Reducing Overhead in the Uintah Framework to Support Short-Lived Tasks on GPU-Heterogeneous Architectures

Brad Peterson
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
bradpeterson@gmail.com

Harish Dasari
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
hdasari@sci.utah.edu

Alan Humphrey
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
ahumphrey@sci.utah.edu

James Sutherland
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
james.sutherland@utah.edu

Tony Saad
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
tony.saad@utah.edu

Martin Berzins
Scientific Computing and Imaging Institute
72 Central Campus Dr
Salt Lake City, Utah
mb@sci.utah.edu

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. The Uintah runtime system is based on a distributed directed acyclic graph (DAG) of computational tasks, with a task scheduler that efficiently schedules and execute these tasks on both CPU cores and on-node accelerators. The runtime system identifies task dependencies, creates a taskgraph prior to an iteration based on these dependencies, prepares data for tasks, automatically generates MPI message tags, and manages data after task computation. Managing tasks for accelerators pose significant challenges over their CPU task counterparts due to supporting more memory regions, API call latency, memory bandwidth concerns, and the added complexity of development. These challenges are greatest when tasks compute within a few milliseconds, especially those that have stencil based computations that involve halo data, have little reuse of data, and/or require many computational variables. Current and emerging heterogeneous architectures necessitate addressing these challenges within Uintah. This work is not designed to improve performance of existing tasks, but rather reduce runtime overhead to allow developers writing short-lived computational tasks to utilize Uintah in a heterogeneous environment. This work analyzes an initial approach for managing accelerator tasks alongside existing CPU tasks within Uintah. The principal contribution of this work is to identify and address inefficiencies that arise when mapping tasks onto the GPU, to implement new schemes to reduce runtime system overhead, to introduce new features that allow for more tasks to leverage on-node accelerators, and to show overhead reduction results from these improvements.

WOLFHPC2015, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2830018.2830023 .

Keywords

Uintah, hybrid parallelism, parallel, GPU, heterogeneous systems, stencil computation, optimization

1. 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 [13], that seek to utilize available accelerators such as graphics processing units (GPUs).

The design of Uintah maintains a clear separation from the applications layer and its runtime system, allowing applications 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 information (*ghost cells*) and the life cycle of data variables, or indeed any details with the multiple levels of parallelization inherent to these heterogeneous systems. Further, the public API exposed to an application developer should remain simple, shielding them from the complex details involved with the parallel programming required on these systems.

Prior work shown in [5], describes an initial design for managing GPU tasks alongside CPU tasks within Uintah. This implementation allowed specific tasks to leverage on-node GPUs, and employed existing runtime system functionality. For example, logic to automatically handle ghost cells exchange within Uintah has been researched and developed for over a decade. This logic allowed the runtime system to distribute and collect ghost cells of simulation variables either within host memory or off-node via MPI. The initial Uintah GPU approach did not implement any new ghost cell logic within the GPU, but rather moved simulation variables from GPU memory to host memory to utilize the existing host-side logic. This approach provides speedups for some tasks, such as those where copying data is a fraction of the computation time. But 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 fast tasks, not by reducing task computation time, but by reducing runtime overhead. The Wasatch project in particular has been a top motivation for these runtime improvements (see section 7.2).

To extend the GPU approach in the initial design [5] 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 ghost cell management occur on the GPU, whether the ghost cells 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 [5]. 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, five optimizations are covered, 1.) persistence of GPU data, 2.) managing multiple ghost cell scenarios, 3.) additional, GPU-specific work queues needed within the task scheduler, 4.) eliminating contention and reducing the size of Uintah's GPU data store 5.) allocating all data variables in one contiguous memory buffer instead of several. 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 those on Knights Corner and Knights Landing Intel Xeon Phis, by providing a framework to allow data to be managed in both high bandwidth memory closer to processors and also in high capacity memory off-chip.

We begin by giving an overview of the Uintah framework in Section 2. Section 3 details work done to enable persistence of simulation data on GPUs and how this work has enabled management of multiple, difficult ghost cell scenarios. In Section 4, changes to the Uintah task scheduler are covered. Section 5 describes changes for how GPU tasks obtain variable data from a data store. Our approach to minimizing GPU API call latency is covered in Section 6. Nodal results from these improvements are shown in Section 7 and the paper concludes in Section 8 with discussion on future work.

2. UINTAH OVERVIEW

The open source Uintah framework [2], [13] 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 as shown in [7] with the multi-material ICE code for both low and high-speed compressible flows, or by using particles on that grid as shown in [1] with the multi-material, particle-based code MPM for structural mechanics. Uintah also provides the combined fluid-structure interaction (FSI) algorithm MPM-ICE [4], the ARCHES turbulent reacting CFD component [6] designed for simulating turbulent reacting flows with participating media radiation, and Wasatch, a general multiphysics solver for turbulent reacting flows.

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 of which the Data Warehouse manages. The metadata indicates the patches on which specific variable data resides, halo depth or number of ghost cell layers, a pointer to the actual data, and the data type (node-centred, 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 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 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 to the GPU prior to task execution.

Uintah task schedulers are responsible for scheduling and executing both CPU and GPU tasks, memory management of data variables, and invoking MPI communication. There are several task schedulers available within Uintah. In this work, we focus on the *Unified Scheduler* [9], shown in Figure 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 [8] (a combination of MPI and 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 [10].

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.

Uintah maintains a clear separation between applications 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. A developer need only supply Uintah with a description of the task as it would run serially on a single patch, namely what variables it will compute, what variables it needs from the previous time step, and how many layers of ghost cell data are needed for a variable. The task developer must supply entry functions to his or her task code, and writes serial C++ code for CPU and Xeon Phi tasks and CUDA parallel code for GPU tasks. The present model for GPU-enabled tasks currently requires that two versions of the task code be maintained, one for CPU code and one for GPU code. In this work, no new requirements were imposed upon the task developer.

Many developers can utilize existing Uintah API tools if they choose. An example of this is shown in Section 7, where we fo-

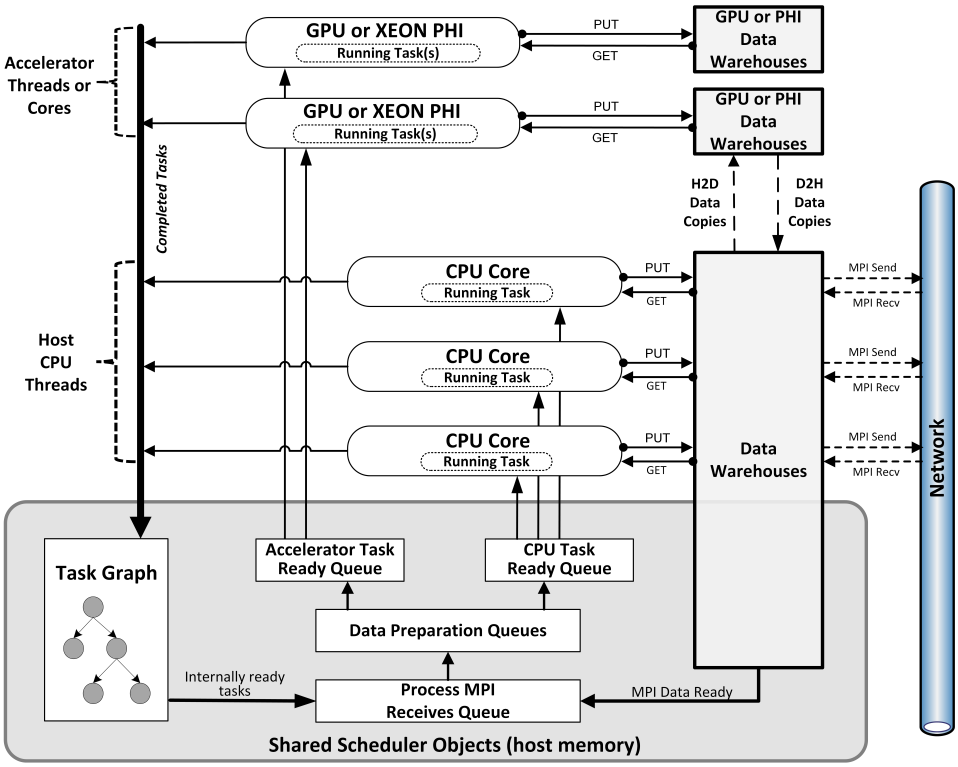

Figure 1: Uintah heterogeneous nodal runtime system and task scheduler. [9]

cus 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 component, a finite volume computational fluid dynamics code that is designed to solve transient, turbulent, reacting flow problems. Wasatch employs a formalism of the DAG approach to generate runtime algorithms [11], and an Embedded Domain Specific Language (EDSL) called Nebo [3], [14]. Nebo allows Wasatch developers to write high-level, Matlab-like syntax that can be executed on multiple architectures such as CPUs and GPUs. Wasatch is still experimental and under development. Uintah itself needs no knowledge of how the Wasatch tasks work, other than the data variables used for each task.

3. PERSISTENCE OF GPU DATA AND MANAGING GHOST CELLS

In the application layer, a developer is responsible for providing task parameters to the runtime system. The developer lists all variables that will be used in the task computation, and indicates whether these variables come from the *Old* or *New Data Warehouse*. Each variable is assigned as *Computes*, *Modifies*, or *Requires*. *Computes* are variables to be allocated by the runtime system to hold data computed by the task. *Modifies* are variables which were previously computed and will be modified by the task. *Requires* are read-only variables computed in the previous time step. The number of needed ghost cells layers for any *Requires* is also indicated. The programmer specifies whether the task is a GPU task or a CPU task. From there, the application layer programmer should not have to worry about the details of memory management. That programmer can write task code assuming the runtime system will have prepared all variables' memory, including gathering ghost cell data.

The original GPU support within Uintah did not make data persistent on the GPU between time steps [5]. Any GPU task with a variable listed as *Requires* would be copied from host-to-device prior to task execution. After task execution task variables listed as *Computes* would be copied device-to-host. This approach was implemented for its simplicity in reusing the runtime system's host-side memory management logic. As described previously in Section 1, for some tasks, this overhead is acceptable. For other tasks, such as those profiled in Section 7, this overhead is inefficient.

In order to avoid unnecessary copies across the PCIe bus, the runtime system now allows data to exist only in host-side memory, or only GPU memory, or both. To start supporting this change, two boolean flags were added to the variable's metadata in the *GPU Data Warehouse*, *ValidOnCpu* and *ValidOnGpu*. The purpose is straightforward, if a boolean flag is set to true, it means the data in that memory location is valid and ready to be accessed. Suppose an upcoming GPU task is analyzed by the runtime system, and it discovers a *Requires* variable only exists in host memory. The runtime system then performs an asynchronous host-to-device copy. When the scheduler verifies the copy has completed, that variable's *ValidOnGpu* flag is changed to true. As for any *Computes* variables, they are allocated device-side prior to task execution. After task execution, then *Computes* variables have their *ValidOnGpu* changed to true. From here, *Computes* variables stay on the GPU. They are *only* copied back into host memory if later CPU task lists that it needs those variables.

The aforementioned design is relatively simple. It allows for pointwise computations to make variable data persistent on GPUs.

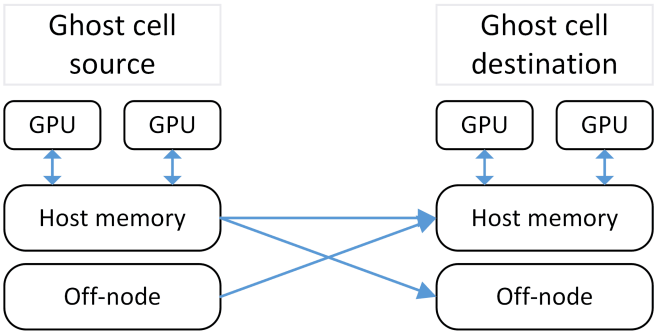

Figure 2: Uintah's initial runtime system to prepare CPU and GPU task variables with ghost cells from adjacent variables.

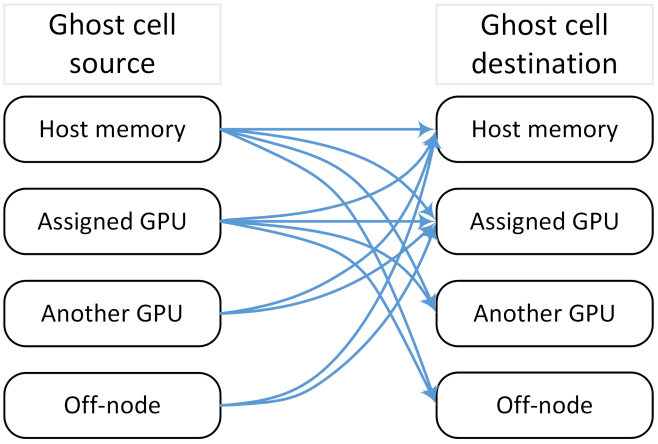

Figure 3: Uintah's current runtime system to prepare CPU and GPU task variables with ghost cells from adjacent variables.

But it does not naturally work for stencil computations as it does not provide any ghost cell management logic in GPUs. With the previous runtime system, there were only three ghost cell copy scenarios (see Figure 2). All ghost cell management was handled in host memory then copied back into GPUs. While functional and simple, it heavily utilized the PCIe bus. For example, suppose each MPI rank was assigned a $4 \times 4 \times 4$ set of patches, and each patch contained $64 \times 64 \times 64$ cells. Also suppose this simulation has only one variable for stencil computations, the data variable existed in GPU memory, the variable held double values, and 1 layer of ghost cells were 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 ghost cells sent out. Then the MPI rank would receive ghost cell 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 variable into host memory would be roughly 14 ms and another 14 ms to copy it back into the device. For many Uintah GPU tasks which compute within a few milliseconds, this is impractical.

With data staying persistent in GPUs, more ghost cell scenarios must be managed. Uintah must prepare variables for both CPU tasks and GPU tasks by obtaining ghost cell data from variables in adjacent patches (adjacent variables) in whatever memory location they exist. These adjacent 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 ghost cell copy scenarios. Because a task does not manage ghost cells for patches or nodes it is not assigned to, this number is reduced to 12 scenarios, as shown in Figure 3. 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 [12] is considered, the number of ghost cell copy scenarios will then increase by needing to determine which data bus to use.

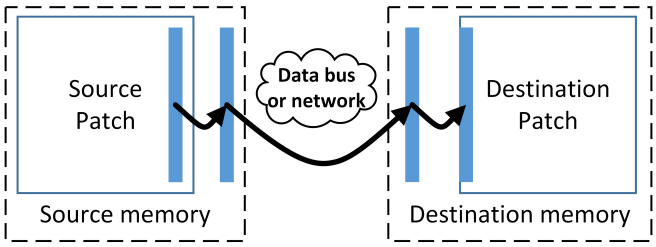

Figure 4: Ghost cells moving from one memory location to another are first copied into a contiguous staging array prior to being copied to that memory location. Later a task on the destination will process the staging array back into the variable.

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

All needed ghost cell dependency logic can be gleaned from analyzing the task graph's dependencies, and analyzing the Data Warehouse's knowledge in which memory locations variables exist and are valid. If ghost cells do not need to be copied into another memory location, then the ghost cell copy can process internally within that memory space. For example, two adjacent variables in GPU memory can simply copy their ghost cells to each other. Otherwise if the ghost cells need to go to another memory location, then contiguous arrays are employed, as shown in Figure 4. By using contiguous arrays to contain only the needed ghost cell data, far less data moves across data buses or network connections, and tools such as CUDA-aware MPI can be employed if needed.

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 device-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 ghost cell approach is that this management system has been merged with the scheduler code as described in Section 4. So whether only ghost cells needs to be copied across the PCIe bus, or a regular task variable without ghost cells also needs to be copied, the scheduler treats both the same. Further, these can all be processed in batches, rather than one at a time. If a GPU task requires N variables each needing ghost cells, then all N can be processed together. To illustrate this with an example, suppose a 27-point stencil task requires that a particular GPU data variable send its ghost cell data to its 26 neighbors, and then receive ghost 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 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 ghost cell copies within that GPU. Af-

ter the kernel completes, the host runtime system identifies those 6 dependencies which belong to another GPU, and so 6 GPU peer-to-peer copies are invoked. The host 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 future tasks to gather these ghost cells back into data variables.

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

There are many benefits to this model. While the prior example demonstrated processing only one variable in stages, the runtime system can process many variables belonging to many patches in bulk, so that all outgoing ghost cell copies for a task can be processed in one kernel. GPU tasks which need to use these ghost cells will also be able to gather together all ghost cells in bulk. And this model can include the ability to reduce the amount of memory allocations and copies by packing individual staging arrays into larger contiguous arrays.

A future enhancement awaiting implementation on the GPU is the ability to store all variables in a contiguous space if those variables are already adjacent to each other in the computational grid. This has already been implemented in host memory. This means that if two adjacent variables are in host memory and require ghost cells from each other, no ghost cell transfer or processing is needed as both variables already exist in the same array. This reduces the amount of internal ghost cell copies needed.

4. TASK SCHEDULER ENHANCEMENTS

There are a number of schedulers in Uintah. The scheduler covered in this section is the *Unified Scheduler* [9], shown in Figure 1. This scheduler functions by having all CPU threads independently checking shared priority queues for available tasks to process. Tasks can proceed through several queues during its execution life cycle. A detailed description of this flow can be found in [5]. To facilitate the changes listed in the previous section, additional scheduler queues are added.

In the prior runtime system, a GPU task proceeded through three queues in its life cycle in the order listed. They were 1.) process all MPI receives, 2.) manage ghost cells gathers, allocate space on the GPU for all data variables, and copy all *Requires* to the GPU, and 3.) execute the task and copy all *Computes* back to host memory. In the current runtime system, a GPU task must proceed through five queues in the order listed. They are 1.) process all MPI receives, 2.) manage some ghost cell gathers in host memory, prepare meta data for some ghost cell gathers in GPU memory, allocate space for some GPU data variables, and copy some *Requires* to the GPU (see Sect. 3), 3.) set *ValidOnGpu* to true for all *Requires* and manage ghost cell gathers within a GPU, 4.) execute the task, and 5.) set *ValidOnGpu* to true for all *Computes*. Previously a CPU task proceeded through only two queues, which were 1.) process all MPI receives and 2.) execute the task. Now a CPU task must proceed through three queues, 1.) process all MPI receives, 2.) copy some variable data from GPU to host memory if needed, and 3.) set *ValidOnCpu* to true for all *Requires* and execute the task.

The *Unified Scheduler* contains a function which accumulates a collection of all variables and ghost cells to be created and processed on the GPU prior to performing these actions. This has two benefits. First, it allows for the creation of smaller GPU Data Warehouses specifically for a GPU task (Section 5), and it allows for employing contiguous memory allocations (Section 6).

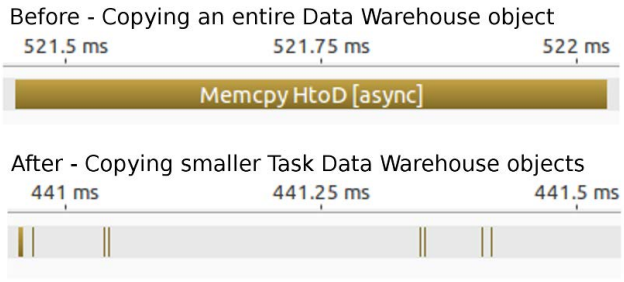

Figure 5: A profiled half millisecond range of an eight patch simulation showing Data Warehouse copies. Before, the initial runtime system had many large Data Warehouse copies (only one shown in this figure). After, the new runtime system's small Task Data Warehouses copy into GPU memory quicker, allowing GPU tasks to begin executing sooner (eight Data Warehouse copies are shown in this figure).

5. GPU DATA WAREHOUSE MODIFICATIONS

The original design of the GPU Data Warehouse contained three problems which limited its potential and necessitated changes for this current work. The problems were first, a GPU Data Warehouse object only contained metadata for variables. With ghost cell management happening within the GPU, it needed to contain information on how to perform internal ghost cell copies. Second, a GPU Data Warehouse memory footprint was becoming larger, 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. Third, every GPU task shared the GPU Data Warehouse in GPU memory, and this resulted in contention and coordination issues. For example, if one GPU task kernel was writing to the New GPU Data Warehouse object in GPU memory, and another GPU task wished to perform a host-to-device update of the New GPU Data Warehouse in GPU memory, the second task must wait until the first kernel completed. Another issue is if one task was copying a GPU Data Warehouse into GPU memory while another was modifying it in host memory. Yet another issue is if multiple tasks on multiple threads simultaneously recognized that the Old and New GPU Data Warehouse were not yet copied into GPU memory, all threads would simultaneously initiate a host to device copy, resulting in many excessive and unnecessary copies. This latter problem was frequently observed.

A solution to these problems was accomplished by creating small Task Data Warehouses independent to each GPU task, gathering all knowledge of a task's GPU memory actions prior to creating and copying data, and by redesigning the GPU Data Warehouse into a compact serialized object.

Task Data Warehouses were created in order to avoid all issues associated with all tasks sharing the same GPU Data Warehouse in GPU memory. The driving concept of Task Data Warehouses is that each GPU task receives its own self contained GPU Data Warehouse objects, wholly independent and not used by other tasks, with only the information it needs to manage ghost cell copies and variables for task computation. These small Task Data Warehouses in GPU memory serve as read-only snapshots of the full GPU Data Warehouse. A GPU task kernel will then have no knowledge or capability to access variables or perform ghost cell copies unrelated to its own task. This eliminates coordination and contention

issues. 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. A further consequence of this model is that the Data Warehouse for host memory variables and the GPU Data Warehouse will now only exist in host memory, which makes it easier for future work to merge both data stores into one object.

In order to create these Task Data Warehouses, the runtime system must know and gather together all upcoming variables and ghost cell copies needed for that task to process. Three host-side collections were created to gather this information. The first collection manages all variable data that is required to be added into the host-side GPU Data Warehouse. The second collection manages all variable information needed in the Task Data Warehouse. The third collection manages all upcoming ghost cell copies that must occur within the GPU. These collections are vital to sizing the arrays within the Task Data Warehouse.

The structure of a GPU Data Warehouse object required streamlining to more efficiently copy into GPU memory. Originally the approach taken was to avoid deep copies by using fixed sized arrays within the object. But as we discovered some GPU tasks required far more variables than others, we likewise noticed the GPU Data Warehouse's memory footprint was becoming too large. Adding an additional array containing ghost cell copy logic within a GPU made this 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 data variables and the array for ghost cell copy logic into one array, and defined in code that it is a very large fixed size array. Doing so preserved the serialized nature of the object in memory. Then use the three collections described in the prior paragraph to count exactly how many variables, staging array variables, and ghost cell copy entries will be needed for this array. Then allocate an object on the host that contains the minimum amount of memory space needed for these items. Doing so means only a fraction of the large fixed sized array is allocated. Then use the three collections to load the Task Data Warehouse in host memory. This results in a compact Task Data Warehouse object whose memory structure consists of a handful of basic variables 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 only a few kilobytes in size. Figure 5 shows the improvements of this approach, demonstrating how more Data Warehouse objects can be copied into GPU memory in less time.

6. EFFICIENT MEMORY MANAGEMENT USING CONTIGUOUS BUFFERS

Moving memory from host-to-GPU is an expensive operation because the PCIe bus to which the GPU is connected has low bandwidth as compared to the GPU internal memory bus. Also, there are significant latencies associated with the CUDA APIs for both allocating and copying memory. The initial GPU runtime allocated memory one variable at a time, which results in large accumulated API latencies. These API latencies become an issue when the accompanying GPU task executes in just a few milliseconds, as seen in Figure 6.

To address this issue, a straightforward method for utilizing a contiguous buffer was implemented. Initially we tested a configuration of allocating two contiguous buffers, one on the host and one on the GPU. The host buffer is sized to hold all *Requires* and all needed ghost cell data for that task. The GPU buffer is sized

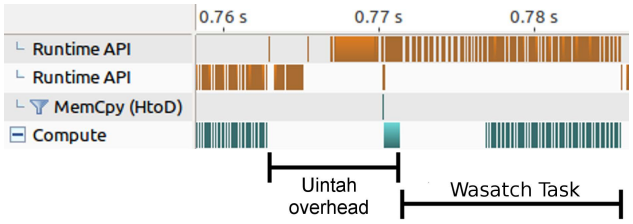

Figure 6: A profiled time step for a Wasatch task using the initial runtime system. Most of the Uintah overhead is dominated by freeing and allocating many data variables.

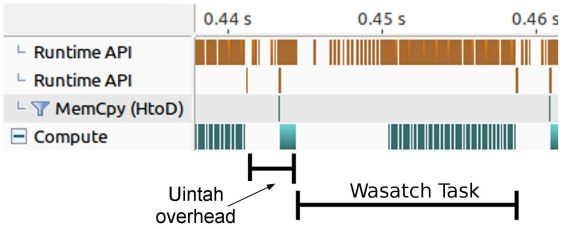

Figure 7: A profiled time step for a Wasatch task using the new runtime system. The runtime system determines the combined size of all upcoming allocations, and performs one large allocation to reduce API latency overhead.

to hold *Computes*, *Requires*, and needed ghost cell data. The host buffer is populated using host-to-host copies. The host buffer is then copied into the first part of the GPU buffer, with the rest of the GPU buffer set aside for *Computes*. The goal was to test the assumption that the combination of multiple smaller host-to-host copies and one single large host-to-device copy will be able to offset the cost of allocating and copying the variables separately. This assumption was found to be false for all scenarios we tested.

We then tested an approach where a contiguous buffer was allocated only in GPU memory instead of both GPU and host memory. Then multiple host-to-device copies are invoked for each *Requires* variable and ghost cell staging variable into the allocated buffer on the GPU. This approach yielded improvements as shown in Figure 7.

Table 1 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 trans-

Table 1: Wasatch GPU tasks profiled with and without contiguous variable buffers.

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 2: Poisson Equation Solver 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. Profiled machine contained a NVidia K20c GPU and an Intel Xeon E5-2620 with CUDA 5.5.

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

port 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 1.27 to 2.00 for a variety of test cases.

7. RESULTS

7.1 Poisson Equaition 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 due to it requiring only a few lines of code. Each time step computes within a few milliseconds, which means that overhead timings become substantial. Such problems are naturally more difficult to achieve speedups.

Table 2 compares this problem on the initial and current runtime system. Within Uintah, the combined memory feature was turned off to provide for an apples-to-apples ghost cell comparison (see the last paragraph in Section 3).

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 $192 \times 192 \times 192$ case demonstrates a major motivation for these GPU runtime enhancements. Here, the initial GPU runtime was 3.5x slower than the CPU task version. Now the current GPU runtime computes this problem 1.33x faster than 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 $192 \times 192 \times 192$ 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.

7.2 Wasatch

As mentioned in Section 6, Wasatch tasks are an ideal case for

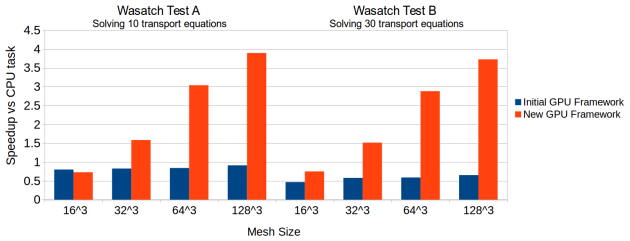

Figure 8: Speedups of Wasatch GPU tasks on the initial runtime and current runtime vs. Wasatch CPU tasks.

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 ghost cell transfers still occur. Table 3 gives time to solutions for two different Wasatch tests which solve 10 and 30 transport PDEs, respectively.

The key aim of this work is to allow Uintah's GPU support to be opened to broader class of computational tasks. As Figure 8 illustrates, 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 does using CPU tasks still perform fastest.

8. CONCLUSIONS AND FUTURE WORK

In this paper we describe modifications to the Uintah runtime system which reduces overhead to allow more categories of computational problems to be executed on a GPU within a heterogeneous architecture. In particular stencil-based computations whose computation time steps are on the order of tens of milliseconds or less and computations which contain many input and output variables now compute faster than their CPU task counterparts. We describe an effective system to keep variable data resident in GPU memory as well as in host memory and off-node, and how ghost 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 variables in a task provides substantial speedup benefits over allocating memory for each individual GPU 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.

With these successes, we plan to improve and optimize the runtime system further. Uintah now opens itself up to a much broader range of computational problems on the GPU, but other classes of problems exist which are not yet efficiently managed. For example, if a simulation problem requires each node to store data in host memory greater than the capacity of GPU memory, then some data must vacate GPU memory during a time step. Future strategies are planned to manage this data movement efficiently. Also, the new Uintah runtime system has implemented the capability to work with multiple GPUs per node, however, more work remains to restructure the logic to process ghost cells for these tasks efficiently. Another optimization which can improve performance is

Table 3: Wasatch computations profiled with the GPU task on the initial GPU framework, the current GPU framwork, and the CPU task. Wasatch tasks internally manages their own CPU thread counts to maximize efficiency. Speedups 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.

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

to have Uintah collect all currently queued and ready GPU tasks, and prepare and launch them in one group rather than processing each task individually. Also, since most time steps tend to reuse the same task graph, Uintah would benefit from a resource pool, where an old time step's resources can be reused in the current iteration. Finally, for some computational problems utilizing Uintah, data layout of variables in memory (row-major, column-major, 2D tiled, 3D tiled, etc.) is crucial for performance gains. With the full GPU Data Warehouse now existing solely in host-memory, it can be merged in with the Data Warehouse managing host memory variables, and then the combined Data Warehouse can be more easily refactored to allow for these memory layout changes.

9. ACKNOWLEDGMENTS

Funding from NSF and DOE is gratefully acknowledged. The work of Peterson, Dasari, Berzins and Sutherland was funded by NSF XPS Award 1337135 and which benefited from background work on the Uintah framework and Wasatch by Humphrey, Saad, and Sutherland. This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. We would also like to thank all those involved with Uintah past and present, Qingyu Meng in particular.

10. REFERENCES

[1] S. Bardenhagen, J. Guilkey, K. Roessig, J. Brackbill, W. Witzel, and J. 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.

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

[3] C. Earl, M. Might, A. Bagusetty, and J. C. Sutherland. Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations. *Journal of Systems and Software, to appear.*, 2015.

[4] 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.

[5] 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.

[6] J.Spinti, J. Thornock, E. Eddings, P. Smith, and A. Sarofim. Heat transfer to objects in pool fires. In *Transport Phenomena in Fires*, Southampton, U.K., 2008. WIT Press.

[7] B. Kashiwa and E. Gaffney. Design basis for cfdlib. Technical Report LA-UR-03-1295, Los Alamos National Laboratory, 2003.

[8] 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.

[9] 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.

[10] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Investigating applications portability with the uintah dag-based runtime system on petascale supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 96:1–96:12, New York, NY, USA, 2013. ACM.

[11] P. K. Notz, R. P. Pawlowski, and J. 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.

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

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

[14] J. C. Sutherland and T. 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.