associated with these receives are then placed into a collection. Later during a timestep, any arbitrary thread may check any MPI receives in this collection for completion, as MPI receives are not necessarily bound to the thread that posted the `MPI_request`. Despite attempts to ensure concurrency on accessing these `MPI_Testsome()` groups, we observed multiple threads allocating buffers for the same MPI message, but only one actually processed the message and invoked a callback through Uintah to deallocate its buffer. Other threads may have allocated buffers which were never released, resulting in a severe memory leak in the Uintah infrastructure, causing the application to quickly fail at large-scale due to *out of memory* errors on the compute nodes. Though this scenario was present in other simulations, it was only evident at large scale, and only significant within the RMCRT radiation model due to the high volume and size of MPI messages. Despite this, the approach to multi-threaded processing of asynchronous MPI within Uintah had worked seemingly well for all cases until now.

A more coarse-grained critical section surrounding the code processing MPI messages was not feasible as it would have serialized a substantial portion of the algorithm. The solution ultimately required a fundamental redesign in the data structure and algorithm used to manage MPI communication records in a multi-threaded environment. The new algorithm leverages a novel wait-free pool, which is thread-scalable and contention-free, to store individual MPI requests. The wait-free pool iterator is implemented as a unique, move-only object which toggles an atomic flag to protect access to the referenced value to prevent data races, i.e. multiple threads modifying the same value.

`C++11` features were used (*atomics, move constructor, move assignment, and disabling copy construction and copy assignment*) to implement a unique protected iterator, that guarantees no two threads can have iterators which dereference to the same object. Once an iterator claims an atomic flag, that iterator can then read, modify, or erase that referenced item, and will release the atomic flag when the iterator is destroyed or advanced to another open item. `MPI_Test()` is then used on each request individually in contrast to the prior design which used `MPI_Testsome()` to test a collection of requests. This solution, outlined in Algorithm 1 results in much simpler code with fewer allocations, and eliminates the complexity of managing the previously used locked vectors of `MPI_Request` objects and their related critical sections.

Algorithm 1 Wait-free MPI_Request pool

```
1: RecvCommList& recv_list = m_recv_lists[id];
2: auto ready_request =
3:  [](CommNode const& n)->bool{return n.test();};
4: iterator = recv_list.find_any(ready_request);
5: if (iterator){
6:  MPI_Status status;
7:  iterator->finishCommunication(m_comm, status);
8:  recv_list.erease(iterator);
9: }
```

4.2 Memory Allocation and Management Strategy

After identifying and addressing the race condition described above in Section 4.1, the RMCRT benchmark problem [35] still failed at scale due to memory-related issues, though it ran longer before failure. Further investigation revealed that extreme heap fragmentation was occurring when running the RMCRT benchmark problem. Persistent small allocations mixed with transient large allocations fragmented the heap such that it grew continually, acting as though a significant memory leak still existed. Using Google's tcmalloc [36], a highly scalable memory allocator for multi-threaded applications, reduced heap fragmentation but the mixture of persistent and transient allocations still resulted in unacceptable fragmentation. Furthermore, frequent small allocations from multiple threads also caused a performance degradation due to contention of shared resources. The performance of the infrequent large allocations was not a factor in the overall performance.

4.2.1 Custom Allocators to Reduce Fragmentation

Developing and using custom allocator classes for Uintah's MPI buffers and `GridVariables` (simulation variables that reside on Uintah's Cartesian mesh patches at cell centers, nodes or faces x,y,z), allowed us to leverage our knowledge of how a data structure would be used to distinguish between large/small and transient/persistent allocations. This greatly improved memory utilization and reduced fragmentation.

To eliminate the observed heap fragmentation, allocators were developed to address the range of allocation sizes causing the fragmentation. For large allocations, the heap was completely avoided by implementing a specialized allocator that uses `mmap` to allocate anonymous virtual memory. Though `mmap` is a system call and can be slower than a standard `malloc`, it was more important to avoid fragmenting the heap than to optimize the throughput of large allocations. Throughput was not a concern for the performance of large allocations, but it is critical for frequent small allocations. To manage our small transient objects, i.e. objects that are frequently

created and destroyed, a lock-free memory pool was developed on top of our `mmap` allocator to avoid the heap and maximize throughput. All other infrequent allocations are still managed using the heap.

Using these techniques, portions of Uintah infrastructure code related to communication were significantly simplified, and nodal throughput was improved by a factor of 2-4X in processing local MPI communication (the time spent posting MPI messages for individual threads).

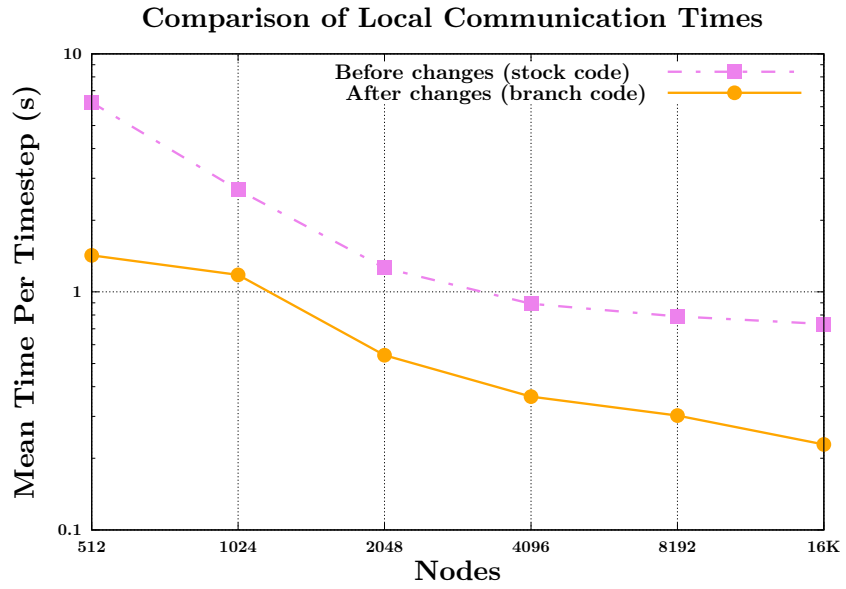

Fig. 2. Comparison of the local communication time (sec) before and after infrastructure improvements.

Figure 2 shows the time spent doing local communication, before and after our infrastructure improvements for our CPU implementation of the Burns and Christen [35] RMCRT benchmark on Titan. These runs were from 512 to 16,384 nodes, with a 2-level problem with 136.31M cells, 512^3 on the fine CFD mesh and 128^3 on the coarse radiation mesh. There were 262k total mesh patches in this problem. The speedups of communication times ranged from 2.27x at 1024 nodes to 4.40x at 512 nodes.

5 Scaling Studies with CPU and GPU Codebases

This section demonstrates strong scalability results on the DOE Titan XK7 [1] system for the Burns and Christen [35] benchmark problem using the GPU imple-

[1] Titan's nodes host a 16-core AMD Opteron 6274 processor and 1 Nvidia Tesla K20x GPU. The entire machine offers 299,008 CPU cores and 18,688 GPUs and over 710 TB of RAM.

mentation of the multi-level mesh refinement approach. *Strong scaling* is defined as a decrease in execution time when a fixed size problem is solved on more cores, and *weak scaling* as the change in execution time as the number of processors and problem size vary proportionally to each other. Parallel efficiency, *E*, is defined as:

$$E = \frac{T_{serial}}{N * T_{parallel}(N)}, \tag{4}$$

where T_{serial} is the time to solution using 1 processing unit, N is the number of processing units and $T_{parallel}(N)$ is the time to solve the same problem with N processing units.

Strong scaling is important in our case as the CCMSC seeks to solve a fixed target problem in a tractable amount of time using more compute resources. To achieve this, the CCMSC needs the whole of machines like Titan. Regarding weak scaling, as the number of cells increase for future work, AMR will be employed to mitigate the challenge of communicating to every MPI rank a copy of the entire domain's radiation data. On a single mesh level, a quadratic growth $\mathcal{O}(N^2)$ (N is the number of communicating MPI ranks) of MPI messages and its associated data would normally occur as the communication would be all-to-all. Using AMR and requiring global halos only on the coarsest level, only the compute nodes owning those coarsest level patches would communicate its radiation data to all other nodes, thus becoming a some-to-all communication. For the results below, and for a recent large coal boiler simulation on DOE Titan and ALCF Mira [4,31], two AMR levels were employed.

Figures 3 and 4 each show the performance and scalability of the multi-level RMCRT:GPU algorithm for three patch sizes. In each fine level cell in both problems, 100 rays per cell were used to compute the divergence of the heat flux. The number of cells in a patch was varied, 16^3 (red), 32^3 (green), and 64^3 (blue). Each of the simulations consisted of a grid with 2 levels, and used a refinement ratio of 4 between the levels. All simulations were run on the DOE Titan system, leveraging the single GPU per node with Uintah's hybrid, multi-threaded task scheduler and runtime system originally designed and tested in [2,7] using 16 threads and 1 GPU per node. This scheduler and runtime system has been heavily modified as outlined in Section 4 to achieve the results shown here.

For the simulation results shown in Figure 3, the total number of cells in the domain was 17.04 million. The fine level contained 256^3 cells and the coarse level contained 64^3 cells. For the larger simulation results shown in Figure 4, the total number of cells in the domain was 136.31 million. The fine level contained 512^3 cells and the coarse level contained 128^3 cells. Using equation 4, the strong scaling efficiency of the large benchmark problem (Figure 4) is 96% going from 4096 to 8192 GPUs, and 89% going from 4096 to 16,384 GPUs.

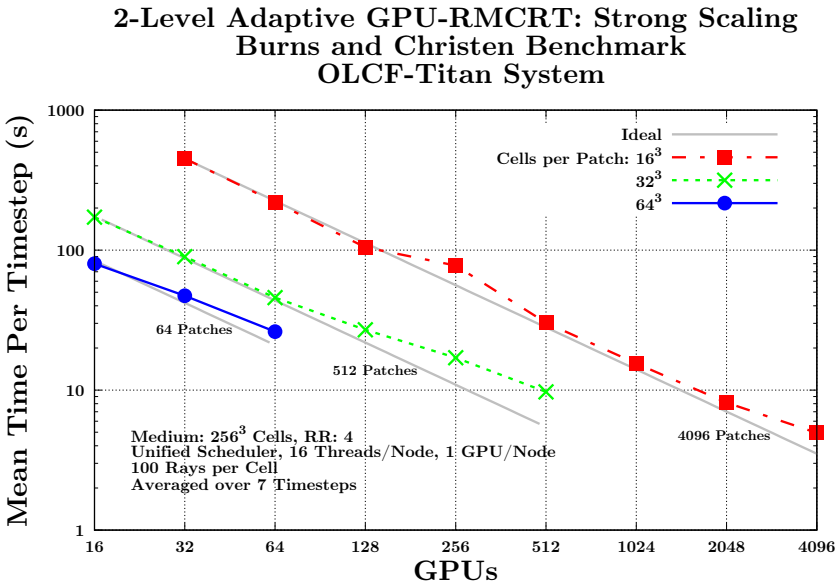

Fig. 3. GPU Strong scaling of the MEDIUM 2-level benchmark RMCRT problem for 3 patch sizes on the DOE Titan system. Refinement ratio of 4 between levels (RR:4). The fine CFD mesh contains 256^3 cells, coarse radiation mesh contains 64^3 cells. Different patch sizes illustrate GPU speedup.

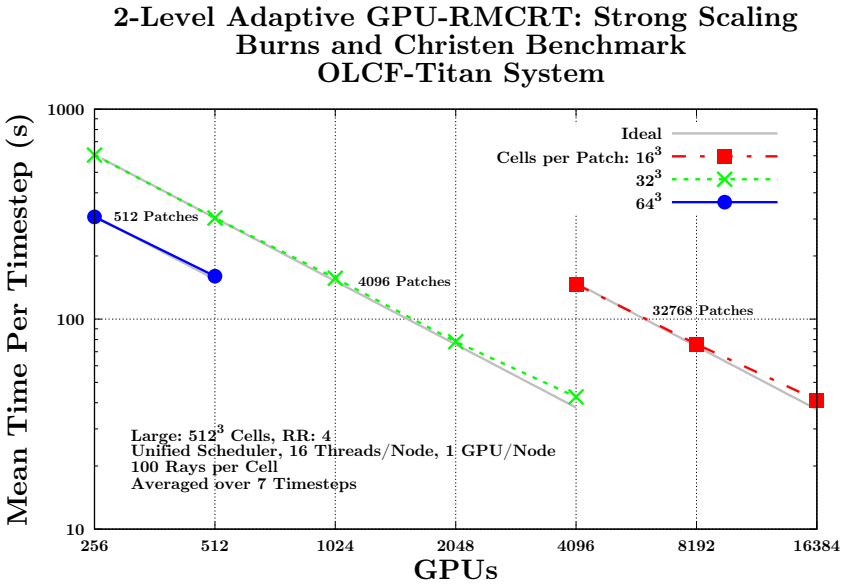

Fig. 4. GPU Strong scaling of the LARGE 2-level benchmark RMCRT problem for 3 patch sizes on the DOE Titan system. Refinement ratio of 4 between levels (RR:4). The fine CFD mesh contains 512^3 cells, coarse radiation mesh contains 128^3 cells. Different patch sizes illustrate GPU speedup.

The tests used for Figures 3 and 4 executed only one RMCRT task at a time on a GPU. This execution model was done under the assumption that production runs using RMCRT would fine-tune patch sizes to utilize all streaming multiprocessors (SMs) of a GPU. A consequence of this approach is that when patches are smaller

than the optimal size, they do not utilize all SMs, and thus total execution times are slower. A different mechanism to efficiently fill all SMs of a GPU regardless of a task's patch size involves assigning an individual RMCRT task multiple CUDA streams and splitting one task into even smaller work units [4]. This mechanism of executing smaller RMCRT work units is used in the results for Section 9, and in those results strong scaling is still observed.

These results show in general that 1.) the RMCRT algorithm strong scales on 16,384 GPUs when work is distributed to more compute nodes, and 2.) improvements to the Uintah infrastructure outlined in this work enables this strong scaling. These results also offer a promising and scalable approach to radiative heat transfer calculations for the CCMSC target boiler problem on current and emerging heterogeneous architectures.

As the problem size grows for RMCRT, more levels of AMR are likely needed. Work is ongoing to support RCMRT on as many as five AMR levels to greatly reduce the amount of MPI communication needed. The two major challenges here are task graph analysis times [4] and data warehouse management of halo data [4]. For upcoming GPU based supercomputers such as OLCF Summit, Uintah itself can supply enough patches and tasks for the increased capacities of these GPUs, and our challenge will be working with this in the context of Kokkos as described in Section 6 and Section 8, specifically managing a limitation based on GPU constant cache memory. For CPU based supercomputers such as NERSC Cori and ALCF Theta, prior and current work [6, 37] is focused using Kokkos with an OpenMP based execution model allowing groups of CPU threads to simultaneously cooperate within a task for faster execution compared to using one CPU thread per task. Finally, results demonstrated here and on ALCF Mira [31] indicate that Uintah compute node scalability is sufficient for upcoming machines.

6 Kokkos GPU Improvements

The need to make use of multiple large-scale parallel architectures both now and in the future makes it desirable to have a single, portable codebase to avoid maintaining multiple implementations. This work leverages Kokkos [38] to achieve a CPU, GPU, and Intel Xeon Phi portable solution for the radiation heat transfer target problem in large computational domains. A major portability challenge is that the current Kokkos Nvidia GPU execution model was not initially designed for the GPU execution models of many asynchronous many-task (AMT) runtimes, including Uintah, that rely on executing finer-grained tasks. The problem lies in Kokkos executing GPU parallel loops in a bulk synchronous manner. However, finer-grained tasks usually cannot be distributed among all streaming multiprocessors of a GPU, resulting in a loss of performance due to unused GPU cores. AMT runtimes avoid this problem by aiming to fully occupy GPU cores by asyn-

chronously executing many concurrent fine-grained tasks. This section details modifications to Kokkos itself to facilitate an execution model compatible with AMT runtimes like Uintah.

An overview of Kokkos is given in Section 6.1. Section 6.2 reviews the current state of similar portability tools and briefly describes their strengths and limitations, with particular emphasis given to GPU portability. Kokkos modifications supporting overlapping concurrent parallel loops are described in Section 6.3, and further parallel reduction work is covered in Section 6.4 to support RMCRT's `parallel_reduce` loops.

6.1 Kokkos Overview

Kokkos [38] provides a C++ programming model for both portability and performance targeting CPUs, GPUs, and Intel MIC (Xeon Phi) platforms. Kokkos currently supports OpenMP, Pthreads, and CUDA as backend programming models and supports GCC, Intel, Clang, IBM, and PGI compilers. Data management abstractions are provided through `Kokkos View` objects which enables parallel loop execution in an architecture-aware manner.

Kokkos uses functors and lambda expressions for both parallel and portable code. An application developer places a single functor or lambda expression inside a `parallel_for`, `parallel_reduce`, or `parallel_scan` construct. An execution policy is also provided describing the loop's iteration pattern. The example code below demonstrates using a functor with a simple Kokkos loop iterating 100 times and writing data into a two-dimensional Kokkos View:

```
typedef Kokkos::View<double*[3]> view_type_2D;

struct Demonstration {
  view_type_2D a;
  Demonstration (view_type_2D a_) : a (a_) {}
  KOKKOS_INLINE_FUNCTION
  void operator() (const int i) const {
    a(i,0) = i;
    a(i,1) = i*i;
    a(i,2) = i*i*i;
  }
};

view_type_2D myView ("A demonstration view", 10);
Kokkos::parallel_for( Kokkos::RangePolicy(0,100),
                      Demonstration (myView));
```

The above code is then compiled targeting the architecture specified in a prior Kokkos configuration step. In the case of a heterogeneous compute node (e.g. a node with CPU cores and an Nvidia GPU), the user may manually specify which memory spaces are used for Views and which architecture executes a given parallel loop.

The current Kokkos Nvidia GPU execution model is bulk synchronous in that Kokkos does not use Nvidia CUDA streams (e.g. a sequence of operations that execute on the GPU in the issued order from host code). Further, only one parallel loop can execute on a GPU at any given time. Kokkos internally partitions parallel loops into many CUDA blocks, and relies on the GPU to distribute those blocks among its many streaming multiprocessors. For this reason, Kokkos is currently most efficient when loops provide iteration ranges large enough to be partitioned into many blocks and distributed uniformly.

6.2 Related Portability Tools

Portability through functors and lambda expressions is also provided by RAJA [39] and Hemi [40]. RAJA is a Lawrence Livermore National Labs project which contains many similarities to Kokkos in its parallel looping design patterns. RAJA is not yet at version 1.0, and its feature set is perhaps not as mature with regard to memory management and architecture-aware execution as Kokkos. The RAJA team is actively developing CHAI [41] to help facilitate portable memory movement of data variables, Sidre for data store management, and Umpire for portable memory allocation and querying. Hemi contains a smaller set of features compared to Kokkos and RAJA. Hemi supports `parallel_for` loops iterating over a configurable range, and provides basic data containers for automatic allocation and data movement between host and GPU memory, with its last release in 2015. Nvidia's Thrust [42] has the ability to provide portable lambda expressions, but most of its feature set targets portable containers and high level algorithms which can operate under CUDA, Intel's Threading Building Blocks (TBB), and OpenMP. Of the three tools listed, Hemi and Thrust support supplying pre-existing CUDA streams for kernel execution, while RAJA supports the default CUDA stream only. Kokkos differs from each of these tools by allowing functor data to be placed in Nvidia constant cache memory to reduce GPU register usage.

OpenACC [43] and OpenMP [44] utilize compiler oriented portability optimization where loops are qualified with pragma directives. OpenACC targets a wide variety of architectures, however, the commonly used PGI compiler for OpenACC does not support Xeon Phi KNL AVX-512 bit vector instructions. OpenMP likewise targets many architectures, and the latest specification targets GPUs [45], however, compilers supporting Nvidia GPUs are still lacking in performance [46, 47]. OpenACC supports asynchronous execution through an `async` clause and supports up to 16

streams. OpenMP provides a `nowait` clause but no ability to explicitly supply or define a stream. OpenCL [48] contains many similarities with the CUDA programming model while allowing for portable compilation. However, OpenCL's performance frequently lags behind CUDA code [49].

These tools take varied approaches for asynchronous parallel reductions. Kokkos itself wraps both the functor and reduction logic inside a single kernel call to avoid latencies of a second kernel invocation. RAJA is implementing stream support, but at this time requires synchronization to retrieve a reduction value computed in an asynchronous parallel reduction. Hemi has no support for parallel reductions. PGI's OpenACC GPU reduction implementation requires two CUDA kernels, one for the loop and a second for the reduction. OpenACC requires a synchronization to obtain the result of the reduction value. Implementing GPU reductions with OpenMP resulted in significant performance losses when using the Clang compiler [47].

6.3 GPU Asynchrony with Kokkos Parallel_For

Kokkos is responsible for executing functors (or a lambda expression which effectively are compiled into a functor) on the GPU. Kokkos does this by first copying a functor's data into GPU memory, then executing a CUDA kernel, and within the kernel invoking the functor. The kernel itself can be partitioned into many CUDA blocks to distribute the computation among a GPU's streaming multiprocessors.

Supplying asynchrony for Kokkos functors requires more than simply attaching a stream to the CUDA kernel. Kokkos can execute functors from two memory locations found within an Nvidia GPU: 1.) local memory or 2.) constant cache memory. The local memory approach simply requires copying the functor data in to GPU memory through a CUDA kernel parameter. The constant cache memory approach requires an explicit copy call into that memory space. Constant cache memory is beneficial as it acts like read-only registers as functor data is fetched in a single clock cycle when read requests are made to the same memory location. In this manner, a GPU's registers are left free for normal code execution. The Kokkos team has indicated that functors requiring less than 512 bytes are best executed through local memory, while larger functors should utilize constant cache memory [50].

6.3.1 Asynchronous Functors in Constant Cache Memory

Constant cache memory is shared among all executing kernels and limited in size (64K bytes on all recent CUDA capable GPUs). A diagram of the current constant cache process for functor copies and functor execution is given in Figure 5. A single functor generates two synchronization points, one for the copy, and another to wait for the functor to complete so a subsequent functor can be placed into constant cache memory.

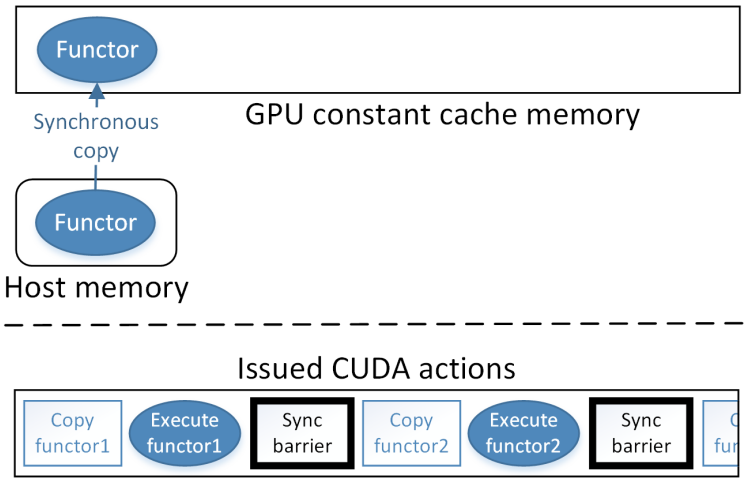

Fig. 5. Current Kokkos GPU execution model using constant cache memory.

Supporting asynchronous execution of kernels requires one mechanism to place multiple functors into constant cache memory and another mechanism to determine when execution completes. This marks the start of this work's modifications of Kokkos. A schematic diagram of the new implementation is shown in Figure 6 and described in more detail below.

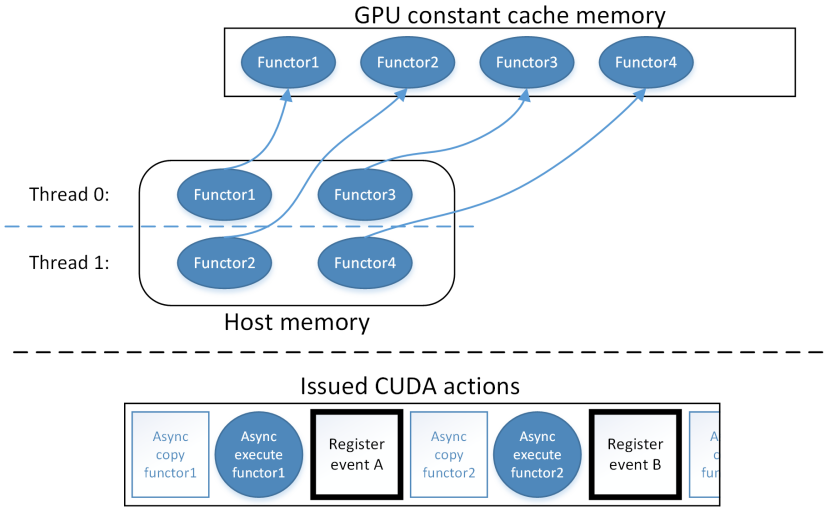

Fig. 6. Multiple functors can asynchronously be placed into GPU constant cache memory, and CUDA events are used to track functor completion.

Within Kokkos a bitset is now used to represent blocks of constant cache memory. When a CPU thread initiates a Kokkos asynchronous parallel loop, Kokkos's engine determines how many data blocks that functor requires, then attempts to locate and atomically claim a contiguous bitset region which corresponds to constant cache memory that can fit the functor data. Upon atomically claiming a region, the functor data is asynchronously copied into constant cache memory, and the functor is executed after supplying the correct offset. As an example, suppose a functor requires 432 bytes, and constant cache memory is split into 128 byte data blocks.

This functor requires 4 data blocks, and Kokkos will atomically attempt to set four contiguous unclaimed bits in the bitset.

Kokkos must have knowledge of when an asynchronous functor completes so that it can mark that region of constant cache memory as reusable. An initial (but ultimately abandoned) approach attempted to use CUDA callback functions which invoke a CPU function upon GPU kernel completion. When callback functions were used among many asynchronously queued streams, significant synchronization was observed among CUDA kernels which increased wall time considerably.

The adopted solution utilizes CUDA events. Immediately after a kernel containing a functor is invoked in a stream, a CUDA event is placed in the same stream (see Figure 6) and Kokkos internally associates that functor with that stream. When the streamed kernel completes, the streamed event is then triggered. Kokkos does not search any events for completion unless the constant cache memory is full with no room for an additional functor. At this point, Kokkos will search through all events and atomically unset bits associated with completed functors. If reuse of functors is desired for optimization, a reference counter could also be associated with each functor. However, Uintah has no use case where functor reuse is possible, as each Uintah task has a unique set of input arguments, and so this optimization was not implemented.

6.3.2 Kokkos API Additions

In the prior Kokkos code example, an execution policy of `Kokkos::RangePolicy(0,100)` indicated 100 total iterations of the parallel loop. The API now supports an additional execution policy parameter for an object containing a CUDA stream, as shown below:

```
Kokkos::Cuda myInstance("some instance description");
Kokkos::RangePolicy myRange(myInstance, 0, 100);
Kokkos::parallel_for(myRange, Demonstration(myView));
```

Kokkos will create a stream upon instantiation of this `Kokkos::Cuda` object, and later reclaim the stream when the object goes out of scope. An application developer wishing to supply a pre-existing stream can do so by supplying it as the argument rather than supplying a descriptive string name. During parallel loop invocation, Kokkos checks if it received an object with a valid stream, and invokes the loop asynchronously or synchronously accordingly. Finally, Kokkos's `DeepCopy` API method to transfer data into another memory space was also modified to support asynchronous copies by supplying a `Kokkos::Cuda` object. This enables Kokkos to asynchronously copy the functor into constant cache memory on the appropriate stream.

6.3.3 Asynchronous Parallel_For Results

These additions were tested and profiled to measure two key items: 1.) any difference in execution times between native CUDA code and Kokkos `parallel_for` code, and 2.) any increase in overhead either from Kokkos itself or from synchronization occurring internally within the GPU. The test code computed simple addition operations on values in global memory and both codes were written using identical logic. Each pair of codes was executed multiple times on 8 streams, with the amount of iterations varying each invocation. The number of threads was deliberately kept low at 256 so as to keep all execution within one CUDA block to simulate the kinds of task codes an AMT runtime may execute. Each kernel was designed to execute between 3 to 25 milliseconds to again simulate short-lived task code common within AMT runtimes. Further, Kokkos functors were executed both in local memory and in constant cache memory. All tests were performed on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 7.5 and on a Intel Xeon E5-2660 CPU.

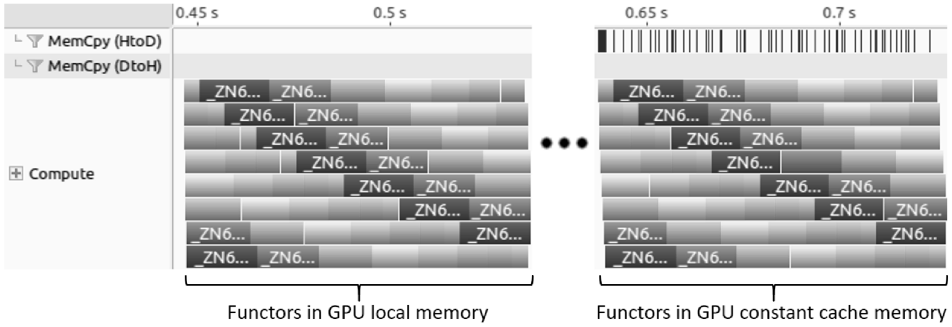

Fig. 7. Profile of executing many `parallel_for` loops on 8 streams and while varying the location of functor data storage. Each block represents a loop. Using constant cache memory requires explicit host-to-device copy calls, but does not significantly affect overhead.

Figure 7 shows an Nvidia Visual Profiler output of a set of `parallel_for` loops executed with the functors in local memory, and another set with the functors in cache memory. These two approaches do an excellent job overlapping computations and avoiding synchronization. Detailed timing indicates the local memory functors approach was only 0.1% to 1% slower than using native CUDA code. The constant cache memory functors approach was roughly 2% slower, which is somewhat expected as this approach requires an additional copy step and additional Kokkos overhead. The average measured additional slowdown per `parallel_for` loop in the constant cache approach is roughly 0.22 milliseconds compared to native CUDA code.

When the original version of Kokkos (prior without modifications) was used to test the same set of `parallel_for` loops described at the start of this section, the computation time was roughly 4.8x slower as these kernels could only be executed serially. The reason this was not 8x slower for the 8 streams is likely due to a lack of global memory bus contention.

RMCRT utilizes `Kokkos::parallel_reduce`, and this section describes work to make the corresponding GPU execution of this parallel pattern asynchronous. Extending the `Kokkos::parallel_for` modifications to `Kokkos::parallel_reduce` required addressing the challenge of copying a reduction result in GPU memory to host memory while avoiding synchronization of other kernels currently executing or queued for execution. Further, the Kokkos API was modified to allow testing if a reduction value is ready without having to explicitly invoke a CUDA synchronization and lock a CPU thread. When implementing these modifications, care was taken to allow future work where a reduction value from a prior loop is used as input into an upcoming loop.

Most of the prior Kokkos `parallel_for` modifications carried forward into `parallel_reduce` logic. Kokkos was further modified to asynchronously copy the reduction value from device-to-host if a stream was used to invoked the reduction kernel. Using `cudaMemcpyAsync` to copy this data may incur a synchronization point. The CUDA documentation explains that even though the function call contains the `Async` suffix, "this [Async] is a misnomer as each function may exhibit synchronous or asynchronous behavior depending on the arguments passed to the function." The documentation further states that when using asynchronous copies to transfer "from device memory to pageable host memory, the function will return only once the copy has completed." [51] For this reason the buffer used to receive the reduction value should be stored in pinned host memory.

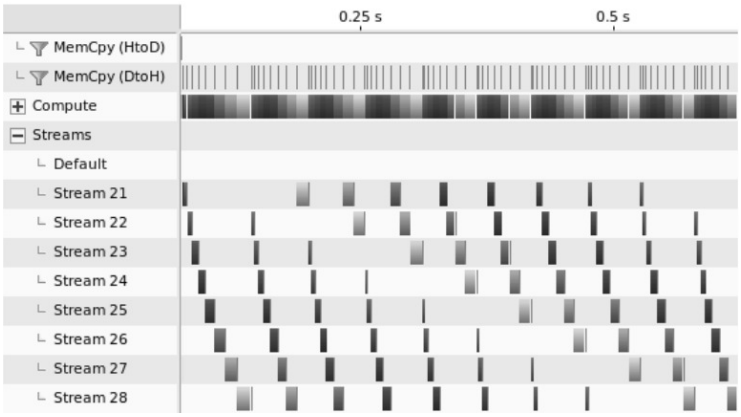

Fig. 8. Profile of executing many `parallel_reduce` loops and their associated streams. Reduction values in pageable host memory invoked repeated synchronization and caused delays as shown by the gaps between executing kernels.

Test code similar to that used in Section 6.3.3 was used to measure many quickly executing reduction loops operating on 8 streams. Figure 8 shows the synchronization incurred by using non-pinned memory for the reduction value. Figure 9 demonstrates the desired overlapping by using pinned memory. Detailed timing indicated minimal differences in using either GPU local memory or GPU constant

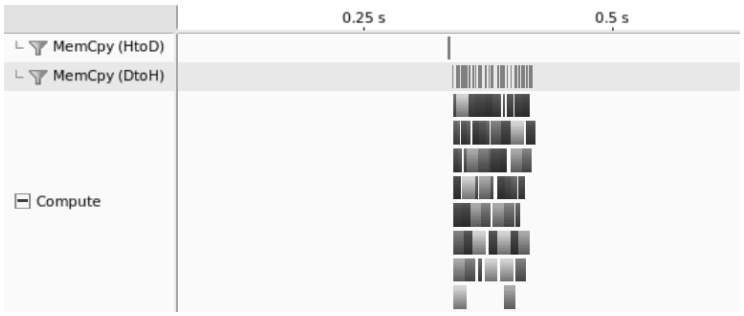

Fig. 9. Profile of the same parallel reduction loops as Figure 8, but utilizing pinned host memory for reduction value buffers. Synchronization is avoided even when pageable host memory is used to copy the functor data host-to-device.

cache memory to store functor data, with the latter computing these test problems roughly 2% slower and requiring an additional overhead of 0.21 milliseconds per kernel compared to the local memory functor approach.

To determine if the reduction value is ready, the `Kokkos::Cuda` instance object described in Section 6.3.2 now offers a `getStatus()` method which indicates if all operations on the stream are ongoing or complete. In future work, the reduction value can be encapsulated inside a specialized Kokkos View object capable of receiving future data. This approach would allow Kokkos to recognize whether the reduction value can stay resident in GPU data if necessary, thus better facilitating nested loops of reductions.

7 Intel Xeon Phis with Uintah and Kokkos

While the focus of this paper is on achieving portability to GPUs via Kokkos, it is also important for this approach to work on Intel Xeon Phi processors. This section reviews prior Uintah work targeting Kokkos portability and performance with the Intel Xeon Phi, overviews Xeon Phi high bandwidth memory (e.g. MCDRAM) and vectorization in the context of the RMCRT problem, and describes portability challenges with iteration patterns for both GPUs and Xeon Phis.

7.1 Review of Prior Performance Work

Initial work [37] provided a foundational strategy to incrementally add Kokkos loops to Uintah. Additionally, a simple mock runtime patterned after Uintah demonstrated the viability of vectorization for a stencil computation on CPUs, GPUs, and Xeon Phis. Follow-up work [6] implemented Kokkos into Uintah for the Xeon Phi and demonstrated substantial speedups by allowing Kokkos to use its OpenMP backend to run all available hardware threads in a task rather than the prior Uintah

model [2] of only one thread per task.

7.2 MCDRAM and Vectorization

Xeon Phi performance is usually significantly degraded if the problem exceeds MCDRAM capacity of 16 GB. The multi-level RMCRT approach [1] prevents the problem of exceeding that limit, even if the problem were scaled to thousands of nodes. For the single-level RMCRT computations in prior work [6] and in Section 10, the problem size also does not exceed 16 GB and thus uses MCDRAM as a cache throughout. Further, ray vectorization for the Intel MIC architecture has been explored [52], but has not yet been implemented into this problem and may not be viable due to the relatively low number of rays used in this RMCRT implementation.

7.3 Portable Challenges for Iteration Patterns

Extending the prior Kokkos RMCRT implementation for GPUs required maintaining an iteration pattern that is viable across CPUs, GPUs, and Xeon Phis. In particular, the prior CUDA implementation in Section 3.4 and later revised for a full production run [4] requires keeping thread counts low (between 256 and 320 threads per kernel) so as to keep GPU register usage low. As a result, the CUDA implementation had each GPU thread assigned to a region of many cells, rather than the alternative method of assigning only one cell per thread. When this CUDA code strategy was adopted into the portable Kokkos RMCRT implementation, results showed that it executed efficiently on CPUs but not efficiently for Xeon Phis. The problem is that the Kokkos iteration range number doesn't correspond with the number of threads created across architectures. For example, suppose a range of 0 to 65,536 is supplied for a loop. The Kokkos OpenMP backend will generally create enough worker threads equal to the number of cores or hyperthreads, while the Kokkos CUDA backend will create 65,536 GPU threads, and 65,536 GPU threads inefficiently utilizes far too many registers. As a result, our strategy changed to allow supplying different sets of range parameters depending on the architecture, and also modifying the portable code to ensure it operated efficiently despite a difference in ranges.

8 Uintah GPU Integration with Kokkos

With Kokkos modifications in place, integrating Kokkos into Uintah to support GPU portability for RMCRT task code required a few additional Uintah modifi-

cations. The changes listed in this section were made while not affecting existing Uintah tasks written using C++ or CUDA code.

8.1 Unmanaged Kokkos Views and Streams

Uintah's `DataWarehouse` (see Section 2) has mechanisms supporting automatic allocation of simulation variables and concurrent, asynchronous data movement of these variables in both host and GPU memory [2–4]. Integrating the `DataWarehouse` with Kokkos utilized Kokkos Unmanaged Views. This allows a Kokkos View to encapsulate already allocated simulation variable data. Previously Uintah's host memory `DataWarehouse` [37] was modified to allow an application developer to request a Kokkos View from a simulation variable. This work modified the `GPU DataWarehouse` to do the same. From here a Kokkos View can be passed into a functor in a portable manner.

In a similar fashion, Uintah manages the creation and reuse of CUDA streams. Like Unmanaged Views, Uintah supplies these pre-existing streams to `Kokkos::Cuda` instance objects. Uintah uses the streams themselves, and not a `Kokkos::Cuda` instance object's `getStatus()` method, to determine when GPU task execution completes.

8.2 Random Number Portability

Uintah maintains a host random number API using C++11 libraries for CPU tasks, and second random number API using CUDA random number libraries for GPU tasks. These random number libraries take distinctly different approaches and are not portable. Kokkos supplies its own portable random number library based on Marsaglia's xorshift generators [53], and this was implemented into the RMCRT task code.

8.3 Invoking Portable Tasks

Invoking Kokkos parallel loops within Uintah required substantial refactoring to the Uintah runtime codebase. Architecture specific branching logic required more compile time decisions to ensure compilation for only the supported architectures. Further, as Uintah mixes both CPU and GPU tasks, some of these tasks must support only a Kokkos OpenMP implementation, while others must support both Kokkos OpenMP and CUDA options.

The first major refactor came in the Uintah task declaration phase by allowing ap-

plication developers to state all architectures that task supports. Here architecture data types are first introduced to the compiler, and these data types are propagated through the rest of Uintah at compile time by using template metaprogramming. This process also utilized C++ macros for code generation to help create function pointers for templated methods.

The second major refactor came by creating a Uintah parallel loop invocation layer which in turn invokes Kokkos parallel statements. This Uintah invocation layer allows the application developer to supply a range of cells within a patch, and Uintah in turn maps these cells to threads. By using template metaprogramming and template partial specialization, a different set of Kokkos execution parameters can be supplied for each architecture. For example, we observed Kokkos portable tasks executed through OpenMP for Intel Xeon Phi KNL architectures required more threads for efficient execution compared to their CUDA counterparts. In another example, RMCRT's CUDA implementation is more efficient when using CUDA's `__launch_bounds__` to manually restrict the total registers a kernel can use to achieve better kernel occupancy in a GPU's streaming multiprocessors. By creating this additional parallel loop layer, Uintah is able to avoid requiring the application developer supply any architecture specific parameters, though these parameters may be supplied if desired.

9 Portability Results

Uintah features three implementations of single-level RMCRT: 1.) RMCRT:CPU which heavily utilizes STL libraries and serially executes using one thread per task; 2.) RMCRT:GPU which is written using CUDA code and executes in parallel using hundreds of CUDA threads; and 3.) RMCRT:Kokkos which uses portable code throughout. The initial implementation of RMCRT:Kokkos was introduced previously [6] to overcome an Intel Xeon Phi scalability barrier attributed to the execution of serial tasks when using RMCRT:CPU on the Xeon Phi. As part of this work, single-level RMCRT:Kokkos has been further refined to enable portability for the GPU. This was accomplished by removing all remaining host system calls, and implementing Kokkos's portable random number library. This implementation can now use either the Kokkos OpenMP backend or the Kokkos CUDA backend. This refinement marks the consolidation of single-level RMCRT approaches into a single codebase implementation.

9.1 Single-Node Experiments

This section shows single-node results using CPUs, GPUs, and Intel Xeon Phis for the Burns and Christen [35] benchmark problem using all three RMCRT code im-

plementations of the single-level approach (RMCRT:CPU, RMCRT:GPU, RMCRT:Kokkos). Table 1 presents results comparing mean times per timestep for two problem sizes (64^3 cells and 128^3 cells). Tasks are each assigned to compute a different region of 16^3 cells, which represents Uintah's most common application driven over-decomposition. The absorption coefficient was initialized according to the benchmark [35] with a uniform temperature field and 100 rays per cell used to compute the radiative-flux divergence for each cell. CPU-based results were gathered on an Intel Xeon E5-2660 CPU @ 2.2 GHz with 2 sockets, 8 physical cores per socket, and 2 hyperthreads per core. For RMCRT:CPU, Uintah's scheduler utilized 32 threads. GPU-based results were gathered on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 7.5. Xeon Phi KNL results were gathered on an Intel Xeon Phi 7230 Knights Landing processor @ 1.30 GHz with 64 physical cores and 4 hyperthreads per core. Xeon Phi KNL simulations were launched using 1 MPI process and 256 OpenMP threads with the *OMP_PLACES=threads* and *OMP_PROC_BIND=spread* affinity settings.

Comparison in Mean Times per Timestep (s)			
		Problem Size	
Processor/Accelerator	Implementation	64^3	128^3
CPU	RMCRT:CPU	14.34	302.02
	RMCRT:Kokkos	7.41	182.81
GPU	RMCRT:GPU	5.94	76.31
	RMCRT:Kokkos after Kokkos changes	3.71	64.72
	RMCRT:Kokkos without Kokkos changes	16.82	274.57
Xeon Phi KNL	RMCRT:Kokkos	4.84	106.89

Table 1
Single-node experiments demonstrating one codebase executed on the CPU, GPU, and Intel Xeon Phi Knights Landing (KNL) processors. The pre-existing C++ and CUDA implementations are also given for comparison purposes.

The CPU speedups by using Kokkos are attributed to: 1.) Kokkos avoids code indirection when requesting simulation data through an `(i,j,k)` interface; 2.) The workload per task is not uniform, and RMCRT:CPU occasionally has more idle CPU cores as it assigns one thread per task, resulting in some CPU threads completing their task computation while other threads have not. RMCRT:Kokkos assigns every thread to compute a task, avoiding the problem of idle CPU threads.

The speedup in GPU execution before and after the changes to Kokkos highlights the work done in Section 6 to support executing finer-grained tasks. Each task was assigned a region of $16x16x16 = 4096$ cells, but these 4096 cells cannot be well-distributed among all streaming multiprocessors on the GPU. Before this work, Kokkos would synchronize between each parallel loop, and these finer-grained

tasks left some GPU cores idle. Now Kokkos can asynchronously execute loops to fill the entire GPU, thus dramatically improving the total wall time computation.

An additional test compared Kokkos GPU RMCRT execution of many finer-grained asynchronous parallel loops against synchronous coarse-grained loops. This test assigned an iteration range across the entire monolithic block of 64^3 and 128^3 cells, respectively, and tested these problems on Kokkos prior to any changes. Here Kokkos had enough work to distribute throughout the GPU, and the total timestep execution times of synchronously executing on these monolithic blocks were similar to asynchronously executing on many smaller 16^3 cell blocks.

9.2 Scalability Studies with Kokkos Codebase

Having previously demonstrated full machine scalability to 16,384 GPUs in Section 5, this section demonstrates that the portable codebase implementation can execute on Nvidia GPUs on the DOE Titan XK7, and on Intel Xeon Phis on the ALCF Theta XC40 [2] . On both machines the Burns and Christen [35] benchmark was used, with 100 rays per cell to compute the divergence of the heat flux, and each task assigned a different 16^3 cell region to simulate the size of tasks used in production runs.

Figure 10 shows Xeon Phi KNL-based performance and strong scalability of the single-level RMCRT:Kokkos algorithm. These results show that the modifications to the portable Kokkos code to support GPUs did not change the strong scaling on the Xeon Phi. Future performance work is required here as over-decomposing this problem into tasks operating on 16^3 cells yields times that are not on par with a single large patch per node [6].

Figure 11 shows Nvidia GPU-based performance and strong scalability of the single-level RMCRT:Kokkos algorithm. These results demonstrate that strong scalability remains when running the portable Kokkos code.

The wall time performance has markedly improved between the original baseline GPU implementation and the portable Kokkos implementation as the portable code requires fewer GPU registers, enabling two kernels to fit per GPU streaming multiprocessor (SM), whereas the Uintah GPU baseline required one kernel per SM. Further, the Nvidia K20X GPU used in Titan nodes have 14 SMs, and so the portable Kokkos code implementation allows for 28 concurrently executing kernels. How-

[2] Theta is a Cray XC40 system located at Argonne National Laboratory, where each node hosts a 64-core Intel Xeon Phi 7230 Knights Landing process running at 1.3 GHz with 16 GB high-bandwidth MCDRAM and 192 GB DDR4 memory. Theta uses an Aries Dragonfly network with the entire machine offering 231,936 cores, 56 TB of MCDRAM, and 679 TB of DDR4 memory.

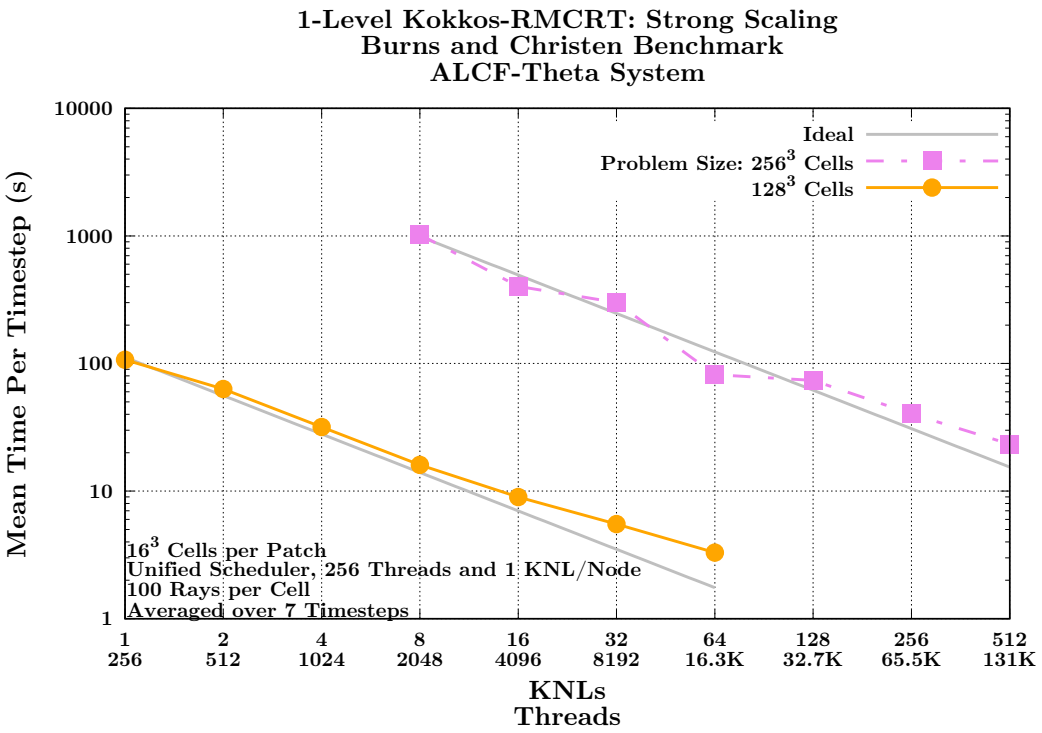

Fig. 10. Intel Xeon Phi strong scaling of the SMALL (128^3 cells) and MEDIUM (256^3 cells) single-level benchmark RMCRT problem on the ALCF Theta system.

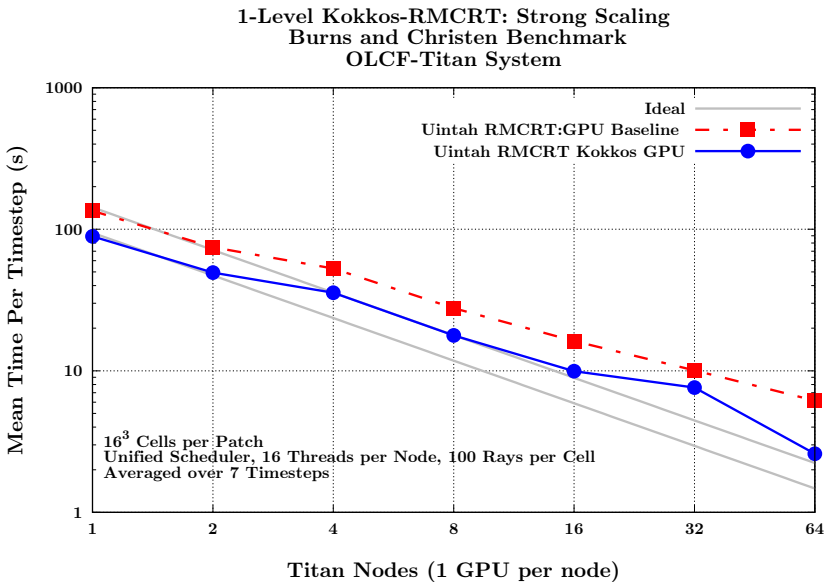

Fig. 11. Nvidia GPU strong scaling of the SMALL (128^3 cells) single-level benchmark RMCRT problem on the OLCF Titan system. The Kokkos GPU tests here use the modifications outlined in Section 6.

ever, it should be noted that the problem is memory bounded (more specifically latency bounded), and benchmarking has indicated this factor becomes dominant after 14 concurrently executing kernels but before 28. The GPU used in Table 1 has 24 SMs, and here the memory bounded feature becomes dominant both in the Uintah GPU baseline and the Kokkos implementation.

Another set of tests measured the Uintah RMCRT Kokkos implementation on Titan's Nvidia K20X GPUs using both the original and modified Kokkos codebase as outlined in Section 6. The same RMCRT test parameters as used for Figure 11 were also used here. On 1, 2, and 4 Titan compute nodes, we observed the new Kokkos enhancements yielded between a 3.13x to 4x speedup compared to the original Kokkos implementation.

This Uintah and Kokkos execution model is expected to scale well to the upcoming generation of GPUs with one exception, the limited resource of GPU constant cache memory to supply functor parameters. The number of SMs per GPU increases from 14 in each Titan GPU to 84 in each Summit GPU. Uintah obtains fast task execution by limiting finer-grained Kokkos loops to one GPU SM. However, the GPU's 64KB of constant cache memory is not enough to support 84 concurrently executing kernels, one for each SM. This problem will likely be solved either through 1) Uintah using coarser grained tasks with patches large enough to efficiently distribute work among multiple streaming multiprocessors, or 2) a mechanism to avoid passing in large amounts of parameter data through constant cache memory.

10 Related Work

Regarding scalability in the context of a radiation transport problem, other models can be found in computational astrophysics and cosmology, involving problems such as neutron star merger, supernova, and high energy density plasma. This work is in the context of codes like ARWIN, the AZEuS adaptive mesh refinement, magnetohydrodynamics fluid code, the more general AMR-based FLASH code [54], based on oct-tree meshes and the physics AMR code Enzo, [55]. At the national labs, radiation codes such as RAMSES and PARTISN [56] exist, but are not generally available. For target problems like neutron transport, CRASH [57] supplies a block adaptive mesh code for multi-material radiation hydrodynamics. There are also radiation transport problems that use CFD codes and AMR techniques [58,59]. Much of the available literature on GPU-based Monte Carlo ray tracing approaches to radiation can be found in the Oncology community where GPUs are used for radiation dose calculation [60]. There are overall very few cases of GPU usage at the scale reported here. Gaburov, et al. [61] have published results on the evolution of the Milky Way galaxy, a calculation done using 18,600 GPUs on DOE Titan. Gray, et al. [62] were one of the first to take advantage of at least 8,192 GPUs in parallel with their Ludwig soft matter physics application.

Regarding portability in the context of an AMT runtime, the ACCEL framework [63] addition to the Charm++ [64] runtime created a model that didn't use functors but did allow supplying a single region of code capable of being compiled for different architectures, including GPUs. ACCEL differs from the Kokkos approach in that ACCEL focuses more on load balancing strategies and avoids providing `parallel_for` or `parallel_reduce` constructs. The Legion [65] project is developing the Regent language [66] with a goal to become architecture independent, but at the moment it relies on the LLVM code translator, and recent work [67] explains that the "LLVM code translator works well for host CPU code, but is not sufficient for tasks that will be run on CUDA-capable processors". More generally, runtimes like Legion, ParSEC [68], and StarPU [69] aid in automatic data movement between CPU and GPU memory space, but leave portability decisions up to the task's code supplied by the application developer and do not integrate a portability library into the runtime.

Regarding portability in the context of software libraries or compiler directives, refer to Section 6.2.

11 Conclusions and Future Work

This work has: (i) Demonstrated a scalable and portable solution for radiative heat transfer problems through a combination of reverse Monte Carlo ray tracing (RMCRT) techniques, adaptive mesh refinement, and adoption and modification of the Kokkos portability library. (ii) Shown the necessity of choosing optimal data structures and algorithms to efficiently expose concurrency. Furthermore, runtime systems like Uintah allow management of limited memory through the use of custom allocators that allow us to choose better allocation policies for different objects and to better utilize available resources, improving nodal throughput. (iii) Demonstrated scalability to 16,384 nodes on a multi-level GPU implementation by dramatically reducing communication costs associated with global data dependencies and reducing the memory footprint required to fit the problem in GPU memory. (iv) Modified Kokkos to allow for GPU asynchronous and concurrent execution of many finer-gained parallel loops. (v) Demonstrated efficiently executing one codebase on CPUs, GPUs, and Intel Xeon Phi Knights Landing processors. The results presented here offer a promising approach to code portability for future Uintah projects, as well as Kokkos users needing finer-grained asynchronous GPU parallel loop execution.

Implementation of RMCRT GPU portability in Uintah required far more work than simply modifying loops to compile for different architectures. In particular, the major changes made to support GPU portability are: (i) Enabling application developers to define which architectures a task supports, and then letting that architecture information propagate through Uintah using template metaprogramming.

(ii) Having data stores and a task scheduler capable of automatically ensuring simulation variable data and its associated halos are prepared and present in the required memory space prior to execution. (iii) Modifying Kokkos itself to enable asynchronous execution of Kokkos loops on CUDA streams, as well as modifying Kokkos to asynchronously support multiple functors copying into GPU constant cache memory. (iv) Allowing Uintah to compile for a mixture of Kokkos OpenMP tasks and Kokkos CUDA tasks in the same build. (v) Implementing a portable random number library into the RMCRT algorithm. (iv) Enabling application developers to launch parallel loops in a portable manner.

As part of future work, the Uintah data stores need to be reworked to support increased portability. At the moment Uintah has two data stores to manage simulation variables, one for host memory and another for GPU memory. These two data stores use fundamentally different approaches and likewise have different API. The data stores and API must be merged to allow an application developer to retrieve simulation variables in an architecture agnostic manner. Additional Uintah scheduler work is actively ongoing for the Intel Xeon Phi to better distribute tasks among the Xeon Phi cores. The end goal being to execute Uintah tasks on a subset of Xeon Phi cores, instead of executing tasks serially with one core per task, or bulk synchronously with all cores per task. Further, while Uintah has demonstrated mixing heterogeneity of CPU and GPU tasks in a production problem at scale [4], more work is needed to efficiently execute Kokkos parallel loops on both CPUs and GPUs in the same build.

12 Acknowledgments

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We would like to acknowledge Oak Ridge Leadership Computing Facility ALCC awards CMB109, “Large Scale Turbulent Clean Coal Combustion” and CSC188, “Demonstration of the Scalability of Programming Environments By Simulating Multi-Scale Applications” for time on Titan. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. Additional support for J. H. comes from the Intel Parallel Computing Centers program. Additionally, we would like to thank all those involved with Uintah past and present.

References

[1] A. Humphrey, T. Harman, M. Berzins, P. Smith, A Scalable Algorithm for Radiative Heat Transfer Using Reverse Monte Carlo Ray Tracing, in: J. M. Kunkel, T. Ludwig (Eds.), High Performance Computing, Vol. 9137 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 212–230. doi:10.1007/978-3-319-20119-1_16.

[2] Q. Meng, A. Humphrey, M. Berzins, The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System, in: Digital Proceedings of Supercomputing 12 - WOLFHPC Workshop, IEEE, 2012.

[3] B. Peterson, H. Dasari, A. Humphrey, J. Sutherland, T. Saad, M. Berzins, Reducing Overhead in the Uintah Framework to Support Short-lived Tasks on GPU-heterogeneous Architectures, in: Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '15, ACM, New York, NY, USA, 2015, pp. 4:1–4:8. doi:10.1145/2830018.2830023.
URL http://doi.acm.org/10.1145/2830018.2830023

[4] B. Peterson, A. Humphrey, J. Schmidt, M. Berzins, Addressing Global Data Dependencies in Heterogeneous Asynchronous Runtime Systems on GPUs, in: Third International Workshop on Extreme Scale Programming Models and Middleware, ESPM2, IEEE Press, 2017.

[5] A. Humphrey, D. Sunderland, T. Harman, 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), 2016, pp. 1222–1231.
URL http://www.sci.utah.edu/publications/Hum2016a/ipdps-pdsec16.pdf

[6] J. K. Holmen, A. Humphrey, D. Sunderland, 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, PEARC17, ACM, New York, NY, USA, 2017, pp. 27:1–27:8. doi:10.1145/3093338.3093388.

[7] Q. Meng, A. Humphrey, J. Schmidt, M. Berzins, Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, ACM, New York, NY, USA, 2013, pp. 96:1–96:12. doi:10.1145/2503210.2503250.
URL http://doi.acm.org/10.1145/2503210.2503250

[8] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[9] M. Berzins, Status of Release of the Uintah Computational Framework, Tech. Rep. UUSCI-2012-001, Scientific Computing and Imaging Institute (2012).

[10] Scientific Computing and Imaging Institute, Uintah Web Page, http://www.uintah.utah.edu/ (2015).

[11] B. Kashiwa, E. Gaffney., Design Basis for CFDLIB, Tech. Rep. LA-UR-03-1295, Los Alamos National Laboratory (2003).

[12] D. Sulsky, S. Zhou, H. L. Schreyer, Application of a particle-in-cell method to solid mechanics, Computer Physics Communications 87 (1995) 236–252.

[13] J. E. Guilkey, T. B. Harman, A. Xia, B. A. Kashiwa, P. A. McMurtry, An Eulerian-Lagrangian approach for large deformation fluid-structure interaction problems, part 1: Algorithm development, in: Fluid Structure Interaction II, WIT Press, Cadiz, Spain, 2003.

[14] J.Spinti, J. Thornock, E. Eddings, P. Smith, A. Sarofim, Heat Transfer to Objects in Pool Fires, in: Transport Phenomena in Fires, WIT Press, Southampton, U.K., 2008.

[15] J. Luitjens, M. Berzins, Improving the Performance of Uintah: A Large-Scale Adaptive Meshing Computational Framework, in: Proc. of the 24th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS10), 2010.

[16] Q. Meng, J. Luitjens, M. Berzins, Dynamic Task Scheduling for the Uintah Framework, in: Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10), 2010.

[17] Q. Meng, M. Berzins, Scalable Large-scale Fluid-structure Interaction Solvers in the Uintah Framework via Hybrid Task-based Parallelism Algorithms, Concurrency and Computation: Practice and Experiencedoi:10.1002/cpe.3099.
URL http://dx.doi.org/10.1002/cpe.3099

[18] A. Humphrey, Q. Meng, M. Berzins, T. Harman, Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System, in: Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment (XSEDE 2012), ACM, 2012. doi:10.1145/2335755.2335791.

[19] P. J. Smith, R.Rawat, J. Spinti, S. Kumar, S. Borodai, A. Violi, Large eddy simulation of accidental fires using massively parallel computers, in: 18th AIAA Computational Fluid Dynamics Conference, 2003.

[20] J. Pedel, J. N. Thornock, P. J. Smith, Large eddy simulation of pulverized coal jet flame ignition using the direct quadrature method of moments, Energy & Fuels 26 (11) (2012) 6686–6694. doi:10.1021/ef3012905.
URL http://dx.doi.org/10.1021/ef3012905

[21] S. Gottlieb, C. Shu, W. Tadmor, Strong Stability-Preserving High-Order Time Discretization Methods, Siam Review 43 (1) (2001) 89–112.

[22] R. Falgout, J. Jones, U. Yang, The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners, in: A. Bruaset, A. Tveito (Eds.), Numerical Solution of Partial Differential Equations on Parallel Computers, Vol. 51 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2006, pp. 267–294. doi:10.1007/3-540-31619-1_8.
URL http://dx.doi.org/10.1007/3-540-31619-1_8

[23] S. B. Pope, Turbulent Flows, Cambridge Press, 2000.

[24] G. Krishnamoorthy, R. Rawat, P. Smith, Parallel Computations of Radiative Heat Transfer Using the Discrete Ordinates Method, numerical Heat Transfer, Part B: Fundamentals, 47 (1), 19-38, 2005.

[25] G. Krishnamoorthy, R. Rawat, P. Smith, Parallelization of the P-1 Radiation Model, Numerical Heat Transfer, Part B: Fundamentals, 49 (1), 1-17, 2006.

[26] X. Sun, P. J. Smith, A Parametric Case Study in Radiative Heat Transfer Using the Reverse Monte-Carlo Ray-Tracing with Full-Spectrum k-Distribution Method, Journal of Heat Transfer 132 (2) (2010) 024501.

[27] I. Hunsaker, T. Harman, J. Thornock, P. Smith, Efficient Parallelization of RMCRT for Large Scale LES Combustion Simulations, paper AIAA-2011-3770. 41st AIAA Fluid Dynamics Conference and Exhibit, 2011.

[28] C. Clouse, Parallel Deterministic Neutron Transport with AMR, in: F. Graziani (Ed.), Computational Methods in Transport, Vol. 48 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2006, pp. 499–512. doi:10.1007/3-540-28125-8_25.
URL http://dx.doi.org/10.1007/3-540-28125-8_25

[29] M. P. Adams, M. L. Adams, W. Hawkins, T. Smith, L. Rauchwerger, N. y M. Amato, T. S. Bailey, R. Falgout, M. L. Adams, T. G. Smith, N. M. Amato, Provably Optimal Parallel Transport Sweeps on Regular Grids, 2013.

[30] I. Veljkovic, P. E. Plassmann, Scalable Photon Monte Carlo Algorithms and Software for the Solution of Radiative Heat Transfer Problems, in: Proceedings of the First International Conference on High Performance Computing and Communications, HPCC'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 928–937. doi:10.1007/11557654_104.
URL http://dx.doi.org/10.1007/11557654_104

[31] S. Kumar, A. Humphrey, W. Usher, S. Petruzza, B. Peterson, J. A. Schmidt, D. Harris, B. Isaac, J. Thornock, T. Harman, V. Pascucci, M. Berzins, Scalable Data Management of the Uintah Simulation Framework for Next-Generation Engineering Problems with Radiation, in: Supercomputing Frontiers, Springer International Publishing, 2018, pp. 219–240.
URL http://www.sci.utah.edu/publications/Kum2018a/Scalable_Data_Management_of_the_Uintah_Simulation_.pdf

[32] K. Viswanath, I. Veljkovic, P. E. Plassmann, Parallel Load Balancing Heuristics for Radiative Heat Transfer Calculations., in: CSC, 2006, pp. 151–157.

[33] M. F. Modest, Backward Monte Carlo Simulations in Radiative Heat Transfer, Journal of Heat Transfer 125 (1) (2003) 57–62.
URL
http://link.aip.org/link/JHTRAO/v125/i1/p57/s1&Agg=doi

[34] B. Hapke, Theory of Reflectance and Emittance Spectroscopy, Cambridge University Press, 1993, Cambridge Books Online.
URL http://dx.doi.org/10.1017/CBO9780511524998

[35] S. Burns, M. A. Christen, Spatial Domain-Based Parallelism in Large-Scale, Participating-Media, Radiative Transport Applications, Numerical Heat Transfer, Part B: Fundamentals 31 (4) (1997) 401–421.

[36] S. Lee, T. Johnson, E. Raman, Feedback Directed Optimization of TCMalloc, in: Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC '14, ACM, New York, NY, USA, 2014, pp. 3:1–3:8. doi:10.1145/2618128.2618131.
URL http://doi.acm.org/10.1145/2618128.2618131

[37] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, 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, ESPM2, IEEE Press, Piscataway, NJ, USA, 2016, pp. 44–47. doi:10.1109/ESPM2.2016.10.
URL https://doi.org/10.1109/ESPM2.2016.10

[38] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202 – 3216, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. doi:https://doi.org/10.1016/j.jpdc.2014.07.003.
URL http://www.sciencedirect.com/science/article/pii/S0743731514001257

[39] R. D. Hornung, J. A. Keasler, The RAJA Poratability Layer: Overview and Status, Tech. Rep. LLNL-TR-661403 (2014).

[40] M. Harris, Hemi Web Page, http://harrism.github.io/hemi (2017).

[41] L. L. N. Labs, CHAI Web Page, https://github.com/LLNL/CHAI (2017).

[42] N. Bell, J. Hoberock, Thrust: A productivity-oriented library for CUDA, GPU computing gems Jade edition 2 (2011) 359–371.

[43] OpenACC member companies and CAPS Enterprise and CRAY Inc and The Portland Group Inc (PGI) and NVIDIA, OpenACC 2.5 Specification, https://www.openacc.org/specification (2015).

[44] L. Dagum, R. Menon, OpenMP: An Industry-Standard API for Shared-Memory Programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55. doi:10.1109/99.660313.
URL https://doi.org/10.1109/99.660313

[45] OpenMP Architecture Review Board, OpenMP Application Program Interface Version 4.5 (2015).

[46] G.-T. Bercea, C. Bertolli, S. F. Antao, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, K. O'Brien, Performance Analysis of OpenMP on a GPU Using a CORAL Proxy Application, in: Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, PMBS '15, ACM, New York, NY, USA, 2015, pp. 2:1–2:11. doi:10.1145/2832087.2832089.
URL http://doi.acm.org/10.1145/2832087.2832089

[47] M. Martineau, C. Bertolli, S. McIntosh-Smith, Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support, Institute of Electrical and Electronics Engineers (IEEE), 2017, pp. 54–64. doi:10.1109/PMBS.2016.011.

[48] A.Bourd,
The OpenCL Specification, https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf (2017).

[49] T. Sörman, Comparison of Technologies for General-Purpose Computing on Graphics Processing Units, Master's thesis, Department of Electrical Engineering, Linköping University (2016).

[50] Kokkos Usage of Cuda Constant Memory, `https://github.com/kokkos/kokkos/issues/606#issuecomment-271905966`, accessed: 2018-4-19.

[51] NVIDIA, CUDA Runtime API Guide (July 2017).

[52] C. Benthin, I. Wald, S. Woop, M. Ernst, W. R. Mark, Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture, IEEE Transactions on Visualization and Computer Graphics 18 (9) (2012) 1438–1448. doi:10.1109/TVCG.2011.277.

[53] S.Vigna,
An Experimental Exploration of Marsaglia's Xorshift Generators, Scrambled, ACM Trans. Math. Softw. 42 (4) (2016) 30:1–30:23. doi:10.1145/2845077.
URL `http://doi.acm.org/10.1145/2845077`

[54] B.Fryxell, K.Olson, P.Ricker, F.X.Timmes, M.Zingale, D.Q.Lamb, P.Macneice, R.Rosner, J. Rosner, J. Truran, H.Tufo, FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes, The Astrophysical Journal Supplement Series 131 (2000) 273–334.

[55] B. O'Shea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, A. Kritsuk, Introducing Enzo, an AMR Cosmology Application, in: Adaptive Mesh Refinement - Theory and Applications, Vol. 41 of Lecture Notes in Computational Science and Engineering, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 341–350.

[56] L. Los Alamos National Security, Los Alamos National Laboratory Transport Packages, http://www.ccs.lanl.gov/CCS/CCS-4/codes.shtml (2014).

[57] B. van der Holst, G. Toth, I. Sokolov, K. Powell, J. Holloway, et al., Crash: A Block-Adaptive-Mesh Code for Radiative Shock Hydrodynamics - Implementation and Verification, Astrophys.J.Suppl. 194 (2011) 23. arXiv:1101.3758, doi:10.1088/0067-0049/194/2/23.

[58] J. P. Jessee, W. A. Fiveland, L. H. Howell, P. Colella, R. B. Pember, An Adaptive Mesh Refinement Algorithm for the Radiative Transport Equation, Journal of Computational Physics 139 (2) (1998) 380–398. doi:10.1006/jcph.1997.5870.

[59] M. Pernice, B. Philip, Solution of Equilibrium Radiation Diffusion Problems Using Implicit Adaptive Mesh Refinement, SIAM J. Sci. Comput. 27 (5) (2005) 1709–1726.

[60] R. W. Townson, X. Jia, Z. Tian, Y. J. Graves, S. Zavgorodni, S. B. Jiang, GPU-based Monte Carlo radiotherapy dose calculation using phase-space sources, Physics in Medicine and Biology 58 (12) (2013) 4341.
URL http://stacks.iop.org/0031-9155/58/i=12/a=4341

[61] J. Bédorf, E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, S. P. Zwart, 24.77 Pflops on a Gravitational Tree-code to Simulate the Milky Way Galaxy with 18600 GPUs, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 54–65. doi:10.1109/SC.2014.10.
URL http://dx.doi.org/10.1109/SC.2014.10

[62] A. Gray, K. Stratford, Ludwig: multiple GPUs for a complex fluid lattice Boltzmann application, Chapman and Hall/CRC, 2013.

[63] D. Kunzman, Runtime support for object-based message-driven parallel applications on heterogeneous clusters, Ph.D. thesis, Dept. of Computer Science, University of Illinois, http://charm.cs.uiuc.edu/media/12-45/ (2012).

[64] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, G. Zheng, Programming petascale applications with Charm++ and AMPI, Petascale Computing: Algorithms and Applications 1 (2007) 421–441.

[65] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing Locality and Independence with Logical Regions, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 66:1–66:11.
URL http://dl.acm.org/citation.cfm?id=2388996.2389086

[66] E. Slaughter, W. Lee, S. Treichler, M. Bauer, A. Aiken, Regent: A High-productivity Programming Language for HPC with Logical Regions, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, 2015, pp. 81:1–81:12. doi:10.1145/2807591.2807629.
URL http://doi.acm.org/10.1145/2807591.2807629

[67] S. Treichler, Realm: Performance portability through composable asynchrony, Ph.D. thesis, Stanford, CA, USA, stanford University (2016).

[68] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: A Generic Distributed DAG Engine for High Performance Computing, Parallel Comput. 38 (1-2) (2012) 37–51. doi:10.1016/j.parco.2011.10.003.
URL http://dx.doi.org/10.1016/j.parco.2011.10.003

[69] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, Concurr. Comput. : Pract. Exper. 23 (2) (2011) 187–198. doi:10.1002/cpe.1631.
URL http://dx.doi.org/10.1002/cpe.1631