Automatic Halo Management for the Uintah GPU-Heterogeneous Asynchronous Many-Task Runtime

Brad Peterson[†], Alan Humphrey[†], Dan Sunderland[§],
James Sutherland[‡], Tony Saad[‡], Harish Dasari[†], Martin Berzins[†]
[†]Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, 84112, USA
[‡]Department of Chemical Engineering, University of Utah, Salt Lake City, UT, 84112, USA
[§]Sandia National Laboratories, PO Box 5800 / MS 1418, Albuquerque, NM, 87185, USA

***Abstract*—The Uintah computational framework is used for the parallel solution of partial differential equations on adaptive mesh refinement grids using modern supercomputers. Uintah is structured with an application layer and a separate runtime system. Uintah is based on a distributed directed acyclic graph (DAG) of computational tasks, with a task scheduler that efficiently schedules and executes these tasks on both CPU cores and on-node accelerators. The runtime system identifies task dependencies, creates a task graph prior to the execution of these tasks, automatically generates MPI message tags, and automatically performs halo transfers for simulation variables. Automating halo transfers in a heterogeneous environment poses significant challenges when tasks compute within a few milliseconds, as runtime overhead affects wall time execution, or when simulation variables require large halos spanning most or all of the computational domain, as task dependencies become expensive to process. These challenges are magnified at production scale when application developers require each compute node perform thousands of different halo transfers among thousands simulation variables. The principal contribution of this work is to (1) identify and address inefficiencies that arise when mapping tasks onto the GPU in the presence of automated halo transfers, (2) implement new schemes to reduce runtime system overhead, (3) minimize application developer involvement with the runtime, and (4) show overhead reduction results from these improvements.**

***Index Terms*—Uintah, hybrid parallelism, parallel, GPU, heterogeneous systems, stencil computation, optimization, concurrency, halo transfer**

I. Introduction

With energy efficiency being a key component in exascale initiatives, namely the 20 MW aspiration for exascale power consumption set by entities like the DOE, supercomputers are now heavily leveraging accelerator and coprocessor-based architectures to meet these power requirements. These heterogeneous systems pose significant challenges in terms of developing software for computational frameworks like the open-source Uintah framework [1], that seek to utilize available accelerators such as graphics processing units (GPUs).

The design of Uintah maintains a clear separation from the application layer and its runtime system, allowing application developers to only be concerned with solving the partial differential equations on a local set of block-structured adaptive mesh patches, without worrying about the runtime details such as automatic MPI message generation, management of halo data and the life cycle of data variables, or indeed any details with the multiple levels of parallelization inherent to these heterogeneous systems. Furthermore, the public API exposed to application developers should remain simple, shielding them from the complex details involved with the parallel programming required on these systems.

Automatic halo processing is a productivity necessity for many simulations utilizing Uintah. Each compute node may have over a thousand computational tasks, thousands of simulation variables, and thousands of halo transfers both intranode and internode. Productivity would be lost if application developers were involved with complicated runtime specific decisions for each halo transfer. Prior work [2] describes an initial design for managing GPU tasks alongside CPU tasks within Uintah. The prior runtime targeted simulations utilizing long-lived GPU tasks (on the order of seconds or minutes) requiring extensive halo dependencies among variables in most or all of the computational domain. For these simulations, it was most efficient to perform all halo logic entirely in host memory. However, for tasks that compute within a few milliseconds, the overhead to prepare the tasks is far larger than their time to compute. The focus of this work is to significantly improve time to solution for simulations containing these short-lived tasks while also supporting other simulations requiring thousands of automatic halo transfers per node. The Wasatch (Section VIII-B) component and reverse Monte Carlo Ray Tracing *(RMCRT)* component (Section VIII-C) in particular are motivations for these runtime improvements.

The extension of the GPU approach in the initial design [2] requires that data remain on the GPU for as long as possible to avoid data movement across any data bus or network. This, in turn, requires that some halo data management occur on the GPU, whether the halo data arrive from (1) other nodes, (2) host memory, (3) within the GPU, or (4) from another GPU on the same node. Similarly, if a task requires high numbers of upcoming GPU memory allocations, this should also be processed in as few API calls as possible. These challenges must be balanced alongside CPU tasks, so that a mixture of GPU and CPU tasks can be used for a computation, allowing each type of task to process where it is most efficient.

This paper describes enhancements and optimizations to the

Uintah runtime system that go well beyond the initial support for GPU tasks in the original design [2]. This paper is an extended form of the conference paper [3]. The extensions beyond the conference paper are contained in Section V, the first two-thirds of Section VI, and a revised related work Section IX. New results in Section VIII show the importance of this work. We provide results demonstrating significant reduction in GPU overhead, allowing tasks where speedups over the CPU version were previously unattainable to now outperform their CPU counterparts. In particular, six optimizations are covered, (1) persistence of GPU data (Section III), (2) managing halo cell scenario in many memory spaces (Section III-B), (3) eliminating data store contention and reducing the size of Uintah's GPU data store (Section IV), (4) managing several concurrency scenarios present with parallel scheduler threads (Section V), (5) additional GPU-specific work queues needed within the task scheduler (Section VI), and (6) allocating all data variables in one contiguous memory buffer instead of several (Section VII). These optimizations can use, but are not dependent on, CUDA paged memory or specific tools such as CUDA Unified Memory or CUDA-aware MPI. These optimizations can be adapted to other multi-tiered memory structures, such as additional NVRAM, by providing a framework to allow data to be managed in both high bandwidth and high capacity memory.

We begin by giving an overview of the Uintah framework in Section II. Section III details work done to enable persistence of simulation data on GPUs and how this work has enabled management of multiple, difficult halo data scenarios. Section IV describes changes for how GPU tasks obtain variable data from a data store. Section V covers concurrency logic added to Uintah to manage memory in a heterogeneous parallel environment. Section VI describes changes to the task scheduler queues. Our approach to minimizing GPU API call latency is covered in Section VII. Nodal results from these improvements are shown in Section VIII. Related works are given in Section IX. The paper concludes in Section X with a discussion on future work.

II. Uintah Overview

The open source Uintah framework [1], [5] is used to solve problems involving fluids, solids, combined fluid-structure interaction problems, and turbulent combustion on multi-core and accelerator based supercomputer architectures. Problems are either initially laid out on a structured grid and over-decomposed into hexahedral blocks of cells *(patches)* [6] with the multi-material ICE code for both low and high-speed compressible flows, or by using particles on that grid [7] with the multi-material, particle-based code MPM for structural mechanics. Uintah also provides the combined fluid-structure interaction (FSI) algorithm MPM-ICE [8], the ARCHES turbulent reacting CFD component [9] designed for simulating turbulent reacting flows with participating media radiation, and Wasatch [10], a general multiphysics solver for turbulent reacting flows.

A. Data Stores

Simulation data is managed by a distributed data store known as a *Data Warehouse*, an object containing metadata for simulation variables. Actual variable data itself is not stored directly in a Data Warehouse, it is instead stored in separate allocated memory which the Data Warehouse manages. The metadata indicates the patches on which specific variable data resides (hereafter referred to as *grid variables*), halo depth or number of halo cell layers, a pointer to the actual data, and the data type (node-centered, face-centered, etc.). Access to simulation data in the Data Warehouse is through a simple *get* and *put* interface. During a given time step, there are generally two Data Warehouses available to the simulation, (1) the *Old Data Warehouse*, which contains all data from the previous time step, and (2) the *New Data Warehouse*, which maintains grid variables to be initially computed or subsequently modified. At the end of a time step, the *New Data Warehouse* is moved to the *Old Data Warehouse*, and another *New Data Warehouse* is created.

With the availability of on-node GPUs, Data Warehouses specific to GPUs are used. Each GPU is assigned its own Old and New Data Warehouse, analogous to the host-side's Data Warehouses. A GPU Data Warehouse contains a reduced set of metadata, and manages only the simulation variables the GPU task will need for a task computation. Through knowledge of the task graph, the Uintah runtime system is able to prepare and stage the GPU Data Warehouses and copy the metadata into GPU memory prior to task execution.

Application developers can utilize existing Data Warehouse API if they choose. An example of this is shown in Section VIII, where we focus on a simple seven-point stencil for the Poisson equation in 3D. In this task data is retrieved from and placed into Data Warehouse objects using simple get and put methods. Other application developers will take existing computational programs originally not written for Uintah, and create tasks with function pointers to their existing code. An example of this is the Wasatch [10] component, a finite volume computational fluid dynamics code that is designed to solve transient, turbulent, reacting flow problems. Uintah itself needs no knowledge of how the Wasatch tasks work, other than the simulation variables used for each task.

B. Task Scheduling and Execution

Uintah task schedulers are responsible for scheduling and executing both CPU and GPU tasks, memory management of grid variables, and invoking MPI communication. There are several task schedulers available within Uintah. In this work, we focus on the *Unified Scheduler* [4], shown in Fig. 1. This scheduler uses a fully decentralized approach without a control thread. All CPU threads are able to obtain work as well as process their own MPI sends and receives. All CPU threads prepare, schedule, and execute CPU and GPU tasks with an arbitrary number of CPU cores and on-node GPUs. All aspects of a GPU task are processed asynchronously, so that a CPU thread can process other tasks while work is occurring on a GPU. Through moving from an MPI-only approach to a nodal shared memory model [11] (a combination of MPI and

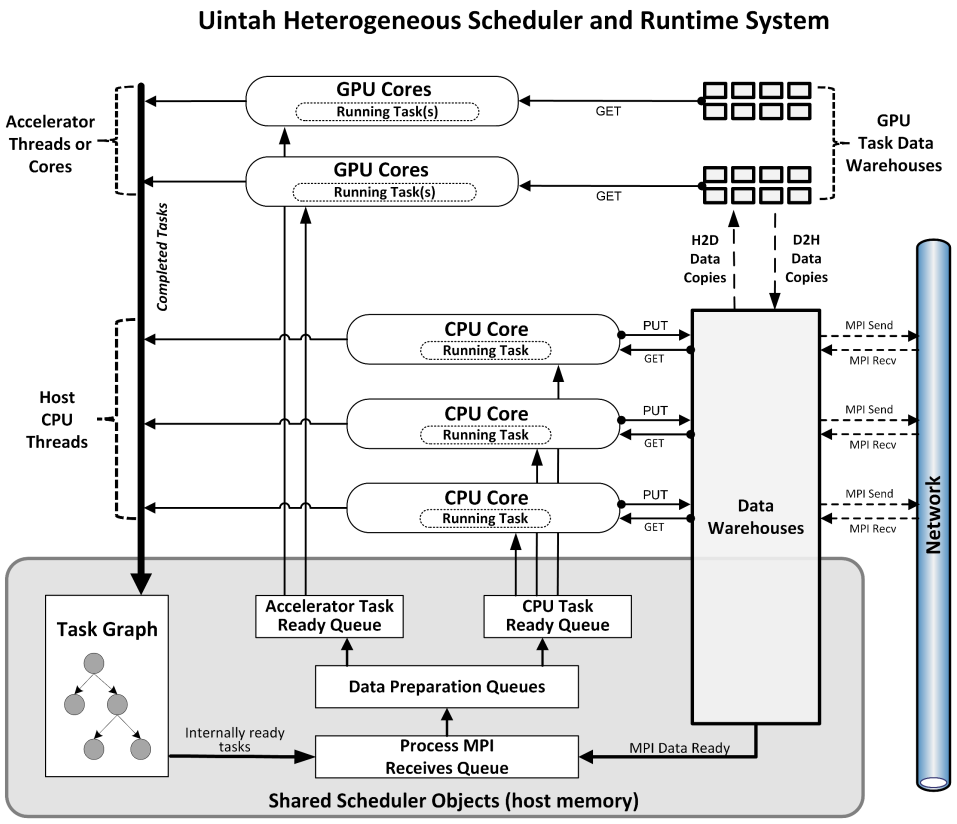

Fig. 1. Uintah heterogeneous nodal runtime system and task scheduler. [4]

Pthreads) where each node has one MPI process and threads execute individual tasks, the Uintah framework has been made to efficiently scale to hundreds of thousands of cores solving a broad class of complex engineering problems [12].

Parallelism within Uintah is achieved in three ways. First, by using domain decomposition to assign each MPI rank its own region of the computational domain, e.g. a set of hexahedral patches, usually with spatial contiguity. Secondly, by using task level parallelism within an MPI rank to allow each task to run independently on a CPU (or Xeon Phi) core or available GPU, and third, by utilizing thread level parallelism within a GPU. Work toward thread-level parallelism for the Xeon Phi is currently underway, and will be based on the idea of a task worker pool, or group of CPU threads that cooperatively execute a single task.

C. Application Developer Interaction with the Runtime

Uintah maintains a clear separation between an application's code and its runtime system and hence the details of the parallelism Uintah provides through its runtime system are hidden from the developer and a task itself. In the application layer, an application developer is responsible for providing task parameters to the runtime system. An example of a 27-point stencil task declaration is given in Fig. 2. The developer lists all grid variables that will be used in the task computation, and indicates whether these grid variables come from the Old or New Data Warehouse. Each grid variable is assigned as *Computes*, *Modifies*, or *Requires*. *Computes* are grid variables

```
void Stencil27::scheduleTimeAdvance(const Level& level,
                                    Scheduler& sched)
{
  Task* task = new Task("Stencil27 method",
                        this, &Stencil27::taskMethod);
  task->requires(OLD_DATA_WAREHOUSE, gridVariable,
                 Ghost::AroundNodes, 1);
  task->computes(gridVariable);
  task->usesDevice(true);
  sched->addTask(task, level->everyPatch());
}
```

Fig. 2. A Uintah task declaration which informs the runtime of a 27-point GPU stencil computation on a single Uintah grid variable.

to be allocated by the runtime system to hold data computed by the task. *Modifies* are grid variables which were previously computed and will be modified by the task. *Requires* are read-only grid variables computed in a prior task. The number of needed halo cells layers for any *Requires* is also indicated. The developer specifies whether the task is a GPU task or a CPU task. Once a task is declared, the application developer should not have to worry about the details of memory management. That developer can write task code assuming the runtime system will have prepared all grid variables' memory, including gathering halo cell data.

All needed halo dependency logic can be gleaned from analyzing the task graph's dependencies. For example, for the task listed in Fig. 2, the Uintah runtime will create one task for every patch on the computational domain. Next, the runtime will identify all simulation variable dependencies among those

tasks. Suppose a patch is surrounded by 26 other patches in the structured grid, then since the simulation variable in the task requires a layer of halo cells, Uintah will generate 26 inbound dependencies and 26 outbound dependencies. If any of these tasks are on different nodes, Uintah will automatically generate and issue MPI sends and receives.

A single task dependency represents one halo region that must be copied from a source grid variable to a destination grid variable. Uintah does not create one task per dependency, rather, a single task can have many (up to thousands) of halo dependencies with other tasks. By not treating each dependency as a separate task allows Uintah to use far fewer tasks to keep the task graph size small for faster analysis. This approach of limiting the amount of tasks is crucial for Uintah to scale to hundreds of thousands of cores for production scale problems containing global halo requirements [13], where tasks on each patch have halo dependencies with all other patches in the computational domain. While Uintah operates with a static task graph, halo dependencies could be added dynamically during a timestep as the task scheduler processes tasks dynamically prior to task execution.

Before a task executes, the Uintah scheduler and data warehouse automatically ensures all halo data is gathered into the simulation variables. Productivity is achieved by allowing the application developer to quickly define all halo dependencies through two simple arguments. For example, in Fig. 2, had the application developer not specified *Ghost::AroundNodes, 1* for one cell layer of halo dependencies, but rather *Ghost::AroundNodes, 100* or *Ghost::AroundNodes, 32767*, Uintah would automatically handle all halo management on a nearly global or global scale. What separates Uintah and the work in this paper from other GPU-enabled runtimes like Legion [14], Charm++ [15], StarPU [16], and PaRSEC [17] is that Uintah provides a combination of three features: (1) minimal application developer interaction to define halo dependencies while allowing the runtime to completely automate all data movement and halo processing, (2) support for a mixture of problems containing both local halo and global or nearly global halos and across multiple adaptive mesh refinement levels, and (3) ability to reach full scale on current machines like Titan [18] and Mira [19]. In production scale multiphysics problems with thousands of simulation variables, thousands of computational tasks per node, and potentially hundreds of thousands of simulation variable dependencies per node [13], it is vital the Uintah runtime assume maximum responsibilities for all halo dependency analysis and halo transfers to aid application developer productivity. support When a task is executed, Uintah executes the user supplied task entry function. Within the entry function an application developer typically writes serial C++ code for CPU and Xeon Phi tasks and CUDA parallel code for GPU tasks. That entry function could utilize other parallel tools such as OpenCL [20], OpenACC [21], OpenMP [22], Raja [23], or Kokkos [24]. Uintah application developers have not used OpenCL as its performance often lags behind CUDA code [25]. OpenACC has not been used as doesn't fully support Xeon Phi KNL vectorization. OpenMP 4.0 introduced GPU support, but current compiler implementations are limited and lacking in GPU performance [26]. Raja and Kokkos share high level similarities in utilizing lambda expressions for portability and performance with minimal disruptions to application developers. Future Uintah work is focused on utilizing Kokkos as its current feature set is more extensive and mature. Regarding memory management, CUDA offers compelling features such as CUDA-aware MPI and Unified Memory to reduce the amount of temporary halo buffers and provides automatic memory movement between host and GPU memory. Uintah does not restrict itself to only CUDA-aware MPI implementations, and Uintah provides automatic packed halo buffers which can then be made to work with GPUDirect if needed. Uintah does not use Unified Memory as CUDA kernels operating in a Unified Memory environment demonstrate significantly slower execution times [27], and any GPU-to-host memory transfer requires a synchronization barrier prior to CUDA Compute Capability 6.x or expensive page faulting for Compute Capability 6.x [28]. We desire performant kernels executing concurrently on numerous streams without any synchronization and as a result we use the Uintah runtime to automate halo transfers and data movement between host and GPU memory without blocking operations.

D. Prior Runtime and Motivations for This Work

Prior work [2] targeted a GPU-based reverse Monte Carlo Ray Tracing (*RMCRT*) simulation, which requires replication of radiative properties among nodes to facilitate local ray tracing. The application developer informed the runtime of the required data replication by specifying very large halos around radiative data variables. The runtime used these large halos to automatically scatter and gather radiative halo data among up to thousands of nodes. All halo logic occurred in host memory for four reasons: (1) it required minimal additional code to debug and maintain, (2) it allowed for quick development to support GPU-enabled tasks with large halos, (3) the host memory halo logic had been proven to scale to 256K cores, and (4) it was more efficient to gather halo data in host memory and then send the simulation variable to GPU memory in a single host-to-GPU copy, rather than using many GPU-to-GPU copies, incurring fewer API latency costs.

The prior runtime had deficiencies. For short-lived GPU tasks with simulation variables requiring small halo regions, performing halo transfers in host memory was prohibitively expensive. Another problem was limited patch over-decomposition options. The prior runtime could only sequentially execute GPU tasks, and so patch sizes were chosen to fill all GPU streaming multiprocessors. We desired a solution that kept simulation variables persistent in GPU memory, performed halo transfers in GPU memory, and allowed for concurrently executing GPU kernels for more over-decomposition options while keeping all GPU streaming multiprocessors full.

One target application in particular is the Wasatch component. Wasatch utilizes many short-lived GPU tasks with small halo regions around simulation variables. Wasatch employs a formalism of the DAG approach to generate runtime algorithms [29], and an Embedded Domain Specific Language (EDSL) called Nebo [30], [31]. Nebo allows Wasatch developers to write high-level, Matlab-like syntax that can be

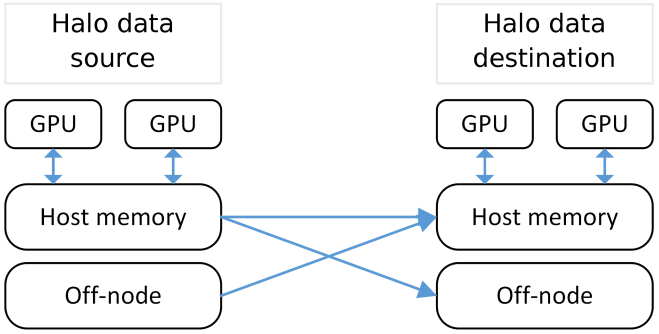

Fig. 3. Uintah's initial runtime system to prepare CPU and GPU task grid variables with halo data from adjacent grid variables.

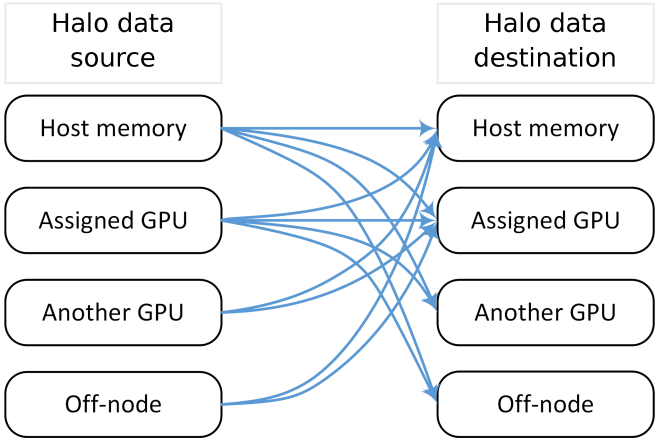

Fig. 4. Uintah's current runtime system to prepare CPU and GPU task grid variables with halo data from adjacent grid variables.

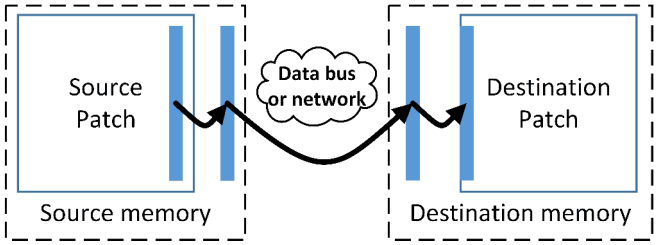

Fig. 5. Halo data moving from one memory location to another are first copied into a contiguous staging array prior to being copied to that memory location. Later a task on the destination will process the staging array back into the grid variable.

executed on multiple architectures such as CPUs and GPUs. RMCRT also remains a target application to verify the new runtime efficiently handles large halos. In this work, RMCRT simulations can now take advantage of concurrently executing kernels.

III. Persistence of GPU Data and Managing Halo Cells

The Uintah runtime system now allows data to exist only in host memory, or only in GPU memory, or both. For tasks employing pointwise computations, this design easily allows application developers to make grid variable data persistent on GPUs. However, for tasks requiring halo data, data persistence does not provide any automatic halo management in GPU memory. This section describes how Uintah now processes halo data in an environment with many memory spaces.

A. Prior Runtime Model

Only three halo data copy scenarios were used in the prior runtime (see Fig. 3). All halo data management was handled in host memory then copied back into GPUs. While functional and simple, this approach heavily utilized the PCIe bus.

For example, suppose each MPI rank is assigned a $4 \times 4 \times 4$ set of patches, and each patch contains $64 \times 64 \times 64$ cells. Also suppose this simulation has only one grid variable for stencil computations, the grid variable exists in GPU memory, the grid variable holds double values, and 1 layer of halo cells are required for all MPI rank neighbors. In this model, an MPI rank can require 56 of the 64 patches to be copied to host, and halo data sent out. Then the MPI rank would receive halo data data for 56 patches, process them in host memory, and then copy them into the GPU. Assuming a bus bandwidth of 8 $\mathrm{GB/s}$, the data transfer time alone for this one grid variable into host memory would be roughly 14 ms and another 14 ms to copy it back into the GPU. For many Uintah GPU tasks which compute within a few milliseconds, this is impractical.

B. Current Runtime Model

With data staying persistent in GPUs, more halo data scenarios must be managed. Uintah must prepare grid variables for both CPU tasks and GPU tasks by obtaining halo data from grid variables in adjacent patches (adjacent grid variables) in whatever memory location they exist. These adjacent grid variables can exist in four memory locations, (1) host memory, (2) GPU memory, (3) another on-node GPU's memory, or (4) off-node. With four possible source locations and four possible destination locations, there are 16 possible halo data copy scenarios. Because a task does not manage halo data for patches or nodes it is not assigned to, this number is reduced to 12 scenarios, as shown in Fig. 4. While it is possible to reduce these 12 into fewer scenarios by employing more MPI ranks per physical node, Uintah's runtime system obtains its performance by allowing a physical node to function in only one MPI rank. If NVLink [32] is considered, the number of halo data copy scenarios will then increase by needing to determine which data bus to use.

Instead of writing specific code for each of the 12 scenarios, the process can be simplified by batching together all source/outgoing halo data copies into a collection of staging grid variables, and then later batching all destination/incoming halo data copies into another collection of staging grid variables.

As shown in Fig. 4, halo copies now occur entirely within the same memory space if the source and destination grid variable reside in the same space. For example, two adjacent grid variables in GPU memory can simply copy their halo data to each other. Otherwise if the halo data must be copied to another memory location, then contiguous arrays with packed data are employed as shown in Fig. 5. Packed buffers are used

because these simulation variables contain non-contiguous halo data in memory. Host-to-GPU and GPU-to-GPU transfers within an MPI rank are simplified and made more efficient, as the cost of many individual transfers between memory spaces is almost always far greater than the cost of creating a packed buffer and sending that buffer. For halo transfers using MPI, packed buffers copied GPU-to-host allow Uintah to use any MPI implementation. Further, packed buffers in GPU memory provide the potential of performance equal to or exceeding non-contiguous data structure solutions provided by CUDA-aware MPI implentations [33].

In the prior runtime system example of a node containing a $4 \times 4 \times 4$ set of patches, a minimum of 14 ms was required simply to transfer the data GPU-to-host over the PCIe bus. This new approach has been implemented in Uintah's runtime system. Profiling this particular problem results in combined transfer and processing times of roughly 1 to 2 ms.

A benefit of this halo cell management is that automated halo copies into GPU memory was merged with the scheduler code as described in Section VI which is responsible for copying simulation variables into GPU memory. So whether only halo cells needs to be copied across the PCIe bus, or a regular grid variable without halo cells needs to be copied, the scheduler treats both the same. These halo cells can all be processed in batches, rather than one at a time, so that if a GPU task requires N grid variables each needing halo cells, then all N can be processed together. This batching enables efficiency gains in two ways, (1) through runtime allocations of contiguous regions as detailed in Section VII, and (2) through utilizing kernel calls to perform GPU-to-GPU halo processing as detailed in the remainder of this section.

1) Batching Example: Suppose a 27-point stencil task requires that a particular GPU data grid variable send its halo cell data to its 26 neighbors, and then receive halo cell data from those same 26 neighbors. Of these 26 neighbors, suppose 11 are found within that GPU, 6 are found within another on-node GPU, and 9 are off-node. This data grid variable will then be assigned a collection of 15 staging regions (6 for the on-node GPU and 9 for off-node), each of which are contiguous arrays. A kernel will be called to perform 15 halo cell copies within that GPU. After the kernel completes, the runtime system identifies those 6 dependencies which belong to another GPU, and so 6 GPU peer-to-peer copies are invoked. The runtime then identifies those 9 dependencies that belong off-node, and MPI is used to send this data as necessary. Once all data is sent out, it is the responsibility of the scheduler processing future tasks to gather these halo cells back into data grid variables.

When a future task needs to use this same grid variable with halo cells, the runtime will recognize that it will need to gather together the halo cells from 26 neighbor patches. It will look in the node's own Data Warehouses and find that halo cells for all 26 exist in various memory locations on that node. It will then process these in bulk and prepare the GPU data grid variable for GPU task execution.

2) Batching Analysis: Batching all halo groups for later processing incurs an overhead cost by delaying halo copies that could otherwise be launched immediately. However, the cost of a launching a single kernel for one batch of halo groups may be preferable to the cost of invoking many smaller kernels or copies. The cost of processing halo data for a Uintah task can be described as $t_{total} = a * t_a + n * t_c$, where a is the number of invoked API actions, t_a is the CUDA API latency to issue a kernel or copy call, n is the number of groups of halo data copies, and t_c is the time required to copy all halo items in a group. For the upcoming measurements, we varied the number of items in a halo group between 16^2 and 128^2 cells and used a machine with an Nvidia K20c GPU and an Intel Xeon E5-2620.

Three halo processing approaches were tested: (1) a single streamed CUDA kernel with code to perform these copies, (2) multiple streamed CUDA kernels with each performing some of the needed copies, and (3) multiple GPU-to-GPU copy calls issued from CPU code. For the first two tests (utilizing CUDA kernels), we observed a kernel launch latency (t_a) of ∼4-5 μs, and time to copy all items in a group (t_c) of ∼1-3 μs. For the third test (multiple GPU-to-GPU API calls isused from CPU code), we observed a copy call latency (t_a) of ∼5-6 μs, and the time per copy (t_c) of less than 1 μs. Unfortunately, for grid variables, not all halo groups are contiguous, with some halo groups requiring cell-by-cell copies as there are no halo cells occupying contiguous memory regions among them. The resulting total API call latency required for these cell-by-cell copies is orders of magnitude worse than the first two approaches and will no longer be analyzed.

Determining whether Uintah should issue halo copies immediately or batch and launch them all as a group is dependent on the length of time the runtime analyzes and adds a group to a batch. Currently the Uintah runtime requires ∼1-3 μs of analysis per halo group to possibly add it into an upcoming batch. Because the kernel latency of ∼4-5 μs is greater than the time required to analyze a single halo group, the runtime should not issue one streamed kernel per halo group as this will always lead to greater halo copy overhead. For these reasons, Uintah's runtime creates and processes only one batch. A possible alternative approach has the runtime utilizing multiple batches total, where a batch begins copying halo data when several groups are queued and several more groups remain to be analyzed. However, this would cut into the efficiency gains of contiguous allocation blocks described in Section VII, as more batches would require more allocation blocks. For current production problems utilizing Uintah, the overhead costs involved with batching only one set of halo groups is minor in comparison to task execution times, and so further optimizing this batching process is not a high priority and left as future work.

IV. GPU Data Warehouse Modifications

The Uintah GPU Data Warehouse is a data store containing metadata for simulation variables in GPU memory. The GPU Data Warehouse itself is entirely contained within an object, updated in host memory, and copied into GPU memory. It aids productivity by allowing application developers to easily retrieve task simulation variables within CUDA code without having to manually pass in each simulation variable's data

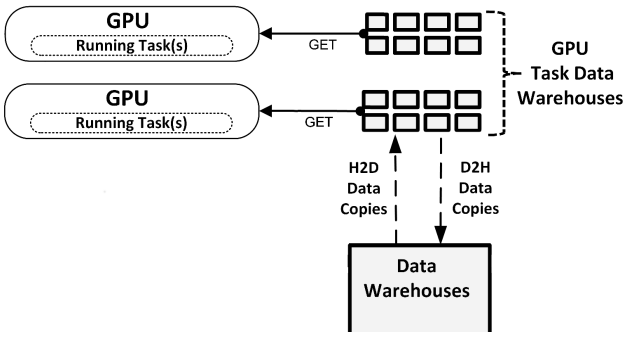

Fig. 6. Each GPU task gets data into its own small Task Data Warehouses, rather than the old approach of all GPU tasks sharing the same large GPU Data Warehouses.

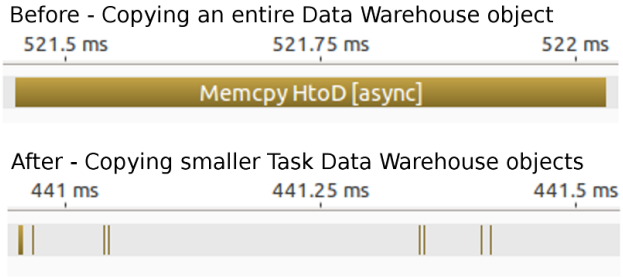

Fig. 7. A profiled half millisecond range of an eight patch simulation showing Data Warehouse copies. Before, the initial runtime system had many large Data Warehouse copies (only one shown in this figure). After, the new runtime system's small Task Data Warehouses copy into GPU memory quicker, allowing GPU tasks to begin executing sooner (eight Data Warehouse copies are shown in this figure).

pointer into a kernel. It also aids the runtime by tracking which simulation variables exist in GPU memory. Previously, the GPU Data Warehouse only operated with sequential kernels and could not function correctly with concurrent kernels. This section describes modifications required for concurrent kernel execution, while also using these data stores to aid in halo gathering entirely within GPU memory.

A. Concurrent GPU Data Warehouses

The prior GPU Data Warehouse model [2] did not function with concurrently executing GPU tasks because the Uintah scheduler updated the GPU Data Warehouse prior to each task executing. The GPU Data Warehouse in GPU memory could not be updated when another task was currently executing and using it. A second problem was the GPU Data Warehouse memory footprint was relatively large, on the order of a few megabytes, due to needing larger fixed-sized array buffers. For GPU tasks that computed within a few milliseconds, the time to copy the GPU Data Warehouse into the GPU was unacceptably large. A third problem related to productivity occurred when an application developer retrieved simulation variables from the data warehouse object despite the task definition not explicitly indicating it would use the variable. Depending on task execution order, those simulations variables may often, but not always, be found in GPU memory, leading to inconsistent behavior.

Task Data Warehouses were created in order to solve these three problems. The driving concept of Task Data Warehouses is that each GPU task receives its own self contained GPU Data Warehouse objects in GPU memory, wholly independent and not used by other tasks, with only the information it needs to manage halo data copies and grid variables for task computation (see Fig. 6). These small Task Data Warehouses in GPU memory serve as read-only snapshots of a subset of the GPU Data Warehouse. A GPU task kernel will then have no knowledge or capability to access grid variables unrelated to its own task. This eliminates coordination and contention issues related to tasks previously sharing a GPU Data Warehouse in GPU memory. This also results in having the full GPU Data Warehouse only existing in host memory, as it is never copied in full into GPU memory as one large object.

B. Utilizing Task Data Warehouses for Halo Gathering

Keeping simulation data persistent in GPU memory required a mechanism to allow halo cell gathering within GPU memory using simulation variables containing non-contiguous halo data (e.g. 3D faces of a grid variable). CUDA Unified Memory was not used because as described previously in Section II, Unified Memory frequently incurred blocking operations and kernel performance reductions. CUDA-aware MPI was not used as most halo gathers would occur within an MPI rank. We utilized a common method to pack and unpack halo buffers [33] using a CUDA kernel in GPU memory.

The Task Data Warehouse object already contained data address and layout information, and so the metadata containing the logic to process these halo transfers in a CUDA kernel was also placed within a Task Data Warehouse object. The scheduler thread responsible for preparing a task's simulation variables prior to execution is also responsible for preparing this metadata. Before a GPU task executes, a CUDA kernel is invoked which reads the Task Data Warehouse for halo copying metadata and copies these halo buffers back into the simulation variables.

Storing both simulation variables and halo copying metadata in a Task Data Warehouse required a change of the internal structure of the data store object. Originally the data store object used fixed sized arrays within the object so as to avoid multiple deep copies. But as we discovered some simulations required far more simulation variables than others, we likewise noticed the GPU Data Warehouse's memory footprint was becoming too large. Adding an additional array containing halo copying metadata within this data store object made the memory footprint even larger. While employing object deep copies was tempting at this point, a more efficient approach was found.

Our solution merged both the array for simulation variables and the array for halo copy logic into one array. This results in a compact Task Data Warehouse object consisting of a few data members followed by one array. Because it is a serialized object in memory, only one copy into GPU memory is required. The end result of these all structural improvements is that each task no longer requires copies of GPU Data Warehouse objects that were megabytes in size. Now they are

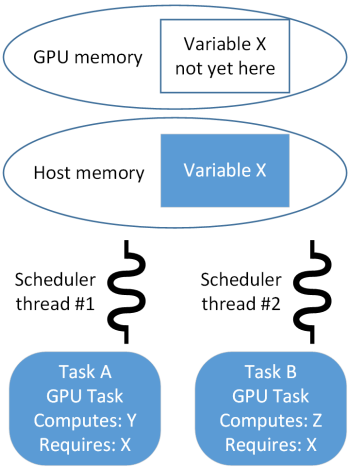

Fig. 8. Two scheduler threads are each assigned a different task to analyze. They do so independently in parallel. Each sees that simulation grid variable X is not yet in GPU memory. The runtime must determine which thread performs the data copy.

only a few kilobytes in size. Fig. 7 shows the improvements of this approach, demonstrating how more Data Warehouse objects can be copied into GPU memory in less time.

V. Simulation Variable Concurrency and Life Cycles

Multiple task scheduler threads concurrently prepare simulation variables for upcoming tasks. Two or more tasks may require the same simulation variable and possibly also require halo data be gathered for a simulation variable. The initial GPU engine [2] targeted problems where tasks did not share simulation variables. Previously it sufficed to simply give each grid variable a boolean value to indicate if a grid variable was valid in host memory and if it was valid in GPU memory. As we have expanded the runtime to more simulations where tasks shared simulation variables, finer control over a grid variable's memory status was required. In this section we describe a solution which is designed to scale to compute nodes with many additional memory hierarchies and memory spaces.

A. Task Scenarios Requiring Scheduler Coordination

Two examples below highlight how simulation variables and possibly halo data may be shared among tasks. For the simplest example, refer to Fig. 8. Suppose a simulation computes grid variable X in host memory on timestep 1. On timestep 2, suppose that grid variable X's status is redefined as a (read-only) *Requires* grid variable, as described in Section III. The task graph has two tasks, task A and task B, which compute on the GPU and requires timestep 1's grid variable X for the computation.

Next, consider an example involving gathering halo cells. Suppose grid variable X was computed in timestep 2 in GPU memory. In timestep 3, tasks C and D need to perform stencil computations using timestep 2's grid variable X. Here Uintah will supply the halo cell data from spatially neighboring patches and place that halo data in GPU memory. However this halo cell data must still be gathered into grid variable X. The scheduler again prepares tasks C and D in parallel on two CPU threads, and notices that grid variable X's halo cell data is not yet prepared. A race condition occurs when both CPU threads are given the responsibility to perform this ghost cell gather step.

These examples demonstrate a need for different task scheduler threads to utilize concurrent solutions. Overall, three types of actions must be considered within Uintah: (1) allocations of simulation variables, (2) copies from one memory region location to another, and (3) gathering in halo cells. These are not limited to simulation variables in GPU memory space, they can occur in any space where tasks utilize simulation variables, such as host memory, NVRAM, and hard disk space.

1) First Attempted Solution Using Duplication: We attempted to preserve what had been a core philosophy of Uintah's schedulers, treating scheduler threads and tasks as fully independent so that no two scheduler threads had to coordinate with one another. This core philosophy required less runtime code and quicker development by avoiding coherency scenarios altogether, and we considered extending it into GPU memory. We proposed allowing multiple copies of data to exist in a memory location. For example, if both tasks A and B need grid variable X from the previous time step to be copied into GPU memory, then two instances of the same grid variable X would be created and copied in GPU memory, with each task gaining ownership over one of these. This approach however created new problems. There would be noticeable overhead when needing to copy multiple instances of grid variables with large patch sizes. Furthermore, if two tasks need halo data to be gathered simultaneously, performing this action twice would again add noticeable overhead. Also, task launch times could be delayed as now these tasks must always wait until their data is prepared in the needed memory location, whereas if two or more parallel tasks shared grid variable data then any task can all proceed the moment the shared data is ready. For these reasons, this approach was not implemented.

2) Implemented Solution: The runtime requires that scheduler threads preparing tasks should coordinate with one another. Our solution adopts a first-touch policy. The first scheduler thread to recognize a necessary preparation action for a simulation variable (such as a halo gather) is the thread which will soon perform that action. If other scheduler threads recognize that action is in progress and not completed, that task is placed back in a work queue for later processing. For example, if both tasks A and B need grid variable X in GPU memory prepared with 2 layers of halo cells, then as two scheduler threads each prepare these tasks, whichever thread recognizes this need first becomes the one to perform the halo gather. Similar first-touch policies are introduced for memory allocation and memory copies. In the case of halo gathering, no scheduler thread is allowed to claim ownership of gathering halo cells for a grid variable until all adjacent halo cell information has been received and is available in that memory space.

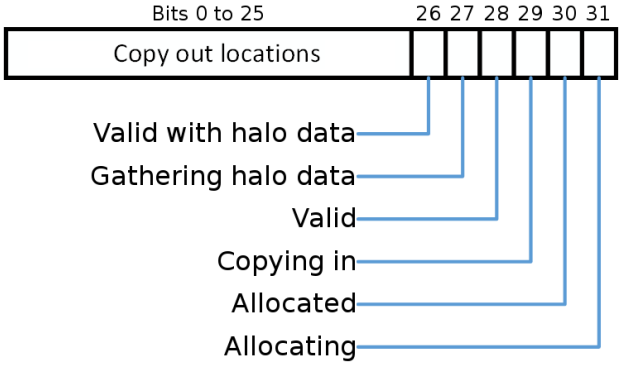

Fig. 9. A bit set layout for the status of any simulation grid variable in a memory location. Every simulation grid variable in every memory space contains a bit set. Reads and writes to this bit set are handled through atomic operations.

B. Simulation Variable Status Bit Set

To accomplish first-touch coordination between scheduler threads, a 32-bit atomic bit set is used (see Fig. 9). If a simulation has two memory spaces (such as host memory and GPU memory), then each grid variable is assigned two of these bit sets. A bit set has six assigned bits for *allocating*, *allocated*, *copying in*, *valid*, *gathering halo data*, and *valid with halo data*. Again consider the example described in Fig. 8 of both tasks A and B needing grid variable X copied from host memory into GPU memory. Each thread assigned to prepare these tasks can perform an atomic read to see if space for this grid variable in GPU memory has been allocated. Suppose the allocation bit was previously set. Then both threads can then see if the GPU grid variable's *copying in* bit or *valid* bit have been set. Suppose neither the *copying in* bit nor the *valid* bit have been set. Each thread attempts an atomic test and set on the *copying in* bit, and the winner must perform the GPU-to-host copy. The other thread continues on analyzing other grid variables needed for that task.

Prior to task execution, the scheduler now checks all grid variables assigned for that task to see if all *valid* or *valid with halo data* bits are set. If some bits are not yet set, then it must be the case that another task started but did not complete its assigned action. The task goes back into an appropriate queue and will be checked again shortly after. The scheduler thread is unaware which other scheduler threads are preparing necessary grid variables, only that a bit has been set indicating the desired action will be completed by another scheduler thread. These work queues are described in more detail in Section VI.

These atomic status bits also allow for better understanding of the state of simulation grid variables at any given time. For example, it is possible to set multiple bits so that a grid variable may be listed as both allocated and valid in host memory. It is also possible that a grid variable in GPU memory can be allocated, valid, and currently gathering halo data. If a grid variable does not exist in a memory space, then no bits are set. With 6 of the 32 bits reserved for these statuses, the other 26 bits can be used to indicate copy out destinations. Suppose a future compute node has multiple hierarchies of host RAM and multiple GPUs. Each memory location can be assigned an ID number corresponding to these bits. Suppose one grid variable is being copied from GPU #1 to GPU #2 and to high capacity host RAM, then the two bits representing those two destinations can be set in that grid variable's bit set. Doing this allows for greater control in case a grid variable needs to be vacated from GPU memory to make room for others. Before deallocation, the grid variable's bit set can be checked to ensure it is not currently being used as part of a data transfer.

The value of this approach is shown in Section VIII-C, in which an order-of-magnitude speedup is shown on a complex ray-tracing radiation calculation on up to 2K GPUs. We also report initial results showing a 4-5X speedup at 16K GPUs on a larger radiation calculation.

VI. Task Scheduler Enhancements

The Uintah Unified Scheduler [4] (see Fig. 1) functions by having all CPU threads independently checking shared priority queues for available tasks to process. Tasks proceed through these several work queues during its execution life cycle, similar to that of an execution pipeline. During this flow through these queues, grid variables are staged in the proper memory location and coupled to halo data if required. The overall flow of these queues has evolved through three distinct themes. This section will briefly describe the prior two themes and explain the current theme now in use.

The first Uintah GPU runtime system [2] targeted tasks with long execution times on the order of seconds to minutes. All halo management occurred in host memory, even for GPU tasks. Grid variables listed as *Requres* were copied into GPU memory, the GPU task kernel executed, and all computed data was copied back into host memory for possible future halo cell processing. This GPU enabled runtime was developed in relatively little time, utilized all host halo cell logic developed in years past, and easily allowed for utilization of multiple GPUs in a compute node.

The second Uintah GPU runtime system [3] targeted tasks with short execution times on the order of milliseconds. The overall theme changed to keep data persistent in GPU memory as long as possible, and to avoid as many host-to-GPU and GPU-to-host transfers as possible. A scheduler thread preparing a task was still responsible for staging *all* of that task's grid variables and gathering in all needed halo data, independent of any other scheduler thread. When a work queue's CUDA stream completed, the scheduler thread could know that phase of the task preparation was completed and could move onto the next work queue.

Implementing this second GPU runtime took considerable effort and required the creation of much smaller Task Data Warehouses (see Section IV), packing and unpacking halo buffers (see Section III), and managing multiple scenarios of copying memory from one memory location to another (see Fig. 4). The end result of these changes allowed for GPU vs. CPU speedups using Uintah for tasks which compute in a few milliseconds.

This work, the third GPU runtime, added additional concurrency checks, which targets simulations with many tasks

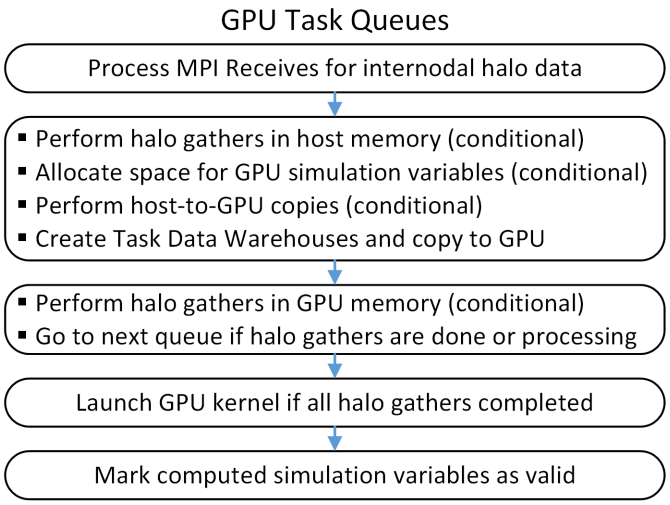

Fig. 10. A simplified flow of all scheduler queues a GPU task must pass through.

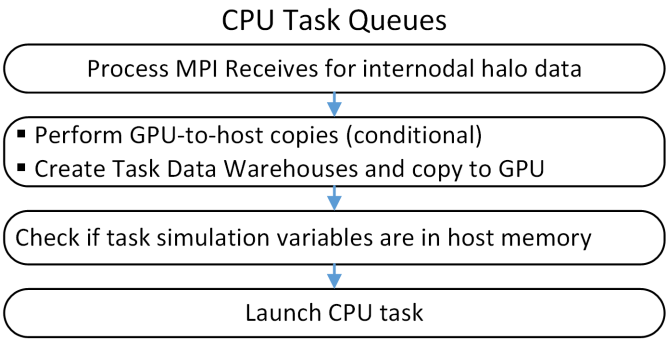

Fig. 11. A simplified flow of all scheduler queues a CPU task must pass through.

that share grid variables and also share halo requirements for simulation variables. Now a task is only responsible for staging grid variables and gathering halo data if it was the first arriver to claim that action. Furthermore preparation of grid variables is no longer the exclusive responsibility of that scheduler thread, but can be accomplished by many scheduler threads working with the same grid variables in parallel. A scheduler thread must now verify if a task's grid variables are ready before that task can move to the next queue.

A. GPU Task Queues

A GPU task now proceeds through five queues as shown in Fig. 10. They are (1) Process all MPI receives. (2) Manage any halo data gathers in host memory if possible. For all potential GPU allocations, copies, and/or halo data gathers necessary for this task, determine which of those becomes assigned to this task using an atomic first arriver policy (see Section V). Perform asynchronous host-to-GPU copies for the task's grid variables for which it is responsible to copy into the GPU. Create Task Data Warehouses (see Section IV) for this task which contains information about each grid variable and necessary meta data for later halo data gathers (see Section III). Asynchronously copy these Task Data Warehouses to the GPU. (3) Set every *valid* bit to true for all *Requires* grid variables this task was assigned to copy into the GPU. For every grid variable requiring halo data, see if all adjacent halo data is valid in GPU memory. If all needed grid variables can proceed with halo data gathering, and this task was assigned responsibility to gather that halo data, asynchronously launch a GPU kernel to complete this action. If halo data gathers cannot yet take place due some adjacent halo data not yet in this GPU, place this task back into this work queue to be checked later. (4) For every grid variable for which this task was responsible for gathering halo data, set those *valid with halo data* bits. Check if all grid variables requiring gathered halo data *valid with halo data* bits set. If not, place this task into this work queue to be checked later. If all grid variables are ready, asynchronously launch the GPU task. (5) Set every *valid* bit to true for all *Computes* and mark the task as done.

B. CPU Task Queues

Previously in the first GPU runtime model [2], the scheduler never required a CPU task to search any GPU memory space for a simulation variable. As a result, the task scheduler utilized only two queues. The first queue for processing MPI receives, and the second for executing tasks.

A CPU task now proceeds through four queues as shown in Fig. 11. They are (1) process all MPI receives. (2) For all potential host memory allocations and/or copies necessary for this task, determine which of those becomes assigned to this task using an atomic first-touch policy. Perform asynchronous GPU-to-host copies for all task grid variables for which it is responsible to copy into host memory. (3) Set every *valid* bit to true for all *Requires* grid variables this task was assigned to copy into host memory. If some task grid variables are not yet valid in host memory, place this task back into this work queue to be checked later. (4) Run the CPU task as all host data is available in host memory. Any needed halo data gathering happens during task execution.

C. Differences Between GPU and CPU Queues

The workflow for GPU queues and CPU queues work flow is fairly similar. A task's life cycle of data preparation, halo data management, execution, and updating of bit sets follows the same general concepts. The only major differences between CPU and GPU task queues are (1) GPU tasks need some form of task data warehouses due to difficulty of managing concurrency for grid variable data warehouses in global GPU memory. (2) GPU tasks are queued and executed asynchronously through streams while CPU tasks are executed on the same CPU scheduler thread which processed that work queue. (3) CPU tasks can prepare simulation variables (allocation and halo gathering) during task execution, rather than relying on the scheduler to allocate and prepare these variables beforehand.

VII. Efficient Memory Management Using Contiguous Buffers

Allocating GPU memory and copying memory host-to-GPU can be expensive operations because of latencies associated

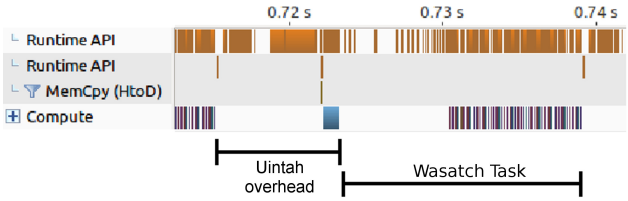

Fig. 12. A profiled time step for a Wasatch task using the initial runtime system. Most of the Uintah overhead is dominated by freeing and allocating many grid variables.

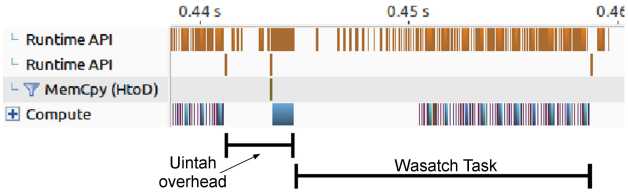

Fig. 13. A profiled time step for a Wasatch task using the new runtime system. The runtime system determines the combined size of all upcoming allocations, and performs one large allocation to reduce API latency overhead.

with these API calls. The initial GPU runtime allocated memory one grid variable at a time and copied these grid varibles one at a time. The associated accumulated API latencies become an issue when the accompanying GPU task executes in just a few milliseconds, as seen in Fig. 12.

A. Reducing GPU Allocation Latency

To address this issue, a straightforward method for utilizing a contiguous buffer was implemented. For a given task, Uintah's runtime computes the total size of all *Computes*, *Requires*, and halo data not yet in a memory location or the in process of allocating or copying into that memory location (see Section V) . A contiguous buffer was allocated in GPU memory, then multiple host-to-GPU copies were invoked for each *Requires* grid variable and halo cell staging variable into the allocated buffer on the GPU. This approach yielded improvements as shown in Fig. 13.

B. Attempts at Reducing GPU Copy Latency

We next investigated reducing the latency overhead when data variables are copied host-to-GPU. We tested forming a packed contiguous buffer in host memory with the goal of performing only one host-to-GPU copy instead of several. The larger difficulty here is that different tasks each require a different set of simulation variables, only sharing some, but not all, of a node's simulation variables among them. We allocated host buffers and filled them using host-to-host copies of individual simulation variables. We then copied these packed buffers to GPU memory. In all cases tested the cost of host-to-host copies outweigh the latency of many host-to GPU copies.

C. Contiguous Buffer Results

Table I gives a one node simulation for processing times using the current GPU engine without contiguous allocations and with contiguous allocations. The initial GPU runtime system is not profiled here. The Wasatch tests profiled solve 10 and 30 transport equations, respectively. Computations were performed on an Nvidia GTX680 GPU and an Intel Xeon E5-2620 with CUDA 6.5. With all these improvements, we observed speedups due to reduced overhead ranging from 1.27x to 2.00x for a variety of test cases.

TABLE I
EFFECT OF CONTIGUOUS BUFFERS ON WASATCH GPU TASKS

Wasatch Test	Mesh Size	Without Contiguous (ms)	With Contiguous (ms)	Speedup Due to Reduced Overhead
Test A - Solving 10 transport equations	16^3	13.36	10.56	1.27x
	32^3	18.25	13.25	1.38x
	64^3	57.99	33.88	1.71x
	128^3	124.51	100.09	1.24x
Test B - Solving 30 transport equations	16^3	41.70	26.61	1.57x
	32^3	51.54	34.89	1.48x
	64^3	173.46	86.62	2.00x
	128^3	374.922	276.22	1.36x

TABLE II
POISSON EQUATION SOLVER GPU VS. CPU SPEEDUP

Mesh Size	CPU only (s)	Initial GPU Runtime (s)	Current GPU Runtime (s)	Speedup - Current vs Initial	Speedup - Current vs CPU
64^3	0.08	0.31	0.11	2.82x	0.73x
128^3	0.31	1.33	0.38	3.50x	0.82x
192^3	0.84	2.96	0.63	4.70x	1.33x
256^3	1.93	6.09	1.13	5.39x	1.71x

The advantage in Uintah's contiguous buffer approach compared to similar runtime GPU buffer allocation schemes [34] is the ability to use runtime temporal knowledge to optimize these allocations. This work demonstrates that buffers yield speedups in all tested scenarios. In future work Uintah can go further and place all Old Data Warehouse simulation variables in one block, and use a second block for New Data Warehouse simulation variables. Uintah can preserve the second block between time steps (when the New Data Warehouse becomes the Old Data Warehouse) and reclaim the first block.

VIII. RESULTS

A. Poisson Equation Solver

A simple seven-point stencil for the Poisson equation in 3D using a simple Jacobi iterative method highlights difficulties of (1) little reuse of data and (2) a short-lived task with a wall time on the order of milliseconds when patch sizes are smaller. Runtime overhead becomes a substantial factor as the timesteps are likewise short-lived.

Table II compares this problem on the initial and current runtime system. Data for this table was computed using 50 iterations on a simulation grid using 12 patches. 12 CPU cores were used for 12 CPU tasks. Speedups provided to show reduction in runtime overhead in GPU tasks and highlight when GPU tasks become feasible over CPU tasks. The profiled

machine contained a Nvidia K20c GPU and an Intel Xeon E5-2620 with CUDA 5.5. Within Uintah, the combined memory feature (Section VII) was turned off to provide for an apples-to-apples halo data comparison.

In all cases, the current GPU runtime performed significantly better than the initial GPU runtime. As the grid sized increased, more data movement was required over the PCIe bus for the initial runtime, and total simulation time naturally increased significantly. For the current runtime, this problem was avoided and the speedup results can be seen by the profiled times.

The 192^3 and 256^3 case demonstrates a major motivation for these GPU runtime enhancements. Here, the initial GPU runtime was 3.5x and 3.2x slower than the CPU task version, respectively. Now the current GPU runtime computes this problem 1.33x faster for 192^3 and 1.71x faster for 256^3 compared to the CPU task version. This result demonstrates we can move more CPU tasks to the GPU to obtain speedups. For smaller grid sizes for this problem, the CPU task overhead is smaller than the GPU task overhead, and this results in faster overall CPU times.

Detailed profiling of the 192^3 case indicated that the previous GPU runtime had overhead between time steps of roughly 49 milliseconds. Under the current runtime, this overhead has been reduced to roughly 2 to 3 milliseconds. The GPU computation portion of this task used 10 milliseconds per time step, indicating a much smaller but still significant portion of the total simulation is spent in overhead. Profiling has indicated that one-third to one-half of the remaining overhead is comprised of GPU API calls such as mallocs, frees, and stream creations. Future work is planned to utilize resource pools so this overhead can be reduced further.

B. Wasatch

As mentioned in Section VII, Wasatch tasks are an ideal case for the work described in this paper. The Wasatch tests we profiled solved multiple partial differential equations (PDEs), and used as many as 120 PDE related variables per time step. Each task computes within milliseconds. Although these tests only run on one patch, they utilize periodic boundary conditions, meaning that each patch edge is logically connected with the patch edge on the opposite side, and thus halo data transfers still occur. Table III gives time to solutions for two different Wasatch tests which solve 10 and 30 transport PDEs, respectively. For the data in this table, the CPU thread counts are managed by Wasatch tasks to maximize efficiency. Speedups are provided to show reduction in runtime overhead in GPU tasks and highlight when GPU tasks become feasible over CPU tasks. Computations were performed on an NVidia GTX680 GPU and an Intel Xeon E5-2620 with CUDA 6.5.

The key aim of this work is to allow Uintah's GPU support to be opened to broader class of computational tasks. As Table III, the original runtime system processed GPU tasks slower than CPU tasks in all tested Wasatch cases. The current runtime system for the same GPU tasks now obtains significant speedups in most cases. Only when patch sizes are small do CPU tasks still perform fastest.

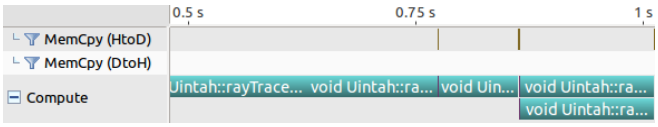

Fig. 14. In the original Uintah GPU engine, overlapping of RMCRT's kernels is infrequent as copying the GPU Data Warehouse prior to task execution is done as a blocking operation to avoid concurrency problems.

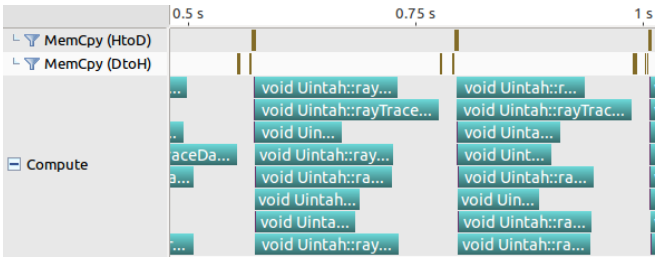

Fig. 15. Because Task Data Warehouses were designed to avoid blocking operations when copied into GPU memory, RMCRT kernel overlap is achieved.

C. Ray Tracing-based Radiation Model

This scaling study focuses on a reverse Monte Carlo ray tracing (*RMCRT*) approach to radiation modeling that makes novel use of Uintah's AMR capabilities to achieve scalability. This problem [35] uses a two-level AMR mesh, and is based on the benchmark described by Burns and Christen [36]. This challenging problem exercises all of the main features of the AMR support within Uintah as well as additional radiation physics. This benchmark problem also requires use of the concurrency improvements detailed in Section V. The radiation portion of this calculation was run on the DOE Titan system, using the single Nvidia K20x GPU available on each node. A fine level halo region of four cells in each direction, x, y, z was used. The AMR grid consisted of two levels with a refinement ratio of four, the fine mesh being four times more resolved than the coarse, radiation mesh.

For two separate cases, the total number of cells on the highest resolved level was 128^3 and 256^3 (blue and red lines respectively in Fig. 16), with 100 rays per cell in each case. The total number of cells on the coarse level was 32^3 and 64^3. In each of these strong scaling cases, a fixed patch size of 16^3 cells was used. Each data point represents a 2X increase in the number of GPUs assigned to the computation.

The principal result illustrated in Fig. 16 is the near order of magnitude speedup in mean time per timestep for each problem. This improvement is largely due to the introduction of non-blocking Task Data Warehouses as described in Section IV, which allows for many smaller patches to execute simultaneously due to kernel overlapping (see Fig. 14 and Fig. 15). Preliminary results from a much larger radiation calculation (512^3 cells on the fine mesh and 128^3 cells on the coarse radiation mesh) also show a 4-5X speedup at 16K GPUs on Titan using the concurrency improvements achieved in this work.

TABLE III
WASATCH TASKS GPU VS. CPU SPEEDUP

Wasatch Test	Mesh Size	CPU Only (s)	Initial GPU Framework (s)	Current GPU Framework (s)	Speedup - Current vs Initial	Speedup - Current vs CPU
Test A - Solving 10 transport equations	16^3	0.08	0.10	0.11	0.91x	0.72x
	32^3	0.19	0.23	0.12	1.92x	1.58x
	64^3	0.79	0.94	0.26	3.62x	3.03x
	128^3	4.75	5.21	1.22	4.27x	3.89x
Test B - Solving 30 transport equations	16^3	0.21	0.45	0.28	1.61x	0.75x
	32^3	0.56	0.97	0.37	2.62x	1.51x
	64^3	2.19	3.72	0.76	4.89x	2.88x
	128^3	13.56	20.79	3.64	5.71x	3.73x

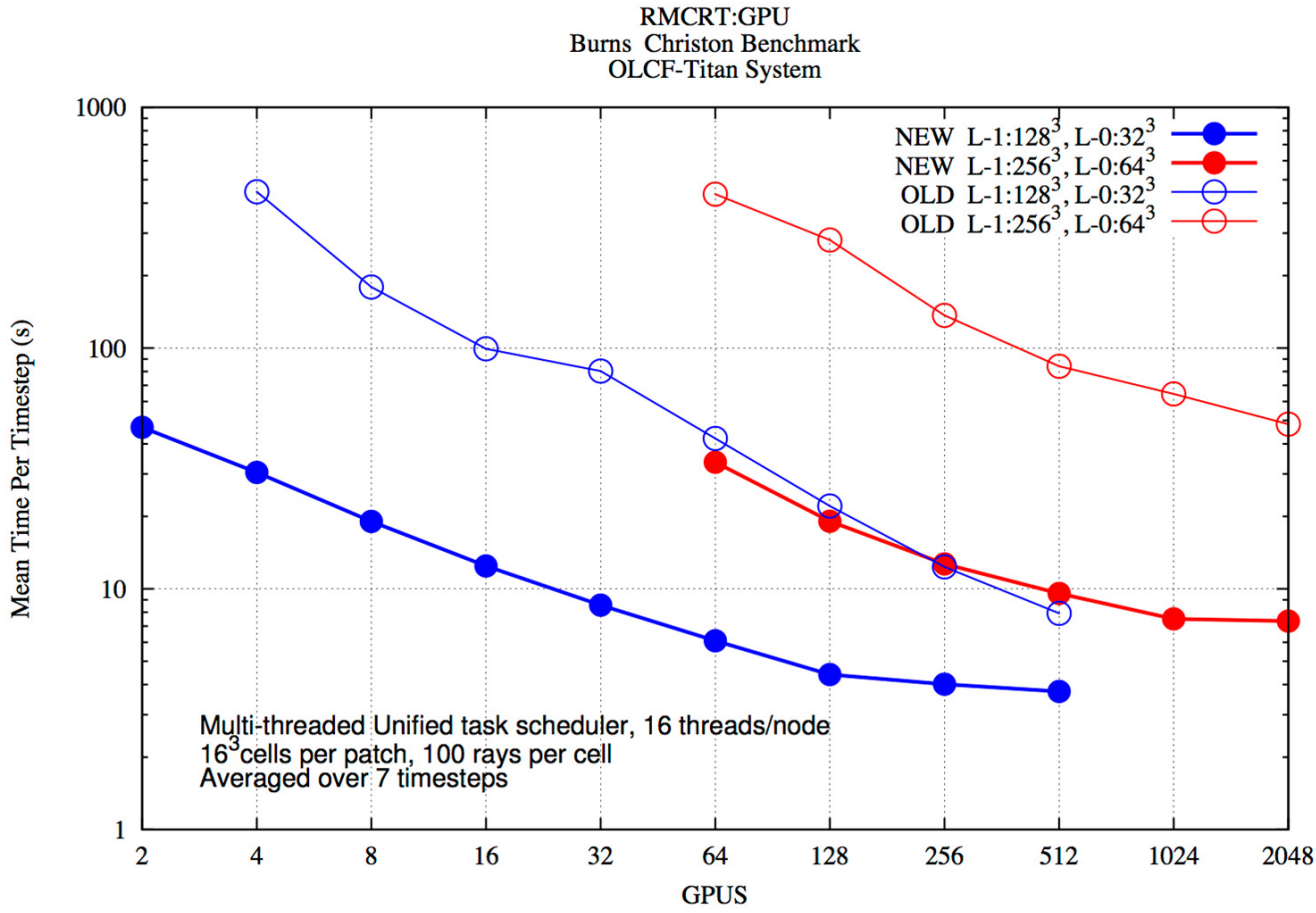

Fig. 16. Strong scaling of the two-level benchmark RMCRT problem [36] on the DOE Titan system. L-1 (Level-1) is the fine, CFD mesh and L-0 (Level-0) is the coarse, radiation mesh.

IX. RELATED WORK

Many other GPU-enabled asynchronous many-task runtimes share similar features of data dependency management and application developer interaction with the runtime. Legion [14] requires application developers define data dependencies with much more detail than Uintah. Developers must work with task syntax that adheres to their Legion C++ programming model or their Regent programming language [37]. Legion leverages a parallel global address space (PGAS) approach using GASNet for all internode communication. Charm++ [15] uses a message passing system similar in nature to MPI, and utilizing that Charm++ can perform well with load balancing. The Charm++ runtime system does not automatically pass messages to facilitate data transfer between neighbor nodes, instead the programmer is responsible for implementing all halo management. GPU kernels can be invoked from within a Charm++ work unit, and the Charm++ GPU Manager helps with task management and synchronization, overlapping of data transfer with kernel computation, and API to notify when a GPU kernel has completed. StarPU [16] manages data copying and data coherency in different memory spaces using a process very similar to cache coherency protocols. Halo transfers must be accomplished through user defined tasks, and some application developer interaction is required to aid

StarPU in MPI transfers among nodes. PaRSEC [17] contains many similarities with Uintah in that the runtime automates data transfer both through MPI for internode communication and between host and GPU memory for intranode communication. In PaRSEC, data coherence utilizes a simplified version of the MOESI cache coherency protocol [38]. Data dependencies are expressed by defining data flows among tasks using their customized JDF Format to help generate PaRSEC's DAG. If MPI is used, the user provides nodal communication information through a process patterned after MPI_Datatypes. Most of PaRSEC's target problems focus on linear algebra computations on mathematical matrices.

Uintah is perhaps the most specialized of these runtimes in that Uintah provides a rich API interface and accompanying runtime for application developers requiring uniform halo requirements around simulation variables on adaptive mesh refinement grids. Uintah focuses heavily on hiding runtime system details from application developers while maintaining both strong and weak scaling to full machine scale. Uintah has been shown to scale to 16K nodes using GPUs on Titan [18] and 768K cores on Mira [19]. In a search of literature, we found that Legion has been demonstrated to scale to 8K nodes on Titan [39], Charm++ to 512K cores on Mira [40], StarPU to 256 nodes [41] on the Occigen cluster located at CINES, France, and PaRSEC to 23868 cores [42] on Kraken.

X. Conclusions and Future Work

In this paper we describe modifications to the Uintah runtime system targeting problems with halo dependencies allowing more computational problems to efficiently execute on GPU/heterogenous architectures with minimal user interaction with the runtime. In particular we targeted stencil-based computations requiring local halo data (such as one layer of halo cells) or computations with nearly global data (data dependencies across the computational domain). We describe an effective system to keep variable data resident in GPU memory as well as in host memory and off-node, and how halo cell transfers can be processed from any source memory location to any destination memory location. We have also described additional work queues to schedule a task during its life cycle necessary for Uintah to process a heterogeneous mix of tasks. We show that allocating one large GPU memory space for all grid variables in a task provides substantial speedup benefits over allocating memory for each individual GPU grid variable. Results show these combined modifications reduced overhead to allow GPU tasks to run up to 5.71x faster versus the initial GPU runtime system, and up to 3.89x faster than their CPU task counterparts.

Uintah now opens itself up to a much broader range of computational problems on the GPU. With these successes, we plan to improve and optimize the runtime system further to aid portability and productivity. Effort is currently underway to utilize Kokkos [24] enabling application developers to only write task code once, instead of the current model where separate CPU and CUDA code must be provided. Implementing Kokkos requires more work merging the host and GPU data stores and task queues into unified logic. For some computational problems utilizing Uintah, data layout of grid variables in memory (row-major, column-major, 2D tiled, 3D tiled, etc.) is crucial for performance gains and Uintah must support these layouts. Utilizing Kokkos for task code also enables Uintah to target Xeon Phi architectures and work is underway to modify the task scheduler to properly execute these tasks with groups of CPU threads.

Acknowledgments

Funding from NSF and DOE is gratefully acknowledged. This material is based upon work supported by the National Science Foundation under Grant No. 1337145. 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 also like to acknowledge Oak Ridge Leadership Computing Facility ALCC award CSC188, "Demonstration of the Scalability of Programming Environments By Simulating Multi-Scale Applications" for time on Titan. We would also like to thank all those involved with Uintah past and present.

References

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

[2] A. Humphrey, Q. Meng, M. Berzins, and 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.

[3] Brad Peterson, Harish Dasari, Alan Humphrey, James Sutherland, Tony Saad, and Martin 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, pages 4:1–4:8, New York, NY, USA, 2015. ACM.

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

[5] M. Berzins. Status of Release of the Uintah Computational Framework. Technical Report UUSCI-2012-001, Scientific Computing and Imaging Institute, 2012.

[6] B.A. Kashiwa and E.S. Gaffney. Design Basis for CFDLIB. Technical Report LA-UR-03-1295, Los Alamos National Laboratory, 2003.

[7] S.G. Bardenhagen, J.E. Guilkey, K.M. Roessig, J.U. Brackbill, W.M. Witzel, and J.C. Foster. An Improved Contact Algorithm for the Material Point Method and Application to Stress Propagation in Granular Material. *Computer Modeling in Engineering and Sciences*, 2:509–522, 2001.

[8] J. E. Guilkey, T. B. Harman, A. Xia, B. A Kashiwa, and P. A. McMurtry. An Eulerian-Lagrangian Approach for Large Deformation Fluid-Structure Interaction Problems, Part 1: Algorithm Development. In *Fluid Structure Interaction II*, Cadiz, Spain, 2003. WIT Press.

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

[10] T. Saad and J. C. Sutherland. Wasatch: an Architecture-Proof Multiphysics Development Environment using a Domain Specific Language and Graph Theory. *Journal of Computational Science*, 2015.

[11] Q. Meng, M. Berzins, and J. Schmidt. Using Hybrid Parallelism to Improve Memory Use in the Uintah Framework. In *Proc. of the 2011 TeraGrid Conference (TG11)*, Salt Lake City, Utah, 2011.

[12] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin 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, pages 96:1–96:12, New York, NY, USA, 2013. ACM.

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

[14] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex 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, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[15] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. *SIGPLAN Not.*, 28(10):91–108, October 1993.

[16] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André; Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *Parallel Comput.*, 38(1-2):37–51, January 2012.

[18] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins. Radiative Heat Transfer Calculation on 16384 GPUs Using a Reverse Monte Carlo Ray Tracing Approach with Adaptive Mesh Refinement. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1222–1231, May 2016.

[19] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, , and C. Wight. Extending the Uintah Framework Through the Petascale Modeling of Detonation in Arrays of High Explosive Devices. *SIAM Journal on Scientific Computing (Accepted)*, 2016.

[20] Alex Bourd. The OpenCL Specification, 2017. https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf.

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

[22] OpenMP Architecture Review Board. Openmp application program interface version 4.0, 2013.

[23] J. Keasler R. Hornung. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, 2014.

[24] H. Carter Edwards and Daniel Sunderland. Kokkos Array Performance-portable Manycore Programming Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 1–10, New York, NY, USA, 2012. ACM.

[25] T. Srman. Comparison of Technologies for General-Purpose Computing on Graphics Processing Units. Master's thesis, Department of Electrical Engineering, Linkping University, 2016.

[26] Matt Martineau, James Price, Simon McIntosh-Smith, and Wayne Gaudin. Pragmatic Performance Portability with OpenMP 4.x. In Naoya Maruyama, Bronis R. de Supinski, and Mohamed Wahib, editors, *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*, pages 253–267, Cham, 2016. Springer International Publishing.

[27] Raphael Landaverde, Tiansheng Zhang, Ayse Kivilcim Coskun, and Martin C. Herbordt. An investigation of Unified Memory Access performance in CUDA. *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014.

[28] Nvidia. CUDA C Programming Guide v8.0 Web page - J. Unified Memory Programming, 2017. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.

[29] Patrick K Notz, Roger P Pawlowski, and James C Sutherland. Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software. *ACM Transactions on Mathematical Software (TOMS)*, 39(1):1, 2012.

[30] Christopher Earl, Matthew Might, Abhishek Bagusetty, and James C. Sutherland. Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations. *Journal of Systems and Software*, 2015.

[31] James C. Sutherland and Tony Saad. The Discrete Operator Approach to the Numerical Solution of Partial Differential Equations. In *20th AIAA Computational Fluid Dynamics Conference*, pages AIAA–2011–3377, Honolulu, Hawaii, USA, June 2011.

[32] Nvidia. Nvlink web page, 2015. http://www.nvidia.com/object/nvlink.html.

[33] Wei Wu, George Bosilca, Rolf vandeVaart, Sylvain Jeaugey, and Jack Dongarra. GPU-Aware Non-contiguous Data Movement in Open MPI. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 231–242, New York, NY, USA, 2016. ACM.

[34] Bin Ren, Nishkam Ravi, Yi Yang, Min Feng, Gagan Agrawal, and Srimat Chakradhar. Automatic and Efficient Data Host-Device Communication for Many-Core Coprocessors. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, pages 173–190, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[35] A. Humphrey, T. Harman, M. Berzins, and P. Smith. A Scalable Algorithm for Radiative Heat Transfer Using Reverse Monte Carlo Ray Tracing. In Julian M. Kunkel and Thomas Ludwig, editors, *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 212–230. Springer International Publishing, 2015.

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

[37] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex 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, pages 81:1–81:12, New York, NY, USA, 2015. ACM.

[38] George Bosilca, Aurelien Bouteiller, Thomas Hérault, Pierre Lemarinier, Narapat Ohm Saengpatsa, Stanimire Tomov, and Jack J. Dongarra. Performance Portability of a GPU Enabled Factorization with the DAGuE Framework. *2011 IEEE International Conference on Cluster Computing*, pages 395–402, 2011.

[39] Michael Edward Bauer. *LEGION: PROGRAMMING DISTRIBUTED HETEROGENEOUS ARCHITECTURES WITH LOGICAL REGIONS*. PhD thesis, Stanford University, 2014.

[40] Abhinav Bhatele, Jae-Seung Yeom, Nikhil Jain, Chris J. Kuhlman, Yarden Livnat, Keith R. Bisset, Laxmikant V. Kale, and Madhav V. Marathe. Massively Parallel Simulations of Spread of Infectious Diseases over Realistic Social Networks. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pages 689–694, Piscataway, NJ, USA, 2017. IEEE Press.

[41] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[42] Anthony Danalis, George Bosilca, Aurelien Bouteiller, Thomas Herault, and Jack Dongarra. PTG: An Abstraction for Unhindered Parallelism. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 21–30, Piscataway, NJ, USA, 2014. IEEE Press.