Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System

Alan Humphrey, Qingyu Meng, Martin Berzins, Todd Harman

UUSCI-2012-003

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

May 3, 2012

Abstract:

The Uintah Computational Framework was developed to provide an environment for solving fluid-structure interaction problems on structured adaptive grids on large-scale, long-running, data-intensive problems. Uintah uses a combination of fluid-flow solvers and particle-based methods for solids, together with a novel asynchronous task-based approach with fully automated load balancing. Uintah demonstrates excellent weak and strong scalability at full machine capacity on XSEDE resources such as Ranger and Kraken, and through the use of a hybrid memory approach based on a combination of MPI and Pthreads, Uintah now runs on up to 262k cores on the DOE Jaguar system. In order to extend Uintah to heterogeneous systems, with ever-increasing CPU core counts and additional onnode GPUs, a new dynamic CPU-GPU task scheduler is designed and evaluated in this study. This new scheduler enables Uintah to fully exploit these architectures with support for asynchronous, outof- order scheduling of both CPU and GPU computational tasks. A new runtime system has also been implemented with an added multi-stage queuing architecture for efficient scheduling of CPU and GPU tasks. This new runtime system automatically handles the details of asynchronous memory copies to and from the GPU and introduces a novel method of pre-fetching and preparing GPU memory prior to GPU task execution. In this study this new design is examined in the context of a developing, hierarchical GPUbased ray tracing radiation transport model that provides Uintah with additional capabilities for heat transfer and electromagnetic wave propagation. The capabilities of this new scheduler design are tested by running at large scale on the modern heterogeneous systems, Keeneland and TitanDev, with up to 360 and 960 GPUs respectively. On these systems, we demonstrate significant speedups per GPU against a standard CPU core for our radiation problem.

Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System

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

Qingyu Meng
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
qymeng@sci.utah.edu

Martin Berzins
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
mb@sci.utah.edu

Todd Harman
Department of Mechanical Engineering
University of Utah
Salt Lake City, UT 84112 USA
t.harman@utah.edu

ABSTRACT

The Uintah Computational Framework was developed to provide an environment for solving fluid-structure interaction problems on structured adaptive grids on large-scale, long-running, data-intensive problems. Uintah uses a combination of fluid-flow solvers and particle-based methods for solids, together with a novel asynchronous task-based approach with fully automated load balancing. Uintah demonstrates excellent weak and strong scalability at full machine capacity on XSEDE resources such as Ranger and Kraken, and through the use of a hybrid memory approach based on a combination of MPI and Pthreads, Uintah now runs on up to 262k cores on the DOE Jaguar system. In order to extend Uintah to heterogeneous systems, with ever-increasing CPU core counts and additional on-node GPUs, a new dynamic CPU-GPU task scheduler is designed and evaluated in this study. This new scheduler enables Uintah to fully exploit these architectures with support for asynchronous, out-of-order scheduling of both CPU and GPU computational tasks. A new runtime system has also been implemented with an added multi-stage queuing architecture for efficient scheduling of CPU and GPU tasks. This new runtime system automatically handles the details of asynchronous memory copies to and from the GPU and introduces a novel method of pre-fetching and preparing GPU memory prior to GPU task execution. In this study this new design is examined in the context of a developing, hierarchical GPU-based ray tracing radiation transport model that provides Uintah with additional capabilities for heat transfer and electromagnetic wave propagation. The capabilities of this new scheduler design are tested by running at large scale on the modern heterogeneous systems, Keeneland and TitanDev, with up to 360 and 960 GPUs respectively. On these systems, we demonstrate significant speedups per GPU against a standard CPU core for our radiation problem.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programing; G.1.8 [**Mathematics of Computing**]: Partial Differential Equations; G.4 [**Mathematics of Computing**]: Mathematical Software; J.2 [**Computer Applications**]: Physical Sciences and Engineering

Keywords

Uintah, hybrid parallelism, scalability, parallel, adaptive, GPU, heterogeneous systems, Keeneland, TitanDev

1. INTRODUCTION

An important trend in high performance computing is the planning and design of software framework architectures for emerging and future systems with multi-petaflop and eventually exaflop performance. With ever imposed demands on system architects for increased density and power efficiency, traditional systems are now being augmented with an increasing number of graphics processing units (GPUs) [30]. This design is most notable in systems such as the Keeneland Initial Delivery System (KIDS)[1] [32]. This architectural trend is also evidenced in the current upgrade path of the DOE Jaguar[2] system to Titan [24].

Significant challenges face those trying to program for such architectures. The first of these challenges is the prospect of significantly less memory per core as the numbers of cores per socket continues to grow. In order to address this challenge, as recognized by a number of authors [1,25], Uintah [4], an open-source software framework has moved from a model that only uses MPI to one that employs MPI to communicate between nodes and a shared mem-

[1]KIDS is an experimental HP-Nvidia GPU cluster located at the National Institute for Computational Sciences with 120 compute nodes each with two Intel Xeon X5660 (Westmere 6-core @2.8GHz) processors, 24GB memory, InfiniBand QDR (single rail) interconnect and 3 Nvidia Tesla M2090 GPUs

[2]Jaguar is a DOE supercomputer located at the Oak Ridge National Laboratory with 18,688 compute nodes each of which contains a single 16-core AMD Opteron 6200 Series (Interlagos cores @2.6GHz) processor on one of its two sockets, 32GB memory, and a Gemini interconnect, giving 299,008 processing cores. Currently on 960 nodes, the second socket contains a single Nvidia Tesla 20-series GPU. This 960 node partition is known as TitanDev.

ory model using Pthreads to map the work onto available cores in a node [21]. The Uintah task-based model lends itself better to the use of Pthreads rather than OpenMP. This approach has led to the development of a multi-threaded MPI runtime system, including a threaded task scheduler that has enabled Uintah to show excellent strong and weak scaling up to 196K cores on the DOE Jaguar XT5 system and good initial scaling to 262k cores on the upgraded DOE Jaguar XK6 system [28]. Using this approach has reduced Uintah's on-node memory usage by up to 80% [21].

A second challenge posed by such architectures is the design of runtime systems that maximize system utilization by fully exploiting all available processing resources on-node. Central to this goal is overcoming the inherent bandwidth bottleneck of PCI-express (PCIe) transfers to-and-from the GPU, as discrete GPUs are typically hosted in PCIe slots. Data copies across the PCIe bus, which has a maximum theoretical bandwidth of 8.0GB/s (PCI Express Gen2 x16 for the Nvidia Tesla C20 series cards). In practice, this rate is closer to 3.3GB/s when using paged memory, and 5.3GB/s using pinned (page-locked) memory. For memory bandwidth bound tasks, this bottleneck requires more advanced techniques to harness the computational power offered by GPUs. Many current approaches to this problem leave CPU cores idle during GPU-based computation, and others simply do not extend focus beyond a single GPU. These approaches waste substantial available computational power.

Uintah is novel in its use of a asynchronous, task-based paradigm, with complete isolation of the application developer from parallelism. The individual tasks are viewed as part of a directed acyclic graph (DAG) and are executed adaptively, asynchronously and often out of order [22]. Uintah uses a novel adaptive meshing approach [18] as well as a variety of fixed mesh and particle solution methods.

In this paper we look at how to extend Uintah's hybrid multi-threaded MPI runtime system [21] to support, schedule and execute both GPU and CPU tasks simultaneously. We examine the design of a CPU-GPU scheduler in the context of a developing scalable hierarchical ray-tracing radiation transport model to provide Uintah with additional capabilities for heat transfer, and electromagnetic wave propagation. This work directly addresses the second major challenge introduced by heterogeneous systems, specifically utilizing all processing resources available on-node. In what follows Section 2 provides an overview of the Uintah software, while Section 3 describes ARCHES, the Uintah component designed for simulation of turbulent reacting flows with participating media radiation, and its ray-tracing radiation transport model for which we are developing GPU-based capabilities. Section 4 briefly describes Uintah's multi-threaded MPI runtime system design [21], which this work extends. The new CPU-GPU task scheduler design and the multitude of techniques used to overlap PCIe transfers and MPI communication with GPU and CPU computation are also described in Section 4. Finally, in Section 5, we describe computational experiments that illustrate the effectiveness of our new hybrid CPU-GPU task scheduler over a range in scales of processor core numbers and GPUs, comparing results with and without GPUs on KIDS and TitanDev, the 960 node parition of the Jaguar system outfitted with GPUs. The paper concludes by describing future work in this area.

2. OVERVIEW OF UINTAH SOFTWARE

The Uintah Software was originally written as part of the University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [8]. C-SAFE, a Department of Energy ASC center, focused on providing science-based tools for the numerical simulation of accidental fires and explosions. The aim of Uintah was to be able to solve complex multi-scale multi-physics problems. Uintah is regularly released as open source software [10].

In order to solve complex multi-scale multi-physics problems, Uintah makes use of a component design that enforces separation between large entities of software that can be swapped in and out, allowing them to be independently developed and tested within the entire framework. This has led to a very flexible simulation package that has been able to simulate a wide variety of problems [3]. The Uintah component approach allows the application developer to only be concerned with solving the partial differential equations on a local set of block-structured adaptive meshes, without worrying about explicit message passing calls or notions of parallelization or load balancing. This approach also allows the developers of the underlying parallel infrastructure to focus on scalability concerns including load balancing, task (component) scheduling and communications. This component based approach to solving complex problems allows improvements in scalability to be immediately applied to applications without any additional work by the application developer.

Uintah currently contains four main simulation algorithms, or components: the ICE compressible multi-material Computational Fluid Dynamics (CFD) formulation, the particle-based Material Point Method (MPM) for structural mechanics, the combined fluid-structure interaction algorithm MPMICE [12], and the ARCHES combustion simulation component that is considered here.

3. THE ARCHES COMBUSTION SIMULATION COMPONENT

The ARCHES component was designed for the simulation of turbulent reacting flows with participating media radiation. It is a three-dimensional, Large Eddy Simulation (LES) code described in [29]. ARCHES uses a low-Mach number ($Ma < 0.3$), variable density formulation to simulate heat, mass, and momentum transport in reacting flows.

The LES algorithm solves the filtered, density-weighted, time-dependent coupled conservation equations for mass, momentum, energy, and particle moment equations in a Cartesian coordinate system [15]. This set of filtered equations is discretized in space and time and solved on a staggered, finite volume mesh. The staggering scheme consists of four offset grids, one for storing scalar quantities and three for each component of the velocity vector. Stability preserving, second and third order explicit time-stepping schemes are used to advance the simulation in time. For the spatial discretization of the LES scalar equations, flux limiting schemes for the convection operator are used to ensure that scalar values remain bounded. For the momentum equation, a central differencing scheme for the convection operator is used for energy conservation. All diffusion terms are computed with a second-order approximation of the gradient. Overall, ARCHES is second-order accurate in space and time. The ARCHES code is massively parallel and highly scalable through its integration in the Uintah framework [27], and also through use of parallel solvers like Hypre [9] and PETSc [2]. As part of the ARCHES development, substantial research has been done on radiative heat transfer using the parallel Discrete Ordinates Method and the P1 approximation to the radiative transport equation [17].

In reacting flow simulations, the main computational cost is the solution of the large number of systems of linear equations required by the Discrete Ordinates Method. While the solution of these systems can be made to scale [28], it is important to reduce this cost. With this cost reduction in mind, more recent work has

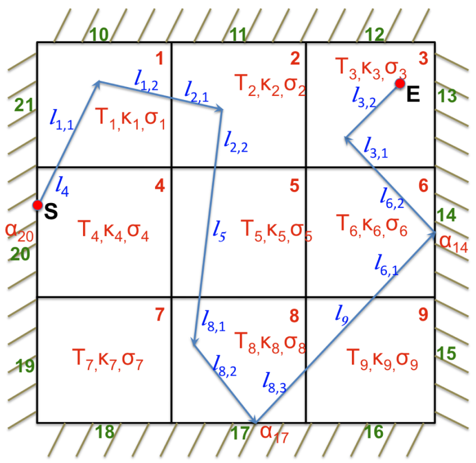

Figure 1: Outline of Reverse Monte Carlo Ray Tracing

been based upon the use of more efficient Reverse Monte Carlo Ray Tracing (RMCRT) methods, e.g. [13, 23, 31]. RMCRT lends itself to scalable parallelism because the intensities of each ray are mutually exclusive. Therefore, multiple rays can be traced simultaneously at any given cell and time step. Creating an efficient, GPU-accelerated software component based on RMCRT methods is the focus of the Computational Experiments section (5).

3.1 Developing a Uintah Radiation Model

In this study, we propose to extend the Uintah framework so that problems involving radiation can *also* be directly supported within Uintah. Some kinds of radiation transport problems already use CFD codes and AMR techniques [14,26]; However, other problems require the concept of tracing rays or particles, such as the simulation of light transport, heat, radiation, or electromagnetic waves.

The approach adopted in Uintah is on using the RMCRT methods, as described by [23]. This approach has the important advantage that by using the principle of reciprocity in radiative transfer, rays are traced backwards from the computational cell thus eliminating the need to track ray bundles that never reach that cell [23]. In RMCRT, rather than following a ray forward and calculating the energy it has lost, the amount of incoming intensity from its path absorbed by the origin where the ray was emitted is calculated. As Sun [31] points out, RMCRT is more amenable to domain decomposition and thus parallel implementation due to the backward nature of the process. Figure 1 shows the back path of a ray from S to the emitter E, on a nine cell structured mesh patch. Each i^{th} cell has its own temperature T_i, absorption coefficient κ_i, scattering coefficient σ_i and appropriate pathlengths $l_{i,j}$. In each case the incoming intensity is calculated, say in cell 4, and then traced back through the other cells. The intensity is integrated along the ray path to compute a divergence of the heat flux or a surface flux. When a ray hits a boundary (as on surface 17 in the figure), it can be either reflected or absorbed depending on the surface properties. Rays are terminated when their intensity is sufficiently small.

Despite the improved efficiency over forward MCRT, there are considerable challenges in the efficient implementation of RMCRT as it is an all-to-all method, where all of the geometry information and property model information for the entire computational domain must be present on each processor [31]. This nature severely limits the size of the problem that can be computed due to memory constraints, especially with large highly resolved physical domains. This challenge is being addressed by using the multi-level mechanisms within Uintah to represent a portion of the domain at a coarser resolution, thus lowering the memory usage [13]. The hybrid memory approach of Uintah also helps as only one copy of geometry is needed per multi-core node. In general, the data required by the RMCRT algorithm is projected to all of the coarser levels, with each level spanning the entire domain. For each fine level patch, data from the coarser levels is retrieved from the Uintah *data warehouse* so it encompasses the patch in a stair step fashion.

CPU-only scalability studies of the RMCRT for the benchmark problem as described by Burns and Christon [5], were run on on a single level [13] with 256^3 cells, using 25 & 100 rays per cell. Each scaling run was run for 10 timesteps, 1 patch per processor, and the mean time per timestep was computed. These preliminary results show reasonable scaling up to 768 cores, above this the loss of scalability is perhaps due to increased communication costs and/or a load imbalance. Nevertheless these results provide a good proof of concept and an excellent starting point for this work.

4. SCHEDULER ARCHITECTURE

As noted in the introduction and in [21], Uintah is a sophisticated computational framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and execute the resulting multi-physics simulation. This is done by utilizing an abstract task-graph representation of parallel computation and communication to express data dependencies between components. The task-graph is a directed acyclic graph of tasks. Each task consumes some input and produces some output (which is in turn the input of some future task). These inputs and outputs are specified for each patch in a structured grid.

Associated with each task is a C++ method which is used to perform the actual computation. In the context of the new hybrid CPU-GPU scheduler, a GPU task is represented by an additional C++ method that is used for GPU kernel setup and invocation. Each component specifies a list of tasks to be performed and the data dependencies between them. The task-graph approach of Uintah shares many features with the migratable object philosophy of Charm++ [16]. In order to increase efficiency, the task graph is created and stored locally [4]. Uintah's CPU-GPU task scheduler is responsible for computing the dependencies of tasks, determining the order of execution and ensuring that the correct inter-process communication is performed [4]. It also ensures that no input or output variable conflicts will exist in any two simultaneously running tasks.

In the migration of the Uintah Computational Framework to hybrid CPU-GPU architectures, we elected to use Nvidia CUDA C/C++ for numerous reasons. Looking at the upgrade path of the DOE Jaguar XK6 system to Titan [24] and also the Keeneland Initial Delivery System (KIDS) [32], we see a trend in the use or planned use of Nvidia GPUs. These are the target machines on which we are already running both CPU and mixed CPU-GPU simulations. Initial runs using ported portions of the CFD component ICE, have demonstrated the ability of our CPU-GPU scheduler to run capability jobs on both KIDS and TitanDev, utilizing all CPU cores and all GPUs simultaneously on each machine. KIDS currently has 1440 CPU cores and 360 Nvidia Tesla 20-series GPUs and TitanDev, 15360 CPU cores and 960 Nvidia Tesla 20-series GPUs.

The principal additions made by our new CPU-GPU scheduler are: significant leveraging of the Nvidia CUDA Asynchronous API [6] to best overlap PCIe transfers and MPI communication with GPU and CPU computation; insulating the component developer from the complexities and details involved with device memory management and asynchronous operations, by automatically managing these operations; using knowledge of the task-graph and task dependencies to pre-fetch data needed for simulation variables prior to task execution. Hence, when a GPU task is ready to run, data needed for the task is already resident in GPU main memory. The GPU task need merely query the scheduler for device pointers and invoke the kernel.

The existing Uintah code base is nearly 700K lines of code, a significant challenge to port in terms of infrastructure and existing simulation components. Although OpenCL [11] has the potential to support more than just GPUs and will be a consideration for use in the future, Nvidia CUDA currently offers far greater support in terms of performance and analysis tools as well as an API allowing for easier performance gains and portability for existing codes. Below we describe the design of our CPU-GPU scheduler and its use of the Nvidia CUDA Asynchronous API [6] in detail.

4.1 Multi-Threaded Runtime System

The overall design of the multi-threaded MPI runtime system is explained in great detail in [21], but to provide context, we review its design briefly here. We then describe in detail how this architecture has been extended by our recent work, adapting Uintah to run on current and emerging heterogeneous systems.

As mentioned in [21], the core scheduler component that stores simulation variables is the *data warehouse*. The *data warehouse* is a hashed-map-based dictionary which maps a variable name and patch ID to a memory address. In the Uintah framework, after the regridder changes the simulation grid and the load balancer generates the patch distribution, the scheduler will create new sets of detailed tasks, compile a new task graph and initialize the *data warehouses*. Uintah's innovative load balancer utilizes space-filling curves in order to cluster patches together [19]. Originally, Uintah used both dynamic and static schedulers, based solely on MPI, in which data structures were created on each MPI process. Although most of Uintah's infrastructure components were carefully designed to be stored in a distributed manner, it was necessary for some data to be stored multiple times, e.g. neighboring patch sets, neighboring tasks and ghost variables. A limitation of pure MPI scheduling was that tasks which were created and executed on the same node could not share data. Uintah's multi-threaded MPI scheduler [21] solves this problem by dynamically assigning tasks to worker threads during execution and shares the same infrastructure components between threads. This design uses one control thread and several worker threads per MPI process. The control thread holds all infrastructure components such as the regridder, the load balancer, the task graph and the *data warehouse* and has read and write access to them.

Central to the design of the dynamic CPU-GPU scheduler (Figure 2) is the multi-stage queuing architecture for efficient scheduling of CPU and GPU tasks. The CPU-GPU scheduler utilizes four task queues: an internal ready queue and an external ready queue for CPU tasks and two queues for the GPU; one for initially ready GPU tasks; those that have requisite simulation variable data copies from host-to-device pending, and a second for the corresponding device-to-host data copies pending completion. First, if a task's internal dependencies are satisfied, then that task will be put in the CPU internal ready queue where it will wait until all required MPI communication has finished. In this same step, if the task is GPU-enabled, the task is then put into the host-to-device copy queue for advancement toward execution. Ultimately, the task goes to the pending device-to-host copies queue. As long as the CPU external queue is not empty, there are always tasks to run. Execution of a task takes place on the first available CPU core or GPU and the scheduler resides on a single, dedicated core per node. CPU tasks are dispatched by the control thread to available CPU cores when they signal the need for work. GPU tasks are assigned in a round-robin fashion to available GPUs on-node once their asynchronous host-to-device data copies have completed. This design helps to overlap MPI communication and asynchronous GPU data transfers with CPU and GPU task execution, significantly reducing MPI wait times.

Algorithm 1 GPU Task Controller Algorithm

```
while $doneTasks < totalTasks$ do
  if $numExternalReadyTasks\ () > 0$ then
    if $highest\ priority\ task\ isGPUEnabled\ ()$ then
      $initiateH2DCopies\ (task, iteration)$
      $task- > markInitiated\ ()$
      $addInitiallyReadyGPUTask\ (task)$
    end if
  end if
  if $numInitiallyReadyGPUTasks\ () > 0$ then
    if $task- > checkH2DCopyDependencies\ ()$ then
      $runGPUTask\ (task, iteration)$
      $addCompletionPendingGPUTask\ (task)$
    end if
  end if
  if $numCompletionPendingGPUTasks() > 0$ then
    if $task- > checkD2HCopyDependencies\ ()$ then
      $postMPISends\ (task, iteration)$
      $reclaimStreams\ (task);$
      $reclaimEvents\ (task);$
      $task- > completed\ ()$
    end if
  end if
end while
```

4.2 Asynchronous GPU Techniques

Significant difficulties arise when mixing concurrency APIs, most notably race conditions, deadlock and general synchronization complexities. Within Uintah's CPU-GPU scheduler is a combination of MPI, Pthreads and Nvidia CUDA, a combination that must be managed with care to avoid such difficulties. Multiple GPUs per node further complicate this situation in the presence of asynchronous memory copies and multiple device contexts (one CUDA calling context per device per process). In the same fashion that Uintah insulates the application developer from the parallelism its infrastructure provides, it also hides and carefully manages details related to GPU memory allocation and transfer. The Fermi-based GPUs found on the target machines mentioned at the beginning of this section offer additional ways to achieve asynchronous concurrent execution of kernels. These GPUs have two copy engines and support multiple kernels running concurrently. Using these features, GPU tasks can be copying data to-and-from the device as well as running multiple kernels simultaneously. In order to exploit these features, the CPU-GPU scheduler creates and manages queues of CUDA *Streams* [6], one for each device on-node. *Streams* provide a means to perform multiple operations simultaneously in that operations from different *streams* can be interleaved and also run concurrently. Our implementation also uses CUDA *Events* [6] which are used for timing and in checking completion of operations such as asynchronous memory copies to-and-from the GPU.

4.3 Extending the Uintah Task Class

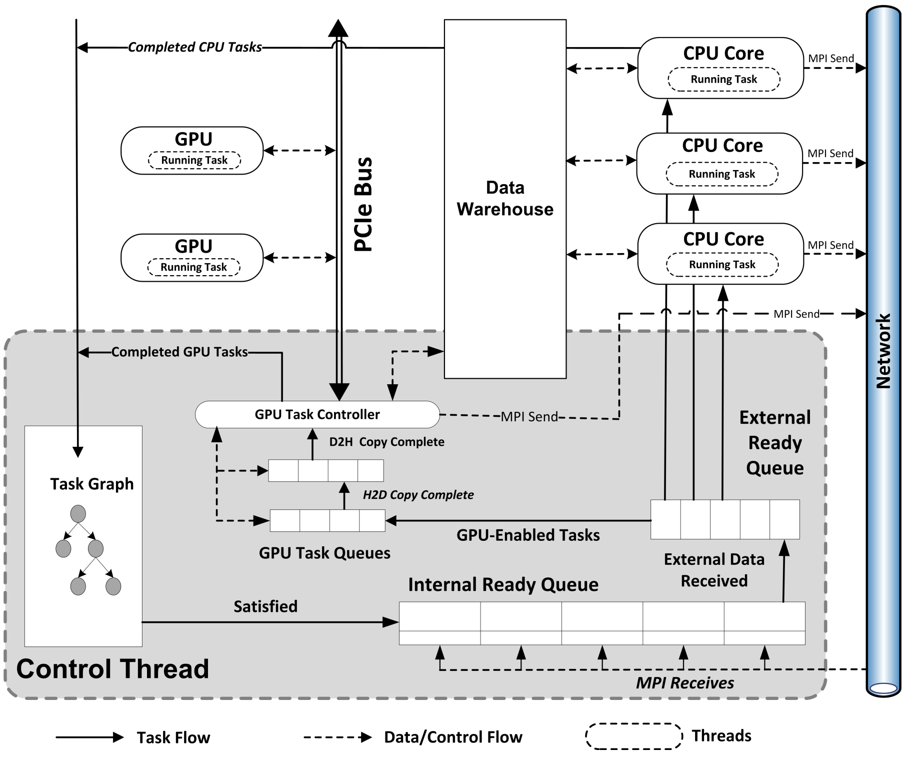

Figure 2: Uintah CPU-GPU Task Scheduler Architecture

Previously, the portion of the Uintah Task class responsible for actual execution of the C++ method representing the computation to perform, was comprised of a single instance of an Action class, which contains a single function pointer to the C++ method to run. With the addition of GPU tasks, we have modified the Uintah Task class to include an additional Action instance with associated pointer to the function containing the GPU kernel setup and invocation. This has been accomplished without altering any existing interface or simulation component.

The design decision to support registration of multiple function pointers was to ultimately add the ability for the scheduler to chose between execution of the CPU or GPU version of the task at runtime. It may be the case that if all on-node GPUs are currently busy or unavailable and there exists an idle CPU core, then it is best to execute a particular task on that CPU core. Currently, if a GPU task has a GPU implementation, it is executed on the GPU. This overall infrastructure design remains broad enough to use other accelerator designs, such as the Intel MIC [7] chip.

4.4 Pre-fetching GPU Task Data

When the CPU-GPU scheduler begins dispatching ready tasks from the CPU external ready queue, it diverts GPU-enabled tasks to the initially-ready GPU task queue. Just prior to this step the CPU-GPU scheduler initiates the device memory allocations and asynchronous host-to-device data copies for the requisite simulation variables. This is accomplished by querying the *data warehouse* for the location and size of the data required for computation and also requesting that the *data warehouse* allocate space for the result of the computation. We have exposed a flat representation of the underlying 3D data structure representing each simulation variable on a patch. This linear array maps relatively easily onto the GPU. To fully exploit the aforementioned levels of concurrency, the host memory to be copied to device must be page-locked. This guarantees the memory will not be paged to disk. The CPU-GPU scheduler then registers for DMA the host memory to be copied to the GPU using a call to `cudaHostRegister()` with the `cudaHostRegisterPortable` flag. This call and flag pair create page-locked (often referred to as pinned) memory from pre-allocated host memory that is considered page-locked by all CUDA contexts. This step avoids a bounce buffer and accelerates PCIe transfers and also eliminates resetting of CUDA contexts when referencing the registered host memory. A call to `cudaHostRegister()` can be cleanly performed from the host without setting a context.

The new scheduler infrastructure maintains a set of queues for *stream* and *event* handles (one per device representing separate contexts for each), and assigns them to each simulation variable per time step to overlap with other host-to-device memory copies as well as kernel execution. These *stream* and *event* handles are stored by the associated Task itself and effectively provide a mechanism to detect completion of asynchronous memory copies without a busy wait, using `cudaEventQuery(event)`. This allows querying the status of all device work preceding the most recent CUDA 4.0 API call to `cudaEventRecord()` [6].

On systems with multiple on-node GPUs such as KIDS, the CPU-GPU scheduler must also manage a CUDA calling context for each device. This is set per device prior to subsequent CUDA API calls on that device. In general, the CPU-GPU scheduler assigns a device to the task itself (round-robin), allocates space on the device, marks the task as initiated and then starts the asynchronous host-to-device memory copies. The entire GPU task processing algorithm is shown in Algorithm 1, where it should be noted that CPU task processing as shown in [21], is interleaved with the GPU task processing.

A call to `cudaEventRecord()` is then made after a call to `cudaMemcpyAsync()` and these *event* pointers are stored with the task itself, and the task is placed into the initially-ready GPU task queue. Priority of GPU tasks is based on the same prioritization algorithm used in the CPU external ready queue, thus the overall task priority is preserved. This is all accomplished asynchronously with respect to the CPU, which is continually responding to requests from idle CPU cores for work. This series of steps essentially prepares the GPU memory needed by the task and is all completed prior to task execution. All data related to each task's host and device pointers are kept in a set of maps maintained by the CPU-GPU scheduler. These maps will ultimately become a separate GPU *data warehouse* in future work.

4.5 GPU Task Execution

During successive iterations of the CPU-GPU scheduler's task controller algorithm, the scheduler checks for existing tasks in the initially-ready GPU task queue and determines if its host-to-device memory copies have completed. This is accomplished by performing `cudaEventQuery(event)` on each of its recorded *events*. The scan is essentially linear in the size of the list of *events* to query, but this size is never greater than say 10 elements, and is essentially constant time, $O(1)$. If all *event* queries return with `cudaSuccess`, the GPU task is ready to run. The C++ method associated with the kernel setup and invocation can then be executed. The component queries the scheduler for device pointers, and a *stream* to associate with the kernel launch. The component then passes these to the kernel routine that performs the computation on the device. To transfer the results of the computation back to the host, the component code requests a device-to-host copy via the infrastructure API. The scheduler in turn initiates the asynchronous memory copy from device to host destination and records the *events* associated with the task. Afterward, the task is placed in the completion pending GPU task queue.

4.6 GPU Task Completion and MPI Sends

Within the CPU-GPU scheduler's task processing loop (Algorithm 1), the *events* in the *stream* associated with the device-to-host memory copy (and kernel used to compute results) of the highest priority GPU task can be queried for completion. Success returned on each of a task's *events* indicates the task has completed execution. The results are then guaranteed to be in the host-side *data warehouse*. At this point, the task can be marked as completed and the CPU-GPU scheduler then reclaims all of the *events* and *streams* used by the task. MPI sends from the GPU task can then be posted. The GPU task is finally removed from the completion pending task queue, allowing other dependent tasks to proceed.

5. COMPUTATIONAL EXPERIMENTS

In this section we examine the performance of Uintah's new hybrid CPU-GPU scheduler and runtime system by running the RMCRT benchmark problem described by Burns and Christon in [5]. This problem is run on a single level using both 41^3 and 128^3 cells. In both cases, the CPU-only version of the *RayTrace()* method consumes more than 90% of the total compute time. Significant speedups in this portion of the code yield significant speedups in time to solution.

We choose to use 41^3 initially so the computed divergence of the heat flux can be compared to the data published in [5]. For these runs we used 25, 50 and 100 rays per cell. The testbed RMCRT component was run for 10 timesteps with one patch per core for the CPU implementation and one patch per GPU for the GPU implementation, with the mean time per timestep computed and compared. In what follows, we describe the approach taken in the GPU implementation of the ray tracer, observing the raw speedups obtained, and compare a single Nvidia M2090 GPU against first a single core and then all cores on a node. These cores were Intel Xeon X5660 (Westmere) @2.8GHz and AMD Opteron 6200 Series (Interlagos) @2.6GHz for KIDS and TitanDev respectively. We also examined the scaling behavior of the CPU and GPU implementations.

As mentioned in Section 3, RMCRT lends itself to scalable parallelism because the intensities of each ray are mutually exclusive. Therefore, multiple rays can be traced simultaneously at any given time step in each cell in every Uintah patch. This leads us to the approach we have taken with the GPU implementation. Our GPU RayTrace kernel uses a patch traversal method similar to that used in the existing GPU port of portions of Uintah's CFD code (ICE algorithm). Here we tile 2D slices of the 3D patch with 2D thread-blocks. These slices are in the two fastest moving dimensions (as the patch cells are traversed), X and Y. We assign a single CUDA thread to each computational cell. Each thread (within a thread-block) is then responsible for tracing the set of rays associated with its respective cell for each slice. Each thread calculates the sum of the intensities from its set of rays, and the divergence of the heat flux for the cell, completely independent of other threads. This avoids potentially costly atomic operations and synchronization. This approach also allows for a single kernel launch per timestep, avoiding the overhead associated with multiple kernel launches.

Single CPU Core vs. Single GPU				
Machine	Rays	CPU (s)	GPU (s)	Speedup
Keeneland	25	28.32	1.16	24.41
1 Core	50	56.22	1.86	30.23
Intel	100	112.73	3.16	35.67
TitanDev	25	57.82	1.00	57.82
1 core	50	116.71	1.66	70.31
AMD	100	230.63	3.00	76.88

All CPU Cores vs. Single GPU				
Machine	Rays	CPU (s)	GPU (s)	Speedup
Keeneland	25	4.89	1.16	4.22
12 Cores	50	9.08	1.86	4.88
Intel	100	18.56	3.16	5.87
TitanDev	25	6.67	1.00	6.67
16 Cores	50	13.98	1.66	8.42
AMD	100	25.63	3.00	8.54

Table 1: GPU Speedups Relative to CPU Implementation on Single Node of Keeneland and TitanDev

Table 1 shows the relative time to solution for both CPU and GPU implementations, and the speedups obtained on the single level RMCRT testbed component using a grid size of 41^3. These timings were a direct comparison on a single node of KIDS and TitanDev for 25, 50 and 100 rays per cell. The first set of timings compare a single CPU core against a single Nvidia M2090 GPU on-node. The second set compare all CPU cores (12 on KIDS and 16 on TitanDev) with the same single GPU. These results show significant speedups on both machines.

As would be expected, the times to solution using the GPU implementation for each run are roughly equal for both machines, however the CPU version of the ray tracer runs considerably faster on Keeneland than TitanDev. An interesting additional result not shown in Table 1, is that when using all three on-node GPUs on Keeneland and comparing against the CPU implementation, the speedups were not as significant. We attribute the slowdown to the NUMA and contention effects within the multi-GPU HP SL390 nodes described in [30]. Currently, our CPU-GPU scheduler has no notion of GPU affinity. Addressing this issue to maximize uti-

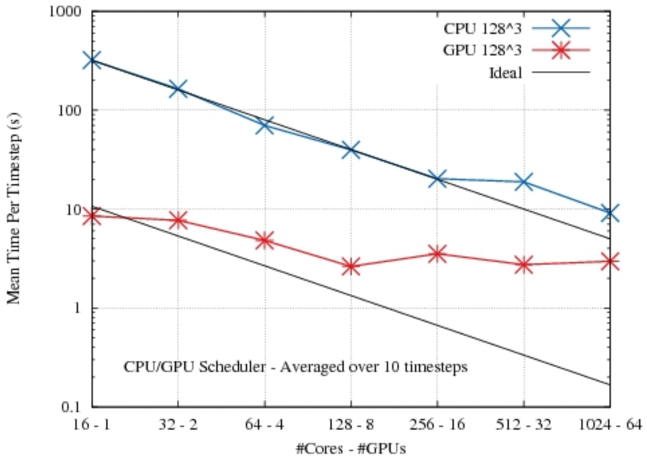

Figure 3: Strong Scaling Comparison on TitanDev

lization of the additional on-node computational resources in multi-GPU systems will be a focal point in future work.

Using the CPU-GPU scheduler, we were able to run capability jobs on both machines, using all CPU cores and GPUs on-node, but saw diminishing returns at larger scale. The all-to-all nature of this problem severely limits the size of the problem that can be computed, and hence does not yet scale well due to memory constraints with large highly resolved physical domains. Figure 3 shows strong scaling results for both CPU and GPU implementation on TitanDev. Similar CPU-only scalability studies of the same single level RMCRT benchmark problem are described in [5]. It is apparent from the figure that the same scalability breakdown shown in [5] on the XSEDE resource Kraken, also occurs on TitanDev (the same result was also evident on Keeneland). Figure 3 additionally illustrates that the GPU implementation quickly runs out of work and strong scaling begins breaking down around eight GPUs. Although the mean time per timestep for the GPU implementation is still considerably lower than the CPU implementation at this point (up to 64 GPUs), ultimately there is insufficient work, and both implementations suffer from the same exorbitant communication costs that are the central difficulty in this problem. Addressing this scalability issue will be a primary focus in future work.

6. FUTURE WORK

We have shown that our present CPU-GPU scheduler design is capable of running Uintah simulations on current and emerging heterogeneous systems, fully utilizing all on-node computational resources. However, we face significant scalability challenges inherent in the RMCRT problem, as shown in our results. Addressing this difficulty will be actively pursued in future work. Other aspects of the CPU-GPU scheduler will be improved upon as well. Most notably, the centralized control thread design will need to be revised by moving to a decentralized design [20]. The central control thread design will become a severe performance bottleneck as CPU core counts on-node continue to grow. An approach that is already being taken on our multi-threaded CPU task scheduler [20], is being considered for the CPU-GPU scheduler. This will allow any thread to execute both CPU and GPU tasks and also to send and receive its own MPI messages. Creation and isolation of a GPU data warehouse is another consideration as is implementing a mechanism for the CPU-GPU scheduler to decide at runtime whether to run a particular task on a CPU core or on a GPU. We are also actively pursuing early access to the Intel MIC [7] chip, with plans to extend Uintah's scheduler to support such accelerator designs.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under subcontracts No. OCI0721659, the NSF OCI PetaApps program, through award OCI 0905068 and by DOE INCITE award CMB015 for time on Jaguar and DOE NETL for funding under NET DE-EE0004449, Uintah was written by the University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE) and funded by the Department of Energy, subcontract No. B524196. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. This research also used resources from the TitanDev program, the 960-node partition of the Cray XK6 Jaguar system outfitted with Nvidia GPUs located at Oak Ridge National Laboratory (ORNL). Our project was granted early access to TitanDev and we would like to thank ORNL and Oak Ridge Leadership Computing Facility for access to TitanDev resources. We would also like to thank all those previously involved with Uintah, Justin Luitjens, Jeremy Thornock, Isaac Hunsaker and Joseph Peterson in particular.

8. REFERENCES

[1] P. Balaji, A. Chan, and R. Thakur E. Lusk W. Gropp. Non-data-communication overheads in MPI: analysis on Blue Gene/P. In *Proc. of the 15th Euro. PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI*, pages 13–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[2] S. Balay, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11, Argonne National Laboratory, 2004.

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

[4] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C.A. Wight, and J.R. Peterson. Uintah - a scalable framework for hazard analysis. In *TG '10: Proc. of 2010 TeraGrid Conference*, New York, NY, USA, 2010. ACM.

[5] 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.

[6] Nvidia Corp. Nvidia Developer Zone Web Page, 2012. http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[7] Intel Corporation. Intel Mic Web Page, 2012. http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[8] J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: A massively parallel problem solving environment. In *Ninth IEEE International Symposium on High Performance and Distributed Computing*, pages 33–41. IEEE, Piscataway, NJ, nov. 2000.

[9] R.D. Falgout, J.E. Jones, and U.M. Yang. *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume UCRL-JRNL-205459, chapter The Design and Implementation of Hypre, a Library of Parallel High

Performance Preconditioners, pages 267–294. Springer-Verlag, 51, 2006.

[10] The Center for the Simulation of Accidental Fires and Explosions. Uintah Web Page, 2012. http://www.uintah.utah.edu/.

[11] The Khronos Group. OpenCL Web Page, 2012. http://www.khronos.org/opencl/.

[12] J. E. Guilkey, T. B. Harman, and B. Banerjee. An eulerian-lagrangian approach for simulating explosions of energetic devices. *Computers and Structures*, 85:660–674, 2007.

[13] I. Hunsaker, T. Harman, J. Thornock, and P. J. Smith. Efficient Parallelization of RMCRT for Large Scale LES Combustion Simulations. Number 2011-3770 in Volume 1, pages 2714–2724, Honolulu, Hawaii, USA, 2011.

[14] J. P. Jessee, Woodrow A. Fiveland, L. H. Howell, P. Colella, and R. B. Pember. An adaptive mesh refinement algorithm for the radiative transport equation. *Journal of Computational Physics*, 139(2):380–398, 1998.

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

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

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

[18] J. Luitjens and M. Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In *Proc. of the 24th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS10)*, 2010.

[19] J. Luitjens, M. Berzins, and T. Henderson. Parallel space-filling curve generation through sorting. *Concurr. Comput. : Pract. Exper.*, 19(10):1387–1402, 2007.

[20] Q. Meng and M. Berzins. Scalable Large-scale Fluid-structure Interaction Solvers in the Uintah Framework via Hybrid Task-based Parallelism Algorithms. Technical Report UUSCI-2012-004, Scientific Computing and Imaging Institute, 2012.

[21] 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.

[22] Q. Meng, J. Luitjens, and M. Berzins. Dynamic task scheduling for the uintah framework. In *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10)*, 2010.

[23] M. F. Modest. Backward Monte Carlo Simulations in Radiative Heat Transfer. *Journal of Heat Transfer*, 125(1):57–62, 2003.

[24] U.S. Department of Energy Oak Ridge Natioanl Laboratory and Oak Ridge Leadership Computing Facility. Titan Web Page, 2011. http://www.olcf.ornl.gov/titan/.

[25] C.D. Ott, E. Schnetter, G. Allen, E. Seidel, J. Tao, and B. Zink. A case study for petascale applications in astrophysics: simulating gamma-ray bursts. In *Proc. of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids.*, MG '08, pages 18:1–18:9, New York, NY, USA, 2008. ACM.

[26] M. Pernice and B. Philip. Solution of equilibrium radiation diffusion problems using implicit adaptive mesh refinement. *SIAM J. Sci. Comput.*, 27(5):1709–1726, 2005.

[27] R.Rawat, J. Spinti, W.Yee, and P.J. Smith. Parallelization of a large scale hydrocarbon pool fire in the Uintah PSE. In *ASME 2002 International Mechanical Engineering Congress and Exposition (IMECE2002)*, pages 49–55, Nov. 2002.

[28] J. Schmidt, J. Thornock, J. Sutherland, and M. Berzins. Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and Hypre. Technical Report UUSCI-2012-002, Scientific Computing and Imaging Institute, 2012.

[29] P. J. Smith, R.Rawat, J. Spinti, S. Kumar, S. Borodai, and A. Violi. Large eddy simulation of accidental fires using massively parallel computers. In *AIAA-2003-3697, 18th AIAA Computational Fluid Dynamics Conference*, June 2003.

[30] K. Spafford, J. S. Meredith, and J. S. Vetter. Quantifying numa and contention effects in multi-gpu systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 11:1–11:7, New York, NY, USA, 2011. ACM.

[31] X. Sun. *Reverse Monte Carlo ray-tracing for radiative heat transfer in combustion systems*. PhD thesis, Dept. of Chemical Engineering, University of Utah, 2009.

[32] J.S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili. Keeneland Web Page, 2009. http://keeneland.gatech.edu/.