PORTABLE AND PERFORMANT GPU/HETEROGENEOUS ASYNCHRONOUS MANY-TASK RUNTIME SYSTEM

by

Bradley Peterson

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

December 2019

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Bradley Peterson**
has been approved by the following supervisory committee members:

Martin Berzins ,	Chair(s)	**May 10, 2019** Date Approved
Mike Kirby ,	Member	**May 10, 2019** Date Approved
Mary Hall ,	Member	**May 10, 2019** Date Approved
Hari Sundar ,	Member	**May 10, 2019** Date Approved
James Sutherland ,	Member	**May 10, 2019** Date Approved

by **Ross Whitaker** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Asynchronous many-task (AMT) runtimes are maturing as a model for computing simulations on a diverse range of architectures at large-scale. The Uintah AMT framework is driven by a philosophy of maintaining an application layer distinct from the underlying runtime while operating on an adaptive mesh grid. This model has enabled task developers to focus on writing task code while minimizing their interaction with MPI transfers, halo processing, data stores, coherency of simulation variables, and proper ordering of task execution. Further, Uintah is implementing an architecture portable solution by utilizing the Kokkos programming portability layer so that application tasks can be written in one codebase and performantly executed on CPUs, GPUs, Intel Xeon Phis, and other future architectures.

Of these architectures, it is perhaps Nvidia GPUs that introduce the greatest usability and portability challenges for AMT runtimes. Specifically, Nvidia GPUs require code to adhere to a proprietary programming model, use separate high capacity memory, utilize asynchrony of data movement and execution, and partition execution units among many streaming multiprocessors. Numerous novel solutions to both Uintah and Kokkos are required to abstract these GPU features into an AMT runtime while preserving an application layer and enabling portability.

The focus of this AMT research is largely split into two main parts, performance and portability. Runtime performance comes from 1) minimizing runtime overhead when preparing simulation variables for tasks prior to execution, and 2) executing a heterogeneous mixture of tasks to keep compute node processing units busy. Preparation of simulation variables, especially halo processing, receives significant emphasis as Uintah's target problems heavily rely on local and global halos. In addition, this work covers automated data movement of simulation variables between host and GPU memory as well as distributing tasks throughout a GPU for execution.

Portability is a productivity necessity as application developers struggle to maintain

three sets of code per task, namely code for single CPU core execution, CUDA code for GPU tasks, and a third set of code for Xeon Phi parallel execution. Programming portability layers, such as Kokkos, provide a framework for this portability, however, Kokkos itself requires modifications to support GPU execution of finer grained tasks typical of AMT runtimes like Uintah. Currently, Kokkos GPU parallel loop execution is bulk-synchronous. This research demonstrates a model for portable loops that is asynchronous, nonblocking, and performant. Additionally, integrating GPU portability into Uintah required additional modifications to aid the application developer in avoiding Kokkos specific details.

This research concludes by demonstrating a GPU-enabled AMT runtime that is both performant and portable. Further, application developers are not burdened with additional architecture specific requirements. Results are demonstrated using production task codebases written for CPUs, GPUs, and Kokkos portability and executed in GPU homogeneous and CPU/GPU heterogeneous environments.

For my wife, Janelle, and my children, who patiently supported me.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge those in the Uintah team who helped guide me with publications and code development. In particular, (listed in alphabetical order), Derek Harris, Todd Harman, John Holmen, Alan Humphrey, John Schmidt, and Dan Sunderland. I would also like to thank my committee for their friendly support. Also, my advisor, Martin Berzins, for never compromising by always insisting on high quality and standards. Finally, my family for supporting me with my numerous responsibilities.

Funding Acknowledgements

- This work was funded in part by National Science Foundation XPS Award under grant number 1337145.
- This work was supported by the Department of Energy, National Nuclear Security Administration, under award number DE-NA0002375.
- This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC05-00OR22725.
- Awards of computer time were provided by the Oak Ridge Leadership Computing Facility ALCC awards CMB109, "Large Scale Turbulent Clean Coal Combustion" and CSC188, "Demonstration of the Scalability of Programming Environments By Simulation Multi-Scale Applications"
- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357.
- Additional support for John Holmen's work cited in this dissertation comes from the Intel Parallel Computing Centers program.

CHAPTER 1

INTRODUCTION

The growing complexity of parallel computing architectures results in applications adopting abstractions to harness the computational potential. Among these abstractions are frameworks called **Asynchronous Many-Task Runtimes** (AMT runtimes), which assist application developers in subdividing parallel computing code into separate tasks, and executing those tasks appropriately on varying computer architectures on many compute nodes. In the last decade, application developers have heavily utilized Graphical Processing Units (**GPUs**) for high-performance parallel computations. GPUs contain hundreds to thousands of processors and can compute specific problems faster and with less energy than CPUs. However, GPUs introduce numerous new challenges not experienced with typical CPU processor architectures. While asynchronous many-task runtimes are evolving to make use of GPUs, no clear dominant methodology pattern has emerged. The lack of a methodology coupled to the need to use GPUs with Uintah has motivated this work.

1.1 Motivation

Nvidia GPUs are currently the most popular accelerators on the market [7]. The November 2018 Top500 supercomputer list [8] features 138 accelerator/co-processor architectures, and of these, 136 are Nvidia GPUs. The Uintah AMT runtime [1, 9] was primarily designed for CPU-based large scale supercomputers, and a full Nvidia GPU implementation is needed.

The Uintah AMT runtime supplies application developers the means to write task code for CPU execution through a model enforcing minimal interaction between the application developer and the underlying runtime. Individuals with a background outside of software development routinely utilize Uintah to simulate large scale computational problems without concerning themselves with the details of task execution or data dependency sharing among tasks. Uintah's runtime is capable of analyzing an application developer's

tasks, automatically partitioning the tasks across many compute nodes, exchanging data dependencies among compute nodes, and executing tasks by utilizing any available CPU resource within a compute node.

Early work by Humphrey et al. [3, 9, 10] implemented an Nvidia GPU-based task execution model for Uintah. While this model was successful in a limited class of problems, it didn't achieve the development ease and computing performance of Uintah's CPU task model. The Uintah AMT runtime itself inefficiently handled short-lived GPU tasks (i.e., those which executed on the order of milliseconds), such that the runtime overhead to prepare GPU tasks took longer than had CPU tasks been used instead. This initial GPU runtime also *synchronously* executed GPU tasks, in contrast with an asynchronous many-task runtime, and this synchronous execution is a potential source of performance slowdown.

Additionally, application developers implementing GPU tasks in Uintah had difficulty in six areas. 1) Duplicating CPU code logic into an Nvidia's GPU programming language. 2) Finding alternatives to common CPU library tools. 3) Specifying efficient parameters for GPU task execution. 4) Partitioning work to fully utilize all GPU resources. 5) Debugging GPU task code. 6) Maintaining this task code in the months or years that followed.

Prior work [11] explored plans to overcome challenges posed by items #1 and #2 in the previous list by implementing the Kokkos framework into Uintah. Kokkos [12] gives application developers the ability to write portable parallel loop code, where the code is only written once and compiled for various architectures like CPUs, GPUs, and Intel Xeon Phi Knights Landing (KNL) processors. Holmen et al. [13] later implemented Kokkos into Uintah targeting CPUs and Intel Xeon Phi KNL processors. However, implementing Kokkos into Uintah for GPUs faced numerous Uintah design challenges far beyond those experienced in the CPU/KNL implementation. Further, Kokkos itself only synchronously executed parallel loop codes on GPUs.

1.2 Thesis Statement

This work has a broad goal of extending an existing AMT runtime and portability layer so that application developers can write portable task code capable of compilation and execution on both CPUs and GPUs as easily as they currently do for CPU-only task

code. Achieving the goal comes through two main parts, *performance* and *portability*. *Performance* is defined as 1) efficiently preparing tasks for execution, 2) supporting numerous simultaneous and asynchronous GPU tasks whose execution parameters can be defined by the application developer for efficient execution, and 3) utilizing the same compilation and execution features had no AMT runtime been used at all. *Portability* is defined as the ability to write C++ compatible task code capable of compiling and executing on multiple architectures without changes. This work goes beyond *portable loops* by implementing *portable tasks*, which encompasses both the loop code and all other supporting C++ code inside the task.

Additionally, application developers should have the flexibility of executing a heterogeneous mixture of CPU and GPU tasks. The implemented solution must be adaptable to other current architectures, such as the Intel Xeon Phi KNL processors, as well as future architectures, such as other new hardware accelerator devices or compute nodes utilizing multiple memory hierarchies. Particular emphasis is given towards maintainability for future application developers and runtime developers by avoiding substantial architecture-specific logic. Also, the AMT runtime should retain its ability to scale to many compute nodes.

This dissertation achieves these goals. The scope of work covers modifications to both the Uintah asynchronous many-task runtime [2, 4, 11, 14–16] and the Kokkos portability framework [16, 17].

The remainder of this introductory chapter covers the following topics. Section 1.3 explains why asynchronous many-task runtimes are desirable solutions to assist application developers. Section 1.4 covers difficulties utilizing the Nvidia GPU execution model with the Uintah AMT runtime. Section 1.5 explains difficulties in achieving task portability. Section 1.6 gives an overview of four computational problems which utilizes this work. Unique contributions of this research are given in Section 1.7. Section 1.8 covers recent related Uintah AMT runtime work by other researchers.

1.3 Asynchronous Many-Task Runtimes

No official definition or standard of an AMT runtime exists, though they share commonalities. AMT runtimes may rely on **over decomposition** by partitioning a compu-

tation problem into individual components, then scheduling and executing tasks within those components. A **task** contains program code and either an explicit or implicit set of dependencies. A **task scheduler** ensures that tasks only run when all prior dependencies are met and can optimize task execution through execution order or architecture location. AMT runtimes often, but not always, utilize one or more **data stores**, which help track and manage data variables needed for the application. Target stencil problems problems utilizing AMT runtimes often, but not always, utilize **halo data**, where a new value in a data cell is computed from values in adjacent data cells. AMT runtimes can help facilitate the data transfer of halo data among tasks and compute nodes.

The asynchronous term in an asynchronous many-task runtime refers to **task parallelism**, where multiple tasks can execute simultaneously and independently of each other. Usually, this task execution occurs on multiple **compute nodes**, where a compute node is defined as a physically distinct computer. AMT runtimes also asynchronously execute different tasks within a compute node as well. Bulk-Synchronous Programming (BSP) differs from an AMT runtime, as tasks here are launched similar to a fork-join model. BSP tasks are simultaneously executed, and then a barrier prevents additional execution of tasks until all prior tasks have completed. An AMT runtime seeks to avoid synchronization barriers and execute any task the moment its dependencies are met.

Another way to view an AMT runtime is in its ambitious goal of abstracting numerous categories of parallelism. Modern high-performance computing (HPC) achieves parallelism in six categories: 1) Utilizing multiple CPU cores for thread-level parallelism (TLP). 2) CPU vectorization where one CPU core can issue one single instruction to multiple data values (SIMD). 3) Accelerator devices like GPUs, which are typically SIMD by default. 4) Instruction-level parallelism (ILP) where upcoming instructions which have no data dependencies on the immediately preceding instructions can be processed simultaneously. 5) Multiple compute nodes which coordinate computations among them (MIMD). 6) Overlapping computation in cores with data movement across buses. Traditionally, AMT runtimes abstract items #1, #5, and #6, with item #3 commonly receiving attention in various forms. Many other tools and libraries seek to assist application developers in simplifying HPC parallelism (see Chapter 2), but are typically limited in focusing on one or two categories only.

1.4 Addressing GPU Difficulties for AMT Runtimes

Nvidia GPUs introduce many usability and portability challenges for AMT runtimes. Four, in particular, are 1) utilizing a proprietary programming similar to C++ as well as an associated proprietary compiler, 2) a memory hierarchy with GPU memory separate from host memory, 3) execution models which should utilize all compute cores on the GPU through many overlapping execution units, and 4) asynchronous data copies and code execution.

The first of these challenges, a proprietary programming model, typically is managed by application developers through code duplication. Application developers will write code for CPU execution, and then duplicate that code with some modifications to make it capable of execution on Nvidia GPUs. Such duplication is usually undesirable and difficult to maintain, especially when the duplicated code is thousands of lines or more.

The additional GPU memory challenge requires strategies to copy data between host memory and GPU memory. When execution only requires a few simulation variables, this copying can be handled with relative ease by the application developer. However, when execution requires numerous simulation variables, and some of these simulation variables are shared among tasks, then this copying becomes unwieldy. Worse, numerous memory copies often take far more time than the actual execution of the code itself, so much so that it's often faster to simply run code on CPUs instead. Nvidia offers abstractions to help automatically copy data between memory spaces, but these have the cost of longer execution times.

As many GPU cores as possible should be kept occupied with work during execution. Very large problems easily utilize all compute cores, as Nvidia GPUs support partitioning and work distribution. For smaller problems, it simply isn't possible to spread work among all GPU compute cores, and so application developers can employ an alternative strategy ensuring that the all cores on a GPU are kept busy through numerous simultaneously executing small problems.

Finally, GPUs seek to overlap memory copies and computation to achieve performance through asynchronous actions. For example, a data copy between host and GPU memory can be invoked asynchronously, and the GPU can perform this copy while its compute cores are currently executing. Nvidia offers GPU **streams** to aid this effort, where a single

stream contains a sequence of asynchronous actions to be performed. An application developer can load many actions on a stream, and check that stream for completion. From an application developer's standpoint, asynchronous actions and streams require more work from the traditional synchronous actions, as additional logic is needed to properly utilize and query streams while also retaining knowledge of actions associated in that stream.

This work seeks to address all four of these GPU challenges by providing new abstractions in the Uintah AMT runtime. The latter three challenges are addressed in Chapter 4 and Chapter 5 using 1) automated data copies within a specially designed data store of simulation variables and 2) an automated task scheduler responsible for the preparation and asynchronous execution of GPU tasks. The first challenge is introduced further in the next section and addressed in detail in Chapter 6 and Chapter 7.

1.5 Addressing AMT Runtime Task Portability

Nvidia GPUs utilize a proprietary code model, which frequently results in application developers maintaining two similar sets of codes, one for CPUs and a second for GPUs. A handful of portability tools (see Section 2), such as Kokkos [12], RAJA [18], OpenMP [19], OpenACC [20], etc., have emerged over the last decade allowing for loop code capable of compiling on both CPUs and GPUs. However, AMT runtime tasks typically require code for preparing simulation variables in addition to its loop code. These portability tools do not yet provide an adequate solution for full task portability. Further, no portability tool adequately allowed for both performant GPU task execution on dozens of streams while also enabling full support for CPU SIMD instructions on Intel Xeon Phi processors, for example.

Enabling task portability required modifications to both Kokkos and Uintah which can be categorized into six separate pieces. 1) Allowing an application developer a mechanism to easily declare which architectures a task can compile to. 2) Enabling a portable interface for Uintah's simulation variable data stores, including the ability to automatically obtain halo data. 3) Enabling a heterogeneous mixture of task code, allowing some tasks to compile only for CPUs or Intel Xeon Phis, others for GPUs, and still other tasks for both CPU and GPU tasks. 4) Writing full task code once and compiling the task for all

supported target architectures. 5) Executing portable loops with different architecture specific parameters while retaining simple portability from the application developer's perspective. 6) Modifying Kokkos itself to support full asynchronous execution of GPU tasks on as many streams as desired.

The combination of these six items results in full task portability. When combined with the new Uintah abstractions to support Nvidia GPUs, this work reaches its broad goal. Application developers can easily write portable task code once with the same level of effort as they previously did for CPU only task code, compile that task code to all desired architectures, and performantly run that task code on either CPUs or GPUs.

1.6 Demonstrating Work on Uintah Applications

This research is applied to four types of computations and applications utilizing Uintah. These are the Poisson 3D equation (Section 1.6.1), a Reverse Monte Carlo ray-tracing technique [21] (Section 1.6.2), tasks in the Wasatch project [22] (Section 1.6.3), and tasks in the ARCHES project [23] (Section 1.6.4). These four target problems highlight 1) differences in task execution times (varying from a few milliseconds to several seconds), 2) differences in halo length (varying from one cell of halo data to encompassing the entire domain), 3) differences in loop code complexity, (varying from just a few lines of code and few simulation variables to hundreds of lines of code and dozens of simulation variables), and 4) differences in utilizing Uintah API (such as heavily relying on Uintah portability tools to using Uintah merely as a task executor). A summary of these four computations and applications are given below.

1.6.1 Poisson 3D Equation

The Poisson equation in three dimensions is a discretization of a partial differential equation used to describe heat transfer through some physical medium over time. The heat equation in 3D Cartesian space is given below, (where α is rate of diffusion through the medium):

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)$$

The discretization using a Jacobi iterative method becomes a seven-point stencil and is given as computer code in Algorithm 1

The Poisson equation in 3D a standard basic Uintah benchmark. Poisson 3D tasks execute very quickly, require halo transfer, and have little reuse of data. These tasks do an excellent job exposing inefficiencies in AMT runtime overhead.

Algorithm 1 Pseudocode for the Poisson 3D equation

```
1: for all cells in a mesh patch do
2:     u[x, y, z] = (1.0 / 6)
3:                * (u[x + 1, y, z] + u[x - 1, y, z] + u[x, y + 1, z]
4:                  + u[x, y - 1, z] + u[x, y, z + 1] + u[x, y, z - 1]);
5: end for
6: compute a sum reduction of u on all cells
```

1.6.2 Reverse Monte Carlo Ray-Tracing Technique

A reverse Monte Carlo ray tracing (RMCRT) technique for simulating radiation [21] is a core component of simulating and evaluating large coal boilers for the Utah Carbon Capture Multidisciplinary Simulation Center (CCMSC). Development of the RMCRT algorithm in the Uintah AMT runtime system is funded under the Predictive Science Academic Alliance Program (PSAAP) II program. GPU results from this work have yielded several publications [3, 4, 10, 14–16, 24].

The RMCRT target problem highlights several difficult challenges for AMT runtimes, namely 1) large halo extents that cover most or all of the computational domain, 2) adaptive mesh refinement (AMR) with non-uniform halo requirements where one AMR mesh level has different halo extents than another, and 3) interaction with numerous simulation variables and CPU tasks in production runs. These RMCRT tasks tend to take seconds, or even minutes, to execute, and thus hide architecture latency in copying data between memory spaces and launching tasks. RMCRT tasks represent challenging problems to abstract into a clean, automated API from both an application developer's perspective and from managing all halo dependencies across nodes [15, 24]

Uintah has implemented RMCRT for different architectures and backends. RMCRT versions covered in this work are CPU only code, GPU CUDA code, and Kokkos portable code capable of execution on CPUs, GPUs, and Intel Xeon Phi KNLs. Further, these problems can be further subdivided into operating on a single mesh level or multiple AMR

Algorithm 2 Ray marching pseudocode for Uintah RMCRT radiation model.

```
1:  procedure ray_trace( patches, materials, olddatastore, newdatastore)
2:      for all (cells in a mesh patch) do          ▷ patch-based region of interest
3:          intensity_sum gets 0                    ▷ initialize intensity to 0
4:          for all (rays in a cell) do
5:              find_ray_direction()
6:              find_ray_location()
7:              update_intensity_sum()
8:          end for
9:          compute divergence of heat flux
10:     end for
11: end procedure
```

mesh levels. The Uintah implemented pseudocode for RMCRT is given as Algorithm 2 and 3.

1.6.3 Wasatch

The Wasatch project [22] and its accompanying Embedded Domain Specific Language (EDSL) called Nebo [25, 26], utilizes Uintah for task execution. Wasatch employs a formalism of the DAG approach to generate runtime algorithms [27]. Nebo allows Wasatch developers to write high-level, Matlab-like syntax that can be executed on multiple architectures such as CPUs and GPUs.

Wasatch differs from the prior listed target problems in that Wasatch implements task graphs which are embedded within Uintah tasks themselves. Further, Wasatch relies on its model to execute GPU code kernels directly while obtaining simulation variables using its own custom API. Wasatch tasks executing within Uintah tasks resulted in numerous short-lived GPU tasks. These GPU kernels in these tasks completed execution in just a few milliseconds. These Uintah tasks represent a class of problems where tasks require numerous simulation variables and halo dependencies, while the tasks themselves execute quickly.

1.6.4 ARCHES Char Oxidation

The Uintah ARCHES turbulent combustion component is used to compute coal multiphysics for large scale coal boiler problems. As previously described [17], ARCHES is a three-dimensional, large eddy simulation (LES) code that uses a low-Mach number

Algorithm 3 Pseudocode for updating sum of radiative intensities.

```
1:  procedure UPDATE_INTENSITY_SUM(ray_origin,ray_direction,num_steps,lo_idx,hi_idx )
2:      initialize all ray marching variables
3:      while (intensity > threshold) do                          ▷ threshold loop
4:          while (current ray is within computational domain) do
5:              obtain per-cell coefficients
6:              advance ray to next cell      ▷ accounts for moving between mesh levels
7:              update ray marching variables
8:              update ray location
9:              in_domain ← cell_type[curr]          ▷ terminate ray if not a flow cell
10:             compute optical thickness
11:             compute contribution of current cell to intensity_sum
12:         end while
13:     end while
14:     compute wall emissivity
15:     compute intensity
16:     update intensity sum
17: end procedure
```

variable density formulation to simulate heat, mass, and momentum transport in reacting flows [23]. Additional fine-grained CPU tasks are responsible for simulating coal combustion, kinetics, and deposition. ARCHES is second-order accurate in space and time and is highly scalable through Uintah to 512K cores [24] and its coupled solvers, such as hypre [28].

The char oxidation computation of a coal particle involves a complex set of physics which includes mass transport of oxidizers from the bulk gas phase to the surface of the particle, diffusion of oxidizers into the pores of the particle, reaction of solid fuel with local oxidizers, and mass transport of the gas products back to the gas phase. The char oxidation code computes the rate of chemical conversion of solid carbon to gas products, the rate of heat produced by the reactions, and the rate of reduction of particle size [29]. These rates are used in the Direct Quadrature Method of Moments (DQMOM) [30], which subsequently affect the size, temperature, and fuel content of the particle field. For each snapshot of time simulated, an assumption of steady-state is made that produces a non-linear set of coupled equations, which is then solved pointwise at each cell within the computational domain using a Newton algorithm.

The char oxidation model is the most expensive model evaluated during the time integration of physics within ARCHES. The pseudocode is given below as Algorithm 4. The

actual code loop is several hundred lines of code and interacts with dozens of simulation variables. Such code is the most impractical to manage manually through separate CPU and GPU code versions, and likewise demonstrating a portable implementation highlights the practical value of this work.

Algorithm 4 ARCHES Char Oxidation Model Loop Structure

```
1:  for all mesh patches do
2:      for all Gaussian quadrature nodes do
3:          for all cells in a mesh patch do
4:              loop over reactions with an inner loop over reactions
5:              multiple loops over reactions
6:              loop over species
7:              loop over reactions with an inner loop over species
8:              for all Newton iterations do
9:                  multiple loops over reactions
10:                 multiple loops over reactions with inner loops over reactions
11:             end for
12:             loop over reactions
13:         end for
14:     end for
15: end for
```

1.7 Unique Contributions

The unique contributions of this dissertation and its associated publications are as follows:

1) Implementation of an asynchronous GPU Data Store enabling both asynchronous task execution and the ability to obtain data store simulation variables within CUDA code [2, 4];
2) Implementation of GPU task scheduler logic allowing for automatic and efficient halo management of GPU tasks in a CPU/GPU heterogeneous environment [2, 4];
3) Modifying the GPU data store and GPU task scheduler logic to avoid halo duplication when halo extents are very large [15];
4) Demonstrating full GPU asynchrony for the combination of task preparation, halo management, and data stores necessary to support both short-lived both tasks with few halo dependencies and long-lived tasks with many halo dependencies [2, 4, 14, 15];

5) Modification of Kokkos to support asynchronous GPU loop execution [16];
6) Implementing Kokkos GPU support into Uintah enabling portable CPU and GPU loops and demonstrating results on complex application developer codes [11, 16];
7) Implementing numerous changes throughout Uintah enabling portable task declaration and full portable task code [16, 17]. Chapter 7 also demonstrates additional novel work for this item.

1.8 Recent Related Work in Uintah

This research has progressed simultaneously with other dissertations utilizing Uintah. All work has clear separation among them. Alan Humphrey implemented novel mechanisms for efficiently supporting globally coupled problems out to hundreds of thousands of CPU cores and tens of thousands of GPUs [31]. John Holmen's ongoing work is utilizing Kokkos's OpenMP for CPU architectures like the Intel Xeon Phi KNL and utilizing multiple CPU threads per CPU task [13, 16, 17]. Damodar Sahasrabudhe is exploring utilizing Uintah and Kokkos so that portable C++ code can compile as CPU SIMD intrinsic instructions or ILP instructions [32] (see items #2 and #4 in Section 1.3), as well as exploring node failure resiliency without data checkpointing [33].

This work focuses on supporting both GPU task portability and GPU execution of tasks. Portability integration into Uintah (Chapter 7) was done in consultation with John Holmen to ensure tasks remained portable for the Kokkos OpenMP backend using Intel Xeon Phi KNL processors [16]. John Holmen also rewrote most nonportable code into portable code [16].

1.9 Document Organization

The remainder of this dissertation consists of the following chapters: Chapter 2 covers related libraries and tools. Chapter 3 overviews the main components of the Uintah AMT runtime relevant to this work. Chapter 4 covers novel work for GPU data stores. New work for GPU task scheduling is given in Chapter 5. Modifications to Kokkos are presented in Chapter 6. Implementing a fully GPU task portable solution in Uintah with Kokkos is covered in Chapter 7. Conclusions, lessons learned, and future work are given Chapter 8.

CHAPTER 2

RELATED FRAMEWORKS, LIBRARIES, AND TOOLS

AMT runtimes that have demonstrated large-scale scalability or plan to reach that goal utilize varied approaches to aid application developers in their GPU implementations [34]. These varied approaches are in part motivated by different target problems and intended audiences. Likewise, many parallel libraries and parallel tools aiding developer productivity also take varied approaches to support targeted applications. This chapter provides a brief overview of Uintah and its comparisons with other related AMT runtimes and tools. A larger overview of Uintah is given in Chapter 3.

2.1 Brief Uintah Overview

The Uintah Asynchronous Many-Task (AMT) runtime [1, 9] is structured at a high level as shown in Figure 2.1. Like many AMT runtimes, Uintah is responsible for managing a collection of executable tasks, forming a directed acyclic graph (DAG) based on task data dependencies, and then scheduling and executing those tasks. The tasks themselves contain the application developer code. Uintah can simultaneously and asynchronously execute many tasks in parallel. Each task can be executed on one CPU thread [9], multiple CPU threads [11, 13, 16, 17], or multiple GPU threads [2, 4, 16, 17].

Most AMT runtimes provide mechanisms for halo transfer across multiple compute nodes, such as through MPI, a partitioned global address space (PGAS), or a customized approach. Most AMT runtimes provide some form of a data store or a data buffer registration system to better automate data dependency needs for both the runtime and the application developer. Uintah itself makes data management a key feature and contains substantial MPI support and data store functionality.

Uintah is unique among the runtimes listed in this chapter due to its combination of three high-level goals. 1) Strong separation of the application developer from the runtime

through automated data stores and halo transfers both within the node and across nodes. 2) Support for a mixture of problems containing both local halos and global or nearly global halos and across multiple adaptive mesh refinement (AMR) levels. 3) Ability to reach full scale on current machines like DOE Titan[1] and DOE Mira.[2] Uintah is also somewhat unique in that its target problems currently operate on structured 3D adaptive mesh grids, whereas the other runtimes seek for a more generalized solution not dependent on a mesh grid. An excellent example of these Uintah features in action is demonstrated in prior work [15], where a simulation requires each compute node contain thousands of tasks, thousands of simulation variables, and millions of data dependencies both locally and globally across many levels of an AMR grid.

2.2 Other AMT Runtimes

The sections below briefly describe Charm++, Legion, HPX, PaRSEC, StarPU, DARMA, STAPL, and OCR. This list is not exhaustive of every AMT runtime found in published literature but was selected due to their similarities with Uintah. Afterward, summaries comparing these runtimes are given.

2.2.1 Charm++

Charm++ [35] is designed for a wide audience as a large, monolithic tool aiding developers requiring a prebuilt, mature, AMT runtime. Central to Charm++'s theme is message passing between tasks. Charm++ does not explicitly define tasks, and likewise does not have an explicit task graph, but rather relies on an event-driven message passing interface using callback functions. When some code unit completes, the developer is responsible for invoking the message and its accompanying callback function. Charm++ is particularly effective at load balancing tasks both within a compute node and across nodes.

Data movement to GPU memory and GPU code execution can be realized through the Charm++ GPU Manager [36]. While it is automatic in the sense that the GPU Manager will allocate GPU memory and perform host-to-GPU and GPU-to-host copies, the amount of

---

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

[2]Mira's nodes host a 16-core 1.6 GHz IBM PowerPC A2 processor. The entire machine offers 786,432 CPU cores across 49,152 nodes and 760 TB of RAM.

development steps required to perform these steps are effectively equivalent to performing them through native CUDA code. The GPU Manager requires the user to provide his or her CUDA kernels, the amount of GPU memory buffers, and size of each buffer. The user is also responsible for providing a callback function when a GPU kernel completes. Data copies and kernel execution can be realized asynchronously to support overlapping kernels.

Regarding portability, the ACCEL framework [37] addition to Charm++ contributes a unique approach to supplying one block of code capable of compilation and execution on different architectures. This approach does not use functors or lambda expressions but does use preprocessor approaches to target a single region of code for later CPU or GPU execution. ACCEL differs from the Kokkos approach in that ACCEL focuses more on load balancing strategies and avoids providing `parallel_for` or `parallel_reduce` constructs. Further, ACCEL works on top of the Charm++ GPU Manager and seeks to provide automatic CUDA kernel code generation, but its feature set is limited by effectively attempting to compile the same C++ code on a CPU compiler and then compiling it a second time on a CUDA compiler.

2.2.2 Legion

The Legion [38] asynchronous many-task runtime system is fundamentally designed around a bottom-up approach of managing both task dependencies and data dependencies. The Legion runtime system automates the dependency analysis after first requiring the application developer to supply a detailed structure of a task's data dependencies. The application developer is expected to have a solid understanding of Legion's theoretical framework and extensive API to properly code application tasks that interact with the runtime. Where Uintah seeks ease of development for application developers, Legion insists developers retain as much control over parallelism and data movement as possible.

Legion's also introduces its own language called Regent [39, 40]. The overall goal of Regent is to become an architecture-independent language that can compile and execute on multiple architectures. For now, Regent relies on an LLVM [41] code translator, and recent work [42] explains that the "LLVM code translator works well for host CPU code, but is not sufficient for tasks that will be run on CUDA-capable processors." More recently,

Legion has switched to runtime code generation of LLVM via Terra [43], with the main intended purpose to provide more efficient CPU vector instructions for LLVM.

Legion utilizes a partitioned global address space (PGAS) instead of MPI or its message passing network. Legion avoids mapping task dependency logic of tasks into MPI messages and also avoids implementing some form of an MPI+X mechanism for heterogeneous nodal architectures [40].

2.2.3 HPX

The HPX runtime [44] recently reached version 1.1.0 but still awaits the introduction of many important features. Its design strategy is both theoretical and bottom-up with the goal of providing a general asynchronous many-task runtime solution that is highly dependent on existing and forthcoming C++ standards. HPX uses task scheduling and message passing to enable asynchrony and proper ordering of tasks.

Much of HPX's development efforts are designed around the overall theoretical framework. Our search of literature indicates HPX has been shown to scale only to 1K nodes [44]. At the moment, HPX has no built-in support data stores, automatic data dependency analysis, halo scattering and gathering, etc. Internodal memory movement is achieved through a global address space. GPU efforts have been focused on finding a standardized framework capable of execution on both NVidia and non-NVidia hardware, with much of the emphasis placed on OpenCL [45].

2.2.4 PaRSEC

PaRSEC [46] contains many similarities with Uintah in that the runtime automates data transfer both through MPI for internode communication and between host and GPU memory for intranode communication. In PaRSEC, data coherence utilizes a simplified version of the MOESI cache coherency protocol [47]. Data dependencies are expressed by defining data flows among tasks using their customized JDF Format to help generate PaRSEC's DAG. If MPI is used, the user provides nodal communication information through a process patterned after MPI_Datatypes.

A frequent target problem of PaRSEC is linear algebra [48]. For example, a triangular matrix solve can have its dependencies implemented cleanly using the JDF format such that when a block of an input matrix has been fully utilized in the computation, subsequent

work in blocks dependent on this newly completed block can proceed. The fine-grained control over these dependencies allows ParSEC to avoid costly synchronization points and achieve excellent nodal utilization [49]. Nvidia GPU support is nonportable in that PaRSEC tasks simply invoke CUDA kernels compiled from host code.

2.2.5 StarPU

StarPU [50] describes itself as a task programming library for hybrid architectures. StarPU handles automatic task dependencies, heterogeneous scheduling, and data transfers between different memory spaces. StarPU targets CPUs, GPUs, Xeon Phis. Portability is managed by giving each task its own architecture specific function call so that a task's parallel code section can have one function for CPU code and a second function for all GPU related code.

StarPU provides an API through C compatible extensions. This API is somewhat similar to Uintah in that it provides some separation of the application developer and the underlying runtime. Data copies between memory spaces are managed using a process very similar to cache coherency protocols. However, halo transfers are not automated and must be accomplished through user-defined tasks. For example, in a stencil-27 problem on a structured grid transferring one cell layer of halo data would require 27 additional tasks. Some application developer interaction is required to aid StarPU in MPI transfers among nodes.

The runtime supports fine-grained tasks, detailed dependencies among these tasks, and has demonstrated efficient nodal usage while avoiding synchronization barriers [51]. Linear algebra solvers are by far the most common use case of those adopting StarPU. Additionally, StarPU has been utilized for climate/weather forecasting, computational astronomy, and N-body simulations.

2.2.6 DARMA

DARMA is an AMT runtime programming model from Sandia National Laboratories. This project was born out of an initial 125-page report [34] which summarized the state of three AMT runtimes (Charm++, Legion, and Uintah). The report's conclusion desired a new programming model that is more generalized and lower level. Quoting the report, "We believe the best design path going forward involves the development of a runtime

that can accommodate and couple a diverse range of application execution patterns, as this approach avoids the reengineering of application-specific ad-hoc solutions." While somewhat similar to HPX (Section 2.2.3) in that DARMA seeks a lower level framework, DARMA is more focused at supporting and standardizing best practices found among the existing AMT community, rather than focusing first on instituting effectively new C++ standards. For clarity, DARMA explicitly states they are not trying to replace all AMT systems or attempting to be a new AMT standard [52].

DARMA's lower level framework and its desire to be application-requirement driven implies that runtimes like Uintah could one day be built on top of DARMA, provided Uintah's API were properly mapped to DARMA's API. DARMA seeks to use a data store generalized far beyond Uintah's approach, requiring the application developer to implement individual data structure logic using DARMA API. Parallel regions of code are assumed to be handled using Kokkos. Regarding task sequencing, DARMA is unique in that it heavily utilizes C++ metaprogramming techniques and incorrect sequencing of tasks can be discovered at compile time through a failed compilation, rather than at runtime with the application generating an error message [53]. A search of literature indicates DARMA has yet to demonstrate effective GPU support or operation on multiple nodes.

2.2.7 STAPL

The Standard Template Adaptive Parallel Library (STAPL) started in 1998 from researchers at Texas A&M University [54]. STAPL is largely focused on a set of parallel tools extending the STL, and ultimately inspired the Intel Thread Building Blocks library (TBB). STAPL goes beyond Intel's TBB with more algorithms, data structures, a runtime, and the ability to manage machines with shared and distributed memory.

STAPL utilizes a task graph called *pRange*, which cover a task graph execution model with similarities to Uintah. Tasks are defined with both an entry function and associated data dependencies. Those data dependencies are then used to ensure the correct execution of tasks. Tasks themselves execute using OpenMP or Pthread back-ends, with multiple threads per node supported. GPUs are not currently supported.

STAPL's runtime has three main features, 1) an Adaptive Remote Method Invoca-

tion (ARMI) library that hides all communication details, whether they be through MPI, OpenMP, or Pthreads, 2) a task scheduler, and 3) a task executor that works with pRanges and ensures task dependencies are met. The ARMI feature, in particular, is more in line with Charm++'s model and is an asynchronous version of Java's RMI. In the context of a runtime system, it gives load-balancing flexibility by passing both data and methods. For tasks requiring a significant amount of communication per work, STAPL's ARMI was demonstrated to perform better than MPI approaches [55].

2.2.8 OCR

The Open Community Runtime (OCR) [56] is another approach toward a standards-based AMT runtime. OCR is community driven and seeks to be a lower-level approach capable of generalization for a broad class of AMT applications. "OCR's low-level API suggests a natural programming model, but most application programmers will prefer higher-level programming models that run on top of OCR" [56]. The initial vision was released in 2012, and version 1.1 was released in April 2016.

OCR has a strong emphasis toward tying data dependencies with its directed acyclic graph, and in many ways is similar in style to Uintah. OCR tasks must first determine that all data dependencies for a task have been met, then must acquire all data blocks needed. From here, the application task code is executed, and then data blocks are released. OCR avoids managing race conditions on data acquires. Internode data transfers are handled via MPI.

Much of the OCR's applications have been a proof-of-concept, demonstrating its potential scaling out to hundreds of nodes. OCR was recently demonstrated with the Intel Xeon Phi Knights Landing processor [57]. A search of literature yields no implementation with GPUs as yet.

2.2.9 AMT Runtime Scaling and Summary

Uintah has been shown to scale to 16K GPUs and 256K CPUs on DOE Titan [3] and 768K cores on DOE Mira [58]. In a search of literature, we found that Legion has been demonstrated to scale to 8K nodes on DOE Titan [59], Charm++ to 512K cores on DOE Mira [60], HPX to 1K nodes [44], PaRSEC to 23,868 cores [61], and StarPU to 256 nodes on the Occigen cluster located at CINES, France [62]. Clearly, work is ongoing with all these

runtimes, however.

Of these, only Charm++ and Uintah has been demonstrated to scale to hundreds of thousands of cores. Uintah is unique among these two as Uintah can further support globally or nearly globally coupled problems with minimal application developer interaction.

A summary of several AMT runtimes is given in Table 2.1. Not all runtimes in this chapter are presented in that table but represent those runtimes with closer similarities to Uintah.

2.3 Parallel and Portability Tools and Libraries

Many tools exist to aid application developer productivity in abstracting the complexities involved with parallel programming. These tools may seek cleaner solutions for code portability, code simplification, or memory management in multiple memory hierarchies. AMT runtimes may choose to utilize these tools to fulfill a specific need not yet met by that runtime.

2.3.1 Kokkos

Kokkos [12] describes itself as "a programming model in C++ for writing performance portable applications targeting all major HPC platforms. For that purpose, it provides abstractions for both parallel execution of code and data management. Kokkos is designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. It currently can use OpenMP, Pthreads, and CUDA as backend programming models." The most fundamental component of Kokkos requires developers write functors or lambda expressions which are then placed inside a `parallel_for`, `parallel_reduce`, or `parallel_scan` construct. Alongside these parallel constructs are arguments specifying the number of threads desired, execution patterns, and targeted execution space. A compiler for that architecture then compiles the functors and lambda expressions, and Kokkos will execute the functors or lambda expressions in the manner specified.

Kokkos parallel loops are generally compiled into the resulting binary. Kokkos heavily utilizes C++ templating and subsequently utilizes compile-time conditional branching to avoid runtime branching. The resulting compiled machine code has little, if any, Kokkos

code. A compiled library option is available, but this approach is typically not used. Overall, performant loop execution is a high priority. For example, a `parallel_reduce` executed in CUDA wraps both the functor and reduction logic inside a single kernel call, rather than employing a separate kernel for the reduction step.

The second major feature of Kokkos is aligning its parallel loops with the layout of data variables. Kokkos maintains abstracted data objects supporting various layouts in up to 8 dimensions. Kokkos Managed Views are data objects maintained by Kokkos itself, while Unmanaged Views are simply wrapped data pointers. Managed Views have API to aid in copying data between memory spaces, such as host memory and GPU memory, but each host-to-GPU and GPU-to-host copy comes with the cost of a synchronization barrier.

Kokkos also maintains additional API to aid developers with portability libraries for atomic operations, locking mechanisms, basic task graph implementations, and random number generation. These extensive portable libraries set Kokkos apart from other portable features mentioned in this chapter.

Kokkos so far actively refrains from encroaching in AMT runtime design space. Kokkos does not yet have any support for a concurrent data store interface, internode memory movement, automatic halo management, asynchrony in data movement, heterogeneity in task scheduling, or overlapping of GPU execution. These issues are left for the application developer to manage manually, or by employing other tools, libraries, or runtimes. Future Kokkos developments may support internode memory movement.

2.3.2 RAJA

RAJA [18] is a Lawrence Livermore National Labs (LLNL) project which contains many similarities to Kokkos in its parallel looping design patterns. RAJA obtains portability through functors and lambda expressions. While RAJA is not yet at version 1.0, and its feature set is perhaps not as mature concerning memory management and architecture-aware execution as Kokkos, it is being used within a number of DOE applications at LLNL.

The RAJA team is actively developing CHAI [63] to help facilitate portable memory movement of data variables, Sidre for data store management, and Umpire for portable memory allocation and querying. RAJA is implementing stream support, but at this time requires synchronization to retrieve a reduction value computed in an asynchronous par-

allel reduction.

2.3.3 Hemi

Hemi [64] contains a smaller set of features compared to Kokkos and RAJA. Like Kokkos and RAJA, portable code is achieved through functors and lambda expressions. Hemi supports `parallel_for` loops iterating over a configurable range, and provides basic data containers for automatic allocation and data movement between host and GPU memory, with its last release in 2015. Hemi has no support for parallel reductions.

2.3.4 OCCA

OCCA [65] describes itself with the following characteristics [66]: "Occa is an open-source library which aims to:

- Make it easy to program different types of devices (e.g., CPU, GPU, FPGA)
- Provide a unified API for interacting with backend device APIs (e.g., OpenMP, CUDA, OpenCL)
- Use just-in-time compilation to build backend kernels
- Provide a kernel language, a minor extension to C, to abstract programming for each backend."

OCCA is fundamentally designed around three concepts: host, device, and kernel. The host runs application code, the device runs the kernel (e.g., CPUs, GPUs, Xeon Phi, etc.), and the kernel contains the portable, parallel code. Each of these three concepts utilizes OCCA syntax for portability.

The kernel's parallel code is managed through one tagged `@outer` loop and zero to many tagged `@inner` loops. The following example is taken from the OCCA user guide [66]:

```
@kernel void loop(const int N, ...) {
  for (int group = 0; group < N; group += blockSize; @outer) {
    for (int id = group; id < (group + blockSize); ++id; @inner) {
      if (id < N) {
        // Parallel work
      }
    }
  }
}
```

From a GPU's perspective, the entire function named `loop()` corresponds to a CUDA kernel, the outer loop corresponds to CUDA blocks, and the inner loop corresponds to

CUDA threads. For OpenCL, these correspond to `workgroup` and `workitem` respectively. The OpenMP implementation has no outer/inner hierarchical parallelism correspondence, and instead simply treats it as a single parallel loop.

Once a kernel is written and saved to file, it is compiled just-in-time (JIT) when invoked in OCCA host code at runtime. The target architecture at runtime determines the compilation to the appropriate back-end (current supported back-ends are Serial, OpenMP, OpenCL, and CUDA). Compiled kernels are cached within an application run and can be reused. The host code which launches the kernels can be written in C, C++, or Python. The device code must be C with OCCA supported attributes.

OCCA also provides portability for memory management. Memory allocation and data copies between memory spaces are directly exposed. For example, a typical GPU related problem in OCCA requires allocating data in both host and device memory, initializing data in host memory, then manually copying the results into device memory.

CUDA streams are likewise abstracted behind OCCA API. Multiple streams are supported, and both the kernels and memory copies can use CUDA streams. OCCA also has support for shared memory, local thread memory, thread synchronization, and atomics.

Overall, OCCA largely contains a one-to-one correspondence of common CUDA features, abstracted into new API, which can then be re-compiled and executed on different back-ends for different devices. Other needs, such as architecture specific iteration patterns, portable tools like random number generation, avoiding unnecessary host-to-device copies, are not provided.

2.3.5 Nvidia Thrust

Nvidia's Thrust [67] can provide portable lambda expressions, but most of its feature set targets portable containers and high-level algorithms which can operate under CUDA, Intel's Threading Building Blocks (TBB), and OpenMP. Nvidia Thrust's algorithm toolkit has loops, reductions, and scans, which are represented through `thrust::for_each`, `thrust::reduce`, `thrust::inclusive_scan`, and `thrust::exclusive_scan` (which offsets by one). These APIs do not function like parallel loop replacements, but are more like the C++ STL model where an array buffer is supplied and iterators are used for the begin and end region. In the case of `thrust::for_each`, a functor must also be supplied. Nvidia

Thrust's execution model is in contrast to other parallel tools like Kokkos and RAJA which don't explicitly tie the parallel API to an array buffer.

Additional tools appear to be borrowed from the C++ STL model, such as `sort`, `copy_if`, `partition`, `remove`, `remove_if`, and `unique`. Iterators again are supplied into these API calls, and the resulting iterators indicate the new region of the array buffer that has undergone the transformation (such as in the case with `remove`).

CUDA streams for asynchronous execution were added to Nvidia Thrust starting in CUDA 6.5 and Thrust 1.8 [68]. This stream support structure breaks Thrust's architecture code portability as the streams must be supplied through a CUDA context. Likewise, CUDA API must be invoked to either determine stream completion or synchronize the stream.

2.3.6 OpenMP

OpenMP [19] has significant and wide adoption in the HPC community. OpenMP contains a rich set of API with compiler directives and library functions. The main target of OpenMP is CPU level parallelism, though OpenMP's version 4.5 specification targeted GPUs, with more support in version 5.0 [69]. However, compilers supporting Nvidia GPUs are still lacking in loop performance [70] and reduction performance [71].

The parallelism's focus is restricted to the node itself, with multiple node communication expected to be supported through a different mechanism. The parallelism follows the fork-join model, where a master thread spawns children threads for an OpenMP parallel statement. OpenMP loop parallelism itself has a static, dynamic, or guided scheduling mechanisms. The static scheduling mode is basic and ensures an equal partitioning of threads to parallel iterations. Dynamic scheduling functions more like a pool of thread workers, where threads which complete their assigned portion can then assist in other pending chunks of parallel iterations. Guided scheduling is like dynamic scheduling, except that the chunk sizes become smaller to potentially better distribute remaining work among remaining open threads.

OpenMP provides many features beyond parallel loop constructs, such as support for atomic operations, SIMD primitives, memory management, and task constructs. With this rich set of API, OpenMP seeks to create a fully portable API that functions at the compiler

level. One drawback in terms of AMT runtimes is that loop and data copy asynchrony is only possible through a `nowait` clause without an abstraction like an asynchronous stream. An AMT runtime looking to use GPU level parallelism would require some alternative mechanism to check for completion of a loop or a data copy, such as a later barrier synchronization or utilizing OpenMP tasks with callback functions.

2.3.7 OpenACC

OpenACC [20] utilizes compiler oriented portability optimization where loops are qualified with pragma directives. OpenACC is in many ways similar to OpenMP. The largest distinction between the two is that OpenACC uses GPUs as the primary target, whereas OpenMP focuses more on CPU level parallelism. This OpenACC targeting is not surprising given that the specification is led mainly by Nvidia.

OpenACC does target many architectures. Vectorized CPU implementations also exist, but vectorization has its limits. The commonly used PGI compiler for OpenACC does not support Xeon Phi KNL AVX-512 bit vector instructions.

Asynchronous execution is supported through an `async` clause and supports up to 16 streams. Regarding reductions, PGI's implementation of OpenACC requires two CUDA kernels, one for the loop and a second for the reduction. This two kernel reduction model is in contrast to Kokkos which requires only one. A synchronization barrier is required to obtain the result of the reduction value.

2.3.8 OpenCL

OpenCL [72] is an open source, standards-based set of API tools maintained by the Khronos Group and is designed to abstract many common features of device accelerator programming. OpenCL contains many similarities with the CUDA programming model while allowing for portable compilation. Unlike CUDA which targets only GPUs, OpenCL targets CPUs, GPUs in general (including AMD's GPUs and Intel's GPUs), and other devices.

An application developer writing code for OpenCL will notice its immediate striking similarity to CUDA's approach. The application developer is responsible for allocating device memory, copying host input into device memory, creating a kernel, launching the kernel, copying results back to host memory from device memory, and then freeing device

resources. In the context of portable execution, OpenCL's device memory and kernel execution map to the actual host memory and CPU, respectively.

OpenCL uses a just-in-time (JIT) compilation approach, where kernels are compiled into an intermediate language called SPIR-V [73]. Khronos is also responsible for maintaining SPIR-V, which itself acts as an abstraction for graphics and shader functionality for OpenCL kernels, OpenGL, and multiple types of architectures.

A major drawback of OpenCL is that its abstractions result in codes which don't map optimally to specific architectures. For example, OpenCL's performance frequently lags behind CUDA code [74]. Much of OpenCL's current efforts are still in the research and design domain, with graphics and FPGAs receiving particular focus.

2.3.8.1 SYCL

Kronos additionally manages SYCL [75], which is meant for "C++ programmers who want all the performance and portability features of OpenCL, but with the flexibility to use higher-level C++ abstractions across the host/device code boundary." SYCL itself uses a `parallel_for` and lambda model much like Kokkos, RAJA, and Hemi. SYCL treats execution units as OpenCL kernels and requires these kernels to be enqueued. From dependency information supplied during the enqueuing process, SYCL builds a directed acyclic graph of dependencies and launches kernels when dependencies are met. Data copies between host and device memory are not explicitly stated through copy API calls, but rather expressed in terms of data accessibility, from which SYCL automatically manages the copies.

2.3.9 CUDA Memory Management

CUDA offers compelling portable memory features. The three discussed here are Unified Memory, and GPUDirect, and CUDA-Aware MPI. A description of their applicability to Uintah is also given.

Unified Memory gives the application developer a single address space, and if managed properly, CUDA will automatically perform all host-to-GPU and GPU-to-host data copies. Unified Memory is an abstraction designed to simplify the development process. CUDA kernels operating in a Unified Memory environment demonstrate significantly slower execution times [76]. Further, any GPU-to-host memory transfer requires a syn-

chronization barrier before CUDA Compute Capability 6.x or expensive page faulting for Compute Capability 6.x [77] and after.

Another CUDA memory management feature, GPUDirect [78], allows for "a direct path for data exchange between the GPU and a third-party peer device using standard features of PCI Express. Examples of third-party devices are: network interfaces, video acquisition devices, storage adapters." GPUDirect is utilized to transmit data from one GPU memory space to another GPU memory space. These GPUs can either be in the same node or different nodes. Central to GPUDirect is creating CUDA pinning memory, as the third-party interacting hardware needs to ensure no paging is used.

CUDA-Aware MPI works on top of GPUDirect to provide an RDMA interface, so an application developer has a seamless way of transmitting data via MPI from GPU memory in one MPI rank to another GPU's memory in another rank. At the time of this writing, supporting MPI implementations are MVAPICH2, OpenMPI, IBM Spectrum MPI moment, and Cray's MPICH. CUDA-Aware MPI has had some difficulty interacting with Unified Memory. For CUDA 6.0 "CUDA Unified Memory is not explicitly supported in combination with GPUDirect RDMA" [78]. OpenMPI v1.8.5 manages Unified Memory "by disabling CUDA IPC and GPUDirect RDMA optimizations on Unified Memory buffer" [79].

Uintah does not utilize any of these features at this time. Unified Memory is largely offered as an abstraction feature at the cost of performance. Uintah can manage data copies automatically through the runtime, and the kernel performance loss is unacceptable. GPUDirect is not used as current Uintah target problems spend a relatively minuscule length of time transferring halo data host-to-GPU and GPU-to-host. However, Uintah is well-situated to utilize GPUDirect if the need arises. Further, pinning memory can be a relatively time-intensive operation. In Uintah's case, pinned memory would be best managed by a pool of previously pinned memory, rather than pinning and unpinning buffers each time step. Cuda-Aware MPI is also not currently utilized used as it requires GPUDirect, and Uintah does not desire to be restricted to a subset of supported MPI implementations. Further, Uintah naturally works within a model where halo packed buffers are staged in host memory and used as needed (Chapter 4 and Chapter 5).

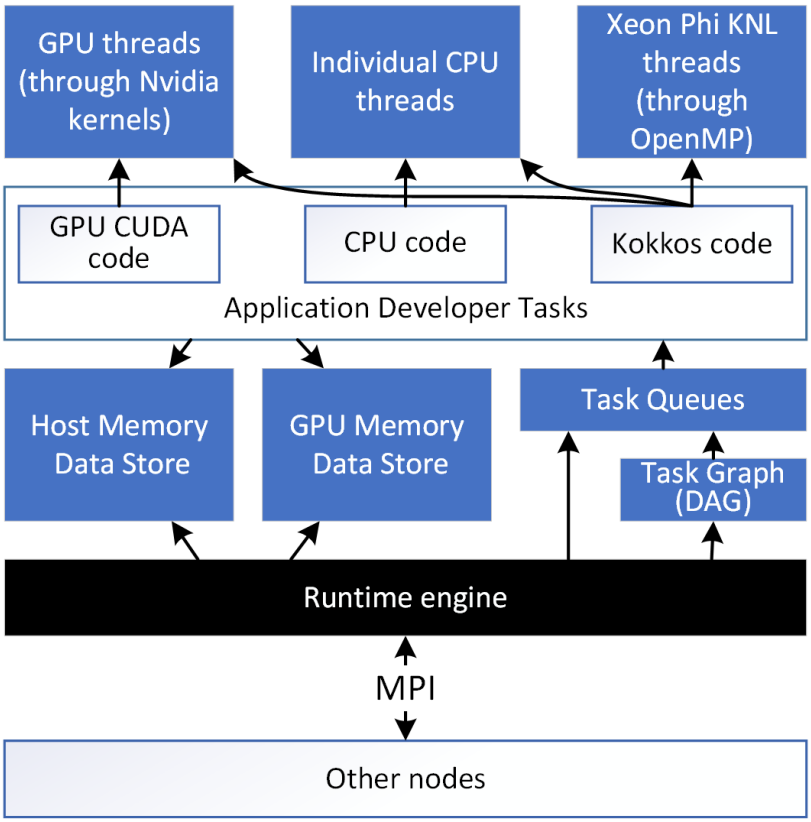

Figure 2.1: Organization of major components and modules comprising Uintah

Table 2.1: A comparison of features of many current AMT runtimes

	Asynchronous Many-Task Runtimes					
	Uintah	Charm++	Legion	HPX	PaRSEC	StarPU
Common usage	Multiphysics on an adaptive mesh grid	Generalized tool	Generalized tool	Generalized tool	Linear algebra	Linear algebra
Developer involvement with runtime	Light	Medium	Heavy	Medium	Medium	Medium
Automatic internodal data movement	Yes	Invoked by user	Yes	Yes	Yes	No
Automatic halo gathering	Host memory (This work for GPU)	No	Yes	No	Yes	No
Data store for application developer	Host memory (This work for GPU)	No	No	No	No	No
Automatic data sharing among tasks	This work	No	Yes	No	Yes	Yes
GPU support	Yes	With add-ons	Yes	No	Yes	Yes
Portable code for CPU and GPU tasks	This work	Some	No	No	No	No

CHAPTER 3

UINTAH OVERVIEW

This chapter covers the constituent parts of Uintah [1, 9], with particular emphasis on relevant material for upcoming chapters. First, a review of traditional applications of Uintah is given in Section 3.1. Section 3.2 will introduce Uintah through the perspective of an application developer, including grid layout, Uintah variable types, task structure, utilizing MPI, and what is meant by automated halo data transfer. Section 3.3 covers the AMT runtime internals, including task graph compilation, task scheduling, and task execution. Sections 3.2 and 3.3 deliberately use two different perspectives to highlight Uintah's fundamental design feature of separating the application layer from the runtime internals.

3.1 Traditional Uintah Applications

Uintah has been used to solve problems involving fluids, solids, combined fluid-structure interaction problems, and turbulent combustion. The most common traditional applications are implemented into Uintah as five separate components. 1) The multimaterial ICE code [80] for both low and high-speed compressible flows, or by using particles [81] with the , particle-based code MPM for structural mechanics. 2) The combined fluid-structure interaction (FSI) algorithm MPM-ICE [82]. 3) The ARCHES turbulent reacting CFD component [23] designed for simulating turbulent reacting flows with participating media radiation. 4) Wasatch [22], a general multiphysics solver for turbulent reacting flows. 5) A multiphysics package for modeling electrochemistry simulations.

3.2 Application Developer Perspective

A Uintah application developer writes task code and interacts with Uintah through three mechanisms. 1) An input file describing the simulation parameters, such as domain space grid layout, simulation variables for output, and total time steps needed to run the

simulation. 2) Declaring all tasks that must run, including the order of the tasks, the input and output simulation variables, and halo extents of simulation variables. 3) The task code itself which obtains simulation variables from Uintah data stores and performs the task computations. The following sections describe these in more detail.

3.2.1 Domain Space Layout

Uintah operates on a structured mesh grid, or more specifically, a three-dimensional Cartesian mesh composed of individual three-dimensional rectangular cuboid cells. The mesh grid is usually a homogeneous set of cells, although Uintah does allow for splicing together different sets of rectangular cuboid cells into one domain. Uintah does not support an unstructured mesh grid at this time.

Adaptive mesh refinement (AMR) is also supported [1]. Application developers typically utilize one coarse mesh level in addition to the fine mesh level, though Uintah can supply more AMR levels as needed. AMR is particularly important for the RMCRT target problem (Section 1.6.2). The application developer is responsible for writing his or her coarsening logic to populate the coarse mesh with data from the fine mesh, as well as determining when to use mesh layers during task code.

3.2.2 Patches

A three-dimensional hexahedral mesh with uniformly shaped cells constitutes a Uintah **patch**. Uintah's design fundamentally relies on this patch concept throughout. For example, each Uintah task routinely operates on a single patch, simulation variables exist on patches, and data stores use patches as a fundamental unit. Most Uintah simulation variables have one or more data items relative to the hexahedral mesh patch.

Uintah allows for varying sizes of patches, though traditionally patch sizes of $10x10x10$, $12x12x12$, $16x16x16$, and $32x32x32$ are the most common. Patch sizes are typically chosen to optimize performance. Finer grained patches and their associated tasks better load balance by spreading work among all available cores within a compute node. Coarser grained patches and their associated tasks better reduce the overhead associated with automated runtime management. The traditional patch sizes tend to provide the best balance for task granularity [83].

3.2.3 Uintah's Five Simulation Variables

Uintah supplies the application developer five types of simulation variables. 1) A collection of data relative to all cells in a single patch, also known as a **grid variable**. 2) A single data unit per patch of cells. 3) One data unit per compute node. 4) Reduction variables, such as a single floating point number that holds an accumulated sum of all cell data in the entire simulation. 5) Particle variables, where individual cells hold particles.

Of these five, the one receiving the most research and usage is the first type listed, the grid variable. The grid variables themselves are further subdivided by Uintah into different flavors to support data that is cell-centered, cell face-centered, or node-centered. Figure 3.1 shows a Uintah patch and some simulation variable types associated with a patch.

These simulation variables are C++ objects, hide the internal details of actual data layout, and provide an application developer an interface to obtain desired data within the object. An application developer can easily access all cells of a grid variable by utilizing C++ compatible iterators associated with the patch. Uintah strongly prefers application developers not have direct access to the underlying data pointer of simulation variables.

3.2.4 Halo Data

Halo data elements are those elements found on other mesh patches that a source mesh patch needs for computation. Stencil computations require data from adjacent mesh patches. Computations that are globally coupled or nearly globally coupled require halo data from mesh patches on most or all of the computational domain. Figure 3.2 demonstrates how an application developer should envision halo data on an individual patch.

Management of halo data (also known as *ghost cells* within the Uintah nomenclature), sets Uintah apart from other AMT runtimes of its class, and likewise has consumed most of Uintah's data store research (Chapter 4). Simulation variables are fundamentally designed to accommodate halo cells, rather than relegating them as an afterthought. Further, Uintah design philosophy insists halo cell management be intuitive and straightforward from the application developer's perspective.

Two of the five Uintah variable types utilize halo data in some form, namely the grid variable types and the particle types. Halo data is simply a layer of additional cells that

belong to neighboring patches, but which are also copied and coupled into a target patch. All patches in the domain will share the same halo cell configuration.

Uintah supplies automatic halo transfer. Figure 3.3 demonstrates how an application developer should envision how halo data transfer functions. Halo data can be copied within simulation variables in a compute node, and can also be copied among compute nodes. In reality, the halo transfers depicted in Figure 3.3 is not a perfect representation of the actual halo transfer mechanics. For example, the green slabs in the figure are not utilized to transfer data within host memory in a compute node, but they are used to transfer to other compute nodes via MPI.

3.2.5 Tasks

A Uintah task contains executable code and dependency information. Uintah uses the dependency information to know when the task should execute relative to other tasks. Uintah application developers are responsible for defining the parameters of each task and writing the executable code.

3.2.5.1 Task Declaration

A task declaration mainly consists of two parts, 1) supplying all simulation variables the task will need, 2) supplying an entry function which will start executing the task code. Additionally, tasks can also declare which architecture or architectures it supports. An example task declaration is shown in Figure 3.4.

In this figure, the task's code will be found in the `Stencil27::taskMethod()` entry function and that task code may be CPU code, GPU CUDA code, or Kokkos parallel code. The task declaration then specifies any *Computes*, *Modifies*, and *Requires* simulation variables. *Computes* are variables which will be allocated for the task and are used to store new data computed during task execution. *Modifies* are variables which will receive updated data during task execution. *Requires* are read-only simulation variables. *Requires* may also specify halo data extents. Typically an application developer will request 1, 2, or 4 layers of halo cells for simulation variables, though Uintah supports much larger extents that are global or nearly global [15]. In this Figure 3.4, a *Requires* variable with the name `gridVariable` will have one layer of halo cells. Additional task parameters can be set, such as specifying that the task will use a GPU device.

3.2.5.2 Task Creation and Execution

After a task is defined, Uintah uses that definition as a template to create many task objects. Typically, Uintah creates one task object per patch assigned to a compute node. So if a compute node is assigned 16 patches, Uintah will create 16 task objects, one for each patch. The runtime processes all dependencies of these tasks to determine the proper sequencing of tasks. The Uintah runtime will automatically scatter and gather all halo data on other compute nodes and prepare simulation variables with halo data already gathered and coupled in it. Tasks execute when all halo dependencies are available and compute cores are available.

3.2.6 Data Stores

Uintah has data stores (also known as **data warehouses** within the Uintah nomenclature) for all simulation variables listed in all tasks. Application developers utilize simple data store API calls such as `get()` to retrieve needed simulation variables during task execution.

Uintah's data stores utilize a key/value pair system, making these data stores associative arrays. The key of a Uintah data store is typically a three-tuple of 1) the simulation variable's label name, 2) the Uintah patch integer ID associated with that simulation variable, and 3) the material media ID this simulation variable is computed against. The value of the data store is one of the five Uintah simulation variable types (Section 3.2.3). These five variable types are polymorphic objects sharing the same base class, thus more easily allowing the data stores to hold every simulation variable type.

Uintah's data stores do not hold the actual simulation variables, but rather merely a reference (in Uintah's case, a C++ pointer) where the actual data can be found in hardware memory. A data store entry can hold a variety of metadata associated with the simulation variable. Common metadata items are 1) its simulation variable type, 2) its memory footprint in bytes, 3) its low and high index in 3D Cartesian coordinates, and 4) padding and stride information.

3.2.7 Simulation Execution

To run a simulation consisting of tasks, the application developer modifies a Uintah driver file indicating simulation parameters. These include the full grid layout informa-

tion, the patch decomposition layout, the cell decomposition layout, AMR mesh levels, the number of timesteps to execute, and what simulation variables should be saved to file. Many additional parameters can be utilized for more advanced simulations.

The user then invokes a simple binary and passes in command line arguments indicating the number of CPU threads, whether a GPU should be used, and the number of MPI ranks desired. When MPI is selected, Uintah's runtime will assign its patches to various MPI ranks, with no two MPI ranks owning the same patch. From the application developer's perspective, the application will generate the same computation output, no matter how many threads or MPI ranks were utilized.

3.3 AMT Runtime Perspective

An overview of the AMT runtime is shown in Figure 3.5. Many of the components shown in this figure have been described in the prior Section 3.2. This section will further describe the AMT runtime features of task creation, task dependency analysis, task graph creation, automatic MPI messages, and task scheduler queues.

3.3.1 Task Creation and Dependency Analysis

Tasks start as task declarations performed by the application developer (Section 3.2.5.1). These task declarations are not yet actual task objects, but merely templates describing how to create a task object. A Uintah instance running in an MPI rank will create task objects and use those for dependency analysis and execution. This process is described in more detail below.

3.3.1.1 Task Graph Compilation

Uintah executing in each MPI rank is responsible for creating its task objects, determining its dependencies with other tasks executing on other Uintah instances in other MPI ranks, creating an explicit **task graph** data structure to represent these task dependencies, and creating its MPI messages to send and receive halo data. Each Uintah instance in an MPI rank is aware of the entire over decomposition layout into patches, and from there, self determines the patches assigned to it.

Next, Uintah will create task objects for tasks to be executed on those patches. For example, if each MPI rank is assigned 16 patches of the domain, then Uintah will first

create a set of tasks to execute associated with those 16 patches.

From here, Uintah begins creating its task graph, or DAG of task objects. Initial dependency relationships between task objects are formed from analyzing both the application developer's stated task execution order as well as the relationship between *Computes*, *Modifies*, and *Requires* among these tasks. Uintah also analyzes the halo dependencies and halo lengths of these newly created task objects. Uintah will identify all tasks objects requiring dependency interaction, both within the MPI rank and tasks in other MPI ranks. A Uintah instance will create additional task objects for all tasks in other MPI ranks that will interact with the Uintah tasks. These additional tasks are likewise placed into the task graph with correct dependencies connecting the tasks in the DAG. Even though these additional tasks objects will only be executed on other MPI ranks, they are still created as task objects to determine proper task ordering and dependency information.

3.3.1.2 Automatic MPI Message Generation

Once a task graph DAG is created, Uintah automatically creates all MPI messages, both for sending data to other MPI ranks and receiving data from MPI ranks. The computed task graph and its associated MPI messages are not recomputed in subsequent time steps if the sequence of tasks remains the same (as is usually the case). Each Uintah instance on different MPI ranks will create and store different task graphs.

This entire task graph compilation and MPI message generation occur without any communication with other Uintah instances running on other MPI ranks. Each Uintah instance computes the same patch distribution for all MPI ranks. Likewise, each Uintah instance computes all MPI messages it must send out, and all MPI messages it must receive.

Overall, this process is crucial to Uintah's philosophy of separation the application developer from AMT runtime concerns. Uintah application developers are never required to manually execute tasks, send data between tasks, or process halo data. This automated task process likewise must be preserved for GPU tasks and is covered further in the upcoming chapters.

Uintah's task graph features and logic is more expansive than what is covered here. For example, Uintah can manage multiple task graphs, support task dependencies across

different AMR mesh grids coarse mesh grids, and is optimized for large task graphs for global or nearly global task dependencies [15]. Humphrey [31] covers these topics in detail.

3.3.2 Task Schedulers

A **task scheduler** is responsible for the preparation and execution of Uintah tasks. Within the Uintah AMT runtime, a task scheduler consists of queues of task objects. Each queue represents a given lifetime state the task will proceed through, and a task advancing to a subsequent queue indicates that the task has advanced to its next lifetime state. For example, a task which requires its simulation variables to obtain halo data from other MPI ranks begins in a queue where the task must wait until all needed MPI messages have arrived. From there, a task can move into a separate queue awaiting task execution. Uintah has many schedulers available, and schedulers differ in the number of states and pools a task may proceed through in its lifetime. As will be shown in Chapter 5, this work utilized a set of five queues for GPU tasks and a set of four queues for CPU tasks.

A **task scheduler thread** is a CPU thread which processes tasks in any task queue. These threads are created upon Uintah initialization, typically one per hardware core or per hyperthread, and no subsequent threads are created afterward. The same thread that processes a CPU task will also start executing a task object. This work focuses on the *Unified Scheduler*, which allows multiple task scheduler threads within a single MPI rank. The Unified Scheduler is essential for both 1) load balancing work within a compute node [9], and 2) minimizing task graph size and task graph compilation times by reducing the amount of MPI ranks needed in a simulation [15]. This work likewise modifies the Unified Scheduler further for GPU task preparation and execution.

3.4 Uintah Summary

The features described here define Uintah's separation of concerns. Application developers need only be concerned with a few features, while the runtime automatically manages the data stores, task scheduling, and automated halo management both within a compute node and across many compute nodes. The next step is to consider how to extend these same separation of concerns to GPUs, by building upon early work [3, 9, 10].

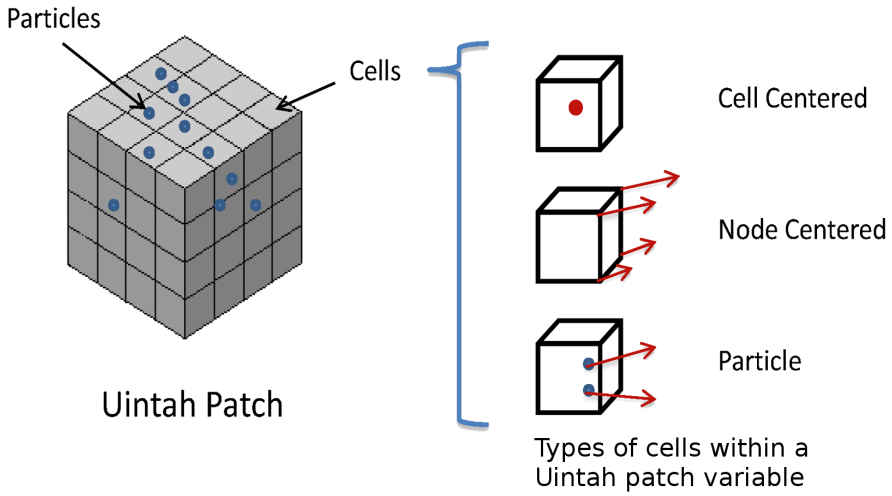

Figure 3.1: A Uintah patch demonstrating types of simulation variables. A grid variable comes in different flavors, such as cell centered and node centered. Particles can also reside within cells [1].

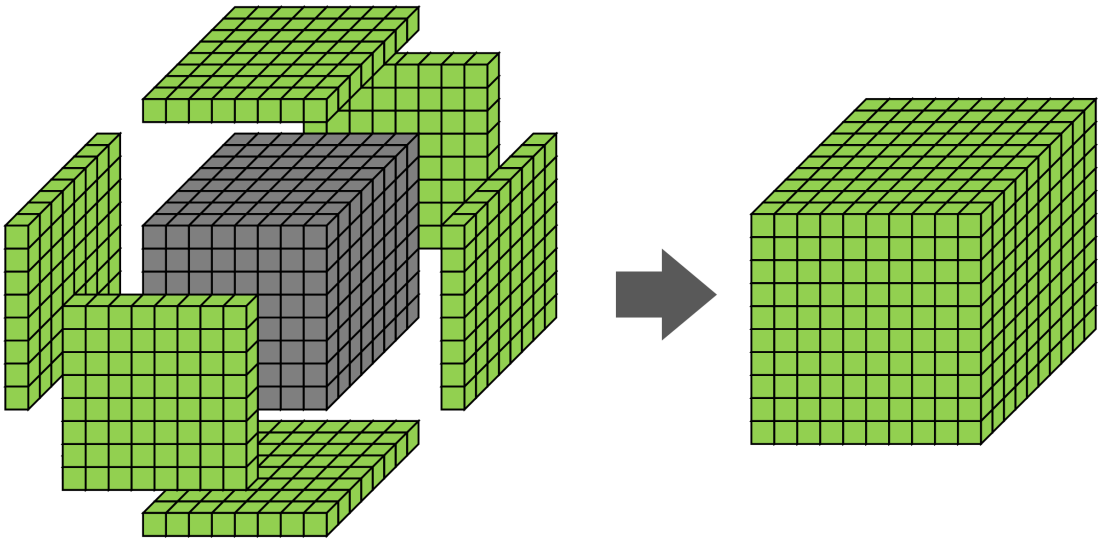

Figure 3.2: Halo cells (green) are copied from adjacent neighbor patches and coupled into this patch (gray). The resulting collection on the right is treated by Uintah as a single simulation variable, and not as a collection of many individual parts. Not shown on the left-hand side are halo data elements gathered in for edges and corners.

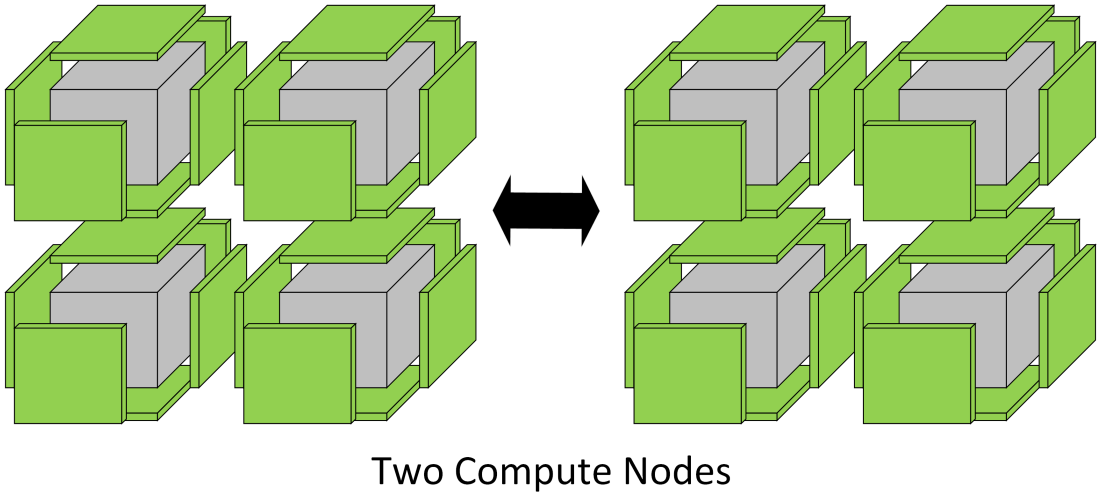

Figure 3.3: A common halo transfer pattern found in Uintah target problems. Each patch of cells (gray cube) must transfer its halo data (green slabs) to adjacent patch neighbors. Each compute node in this example owns four patches, and data is stored in host memory. Halo transfer can occur within a compute node via host memory copy calls, or to other compute nodes via MPI.

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

Figure 3.4: A Uintah task declaration which informs the runtime of a 27-point GPU stencil computation on a single Uintah grid variable.

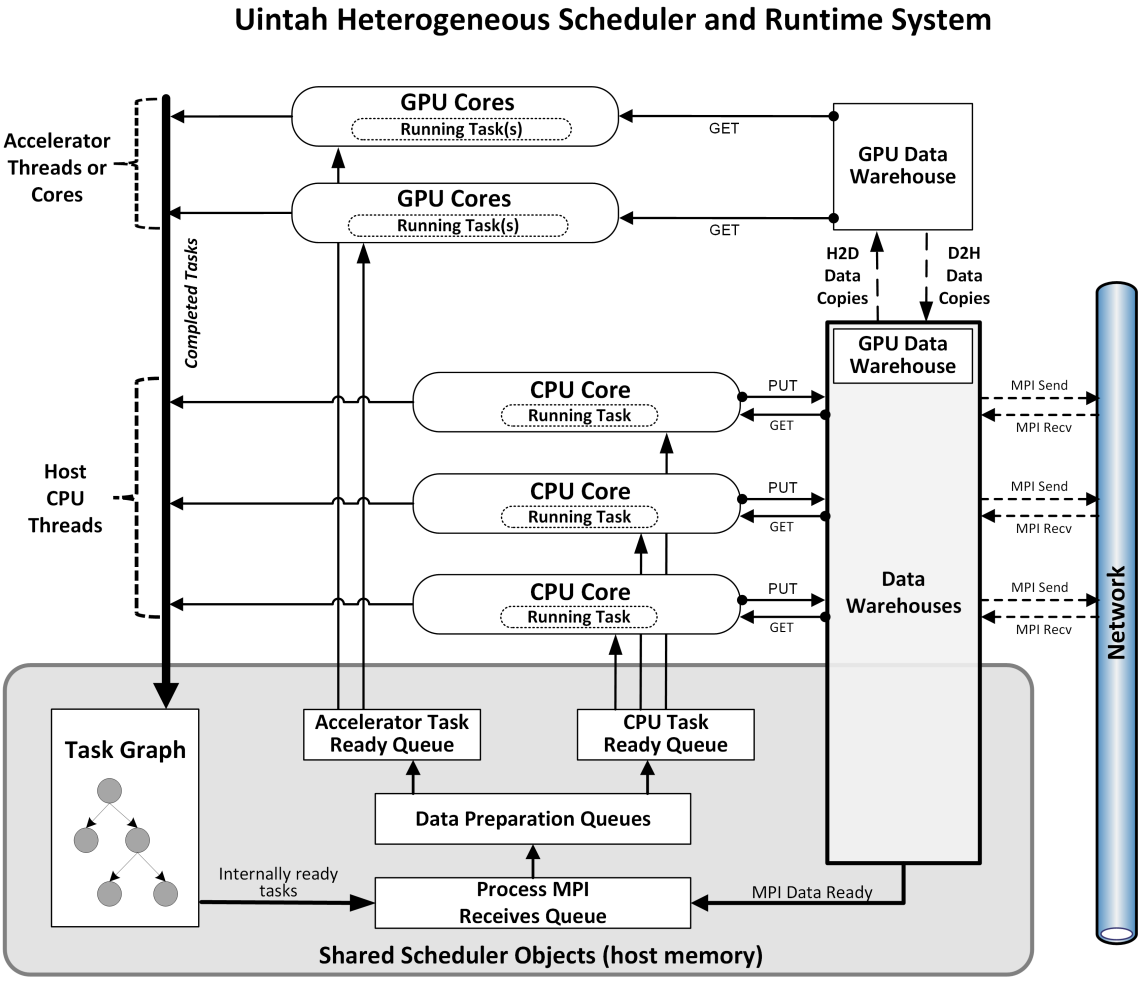

Figure 3.5: Uintah heterogeneous nodal runtime system and task scheduler. [2]

CHAPTER 4

A HOST AND GPU DATA STORE ENABLING CONCURRENCY, ASYNCHRONY, AND DATA SHARING

This chapter describes a GPU data store capable of supporting full asynchrony of numerous CPU and GPU tasks executing simultaneously in parallel. This GPU data store must help Uintah **prepare** simulation variables, and also help **scatter** and **gather** halo data. variables. Preparing simulation variables requires allocation, copying data to and from memory spaces, and coupling halo data. Halo scattering and gathering must now occur in multiple memory spaces in addition to multiple compute nodes. This chapter explains how the GPU data store assists task schedulers in avoiding simulation variable race conditions, and provides new mechanisms for simulation variables with large halos. This chapter is closely linked to Chapter 5 which covers task schedulers. In short, Chapter 4 describes the data structures to facilitate concurrent simulation variable storage, and Chapter 5 describes how multiple simultaneously executing task scheduler threads use these data structures to prepare and execute multiple tasks in parallel.

Designing a data store for an accelerator device memory space, such as GPU memory, faces numerous challenges not experienced with a host data store. Before this research, Uintah utilized a host memory data store called the **OnDemand Data Warehouse** [9]. Additionally, a simple GPU data store called the **GPU Data Warehouse** was also implemented by Humphrey et al. [3, 10]. This GPU data store was sufficient for its target problem while also demonstrating strong scaling up to 16K nodes on the DOE Titan supercomputer [3].

This initial GPU Data Warehouse required a GPU task scheduler limited to only executing one GPU task at a time. Further, all GPU simulation variables and associated data were prepared in host memory and then later copied into GPU memory. The GPU Data Warehouse could not operate asynchronously with multiple tasks sharing simulation

variables. As a result, short-lived and finer-grained GPU tasks could not be distributed throughout the GPUs cores [2].

The initial GPU Data Warehouse and its associated task scheduler laid the foundation this work as described in this and the next chapter. In particular, the prior work introduced 1) pre-allocating simulation variables prior to task execution, 2) moving the halo gathering phase prior to task execution instead of during task execution, 3) using a modified scheduler to automatically perform all host-to-GPU and GPU-to-host simulation variable data copies, and 4) a mechanism to manage simulation variables with global halos [3].

This chapter first explains Uintah's data store design principles in Section 4.1. This is followed by a review of the existing host memory data store (Section 4.2), its internal structure (Section 4.2.1), existing concurrency mechanisms (Section 4.2.2), halo gathering (Section 4.2.3), and allocation/dealloaction approach (Section 4.2.4). Section 4.2.5 describes why this host data store is an insufficient model for a GPU data store. Section 4.3 reviews the initial GPU Data Warehouse.

This work significantly expands Uintah's GPU data store into one that is production ready and capable of asynchronously handling concurrency issues without task restrictions. This chapter also presents work from prior publications [2, 4, 15], and covers solutions to five major data store challenges that were not solved in the initial GPU Data Warehouse: 1) Halo transfers should occur within GPU memory where reasonable (Section 4.4.2). 2) Halo data found in GPU memory and required in other memory spaces should be efficiently copied out of GPU memory (Section 4.4.3). 3) Concurrency mechanisms must allow tasks to simultaneously utilize and share the same simulation variables (Section 4.4.4). 4) Concurrency mechanisms must allow halo gathering without race conditions (Section 4.4.5). 5) Proper allocation and deallocation of simulation variables while avoiding common user errors (Section 5.3.5).

4.1 Uintah Data Store Design Principles

Both the host memory and GPU memory data stores shared basic fundamental approaches. They are at their core a key/value associative array of simulation variables. The data stores have straightforward APIs to allow application developers to obtain a simulation variable given a key.

Application developers writing task code typically have access to two data stores, one for the current timestep (*New Data Warehouse*) and one for the previous timestep (*Old Data Warehouse*). Between timestep transitions, the *Old Data Warehouse* is removed, the *New Data Warehouse* becomes the *Old Data Warehouse*, and a new *New Data Warehouse* is created. The prior *Old Data Warehouse* can deallocate any remaining simulation variables at this stage. The *New Data Warehouse* that turned into this timestep's *Old Data Warehouse* will have all computed variables from the prior timestep. The new *New Data Warehouse* will be empty.

4.1.1 Halo Design

Uintah considers halo data to be a fundamental property of a single simulation variable itself. This model is in contrast to many other runtimes which offload halo management to the application developer and likewise treat each set of halo buffers as separate data store entries. Uintah's model is crucial for large scale problems where simulation variables require a mixture of local halo extents and halo extents covering most or all of the simulation domain [15].

For example, suppose a simulation variable is needed to perform an upcoming 27-point stencil operation. Uintah will gather that variable's halo data into the one data entry for the simulation variable, rather than create 26 additional halo data store entries in the data store. This process scales, so that if halo data is needed from many neighboring patches, the result is again gathered into one data store entry without requiring any additional work from the application developer.

Uintah's automated halo management comes in two distinct phases, halo scattering, and halo gathering. The scattering phase is handled by task scheduler logic, and Uintah's existing data stores are largely uninvolved [9, 31]. The gathering phase is more sophisticated and handled by Uintah's data stores.

CPU-only simulations transfer halo data as shown in Figure 4.1 (see also Figure 3.3). Halo gathering obtains its data from two sources, 1) the halo data already within that MPI rank's host memory, or 2) the halo data in another MPI rank. When Uintah in a receiving MPI rank has obtained halo data from the second source (another MPI rank), the Uintah instance in the receiving MPI rank treats that data as if it came from the first source (host memory).

The initial GPU Data Warehouse [3, 10] borrowed on the prior design and is shown in Figure 4.2. For halo gathering, all variables were created and managed in host memory, then copied host-to-GPU. For halo scattering, data computed in GPU memory is copied GPU-to-host.

4.2 OnDemand Data Warehouse

Uintah's host memory data store, the OnDemand Data Warehouse [83] was given the term *on demand* due to its concept of allocating new variables and gathering halo data on-the-fly. New variables are allocated during task execution in task code. Halo data is also gathered together at the moment it is requested during task execution, rather than in some earlier Uintah runtime phase.

To better understand this process, the sample code below demonstrates an example of OnDemand Data Warehouse code usage:

```
constNCVariable<double> phi;
old_dw->get(phi, phi_label, matl, patch, Ghost::AroundNodes, 1);
NCVariable<double> newphi;
new_dw->allocateAndPut(newphi, phi_label, matl, patch);
newphi.copyPatch(phi, newphi.getLow(), newphi.getHigh());
double residualVar =0;

// Task code logic for all cells in the patch

new_dw->put(sum_vartype(residualVar), residual_label);
```

This code interacts with three OnDemand Data Warehouse simulation variables. The first simulation variable, `phi`, is obtained from the prior timestep via the old data store (`old_dw`). The second simulation variable, `newphi`, is allocated for the current time step and stored in the new data store (`new_dw`). This second simulation variable also requests one layer of halo cells (also known as ghost cells) in addition to its data cells. A residual value, `residualVar`, is created locally and its values accumulate into the appropriate data store entry at the end of the task.

4.2.1 OnDemand Data Warehouse Structure

The OnDemand Data Warehouse uses a key-value pair approach and stores one entry for each simulation variable. The value of the data structure is an object of one of the five Uintah variable types described in Section 3.2.3. All five Uintah variable types are polymorphic and derived from a base variable class, thus allowing all Uintah simulation

variable types to be stored in the associative array.

4.2.2 OnDemand Data Warehouse Concurrency

The OnDemand Data Warehouse is designed to work concurrently in a multithreaded environment. The Uintah AMT runtime has the capability of supplying many CPU threads per MPI rank, rather than a model where every thread is given its own MPI rank and subsequently also avoiding data store concurrency issues entirely. All threads in an MPI rank can perform various runtime actions independently. For example, one thread can perform an MPI send action while another executes a task whose simulation variables are available. Thus, this data store must account for two or more tasks simultaneously interacting with its variable entries.

Initially, the OnDemand Data Warehouse had a lock-free design by preallocating and preloading the array of simulation variables at the start of a timestep. This approach ensured no array resizing was needed later during the timestep. Further, almost all data store entry actions are reads which can occur concurrently without locks. The only writes come when a task is computing a new simulation variable and places a new simulation variable in its predetermined array index slot. Uintah helps ensure that either only one write can occur with no reads to that array index through proper task sequencing gleaned from task graph analysis. Once a simulation variable is written to an array index, no further writes occur at that index, and only reads may occur. Thus, a lock-free pattern is realized for simulation variable read and writes. The halo gathering process, however, is not lock free, and is instead managed by three different sets of locks.

4.2.3 Halo Gathering

A brief description of how the OnDemand Data Warehouse receives halo data from other MPI ranks is given here for later comparison to this work's more expansive approach (Section 4.4.4.1). When an MPI rank prepares to receive halo data via MPI from another MPI rank, it creates a new data store entry with a buffer large enough to hold what it must receive from another MPI rank. The OnDemand Data Warehouse has two boolean values to track the simulation variable's status: 1) a *foreign* boolean, which marks that this variable is owned by another MPI rank, and 2) a *valid* boolean, which marks that a foreign variable has been received (called *valid*). The *foreign* boolean is always initialized to true for data

store entries storing halo data. The *valid* boolean is initialized false and later marked true when the scheduler verifies the MPI halo data has arrived.

When halo data has arrived, and both booleans are set to true, the halo data is still kept separate from other data store entries, having not yet been gathered and merged into a simulation variable. The merging process occurs during task execution. During task execution, an application developer may request a grid variable with certain halo extents. At that moment, the OnDemand Data Warehouse creates a new array, large enough to later hold the grid variable and its gathered halo cells. Then the OnDemand Data Warehouse copies cell-by-cell all values from the existing grid variable and halo data as found from all neighboring patches (see Fig 3.2). This new merged grid variable then replaces the original simulation variable in the data store. Its 3D coordinate low point (in the x-, y-, z- corner) and high point (in the x+, y+, z+ corner) are also updated in the data store. Ultimately, this process can be expensive, and this overhead has been tested and measured at using between 5 to 10% of task wall clock execution time for short-lived tasks like the Poisson 3D simulation.

4.2.3.1 Concurrency Flaws in This Halo Design

The prior halo gathering model was designed piecemeal to support existing use cases but contains potential race conditions not observed in existing use cases. These flaws help guide new data store design principles for this work. All race conditions occur when the following criteria are met: 1) two or more Uintah tasks may utilize the same simulation variable, 2) these tasks execute simultaneously, 3) at least one of these tasks invokes the *on demand* halo gathering process during task execution. Solutions are covered later in this chapter and in the next chapter in Sections 4.4.4, 4.4.4.1, 4.4.4.2, 4.4.5, and 5.2.1.

4.2.4 Simulation Variable Lifetime

The application developer must manually allocate OnDemand Data Warehouse simulation variables. Such variables do not exist until the task executes and a call is made to the data store's `allocateAndPut()` API call. Only variables listed as *Computes* may be manually allocated in this fashion.

The OnDemand Data Warehouse utilizes a **scrub counter**. Before a timestep execution, Uintah creates a task graph indicating all upcoming tasks and the simulation variables

these tasks utilize. Uintah uses this task graph to count all usage instances of all simulation variables. For example, suppose variable `phi` on patch ID 84 using material ID 0 is declared for use in four upcoming Uintah tasks, then Uintah assigns a scrub count of 4 for this particular data store entry. After each one of these four tasks executes, Uintah decrements the scrub counter. When the scrub counter hits 0, the simulation variable is deallocated.

4.2.5 OnDemand Data Warehouse Is an Insufficient GPU Data Store Model

The OnDemand Data Warehouse has fulfilled its role as a concurrent host memory data store for all Uintah target problems. The data store model handles common stencil computation halo problems requiring one or two additional halo cell layers and for problems requiring halos that extend through most or all of the computational domain. Application developers have readily adopted this data store's API with little to no confusion.

Unfortunately, the OnDemand Data Warehouse cannot serve as a perfect model for GPU data store in a heterogeneous environment because of the five following reasons: 1) Simulation variable allocations occur within tasks themselves, whereas GPUs cannot reliably allocate variables within CUDA code. 2) Halo gathers also occur within the task themselves creating larger versions of the simulation variable, which suffers from possible race conditions and again relies on memory allocations on-the-fly. 3) Application developers frequently neglect to inform Uintah of task simulation variables that will be used compared to what is used in task code. 4) Deallocation of memory space on GPUs creates a synchronization barrier which conflicts with the goals of an asynchronous runtime. 5) The OnDemand Data Warehouse duplicates halo data when large halos are utilized, which may not fit within a GPU's limited memory.

4.3 The Initial GPU Data Warehouse

The first version [3, 9, 10] of the GPU Data Warehouse was designed for the RMCRT implementation target problem (Section 1.6.2). The overarching concept required the developer to implement all task code as CUDA code, including obtaining simulation variables within CUDA code. At the time, writing native CUDA code offered the most feasible means of GPU task execution.

The OnDemand data store object was not used, and instead, a new GPU data store

object was created from scratch. The object contained an array of structs to hold all simulation variables and their associated metadata. Before a GPU task executed, the GPU Data Warehouse object in host memory was loaded with appropriate metadata, and then copied into GPU memory.

For halo scattering, simulation variables were copied to host memory before any needed halo data was used elsewhere. All halo logic occurred in host memory for four reasons: 1) it required minimal additional code to debug and maintain, 2) it allowed for quick development to support GPU-enabled tasks with large halos as found in the RMCRT target problem, 3) the host memory halo logic had been proven to scale to 256K cores, and 4) it was more efficient to gather halo data in host memory and then send the simulation variable to GPU memory in a single host-to-GPU copy, rather than using many GPU-to-GPU copies, incurring fewer API latency costs. While functional and simple, this approach heavily utilized the PCIe bus that connects the GPU with the host CPU.

Prior work [9, 10] for RMCRT simulations required replication of radiative properties among compute nodes to facilitate local ray tracing. Replication was accomplished by specifying very large halos to surround radiative data variables. These RMCRT target problems used global halos both for problems where AMR was not utilized and for AMR using coarse levels. The main challenge was that each grid variable required halo data for the entire domain, and thus grid variables were frequently duplicating halo data and could not fit within limited GPU memory. To resolve this problem, the initial GPU Data Warehouse implemented an additional data store called a **level database** to store one single instance of a simulation variable containing all data for the entire AMR level, rather than many individual grid variables which duplicated global halo data.

The prior GPU Data Warehouse and its accompanying task scheduler had deficiencies for this work's target problems. GPU tasks could only be sequentially executed, and so larger patch sizes were chosen to fill all GPU streaming multiprocessors. For short-lived GPU tasks with simulation variables requiring small halo regions, performing halo transfers in host memory was prohibitively expensive compared to similar CPU tasks [2]. Another problem was limited patch over-decomposition options, as problems may require large halos whose extents are not fully global halos [15].

One of this work's target problems, Wasatch tasks (Section 1.6.3), greatly exposed these

GPU Data Warehouse deficiencies. Wasatch utilizes many short-lived GPU tasks with small halo regions around simulation variables. Frequent data copies across the PCIe bus between host memory and GPU memory often took more time than CUDA kernels executed. Further, Wasatch preferred executing many smaller tasks which were not sufficiently occupying all cores of the GPU.

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

4.4 This Work's GPU Data Store

This section marks the start of new data store work [2, 4]. The use cases described in Chapter 1 require a concurrent GPU data store free from the deficiencies of the prior data stores. As described in this chapter's introduction, the five new features are: 1) Halo transfers occur within GPU memory where reasonable (Section 4.4.2). 2) Halo data required in GPU memory and required in host memory or other MPI ranks are efficiently copied out of GPU memory (Section 4.4.3). 3) Concurrency mechanisms allow tasks to simultaneously utilize and share the same simulation variables (Section 4.4.4). 4) Concurrency mechanisms allow halo gathering without race conditions (Section 4.4.5). 5) Simulation variables are properly allocated and deallocated to avoid common application developer errors (Section 5.3.5).

4.4.1 GPU Halo Copies to Various Memory Spaces

A new requirement of the GPU data store is keeping simulation variables persistent in GPU memory, and refraining from copying them in and out of host memory for halo

gathering and scattering. With data staying persistent in GPU memory, more halo data scenarios must be managed. Uintah must prepare grid variables for both CPU tasks and GPU tasks by obtaining halo data from grid variables in adjacent patches (adjacent grid variables) in whatever memory location they exist. These adjacent grid variables can exist in four memory locations, 1) host memory, 2) GPU memory, 3) another on-node GPU's memory, or 4) off-node.

With four possible source locations and four possible destination locations, there are 16 possible halo data copy scenarios. Because a task does not manage halo data for patches or nodes it is not assigned to, this number is reduced to 12 scenarios, as shown in Figure 4.3. This number could be reduced to only eight scenarios if Uintah employed only one MPI rank per GPU, rather than one MPI rank per node, as the four scenarios arrows involved in "Another GPU" as labeled in Figure 4.3 would be removed from consideration. NVLink [84] offers additional data bus connectivity. NVLink is a fast interconnect between GPU and host memory, as well as between multiple GPUs, and thus NVLink increases the number of halo data copy scenarios. This section works with the 12 scenarios in Figure 4.3, which is one MPI rank per node, multiple GPUs in a node. NVLink is new as of the Nvidia Volta GPU series, and its support in Uintah is left as future work.

Implementing specific code for each of the 12 scenarios is infeasible and impractical. Instead, the process can be broadly treated as four categories, specifically 1) halo copies within the same host memory space, 2) halo copies within the same GPU memory space (Section 4.4.2), 3) halo copies out of host memory and into GPU memory (Section 4.4.3), and 4) halo copies out of GPU memory and into host memory (also Section 4.4.3). The first category is already implemented in Uintah and works with data to and from other MPI ranks. The other categories are covered below.

4.4.2 Halo Copies Within the Same GPU Memory Space

Halo copies now occur entirely within the same memory space if the source and destination grid variable reside in the same space [2, 4]. For example, two adjacent grid variables in GPU memory can simply copy their halo data to each other. This process avoids a temporary buffer in between the scattering and gathering of halo data. The data is copied cell-by-cell because most grid variable halo copies among each other utilize data

layouts where the halo cells are not contiguously stored. Further, the cell-by-cell copies are performed within CUDA code using a specialized kernel specific for this purpose. GPU memory cell-by-cell copies are not processed through host API calls as the resulting overhead increased and bandwidth would be orders of magnitudes worse.

This CUDA kernel to perform cell-by-cell halo copies is also adaptable and reused for other halo copies described in Section 4.4.3. Further, Uintah can batch together copy logic for all halos for grid variables required for a task, resulting in invoking this CUDA kernel once for all halo scatters, and once more for all halo gathers. This process will be further described in the upcoming sections and the next chapter.

4.4.3 Halo Copies to Other Memory Spaces

If the halo data must be copied to another memory location, then contiguous arrays with packed data are employed as shown in Figure 4.4 [2, 4]. Packed buffers are used because these simulation variables contain non-contiguous halo data in memory. The halo copy CUDA kernel described in Section 4.4.2 is again used here to copy halo data from grid variables into these packed buffers, as well as out of packed buffers and into grid variables. Host-to-GPU and GPU-to-GPU transfers within an MPI rank are simplified and made more efficient, as the cost of many individual transfers between memory spaces is almost always far greater than the cost of creating a packed buffer and sending that buffer. For halo transfers using MPI, when packed buffers are copied GPU-to-host, this allows Uintah to use any MPI implementation, rather than specific MPI implementations that utilize CUDA's GPUDirect, Unified Memory, or CUDA-aware MPI. Further, packed buffers in GPU memory provide the potential of performance equal to or exceeding non-contiguous data structure solutions provided by CUDA-aware MPI implementations [85].

In the prior runtime system example given in Section 4.3 of a node containing a $4 \times 4 \times 4$ set of patches, a minimum of 14 ms was required simply to transfer the data GPU-to-host over the PCIe bus. This new approach has been implemented in Uintah's runtime system. Profiling this particular problem results in combined transfer and processing times of roughly 1 to 2 ms.

These halo cells can all be processed in batches in a single CUDA kernel, rather than one CUDA kernel per grid variable. For example, suppose a GPU task requires N grid

variables each needing to gather in halo cells, then all N can be processed together in the same CUDA kernel. In another example, suppose a Uintah task responsible for sending MPI halo data to other MPI ranks requires halo data from N grid variables currently in GPU memory. A single CUDA kernel can process all halo scatters from grid variables in GPU memory into packed buffers in GPU memory. From here, a modified Uintah scheduler (Chapter 5) copies the halo data in packed buffers into host memory.

4.4.3.1 Batching Example From Many Sources

Suppose a hypothetical 27-point stencil task requires that a single GPU data grid variable send its halo cell data to 26 neighboring grid variables, and then receive halo cell data from those same 26 neighbors. Of these 26 neighbors, suppose 11 are found within that GPU, six are found within another on-node GPU, and 9 are off-node. This grid variable will then be assigned a collection of 15 appropriately sized buffers for later packing (6 for the on-node GPU and 9 for off-node). A kernel will be called to perform 15 halo cell copies within that GPU and create 15 packed buffers. After the kernel completes, the runtime system identifies those six dependencies which belong to another GPU, and so 6 GPU peer-to-peer GPU-to-GPU copies are invoked. The runtime then identifies those nine dependencies that belong off-node, the packed buffers are copied into host memory, and MPI is used to send this data as necessary to other nodes. Once all data is sent out, it is the responsibility of the scheduler processing future tasks to gather these halo cells back into data grid variables.

When a future task needs to use this same grid variable with halo cells, the runtime's scheduler will recognize that it will need to gather together the halo cells from 26 neighbor patches. The scheduler will look in the node's Data Warehouses and find that halo cells for all 26 exist in various memory locations on that node. Halo data for that grid variable will be sent into that GPU memory space as needed. The scheduler will then invoke the CUDA kernel responsible for copying halo cell into the grid variable in GPU memory.

4.4.3.2 Batching Analysis

Batching all halo groups for later processing incurs an overhead cost by delaying halo copies that could otherwise be launched immediately. However, the cost of a launching a single kernel for one batch of halo groups may be preferable to the cost of invoking

many smaller kernels or copies. The cost of processing halo data for a Uintah task can be described as $t_{total} = a * t_a + n * t_c$, where a is the number of invoked API actions, t_a is the CUDA API latency to issue a kernel or copy call, n is the number of groups of halo data copies, and t_c is the time required to copy all halo items in a group. The upcoming measurements [2] varied the number of items in a halo group between 16^2 and 128^2 cells and used a machine with an Nvidia K20c GPU and an Intel Xeon E5-2620.

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

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

4.4.4 Concurrent Sharing of Simulation Variables

The prior GPU store model [10] did not function with concurrently executing GPU tasks because the Uintah scheduler performed memory writes to the GPU data store before each task executing. Previously, the GPU Data Warehouse only operated with sequential kernels. Three problems surfaced while attempting to execute GPU-enabled tasks concurrently, and these three problems were rooted in tasks sharing simulation variables in the GPU Data Warehouse.

The first problem was a read-after-write race condition, where the GPU Data Warehouse object in GPU memory was updating while another GPU task simultaneously used the object. The second problem occurred when an application developer requested simulation variables inside a GPU task, despite the task definition not explicitly indicating it would use the simulation variable. A race condition occasionally occurred when another GPU task also indicated it would use that same simulation variable. Depending on the task execution order, those simulations variables may often, but not always, be found in GPU memory. A third, loosely related, problem increased the memory footprint of the GPU Data Warehouse object to a few megabytes due to larger array buffers holding significantly more simulation variable metadata. For GPU tasks that computed within a few milliseconds, the time to copy the GPU Data Warehouse into GPU memory was unacceptably large [2].

This section describes a combination of two mechanisms to fix these problems. The first solution added an atomic bit set to each simulation variable to indicate its current status in host memory or GPU memory and allow Uintah's runtime threads to coordinate and prevent race conditions (Section 4.4.4.1). The second solution utilized a novel approach of smaller, thread local GPU data stores (Section 4.4.4.2).

4.4.4.1 Simulation Variable Atomic Bit Set

To give a GPU task scheduler the ability to prepare and distribute simulation variables in more than one memory space, Uintah adopted a status model similar to the MSI cache coherency protocol [86]. Each MSI state has analogs in Uintah. The *Modified* (M) state is covered by the task scheduler and task graph logic, which strictly controls that no two tasks will modify the same simulation variable at the same time (assuming the application

developer correctly declared which tasks use these variables). The *Shared* (S) state corresponds to Uintah simulation variables available for read-only use. The *Invalid* (I) state corresponds to simulation variables needing to inform the task scheduler that it is not currently available for use.

This work implemented two variations of the *Shared* (S) state. One state indicates if a simulation variable is available for use, and a second indicates whether all halo data is also available. This work also implemented two variations of the *Invalid* (I) state, one to indicate if the simulation variable is holding onto allocated space without any valid data (such as a *Computes* variable that hasn't yet been computed), and another to indicate if no allocated space exists at all. Uintah has no need at this time for any MESI or MOSI cache coherency approaches for *Exclusive* (E) states or *Owned* (O) states, though these may be useful if Uintah switches to a dynamicically generated task graph instead of its current task graph that is static for any one timestep but may change between timesteps.

Additionally, this work implemented a two-step policy lock-free implementation where the first step atomically indicates an action will occur or is occurring, the second step indicates the action has completed [2, 4]. This process is similar to other lock-free models where an atomic declaration of an intended action precedes the actual action [87].

A 32-bit atomic bit set was created to store multiple simulation variable states (see Figure 4.5). A bit set has six assigned bits for *allocating*, *allocated*, *copying in*, *valid*, *gathering halo data*, and *valid with halo data*. Each simulation variable entry in a data store is assigned a bit set. If a simulation has two memory spaces (such as host memory and GPU memory), then each simulation variable is assigned two of these bit sets.

Invalid states can be indicated by having the valid bit not set or the allocated bit not set. Shared states are indicated by having the *valid* or *valid with halo* data bits set. The action bits (*gathering halo data*, *copying in*, and *allocating*), are useful for task schedulers to ensure only one scheduler thread performs a given action.

With 6 of the 32 bits reserved for these statuses, the other 26 bits can be used to indicate copy out destinations. Suppose a future compute node has multiple hierarchies of host RAM and multiple GPUs. Each memory location can be assigned an ID number corresponding to these bits. Suppose one grid variable is being copied from GPU #1 to GPU #2 and high capacity host RAM, then the two bits representing those two destinations can be

set in that grid variable's bit set. Doing this allows for greater control in case a grid variable needs to be vacated from GPU memory to make room for others. Before deallocation, the grid variable's bit set can be checked to ensure it is not currently being used as part of a data transfer.

Additional work for Uintah's task scheduler utilizing these atomic bit sets are given in Section 5.3.2.

4.4.4.2 Task Data Warehouses

Task Data Warehouses were created to prevent read-after-write race conditions of simultaneously updating the GPU Data Warehouse in GPU memory while allowing GPU tasks to use the same data warehouse object [2, 4]. The driving concept of Task Data Warehouses is that each GPU task receives its own self contained GPU Data Warehouse objects in GPU memory, wholly independent and not used by other tasks, with only the simulation variable information that task needs for computation (see Figure 4.6). These small Task Data Warehouses in GPU memory serve as read-only snapshots of a subset of the GPU Data Warehouse. This design 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.

The creation and loading of the Task Data Warehouse is managed by the task scheduler described in Chapter 5 . In summary, the task scheduler first gathers together a collection of all simulation variables the task will need. Then it uses a C++ *placement new* technique to allocate a single buffer in memory containing a GPU Data Warehouse object exactly large enough to hold all simulation variable metadata for the task. The task metadata is then loaded into the Task Data Warehouse, and the task scheduler copies this Task Data Warehouse into GPU memory.

This Task Data Warehouse design has several desirable qualities. A GPU task kernel will have no knowledge or capability to access grid variables unrelated to its task (an issue that affects the OnDemand Data Warehouse). Task Data Warehouses eliminate race conditions related to tasks sharing a GPU Data Warehouse in GPU memory.

Task Data Warehouses occupy a smaller memory footprint than the prior GPU Data Warehouses. The reduction is accomplished by placing all data store metadata into a single

array and also by using the *placement new* approach for allocation. This process allows one fast host-to-GPU memory to copy the full Task Data Warehouse object into GPU memory.

4.4.5 Concurrent Halo Gathering

The atomic bit set and the Task Data Warehouse model together allow for a clean solution to ensure concurrency during a halo gathering phase. As described more in Section 5.3.2, a GPU task scheduler thread can atomically set a bit indicating a simulation variable's halo gathering phase will soon commence. The task scheduler thread that succeeded in setting that bit must initiate the halo gather. Other tasks using that simulation variable wait until the needed halo gathers have completed.

Uintah must have a specific mechanism to perform the halo copies. As previously described in Section 4.4.2 and Section 4.4.3, Uintah utilizes a CUDA kernel to aid in halo scattering and gathering, prior to GPU task execution. These CUDA kernels are covered in more detail here.

The Task Data Warehouse was modified to carry halo copying logic [2, 4]. Task Data Warehouse object entries support holding two types of information, the first holds simulation variables containing data address and layout information, and the second holds metadata logic describing how to perform halo copies. The metadata describes a grid variable's halo region to copy from, and another grid variable's halo region to copy to. The CUDA kernel responsible for halo copies in GPU memory then reads the Task Data Warehouse entries to identify its actions it must perform. Further, the kernel can use the same Task Data Warehouse to look up the capacity, dimensions, and data pointer of both the source and destination region. From here, the kernel can identify how it can perform the halo cell-by-cell copies in GPU memory.

At first glance, allowing a Task Data Warehouse entry to store either a simulation variable or halo copy metadata appears ill-fitting. The reason for this design is shrinking the data store's memory footprint. The original GPU Data Warehouse object's size was on the order of several megabytes [3] due to it internally having many fixed size arrays. This work reorganized the GPU Data Warehouse object into a few data members and *one* large array, where this one large array stores entries for completely different roles. C++ *placement new* allowed for a further reduction in object size by allocating only as many array entries

as needed. The result was a single data store object with a memory footprint of only a few kilobytes. This single object allowed for the Task Data Warehouse to quickly be copied into GPU memory.

Figure 4.7 shows the improvements of this approach, demonstrating how more Data Warehouse objects can be copied into GPU memory in less time. Each task copies two such data store objects, one for the prior timestep and a second for the current timestep. These smaller Task Data Warehouses drastically reduces the launch overhead, opening up Uintah to utilize GPU tasks that execute on the order of a few milliseconds.

Before Task Data Warehouses, GPU tasks could only synchronously execute as the GPU Data Warehouse could not update in GPU memory while another task was executing. Task Data Warehouses enabled parallel task execution, as they can copy into GPU memory without interfering with other GPU tasks, thus also enabling parallel execution of the GPU tasks [4]. Figure 4.8 demonstrates kernel overlapping while running long-lived RMCRT tasks. Such overlapping is critical to allowing application developers far greater flexibility in domain decomposition strategies.

4.5 Data Store Summary

The work in this chapter creates data stores to facilitate concurrent asynchronously executing GPU tasks without synchronization barriers. Small data stores are created for each individual GPU task, which 1) reduces host-to-GPU copy times, 2) avoids read after write race conditions, and 3) avoids problems of accessing simulation variables not listed in the task declaration phase. Halo scattering and gathering now can occur entirely in GPU memory space and avoid host memory. Simulation variables themselves are associated with an atomic bitset to track the status of data copies and halo gathers to help avoid race conditions caused by duplicated actions. The next chapter describes how a GPU task scheduler uses these data stores and atomic bitsets for heterogeneous task execution.

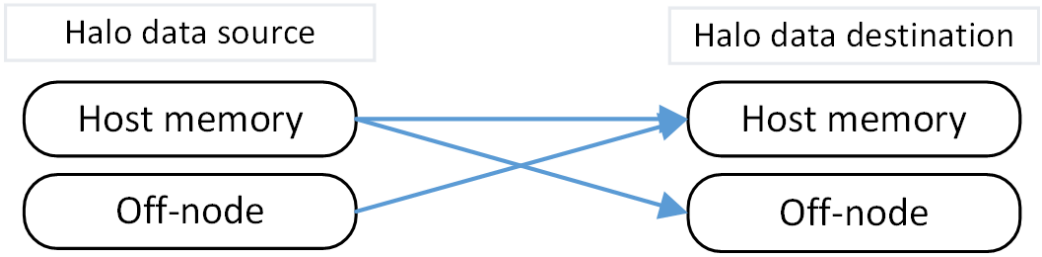

Figure 4.1: Uintah's initial runtime system to transfer to prepare only CPU tasks. Three scenarios are shown, one that keeps all halo data resident in memory, two others which transfer halo data to another compute node. (See also Figure 3.3).

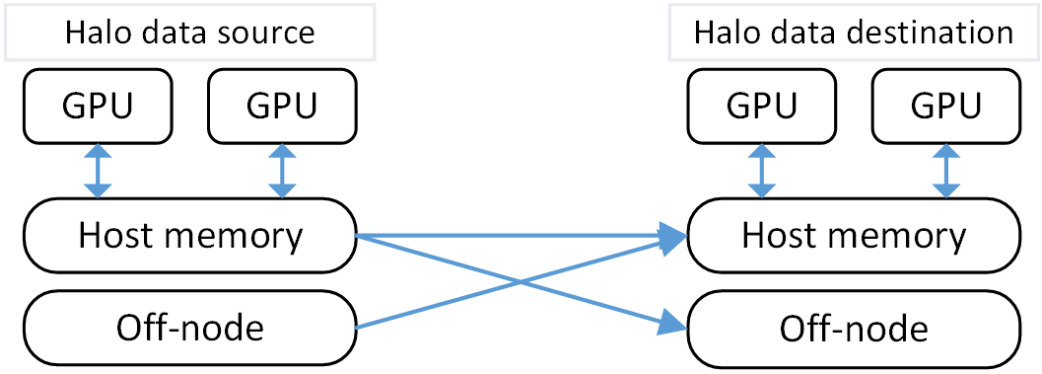

Figure 4.2: Uintah's modified initial runtime system to prepare CPU and GPU task grid variables with halo data from adjacent grid variables. The GPU halo scenarios are all managed in host memory, necessitating that GPU grid variables are copied in and out of GPU memory.

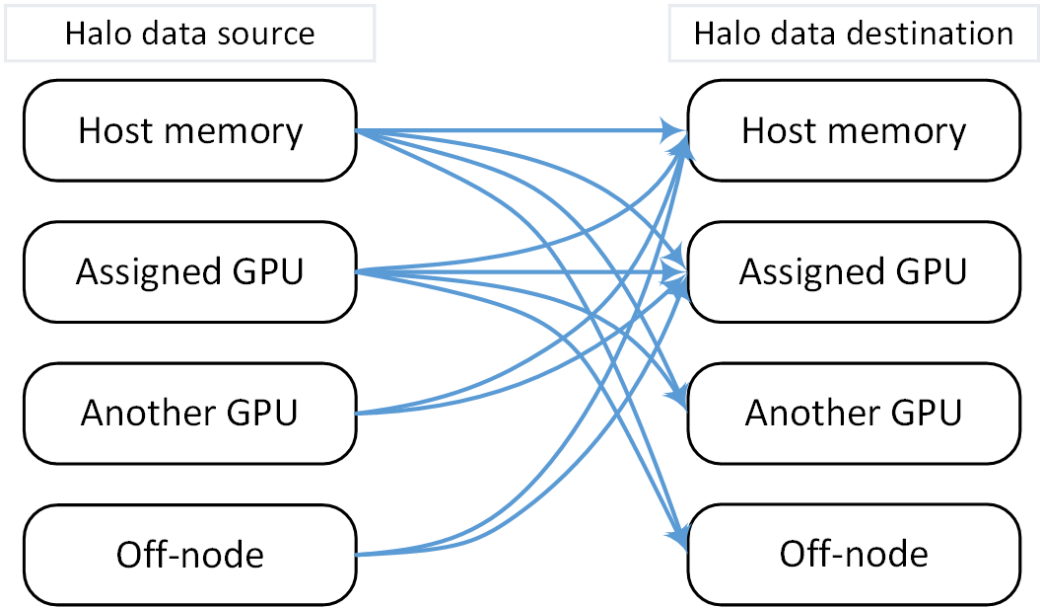

Figure 4.3: Uintah's current runtime system to prepare CPU and GPU task grid variables with halo data from adjacent grid variables.

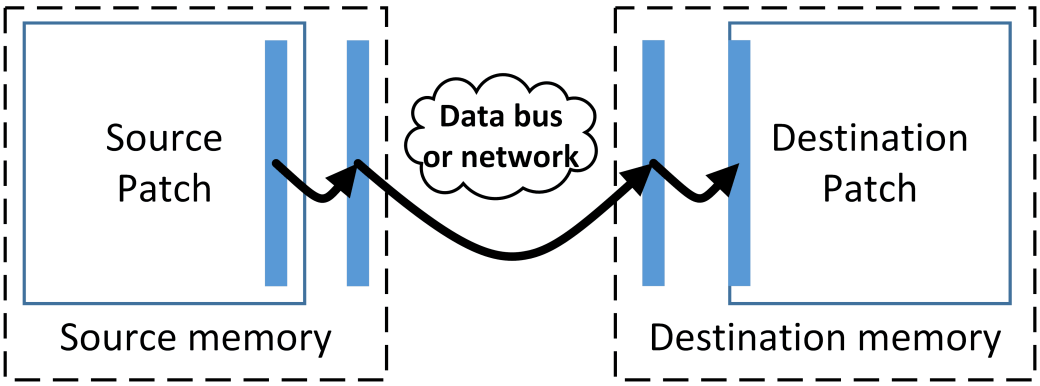

Figure 4.4: Halo data moving from one memory location to another are first copied into a contiguous staging array prior to being copied to that memory location. Later a task on the destination will process the staging array back into the grid variable.

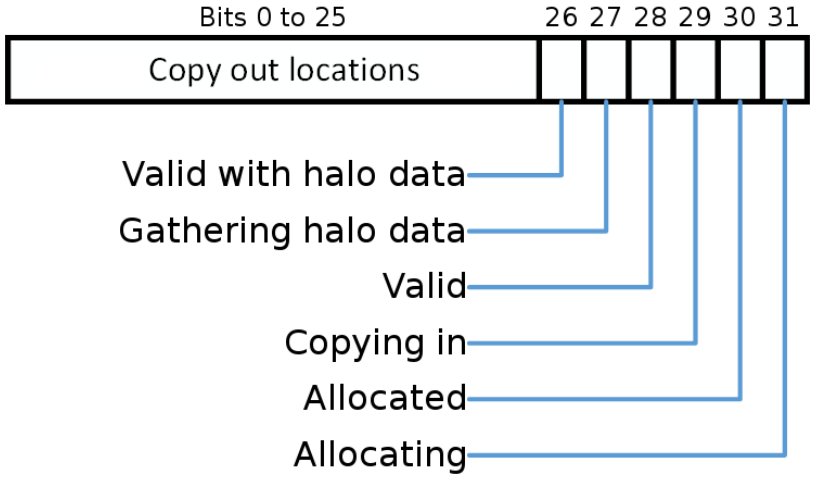

Figure 4.5: A bit set layout for the status of any simulation grid variable in a memory location. Every simulation grid variable in every memory space contains a bit set. Reads and writes to this bit set are handled through atomic operations.

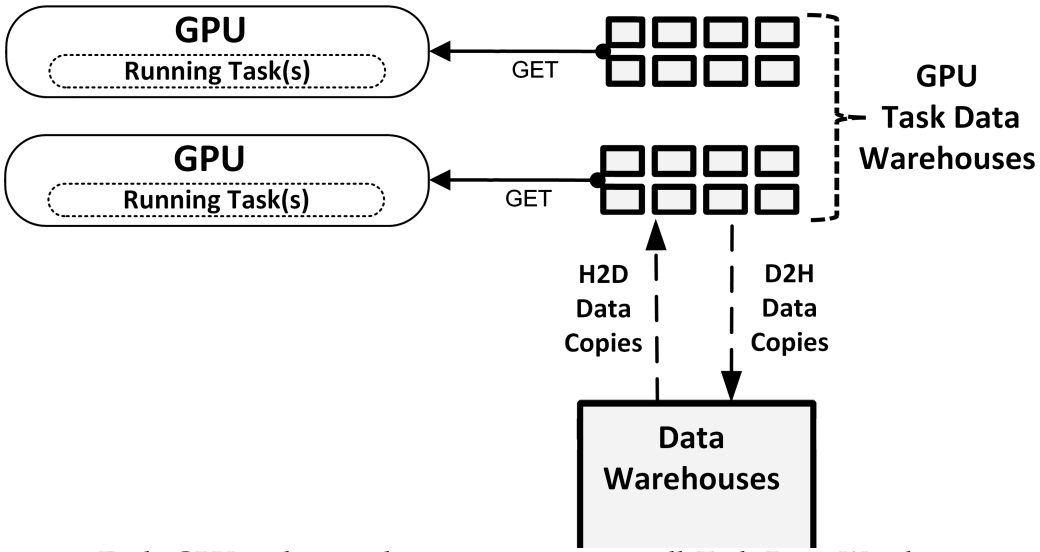

Figure 4.6: Each GPU task gets data into its own small Task Data Warehouses, rather than the old approach (Figure 3.5) of all GPU tasks sharing the same large GPU Data Warehouses.

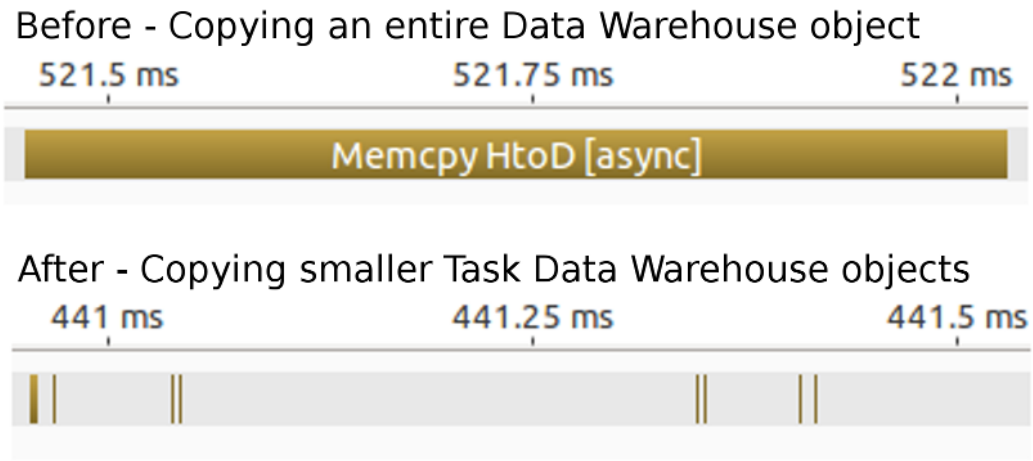

Figure 4.7: A profiled half millisecond range of an eight patch simulation showing Data Warehouse copies. Before, the initial runtime system had many large Data Warehouse copies (only one shown in this figure). After, the new runtime system's small Task Data Warehouses copy into GPU memory quicker, allowing GPU tasks to begin executing sooner (eight Data Warehouse copies are shown in this figure). Each task requires two such copies. Thus, launch latency was improved by over 1000 microseconds per task.

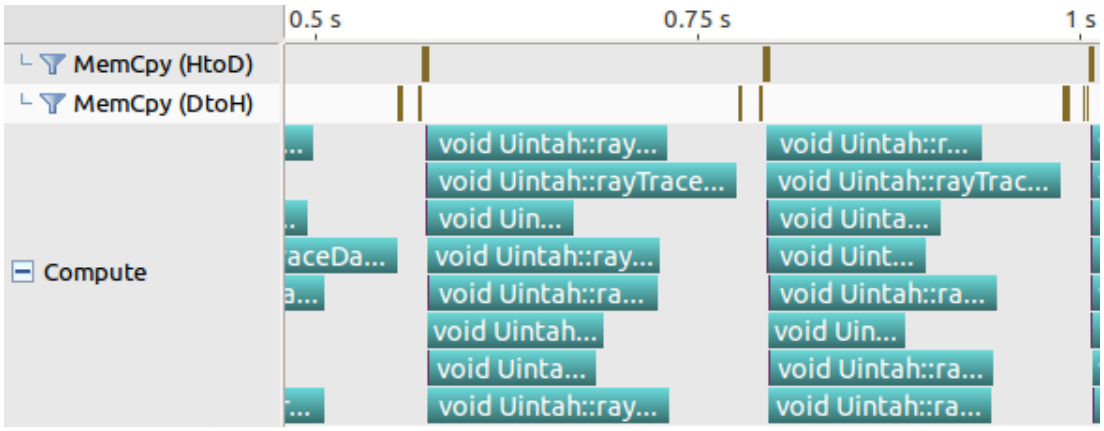

Figure 4.8: Data sharing among tasks and GPU Task Data Warehouses allows the Uintah runtime to asynchronously invokes sets of host-to-GPU copies followed by GPU task kernel executions. The brown lines in the MemCpy rows represents Task Data Warehouse copies, and each teal bar in the Compute row represents a single GPU RMCRT task. Computations were performed on an Nvidia K20c GPU.

CHAPTER 5

UINTAH HETEROGENEOUS/GPU TASK SCHEDULING AND EXECUTION

This chapter describes how the GPU task scheduler was modified to support full asynchrony of many CPU and GPU tasks simultaneously executing in parallel [2, 4, 15]. In particular, this chapter focuses on challenges related to multiple task scheduler threads preparing tasks, copying simulation data between the host and GPU memory spaces, coordination among scheduler threads, and execution of both CPU and GPU tasks. This chapter utilizes data stores with built-in concurrency mechanisms as outlined in Chapter 4.

The task scheduler in this chapter is the latest in a series of task scheduler designs. The first scheduler utilizes one thread per MPI rank and launched only CPU tasks [9]. Concurrency issues were entirely avoided as no two tasks could be prepared or executed simultaneously. The next scheduler utilized multiple Pthreads within an MPI rank [9] to prepare and launch CPU tasks. All threads performed work independently, and scheduler threads utilized work pools to prepare and launch CPU tasks, send data via MPI to other MPI ranks, or process data received from other MPI ranks (see Figure 3.5). Major benefits to this Pthread-based scheduler model were better load balancing within a physical compute node and reducing dependency analysis across MPI ranks.

The next task scheduler created by Humphrey et al. [3, 10] added GPU task scheduling alongside existing CPU task execution capabilities. This GPU task scheduler took a basic approach for preparing and executing GPU tasks. A task scheduler thread would prepare simulation variables in host memory, gather halo data if necessary, copy these simulation variables into GPU memory, execute a GPU enabled task, then copy all computed variables back into host memory. The application developer was required to manually configure and launch a CUDA kernel. Support was started for asynchronous CUDA actions, but at the time, synchronization still occurred between task copies and task execution.

The target problem driving the first GPU task scheduler was Uintah's reverse Monte Carlo Ray Tracing (RMCRT) component (Section 1.6.2). These RMCRT tasks required relatively straightforward parallelism amenable for GPUs and could be easily partitioned to utilize all processors in the GPU. Further, the tasks themselves were long-lived (on the order of seconds to minutes), which resulted in a small fraction of total execution time spent in task scheduler overhead.

Multiple concurrency issues existed when trying to adopt this first GPU task scheduler in a fully asynchronous and parallel heterogeneous environment for all target problems listed in Chapter 1. As a result, an overhaul of the GPU task scheduler was required and is the basis for the remaining sections in this chapter.

This chapter overviews the initial task scheduler and associated automated halo gathering logic in Section 5.1. Section 5.2 covers concurrency problems associated with the first GPU scheduler. The chapter next gives this work's six modifications to the GPU scheduler: 1) Asynchronous GPU actions used throughout, including simulation variable copies and task invocation, and all synchronization barriers are avoided (Section 5.3.1). 2) Multiple scheduler thread coordination to share host-to-GPU and GPU-to-host simulation variable copies and halo gathers (Section 5.3.2). 3) New CPU and GPU task scheduler queues for asynchrony and sharing of task scheduler work (Section 5.3.3). 4) An optional approach using contiguous buffers to reduce API call latencies (Section 5.3.4). 5) New mechanisms to avoid concurrency issues with allocation and deallocation of simulation variables (Section 5.3.5). 6) A new mechanism for sharing halo data for large halo extents shared among multiple simulation variables (Section 5.3.6).

5.1 Initial GPU Task Scheduler

The initial task scheduler design [3, 10] executed GPU tasks through four main steps. 1) Prepare the GPU task by synchronously copying every task simulation variable from host memory to GPU memory. 2) Synchronously copy the entire GPU Data Warehouse Object managed in host memory into GPU memory. 3) Synchronously execute the GPU kernel found within the task (this meant only one GPU kernel could execute at a time in the GPU). 4) Upon task completion, synchronously copy all computed simulation variables back to the host memory space, then deallocated all allocated GPU space. All existing CPU

tasks and queues remained unchanged.

This task scheduler utilized multiple scheduler threads within an MPI rank. Each thread performed its task work completely independent of others. In other words, one scheduler thread preparing one task could not coordinate with another scheduler thread preparing another task, even if they were preparing the same simulation variables, leading to race conditions. Other scheduler threads performing GPU actions would be blocked waiting on one CUDA action to complete.

Tasks were executed by first invoking the task's CPU callback function. The task scheduler itself had no knowledge how to partition the upcoming CUDA kernel to maximize its occupancy throughout the GPU, or whether the task launched any CUDA kernels at all. The application developer was responsible for configuring CUDA's threading and blocking parameters and manually invoking kernels.

As Uintah commonly utilized tasks with patches of 16^3 cells or even 32^3 cells, many of Uintah's existing problems simply could not be ported to this model. For problems like a simple pointwise or stencil computation on 16^3 cells, even assigning one GPU thread to every cell would leave many GPU processors starved for work. For other problems like the RMCRT computation, each cell had significant ray tracing work per cell, and could better utilize a GPU.

5.1.1 Initial Halo Gathering and Data Copies Into GPU Memory

All halo data for all simulation variables were first processed in host memory, then copied into GPU memory, as shown in Figure 4.2. The GPU task scheduler relied on the OnDemand Data Warehouse's ability to perform the halo gathers before task execution. The initial halo gathering model was designed such that all CPU task scheduler logic could remain unchanged.

Recall that Uintah simulation variables are defined in one of three states: *Computes* (for new simulation variables), the second state is *Modifies* (for modifying a prior Computes in a prior task), and *Requires* (for read-only simulation variables that will never be modified again). When a task scheduler thread obtained a task to execute, the scheduler thread would first loop through all simulation variable requirements. For all *Computes* variables, the scheduler thread would allocate appropriate space on the GPU. The GPU Data

Warehouse object in host memory would be updated with a new entry for that *Computes* variable. For all *Requires* variables, the simulation variable object is obtained through an OnDemand Data Warehouse `get()` API call, which may also gather and couple in the halo data into the simulation variable. A similarly sized buffer would be allocated in GPU memory, and then a host-to-GPU data copy is invoked to copy the simulation variable into GPU memory. *Modifies* were not implemented, but could be like the *Requires* model. Once all simulation variables are in GPU memory, the task scheduler copes the GPU Data Warehouse object into GPU memory.

5.1.2 Initial Global Halo Management

The RMCRT target problem exposed some of the challenges with the prior halo model, as the memory footprint required was too large for GPU memory. Every MPI rank running RMCRT tasks required access to all radiation data on the entire computational domain for its ray tracing. Uintah has two mesh grid options for RMCRT: 1) using global data on a single mesh level or 2) utilizing AMR where the coarse mesh level covers the entire domain while the fine mesh level would have localized halos.

The memory footprint issue occurred as each Uintah instance assigned each MPI rank a unique set of patches, and grid variables on each patch required a global halo. For example, suppose an MPI rank was assigned 16 patches. Uintah would then prepare simulation variables for each of those 16 patches with global radiative data. This ultimately resulted in the global radiative data being duplicated 16 times in one MPI rank. The memory requirement routinely exceeded GPU memory capacity.

To avoid duplicating halo data, the task scheduler was further implemented to first recognize these global halo requirements in the task preparation, then prepare one of these simulation variables with global radiative data, and store these variables in another data store called the *level database* [3]. Other simulation variables with global halos assigned to other patches could use the same simulation variable in the level database. The application developer was further required to write task code obtaining these simulation variables from the level database, and not through the normal GPU Data Warehouse `get()` API call.

5.2 Initial GPU Scheduler Concurrency Challenges

Applying the initial GPU task scheduler to the Wasatch problem (Section 1.6.3) exposed new challenges. Initially, this work removed all CUDA blocking calls to allow asynchronous GPU task computation. This change led to numerous race conditions. These issues were all associated with allocation, halo preparation, data copies, and deallocation. An overview of these challenges is given below.

5.2.1 Task Scenarios Requiring Scheduler Coordination

The initial GPU task scheduler [10] targeted problems where tasks did not share simulation variables. Previously, it sufficed to simply give each grid variable a boolean value to indicate if a grid variable was valid in host memory and if it was valid in GPU memory. However, tasks from the Wasatch and ARCHES problems required sharing simulation variables. When two or more task scheduler threads processed tasks sharing simulation variables, race conditions emerged.

Two examples below highlight how simulation variables and possibly halo data may be shared among tasks. For the simplest example, refer to Figure 5.1. Suppose a simulation computes grid variable X in host memory on timestep 1. On timestep 2, suppose that grid variable X's status is redefined as a (read-only) *Requires* grid variable. The task graph has two tasks, task A and task B, which compute on the GPU and requires timestep 1's grid variable X for the computation. Two scheduler threads may both simultaneously attempt to allocate GPU memory space and prepare halo data.

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

These examples demonstrate a need for different task scheduler threads to utilize concurrent solutions. Overall, three types of actions must be considered within Uintah: 1)

allocations of simulation variables, 2) copies from one memory region location to another, and 3) gathering in halo cells. These are not limited to simulation variables in GPU memory space; they can occur in any space where tasks utilize simulation variables, such as host memory, NVRAM, and hard disk space.

5.2.1.1 First Attempted Solution Using Duplication

This work attempted to preserve what had been a core philosophy of Uintah's schedulers, treating scheduler threads and tasks as fully independent so that no two scheduler threads ever coordinated with one another. This core philosophy required less runtime code and quicker development by avoiding coherency scenarios altogether, and it was hoped this philosophy could be extended for GPU tasks. We proposed allowing multiple copies of data to exist in a memory location. For example, if both tasks A and B need grid variable X from the previous time step to be copied into GPU memory, then two instances of the same grid variable X would be created and copied in GPU memory, with each task gaining ownership over one of these. This approach created new problems. Runtime overhead would increase due to copying multiple instances of grid variables. Runtime overhead would also increase if boht tasks required halo data gathering. Perhaps most importantly, task launch times would always be delayed as all tasks must always wait until their data is prepared in the needed memory location, whereas if two or more parallel tasks shared grid variable data, then any task can proceed the moment the shared data is ready. For these reasons, this approach was not implemented.

5.2.2 Task Declaration Design Flaw

A completely separate issue occurs due to a disconnect from an application developer's perspective. He or she must first declare a list of simulation variables a task *will* utilize, and that list may differ from the simulation variables the application developer *actually* utilizes in task code. Effectively, the application developer must specify simulation variables twice. This Uintah model requires the application developer to enforce correctness between the task declaration and task code. When correctness does not occur, Uintah crashes in subtle and hard to detect ways. This flaw affects both the existing OnDemand Data Warehouse and the GPU Data Warehouse in different ways.

An example is found when simulation variables are not declared but still used. Sup-

pose an application developer uses a simulation variable in two tasks but only declared that variable as belonging to one task. Further, suppose that these two tasks execute simultaneously, and then the following sequence occurs. 1) The first task obtains the simulation variable. 2) The second task obtains the simulation variable. 3) The first task ends, and Uintah has the option to deallocate the space as it assumed only one task would use the simulation variable. 4) The second task still has a simulation variable with an address to deallocated memory. Experience has demonstrated that application developers will often create these race condition scenarios simply by forgetting to add another *Requires* in a task declaration. Further, these race conditions are difficult to identify when thousands of tasks and thousands of simulation variables are used within an MPI rank.

Another example is found when simulation variables are declared but not used. Here Uintah spends additional overhead preparing a simulation variable for use, but never uses it. Further, Uintah cannot deallocate the variable efficiently and must wait until the end of a timestep when the entire data store object is deallocated.

5.3 This Work's GPU Task Scheduler

The remainder of this chapter covers all the modifications to the GPU task scheduler as outlined at the beginning of this chapter. This chapter relies on the data structures already described in Chapter 4.

5.3.1 Utilizing Asynchronous CUDA Streams

Nvidia GPUs can efficiently compute very large problems either synchronously or asynchronously. Very large problems can be decomposed into hundreds or thousands of CUDA blocks, and the GPU efficiently distributes those GPU blocks among its processors as the processors become available. Some small inefficiencies can occur at the end of computing large problems when only a few blocks remain and no new work can continue.

Unfortunately, this model cannot work for Uintah. Even though simulations utilizing Uintah typically use task sizes considered coarse-grained relative to task sizes in other AMT runtimes, they are too small to spread among the GPU. For example, suppose a task operating on 16^3 cells assignes one GPU thread per cell and one block per two-dimensional patch slice. That decomposition would not be distributed among GPUs that have more

than 16 sets of streaming multiprocessors. Thus, the GPU becomes starved for work in a synchronous execution model.

A strategy to overcome starvation is by leveraging asynchronous CUDA actions, so that data copies and kernel executes can all overlap with each other and fill all GPU processors. CUDA synchronization points must be avoided, as they prevent CUDA actions issued after the synchronization point from overlapping with current actions.

The task scheduler was first modified by assigning every Uintah task a CUDA stream. GPU tasks naturally utilized these streams for all data copies into the GPU and CUDA kernel execution. CPU tasks were given CUDA streams as well, as the task scheduler thread may require copying data from GPU memory into host memory, and would likewise need an asynchronous stream for that action. Tasks interacting with many GPUs receive many CUDA streams, one CUDA stream per GPU. An example is a CPU task responsible for archiving all simulation variables to disk, as this task must copy data GPU-to-host from all GPUs in the MPI rank.

The streams also supply critical information needed to determine when a prior asynchronous CUDA action has completed. Once all possible CUDA actions have been asynchronously invoked for a given task, Uintah places the task in an appropriate task queue. Later, a task scheduler thread queries all streams in tasks in these queues to determine if the CUDA actions have completed. This process is further described in Section 5.3.3.

5.3.1.1 Many Streams Needed for Overlapping Work

Assigning every task a unique CUDA stream is crucial for GPU work overlapping. Consider a counterexample where Uintah limited itself to only one stream. Kernels executed on that stream must proceed sequentially, and cannot overlap. Also consider a counterexample where Uintah utilized a fixed number of N streams. Kernels on each stream must proceed sequentially, but kernels in different streams can overlap, and so only N amounts of overlapping are possible. By giving every task a unique stream, Uintah can optimally overlap kernels on the GPU where possible. In our testing, CUDA freely managed actions on tens of thousands of streams without any noticeable additional overhead.

5.3.2 Runtime Work Coordination Among Scheduler Threads

Consider again the example described in Figure 5.1 of both tasks A and B needing grid variable X copied from host memory into GPU memory. This work now solves the concurrency problems through scheduler thread coordination [2, 4]. The two scheduler threads preparing these two tasks utilize the atomic bit set described in Section 4.4.4.1. Each scheduler thread performs an atomic read to see if space for this grid variable X in GPU memory has been allocated. Suppose the *allocated* bit was previously set. Then both scheduler threads can then see if the GPU grid variable's *copying in* bit or *valid* bit have been set. Suppose neither the *copying in* bit nor the *valid* bit have been set. Each scheduler thread attempts an atomic test and set on the *copying in* bit, and the winner must perform the GPU-to-host copy. The other scheduler thread continues analyzing other grid variables needed for the task. Similarly, Uintah can now overcome all task scheduler race conditions that affected the OnDemand Data Warehouse and the Pthread driven scheduler. (Additional examples of these race conditions are given in Section 4.4.4 and Section 5.2.1.)

The atomic bit sets allow a second key feature, ensuring a task is ready for execution, whether these tasks are executed on CPUs or GPUs. In the prior schedulers, enforced synchronization meant that a task scheduler thread could perform all necessary simulation variable copies between memory spaces, and then immediately afterward execute the task. In this work's task scheduler, a scheduler thread checks and ensures *all* simulation variables assigned for that task have the appropriate *valid* or *valid with halo data* bits set. If some bits are not yet set, then it must be the case that another task started but did not complete its assigned action. The task goes back into an appropriate queue and will be checked again shortly after. The scheduler thread is unaware which other scheduler threads are preparing necessary simulation variables, only that a bit has been set indicating the desired action will be completed by another scheduler thread. These work queues are described in more detail in Section 5.3.3.

Section 4.4.4.1 described how task scheduler threads utilizing the atomic bit set could be viewed as an extension of the MSI cache coherency protocol. This work's task scheduler could theoretically reduce all the bits in the atomic bit set down to just two bits, one to indicate if the variable is ready for use (the S state), and another to indicate if the variable is in some invalid state (the I state), with variable modification (the M state) already handled

through task graph dependency logic. Where the atomic bit set's additional states have demonstrated considerable value is in debugging errors during scheduler code development. Multiple bits can be simultaneously set, which better allows for understanding the state of simulation grid variables at any given time. For example, a grid variable may be listed as both allocated and valid in host memory. Another example is listing a variable as allocated, valid, and currently gathering halo data. If a grid variable does not exist in a memory space, then no bits are set. In essence, this model allows Uintah runtime developers to perceive simulation variables as having a lifetime of states, spanning allocation, usage, halo gathering, and deallocation. Uintah runtime developers can query a simulation variable's bit set during a timestep and glean how far it has progressed through its lifetime. This information allows the runtime developer to narrow down a set of conditions that likely occurred leading up to a bug.

5.3.3 New Task Queues

The task asynchrony described thus far requires that tasks proceed through multiple phases, such as receiving MPI data from other MPI ranks, allocating space for simulation variables, copying simulation variables, gathering halo data into simulation variables, executing tasks, and deallocating simulation variables. When task actions are invoked asynchronously, Uintah is not automatically notified when a task has completed a given phase, and so a task scheduler thread must check when a task has completed one phase so that it can be moved to a new phase. This process is accomplished by having a series of task queues.

5.3.3.1 GPU Task Queues

A GPU task now proceeds through five queues as shown in Figure 5.2 [2, 4]. 1) Process all MPI receives. 2) manage any halo data gathers in host memory if possible. For all potential GPU allocations, copies, and halo data gathers necessary for this task, determine which of those becomes assigned to this task using an atomic first arrival policy. Perform asynchronous host-to-GPU copies for the task's grid variables for which it is responsible for copying into the GPU. Create Task Data Warehouses (Section 4.4.4.2) for this task which contains information about each grid variable and necessary meta data for later halo data gathers (Section 4.4.5). Asynchronously copy these Task Data Warehouses to the GPU. 3)

Set every *valid* bit to true for all *Requires* grid variables this task was assigned to copy into the GPU. For every grid variable requiring halo data, see if all adjacent halo data is valid in GPU memory. If all needed grid variables can proceed with halo data gathering, and this task was assigned responsibility to gather that halo data, asynchronously launch a GPU kernel to complete this action. If halo data gathers cannot yet take place due to some adjacent halo data not yet in this GPU, place this task back into this work queue to be checked later. 4) For every grid variable for which this task was responsible for gathering halo data, set those *valid with halo data* bits. Check if all grid variables requiring gathered halo data *valid with halo data* bits set. If not, place this task into this work queue to be checked later. If all grid variables are ready, asynchronously launch the GPU task. 5) Set every *valid* bit to true for all *Computes* and mark the task as done.

5.3.3.2 CPU Task Queues

Previously in the first GPU runtime model [10], the scheduler never required a CPU task to search any GPU memory space for a simulation variable. As a result, the task scheduler utilized only two queues. The first queue for processing MPI receives, and the second for executing tasks.

A CPU task now proceeds through four queues as shown in Figure 5.3 [2, 4]. 1) Process all MPI receives. 2) For all potential host memory allocations and copies necessary for this task, determine which of those becomes assigned to this task using an atomic first-touch policy. Perform asynchronous GPU-to-host copies for all task grid variables for which it is responsible for copying into host memory. 3) Set every *valid* bit to true for all *Requires* grid variables this task was assigned to copy into host memory. If some task grid variables are not yet valid in host memory, place this task back into this work queue to be checked later. 4) Run the CPU task as all host data is available in host memory. Any needed halo data gathering happens during task execution.

5.3.3.3 Differences Between GPU and CPU Queues

The workflow for GPU queues and CPU queues work flow is fairly similar. A task's life cycle of data preparation, halo data management, execution, and updating of bit sets follows the same general concepts. The only major differences between CPU and GPU task queues are a follows: 1) GPU tasks need some form of task data stores due to the difficulty

of managing concurrency for grid variable data stores in global GPU memory. 2) GPU tasks are queued and executed asynchronously through streams while CPU tasks are executed on the same CPU scheduler thread which processed that work queue. 3) CPU tasks can prepare simulation variables (allocation and halo gathering) during task execution, rather than relying on the scheduler to allocate and prepare these variables beforehand.

The results of combined work so far can be seen in Figure 5.4. Simulation variables can stay resident in GPU memory. Further, halo scatters and gathers are more efficiently managed. Altogether these changes open up Uintah to efficiently execute a broad class of problems where short-lived GPU tasks execute on the order of milliseconds [2, 4].

5.3.4 Efficient Memory Management Using Contiguous Buffers

Allocating GPU memory and copying memory host-to-GPU can be expensive operations because of latencies associated with these API calls. This section describes a mechanism where a strategy utilizing fewer API calls results in faster task execution times [2, 4].

The initial GPU runtime allocated memory one grid variable at a time and copied these grid variables one at a time. The associated accumulated API latencies become an issue when the accompanying GPU task executes in just a few milliseconds, as seen in Figure 5.5.

5.3.4.1 Reducing GPU Allocation Latency

To address this issue, a straightforward method for utilizing a contiguous buffer was implemented. For a given task, Uintah's runtime computes the total size of all *Computes*, *Requires*, and halo data not yet in a memory location or the in the process of allocating or copying into that memory location (Section 5.3.2). A contiguous buffer was allocated in GPU memory. Then multiple host-to-GPU copies were invoked for each *Requires* grid variable and halo cell staging variable into the allocated buffer on the GPU. This approach yielded improvements as shown in Figure 5.6.

5.3.4.2 Attempts at Reducing GPU Copy Latency

This work investigated reducing the latency overhead when data variables are copied host-to-GPU. The mechanism is by forming a packed contiguous buffer in host memory with the goal of performing only one host-to-GPU copy instead of several. The larger difficulty here is that different tasks each require a different set of simulation variables,

only sharing some, but not all, of a node's simulation variables among them. Host buffers were allocated and filled them using host-to-host copies of individual simulation variables. These packed buffers were then copied into GPU memory. In all cases tested the cost of host-to-host copies outweigh the latency of many host-to GPU copies.

5.3.4.3 Contiguous Buffer Results

Table 5.1 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 transport equations, respectively. Computations were performed on an Nvidia GTX680 GPU and an Intel Xeon E5-2620 with CUDA 6.5. With all these improvements, speedups were observed due to reduced overhead ranging from 1.27x to 2.00x for a variety of test cases.

The advantage in Uintah's contiguous buffer approach compared to similar runtime GPU buffer allocation schemes [88] is the ability to use temporal runtime knowledge to optimize these allocations. This work demonstrates that buffers yield speedups in all tested scenarios [2, 4].

5.3.5 Avoiding Allocation and Dealloction Issues

This section covers solutions to two additional concurrency challenges. The first challenge is that the GPU Data Warehouse [10] deallocated simulation variables in GPU memory at the end of a GPU task, limiting reuse of data. The second challenge was described in Section 5.2.2, where a data store is affected by a race condition problem caused by incorrect task declaration. Both challenges are rooted in deciding and ensuring when simulation variables are allocated and deallocated.

5.3.5.1 Delaying Deallocation

Parallel execution of tasks requires that simulation variables are not automatically deallocated once a task completes. The OnDemand Data Warehouse for host memory simulation variables utilizes a scrub counter, with one counter per simulation variable, and the counter computed during the task graph compilation phase. The scrub counter design cannot be applied to GPU simulation variables. Each `cudaFree()` call also invokes a synchronization barrier, which would negate the gains from the asynchronous work

accomplished in Sections 5.3.1, 5.3.2, and 5.3.3.

As a result, the GPU task scheduler now frees all simulation variables by placing allocated buffers back into a resource pool. New allocations first check the pool for a buffer of the requested size, and a `cudaMalloc()` is invoked only if no such buffer was found. Uintah performs the deallocation process at the end of each timestep. Future work can apply a scrub counter to GPU simulation variables to allow buffers to be placed back into a pool during a timestep.

5.3.5.2 Avoiding Issues With Improperly Defined Tasks

As described in Section 5.2.2, occasionally, an application developer would neglect to declare that a task will use a simulation variable, and would then write task code using that simulation variable. This user error is common and difficult to debug, as sometimes the data store object will have that simulation variable available for that task, due to a previously executing task utilizing it. This same design flaw resurfaced when the initial GPU Data Warehouse was utilized by many GPU tasks executing in parallel.

The Task Data Warehouse design aids in fixing this issue. Each Task Data Warehouse only contains entries for simulation variables the user declared in the task declaration phase. Further, the combination of new concurrency mechanisms described in the prior sections ensures those simulation variables will be available for concurrent reads. Now when a user attempts to wrongly obtain a simulation variable that wasn't declared in the task declaration phase, Uintah can guarantee the simulation variable will not exist, and can likewise inform the user with an appropriate error message.

A simple future work addition is also possible. Uintah can now be modified to detect if a simulation variable goes unused in a task, and warn the application developer appropriately. Each Task Data Warehouse can supply a counter associated with each simulation variable entry. Whenever the application developer requests a specific simulation variable, the associated counter is incremented. After task execution, Uintah can check that Task Data Warehouse and ensure all counters are non-zero. If any are zero, this guarantees a variable was declared in the task declaration phase, but not used. This mechanism also augments the scrub counter approach (Section 4.2.4) and properly guarantees deallocation can and will occur in the middle of a timestep when all tasks have finished using that

simulation variable.

5.3.6 Large Halo Sharing

In the RMCRT target problem, multiple tasks in an MPI rank routinely required different simulation variables (grid variables) while using the same data dependencies (halo data). Prior task scheduler logic could share halo data at the cost of locking mechanisms [3], or the task scheduler could retain asynchrony at the cost of duplicating halo data [4]. The target problem required sharing both halo data and retaining asynchrony as the prior task schedulers either executed inefficiently due to contention caused by locks or the problem could not fit into GPU memory due to duplicated halo data. This section describes the problem in detail and presents a novel solution [15].

Properly sharing simulation variable data and its halo dependencies evolved through three distinct methods. The first approach [3], termed **Method I** (Section 4.3), created one shared data object that is composed of all patch variables and all halo data for a simulation variable. This approach used a third data store, called the *level database*, to manage simulation variables that encompassed an entire mesh level. Application developers were required to manually retrieve these shared data objects from the level database and not from the GPU Data Warehouse. Method I used CPU locks and GPU barriers that affected performance by preventing GPU kernel overlap, resulting in GPU tasks executing serially. A profiled run of RMCRT tasks in Method I is shown in Figure 5.7.

The second approach, **Method II** [2, 4], allowed sharing of patch variables among tasks but did not allow the sharing of halo data. Scheduler threads coordinated among themselves through atomic bitsets which represent lifetime states of a simulation variable. One bit set was assigned to each patch variable to ensure only one scheduler thread can allocate, prepare halo data, and copy a simulation variable. The overall goal was asynchrony and overlapping of GPU tasks, as can be seen in Figure 5.8. Method II avoided the *level database* in Method I. However the duplication of halo data drastically increased memory usage overhead in the target problem. A simplified visual representation of the new data store layout is given in Figure 5.9.

Method III [15] combines the best attributes of Method I and Method II. Method III introduces a new task scheduler and data store changes to share both patch variables

and halo data in shared data objects. The task scheduler is asynchronous and lock-free. Method III avoids a level database approach, a large data store code rewrite, and overly complicated logic.

This solution 1) decouples components of a data store item, 2) allows multiple data store items to share decoupled objects, 3) introduces additional asynchronous task scheduler logic for shared decoupled objects, and 4) fits into existing data store logic throughout Uintah. A simplified visual representation of the new data store layout is given in Figure 5.10. A data store entry now only contains identifying metadata (e.g., simulation variable name and patch ID). A **data description object** contains metadata for simulation variable layout and usage status. Multiple data store entries may share a data description object using shared pointers. If data sharing is not needed (which is the case for most existing simulations using Uintah), existing data store functionality is retained by having each data store entry point to its own data description object.

Next, the atomic status bit set representing lifetime states of the simulation data is decoupled out of the data store entry and moved into the shared data object. This change is needed as atomic operations must occur on a single shared bit set. The shared data object will occur only once, while multiple data store entries are possible for that data object. Had the atomic bit set stayed in the data store entries, then more convoluted solutions would be needed, such as making multiple copies of the bit set and using locks to update them.

Creation of data description objects is managed by task scheduler logic. When a scheduler thread analyzes an upcoming task for simulation variable preparation, it computes a spatial box necessary to encompass all halo data for a group of simulation variables assigned to various patches on that node. If the halos are small (such as one cell of halo data), the box only spatially encompasses one simulation variable and requires no additional work. If the halos are large and multiple simulation variables are spatially contained within a box, the scheduler thread begins a process to share a data description object among all the corresponding data store entries.

As an example, suppose a simulation is laid out in a 2D grid exactly as shown in Figure 5.10. Each node is assigned a square block of four patches for task computation. Further, suppose the halo for a simulation variable requires a full patch of 16 cells in every direction. One possible box, shown in Figure 5.10, encompasses a region of 16 patches numbered as

shown. The four patches assigned to the node would then share a data description object, and the four data store entries for this simulation variable would all refer to the same shared object.

Multiple scheduler threads preparing different tasks may simultaneously attempt to form the same shared data description object for a contiguous group of simulation variables. An atomic bit set is employed for coordination among scheduler threads. As no shared object has yet been created, the bit set used is the one associated with the smallest unique integer patch ID among the boxed entries. Once the separate patch variables have been merged into one shared data description object, the atomic bit set is copied into the shared object and a bit updated to indicate the simulation variable and its associated halo data are all valid and ready for use. Any other scheduler threads processing a task requiring any of these data store entries in this box must either wait or seek new work from a work queue.

Previously, seven bits of the bit set described these different states of simulation variables: `allocating`, `allocated`, `copyingIn`, `validForUse`, `gatheringHaloCells`, `validWithHaloCells`. This work added bits for `mergingDataObjects` and `mergedDataObject`. Additionally, the status bit set now uses an additional 16 bits to hold a reference-counting short integer. This count represents how many data store entries are sharing the data description object.

Overall, this new data store design has many desirable qualities. There is no need for a third data store (the *level database*) as described in Method I. Noncubic domains are supported by allowing multiple data description objects to exclude void regions where no patches exist. Task scheduler thread asynchrony is retained. Memory overhead is kept low since duplication is avoided. All variables are managed in current data stores without requiring a large refactoring of the infrastructure code. All logic for this data store can co-exist for other projects using Uintah which have dramatically different data dependency requirements.

The most important improvement is a reduction of memory usage on a node. Table 5.2 shows with this approach, the target problem fits within GPU memory, whereas before it could not fit within GPU memory in Method II. Results were computed on a single node with an Intel Xeon CPU E5-2660 @ 2.20GHz with 32 GB RAM running Uintah on 16 CPU

threads and an Nvidia GeForce GTX TITAN X with 12 GB of RAM.

5.3.7 Improving GPU Occupancy

This subsection describes an optimization to this scheduler model enabling wall time speedups through better GPU occupancy. The target problem assigned 15 or 16 patches per node. However, as seen in Figure 5.8, this patch count inefficiently occupied a GPU throughout a timestep. Titan's NVIDIA K20X GPUs contained 14 streaming multiprocessors (SMs), not enough for the 15 or 16 GPU tasks to compute simultaneously, slightly oversubscribing the total SMs. Instead of assigning only 14 patches per node (and thus using $\sim$10% more compute hours on DOE Titan), a solution was desired which retained a low patch count per node for faster dependency analysis while also providing good GPU occupancy.

The first attempt split a kernel into multiple blocks, but a kernel would not vacate SMs until all of its blocks computed, leaving some SMs idle. A second attempt split each GPU task into multiple kernels and launched them on a shared task GPU stream, but this created serialization among kernels when multiple CPU threads asynchronously launched multiple kernels intermixed with host-to-device transfers. The adopted solution [15] assigns each GPU task multiple GPU streams and splits a task into multiple kernels. The task scheduler does not consider a GPU task complete until each GPU stream assigned to that task completes all of its operations. Figure 5.11 demonstrates significantly better SM occupancy with smaller kernels on multiple streams. Speedups of 1.2x were measured compared to the baseline of supplying only one kernel and one stream per task. This approach of launching higher numbers of small kernels to a GPU best allows Uintah to target GPUs with a differing number of SMs. The only requirement is that a GPU's compute capability version must support more concurrent kernels than SMs.

5.4 Results

Results for Poisson tasks (Section 1.6.1), RMCRT tasks (Section 1.6.2), and Wasatch tasks (Section 1.6.3) are given below. These three problems highlight different Uintah computation characteristics and their associated difficulties. The Poisson tasks highlight Uintah overhead by executing tasks that are short-lived and have very little reuse of data.

The Wasatch tasks highlight a category of tasks which make heavy use of numerous data variables requiring halo management, and the tasks themselves typically execute within a few milliseconds or less. RMCRT tasks highlight tasks with non-uniform halo requirements, where tasks have a mixture of localized halos and global or nearly global halos. RMCRT tasks typically take seconds to minutes to execute, and thus are less affected by Uintah launch overhead times, but are affected by memory requirements.

5.4.1 Poisson Equation Solver Results

Poisson tasks (Section 1.6.1) are used for this work explicitly to expose runtime overhead. As shown in Algorithm 1, the task computes relatively quickly, as each cell in a timestep requires six memory read operations, six basic arithmetic operations, and one memory store operation. Cell iteration for both the CPU and GPU computations starts on a cell on one patch face, then iterates down one dimension until reaching a cell on the opposing patch face. This iteration pattern limits data reuse to just one cell likely being found in the L1 cache. Poisson problems are memory bounded due to little reuse of data (e.g., low flops per byte).

Table 5.3 compares this problem on the initial and current runtime system. Data for this table was computed using 50 iterations on a simulation grid using 12 patches [2]. 12 CPU cores were used for 12 CPU tasks. Speedups provided to show a reduction in runtime overhead in GPU tasks and highlight when GPU tasks become feasible over CPU tasks. The profiled machine contained an Nvidia K20c GPU and an Intel Xeon E5-2620 with CUDA 5.5.

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

The 192^3 and 256^3 case demonstrates a major motivation for these GPU runtime enhancements. Here, the initial GPU runtime was 3.5x and 3.2x slower than the CPU task version, respectively. Now the current GPU runtime computes this problem 1.33x faster for 192^3 and 1.71x faster for 256^3 compared to the CPU task version. This result demonstrates

Uintah can move more CPU tasks to the GPU to obtain speedups. For smaller grid sizes for this problem, the CPU task overhead is smaller than the GPU task overhead, and this results in faster overall CPU times.

Detailed profiling of the 192^3 case indicated that the previous GPU runtime had overhead between time steps of roughly 49 milliseconds. Under the current runtime, this overhead has been reduced to roughly 2 to 3 milliseconds. The GPU computation portion of this task used ten 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. Work is underway to implement the memory pool and stream pools to reduce these times further.

The Current vs CPU speedup column in Table 5.3 highlights Uintah GPU task preparation overhead. For 64^3 and 128^3 cases, Poisson tasks execute faster on CPUs. The major drivers of this GPU task overhead are preparing a Task Data Warehouse (Section 4.4.4.2) for each task, invoking a CUDA kernel to perform all halo copies in GPU memory, and many non-coalesced memory invocations during the halo copy process. Future work is planned to reduce this overhead.

5.4.2 Wasatch Results

Wasatch tasks are an ideal case for the work in Chapter 4 and Chapter 5. The Wasatch tests profiled solved multiple partial differential equations (PDEs) and used as many as 120 PDE related variables per time step. Each task computes within milliseconds. Although these tests only run on one patch, they utilize periodic boundary conditions, meaning that each patch edge is logically connected with the patch edge on the opposite side, and thus halo data transfers still occur. Table 5.4 gives time to solutions for two different Wasatch tests which solve 10 and 30 transport PDEs, respectively. For the data in this table, the CPU thread counts are managed by Wasatch tasks to maximize efficiency. Speedups are provided to show a reduction in runtime overhead in GPU tasks and highlight when GPU tasks become feasible over CPU tasks. Computations were performed on an NVidia GTX680 GPU and an Intel Xeon E5-2620 with CUDA 6.5.

The key aim of this work is to allow Uintah's GPU support to be opened to a broader

class of computational tasks. As Table 5.4 shows, the original runtime system processed GPU tasks slower than CPU tasks in all tested Wasatch cases. The current runtime system for the same GPU tasks now obtains significant speedups in most cases. Only when patch sizes are small do CPU tasks still perform fastest.

5.4.3 RMCRT results

This scaling study [2] focuses on a reverse Monte Carlo ray tracing (RMCRT) approach to radiation modeling that makes novel use of Uintah's AMR capabilities to achieve scalability. This problem [21] uses a two-level AMR mesh and is based on the benchmark described by Burns and Christen [5]. This challenging problem exercises all of the main features of the AMR support within Uintah as well as additional radiation physics. This benchmark problem also requires the use of the concurrency improvements detailed in Chapter 4 and Chapter 5. The radiation portion of this calculation was run on the DOE Titan system, using the single Nvidia K20x GPU available on each node. A fine level halo region of four cells in each direction, x, y, z was used. The AMR grid consisted of two levels with a refinement ratio of four, the fine mesh being four times more resolved than the coarse, radiation mesh.

For two separate cases, the total number of cells on the highest resolved level was 128^3 and 256^3 (blue and red lines respectively in Figure 5.12), with 100 rays per cell in each case. The total number of cells on the coarse level was 32^3 and 64^3. In each of these strong scaling cases, a fixed patch size of 16^3 cells was used. Each data point represents a 2X increase in the number of GPUs assigned to the computation.

The principal result illustrated in Figure 5.12 is the near order of magnitude speedup in mean time per timestep for each problem. This improvement is largely due to the introduction of non-blocking Task Data Warehouses as described in Section 4.4.4.2, which allows for many smaller patches to execute simultaneously due to kernel overlapping (see Figure 5.13 and Figure 5.14).

5.5 Task Scheduler Summary

The work in this chapter modifies the GPU task scheduler enabling asynchronous and concurrent execution of GPU tasks without synchronization barriers. New task scheduler

logic allows the scheduler to properly stage simulation variables in CPU or GPU memory, as well as preparing halo data prior to GPU task execution. Tasks scheduler threads now cooperate in preparing simulation variables needed for each other's tasks. Each CPU and GPU task is assigned CUDA streams, allowing for asynchronous GPU task execution and overlapping asynchronous data copies host-to-GPU and GPU-to-host. Task Data Warehouses are prepared in host memory by the GPU task scheduler and these task data warehouses are loaded with both a task's simulation variables and logic how to complete the halo gathering process. Additional data store and task scheduler logic allows the GPU to share large halo data among many simulation variables, which allows full production runs of simulations using RMCRT on GPUs.

Chapter 4 and Chapter 5 now allow application developeres to run a heterogeneous mixture of CPU and GPU tasks in many production ready problems. The next key challenge is supporting ease of development through portable loop code and portable task code. These challenges are addressed in Chapter 6 and Chapter 7.

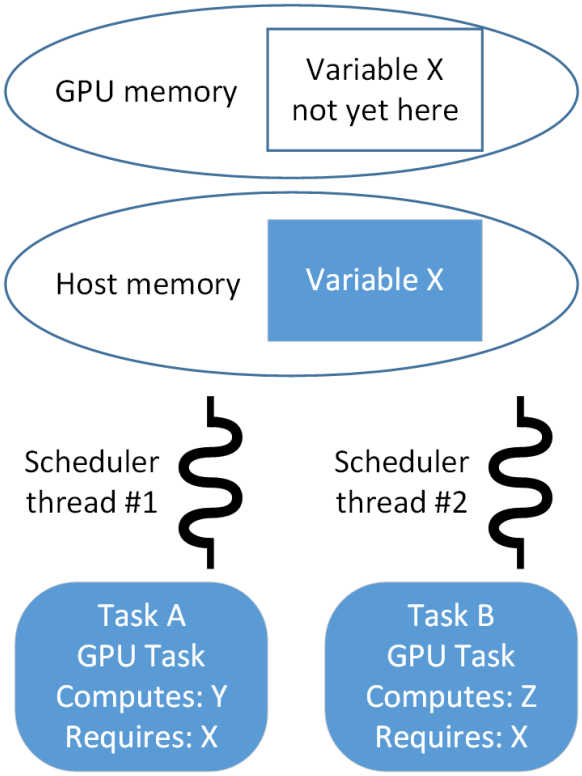

Figure 5.1: Two scheduler threads are each assigned a different task to analyze. They do so independently in parallel. Each sees that simulation grid variable X is not yet in GPU memory. The runtime must determine which thread performs the data copy.

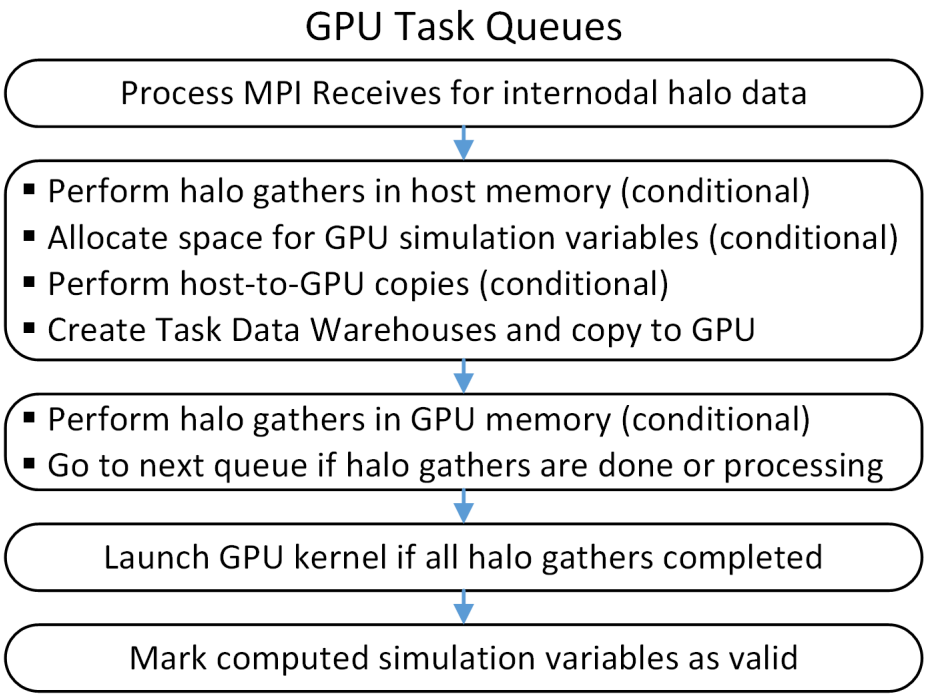

Figure 5.2: A simplified flow of all scheduler queues a GPU task must pass through.

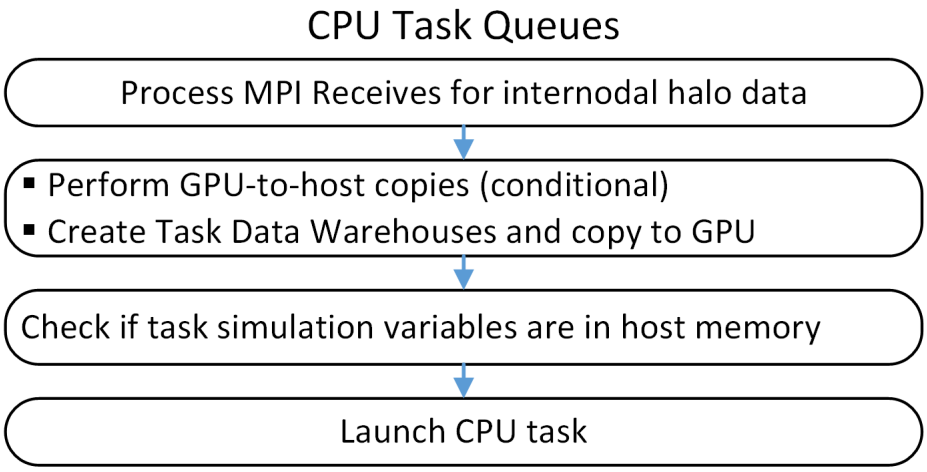

Figure 5.3: A simplified flow of all scheduler queues a CPU task must pass through.

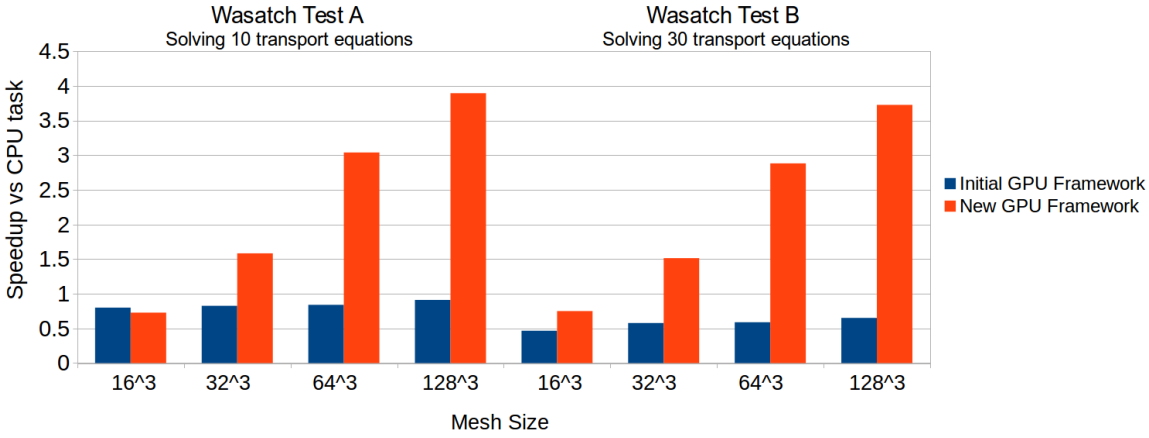

Figure 5.4: Short-lived GPU tasks were most susceptible to runtime overhead. Performing halo gathers entirely in GPU memory helped make total GPU simulation wall times tasks faster than CPU simulations. Computations performed on an Nvidia GTX 680 GPU with CUDA 6.5 and an Intel Xeon E5-2620. [2]

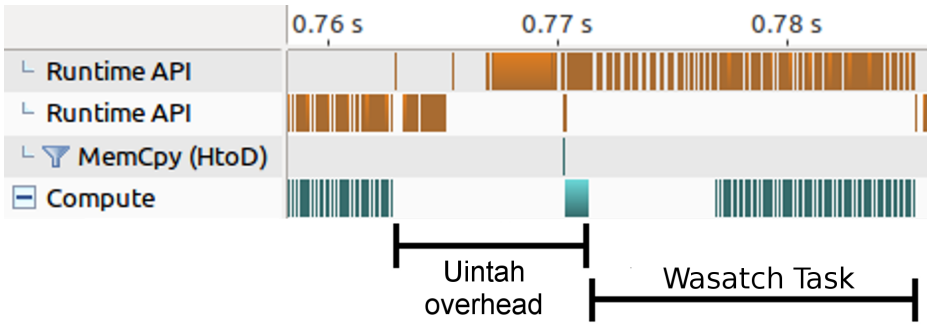

Figure 5.5: A profiled time step for a Wasatch task using the initial runtime system. Most of the Uintah overhead is dominated by freeing and allocating many grid variables.

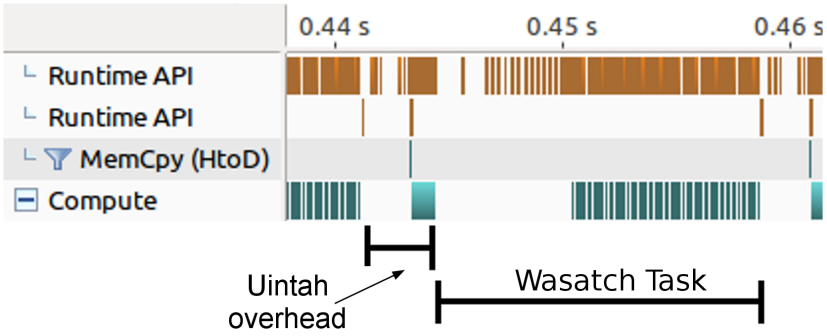

Figure 5.6: A profiled time step for a Wasatch task using the new runtime system. The runtime system determines the combined size of all upcoming allocations, and performs one large allocation to reduce API latency overhead.

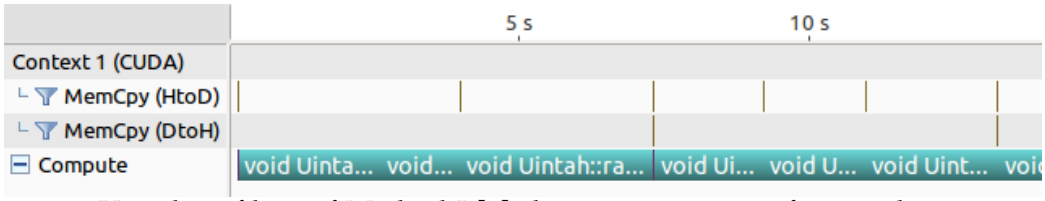

Figure 5.7: Visual profiling of Method I [3] showing a portion of one radiation timestep using RMCRT GPU kernels. Memory copies are shown in the first two `MemCpy` line. Seven RMCRT GPU task executions are shown in teal. This figure demonstrates no overlapping of kernels due to blocking data store synchronization.

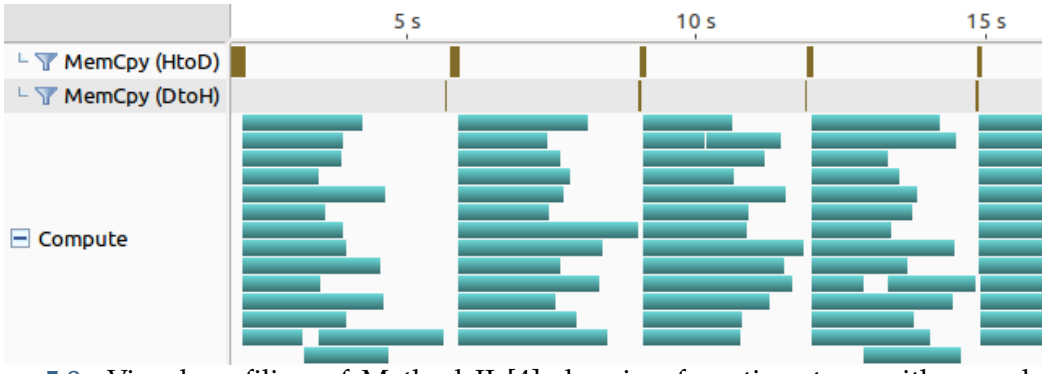

Figure 5.8: Visual profiling of Method II [4] showing four timesteps with asynchrony and overlapping of GPU tasks. The gaps between timesteps illustrates lack of full GPU occupancy.

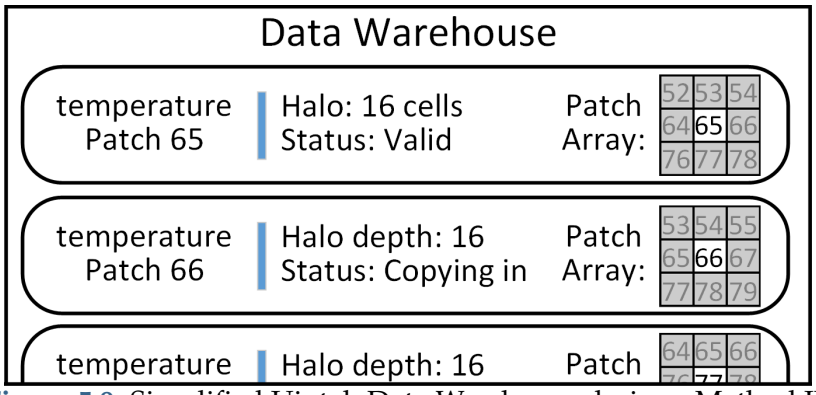

Figure 5.9: Simplified Uintah Data Warehouse design - Method II.

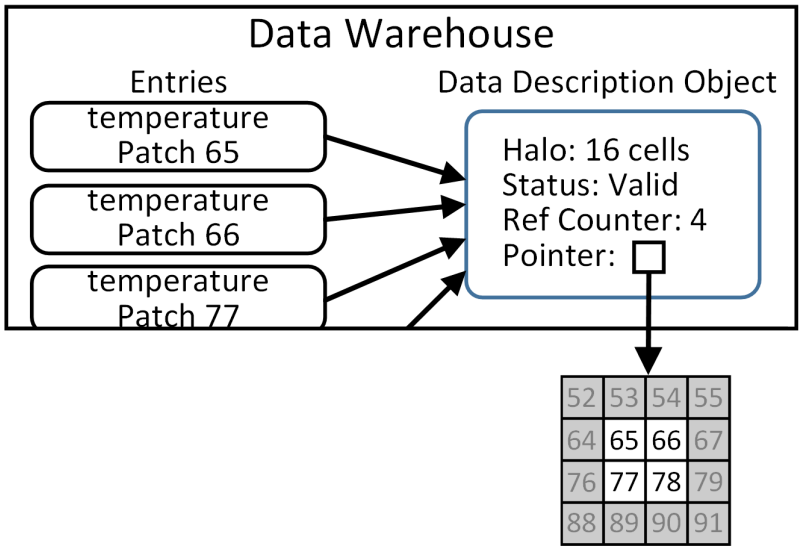

Figure 5.10: Simplified Uintah Data Warehouse design - Method III.

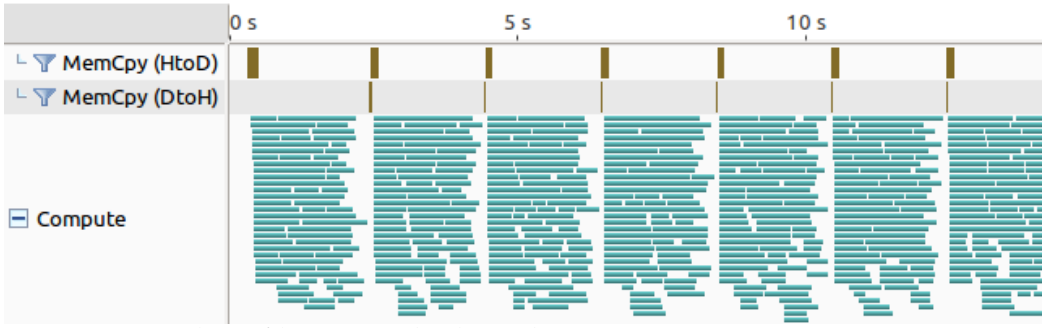

Figure 5.11: Visual profiling - Method III showing six successive timesteps. Uintah's scheduler supplies each GPU task multiple streams so that task can be split into multiple kernels, executed concurrently and achieving better GPU occupancy.

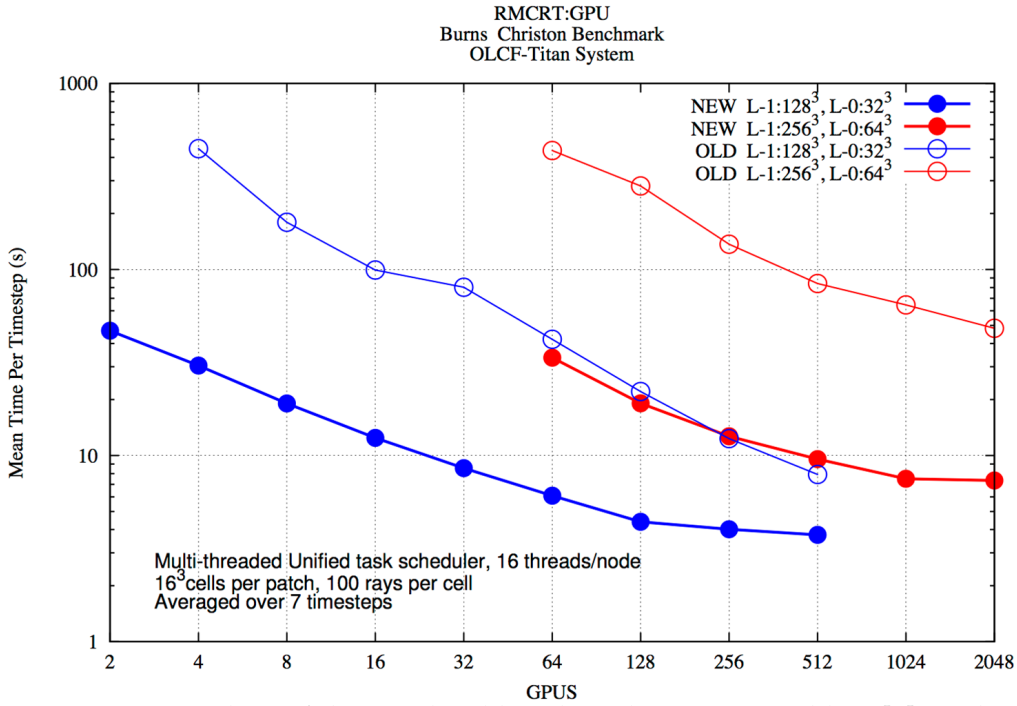

Figure 5.12: Strong scaling of the two-level benchmark RMCRT problem [5] on the DOE Titan system. L-1 (Level-1) is the fine, CFD mesh and L-0 (Level-0) is the coarse, radiation mesh [2]. (Computations performed by Alan Humphrey.)

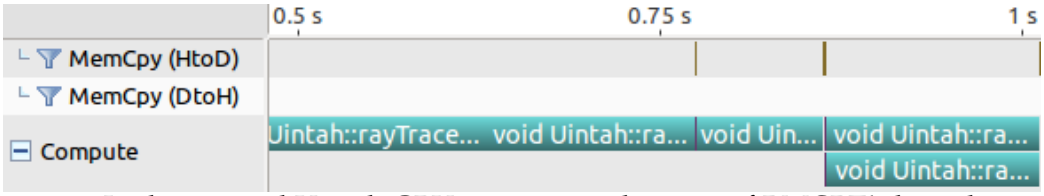

Figure 5.13: In the original Uintah GPU engine, overlapping of RMCRT's kernels is infrequent as copying the GPU Data Warehouse prior to task execution is done as a blocking operation to avoid concurrency problems.

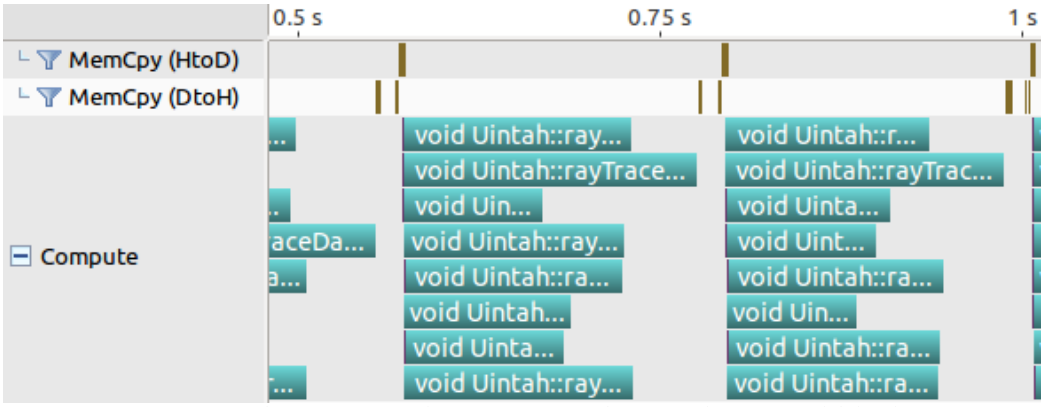

Figure 5.14: Because Task Data Warehouses were designed to avoid blocking operations when copied into GPU memory, RMCRT kernel overlap is achieved.

Table 5.1: Effect of contiguous buffers on Wasatch GPU tasks. Results computed by both Brad Peterson and Harish Dasari [2].

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 5.2: Results of running only GPU RMCRT benchmark tests for the three methods detailed in this section. Method I has low memory usage but high wall time overhead due to frequent GPU blocking calls. Method II improves wall times, but memory usage is unacceptably large. Method III's low overhead results in both faster wall times and low memory usage.

Improvements in Overhead			
Type of Mesh		Timestep (s)	Host memory usage after (MB)
Coarse: 32^3 cells, 4^3 patches Fine: 64^3 cells, 4^3 patches	Method I	1.79	57
	Method II	0.21	3073
	Method III	0.15	65
Coarse: 32^3 cells, 4^3 patches Fine: 64^3 cells, 4^3 patches	Method I	4.95	213
	Method II	1.41	23229
	Method III	0.82	279
Coarse: 32^3 cells, 4^3 patches Fine: 64^3 cells, 4^3 patches	Method I	6.44	218
	Method II	Exceeded memory	
	Method III	1.08	311

Table 5.3: Poisson Equation Solver GPU vs. CPU Speedup

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

Table 5.4: Wasatch tasks GPU versus CPU speedup.

Wasatch Test	Mesh Size	CPU Only (s)	Initial GPU Framework (s)	Current GPU Framework (s)	Speedup - Current vs Initial	Speedup - Current vs CPU
Test A - Solving 10 transport equations	16^3	0.08	0.10	0.11	0.91x	0.72x
	32^3	0.19	0.23	0.12	1.92x	1.58x
	64^3	0.79	0.94	0.26	3.62x	3.03x
	128^3	4.75	5.21	1.22	4.27x	3.89x
Test A - 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

CHAPTER 6

KOKKOS MODIFICATIONS FOR ASYNCHRONOUS GPU EXECUTION

This work makes us of Kokkos [89] to achieve a CPU, GPU, and Intel Xeon Phi portable solution for Uintah task code. Kokkos provides application developers a C++ approach to write parallel loop code once and compile and execute that code on multiple architectures. Kokkos also provides portable abstractions for common API tools, such as random number generation, and API for parallel loop launch parameters.

One current challenge of using Kokkos with AMT runtimes is executing tasks to efficiently hide latency costs while fully utilizing the GPU. Volkov [90] described this process relative to Little's Law [91]: "Executing more threads at the same time typically results in better throughput, but only up to a limit." Further, "hiding both arithmetic and memory latency at the same time may require more warps than hiding either of them alone." For example, suppose a compute-bound CUDA kernel best avoids arithmetic latency by using eight warps (32 threads per warp) per streaming multiprocessor (SM), and that GPU has 24 SMs. This kernel requires $8 * 32 * 24 = 6144$ threads for full throughput. A standard Uintah task operating on less than 6144 cells would incur arithmetic latency and starve GPU cores of work.

Currently, Kokkos only synchronously executes GPU parallel loops. Synchronous CUDA kernel execution incurs launch latency. AMT runtimes seek to reduce the relative cost of launch latency by executing sufficiently large tasks [92]. Asynchronous CUDA kernel execution can reduce this launch latency overhead by having the GPU perform work on some active kernels while the GPU is in the process of initializing others. Thus, asynchronous kernel execution allows an AMT runtime to efficiently execute smaller tasks.

This work modifies Kokkos itself to enable asynchronous task execution. This process

requires modifications in three areas. 1) Adding CUDA stream support to Kokkos CUDA loops. 2) Allowing Kokkos to store multiple functors in the GPU's constant cache memory. 3) Modifying other relevant Kokkos API.

This chapter introduces a common portable code design pattern using functors and lambda expressions in Section 6.1. Kokkos is introduced in Section 6.2, including its parallel constructs, data management objects, execution policies, and additional Kokkos tools. Section 6.3 describes the current Kokkos GPU bulk-synchronous model and explains its challenges relative to an AMT runtime. This work's modifications to Kokkos itself are presented in Section 6.4 and Section 6.5. Section 6.6 analyzes launch latency overheads. Section 6.7 describes how to efficiently execute Kokkos CUDA asynchronous reductions. Section 6.8 gives results for Poisson and RMCRT codes rewritten into portable loops and executed on various back-ends and architectures.

6.1 Portability Through Functors and Lambda Expressions

C++ functors and lambda expressions are utilized by Kokkos, RAJA [18], and Hemi [64] for portability. This section briefly reviews functors and lambda expressions and demonstrates their use in a software design pattern for portability.

A C++ functor is a struct of data members, a constructor, and an overloaded `operator()` method. All portable C++ code is placed within the `operator()` body, with the assumption the C++ code can compile for multiple back-ends, such as OpenMP for CPU execution or CUDA for GPU execution. All C++ code within the `operator()` method body yields compiled code no more or less efficient than if the same C++ code is compiled in another function. An example of a basic functor for CPU execution is given below:

```
struct Functor {
  int val;                                    // Data member
  Functor ( int val ) {                       // Constructor
    this->val = val;
  }
  void operator() ( const int i ) const {     // Portable code. Prints both
    printf( "Functor iteration %d, "          // the passed in argument and
      "data member value %d\n", i, val );     // the data member
  }
};

int main() {
  Functor myFunctor( 1 );                     // Creates the functor
  for ( int i = 0; i < 100; i++ ) {           // Invoke functor 100 times
    myFunctor ( i );
  }
```

```
  return 0;
}
```

CUDA code can only be launched through a kernel. Thus, the design pattern is modified with a wrapper function to launch the kernel. Further, the `operator()` method must be prefixed with CUDA's `__device__` so the portable code is compiled for CUDA code, or `__host__ __device__` syntax to compile the code once for CPU code and a second time for CUDA code. These changes are shown below:

```
struct Functor {
  int val;                                        // Data member
  Functor ( int val ) {                           // Constructor
    this->val = val;
  }
                                                  // CUDA annotation enabling
  __host__ __device__                             // CPU and GPU compilation
  void operator() ( const int i ) const {
    printf( "Functor iteration %d, "              // Portable code
      "data member value %d\n", i, val );
  }
};

template <typename FunctorType>
__global__
void executeFuntor( FunctorType func ) {
  func( threadIdx.x );                            // GPU code - launches functor
}

template <typename FunctorType>
void parallelForLaunch( FunctorType func,
                        int numIterations ) {
                                                      // Invoke kernel for
  executeFuntor<<< 1, numIterations >>>( func );      // 100 GPU threads
  cudaDeviceSynchronize();
}

int main() {
  int val = 1;
  int iterations = 100;
  Functor myFunctor( val );                       // Creates the functor
  parallelForLaunch( myFunctor, iterations );     // Pass functor into
                                                  // architecture launcher
  return 0;
}
```

C++ lambda expressions [93] are effectively functor shorthand, and are used to simplify the code. Below is the prior example rewritten with a lambda expression:

```
template <typename FunctorType>
__global__
void executeFuntor( FunctorType func ) {
  func( threadIdx.x );                          // GPU code - launches functor
}

template <typename FunctorType>
void parallelForLaunch( FunctorType func,
                        int numIterations ) {
                                                          // Invoke kernel for
  executeFuntor<<< 1, numIterations >>>( func );          // 100 GPU threads
  cudaDeviceSynchronize();
}

int main() {
  int val = 1;
  int iterations = 100;
  parallelForLaunch( [=] __host__ __device__               // CUDA annocation
                (const int i) {
                   printf( "Lambda iteration %d, "         // Portable code
                     "data member value %d\n", i, val );
                }, iterations );

  return 0;
}
```

The portable software design pattern is almost complete. A second `parallelForLaunch` function is needed to launch CPU tasks, and both functions require template specialization to target CPU and GPU modes. Now the design pattern closely resembles Kokkos, RAJA, and Hemi code. Uintah could adopt the above strategy for portability and avoid Kokkos entirely. However, Kokkos provides enough additional features for different back-ends, parallel reductions, portable data objects, and portable API tools, to justify using Kokkos, rather than attempting to duplicate what Kokkos has already accomplished.

6.2 Kokkos Overview

Kokkos [89] provides a C++ programming model for both portability and performance targeting CPUs, GPUs, and Intel Xeon Phi platforms. Kokkos currently supports OpenMP, Pthreads, and CUDA as backend parallel programming models and supports GCC, Intel, Clang, IBM, and PGI compilers. Three features classify the Kokkos codebase: 1) code which executes C++ functor objects, 2) portable objects which help developers access data variables in an architecture-agnostic manner, and 3) additional tools, such as random number generators, which utilize a portable API. The Kokkos code example below demonstrates the first two of these three features:

```
typedef Kokkos::View<double*[3]> view_type_2D;       // Portable 2D array

struct Demonstration {                               // A portable functor
  view_type_2D arr;                                  // Data member
  Demonstration (view_type_2D arr) : arr (arr) {}    // Constructor
  KOKKOS_INLINE_FUNCTION
  void operator() (const int i) const {
    arr(i,0) = i;                                    // Portable code
    arr(i,1) = i*i;
    arr(i,2) = i*i*i;
  }
};

view_type_2D myView ("A demonstration view", 10);    // Create 2D array
Kokkos::parallel_for( Kokkos::RangePolicy(0,100),    // Set 100 iterations
                      Demonstration (myView));       // Launch functor
```

That portable code writes values to a 2D array. Notable features are 1) a portable functor which can be compiled and executed on multiple architectures, 2) a portable `parallel` construct, with `parallel_for` used in this example (see Section 6.2.1), 3) an execution policy, *RangePolicy*, using a basic iteration pattern for 100 iterations (see Section 6.2.2), and 4) portable data containers called Kokkos Views (see Section 6.2.3). These and additional Kokkos features are given in the following subsections.

6.2.1 Kokkos Portable Parallel Constructs

Kokkos provides three classes of parallel loops, specifically `parallel_for`, `parallel_reduce`, and `parallel_scan` constructs. A `parallel_for` loop is used when simply needing to loop over some iteration range. A `parallel_reduce` loop performs the same action as a `parallel_for` with the additional feature of computing a reduction of a value (e.g. the accumulation of a single value among all iterations). A `parallel_scan` loop expresses a parallel pattern somewhat like a reduction, but instead of reducing results into a single variable, results are successively reduced and applied to all indexes of a data container.

Kokkos supports a variety of arguments to support portable execution on various execution spaces. For example, supplying the template argument `Kokkos::RangePolicy<Kokkos::OpenMP>(0,100)` would ensure the loop always executes using an OpenMP back-end. Likewise `Kokkos::RangePolicy<Kokkos::Cuda>(0,100)` ensures the loop compiles into CUDA code for GPU execution.

Shown below is the Kokkos code example rewritten using a lambda expression.

```
view_type_2D myView ("A demonstration view", 10);
Kokkos::parallel_for ( Kokkos::RangePolicy(0, 100),
                       KOKKOS_LAMBDA (const int i) {
  myView(i,0) = i;
  myView(i,1) = i*i;
  myView(i,2) = i*i*i;
});
```

To compile the above code for OpenMP support, `KOKKOS_LAMBDA` is replaced with `[&]` to enable lambda captures of other variables by value. When compiled for a CUDA environment, `KOKKOS_LAMBDA` is replaced with `[=]__host__ __device__` to support Nvidia's mechanism to compile the lambda expression for both CPU and CUDA code.

6.2.2 Kokkos Execution Policies

Three types of iteration patterns are included in Kokkos, the *RangePolicy*, *MDRangePolicy*, and *TeamPolicy* [94]. These three are summarized in Table 6.1. The *RangePolicy*, as shown in both preceding Kokkos code examples, uses straightforward 1D iteration patterns. *MDRangePolicy*, short for multidimensional range policy, supports 2D and 3D iteration patterns into (i, j) or (i, j, k) indexes. *MDRangePolicy* also supports dimensional iteration corresponding to Kokkos View memory layouts, such as row-major iteration for CPU execution spaces or column-major iteration for the CUDA environment. *TeamPolicy* supports grouping threads into teams, iterating an individual team of threads into a 1D range, as well as allowing individual teams of threads to have additional coordination mechanisms. *TeamPolicy* also supports hierarchical parallelism where a team of threads is used within inner nested loops one or two levels deep.

6.2.3 Kokkos Views

Portable data management abstractions are provided through **Kokkos View** objects. Using Kokkos Views is not a necessary component of Kokkos portable code, but they are useful for performance through 1) better direct access to simulation variables by avoiding function calls and indirection, and 2) enabling architecture aware looping patterns. Kokkos Views support containers in up to 8 dimensions and also support row-major alignment, column-major alignment, or various tiled approaches. Code which accesses values within a Kokkos View object does not need to change even if the underlying alignment does change.

Kokkos Views can be managed or unmanaged, and are called *Kokkos Managed Views* and *Kokkos Unmanaged Views* A Kokkos Managed View manages its allocation, contains a reference counter, and will deallocate when the reference counter becomes zero. A Kokkos Unmanaged View is essentially a wrapper over an existing value or array, and the application developer is responsible for all allocation and deallocation of the underlying data. The Uintah data stores use Unmanaged Views, as Uintah manages its own allocation and deallocation.

6.2.4 Additional Kokkos Tools

The Kokkos team provides other portable tools to aid application developers. These include support for atomic operations, a Kokkos random number library based on Marsaglia's xorshift generators [95], a `Kokkos::DualView` class to assist developers in transferring data between host and GPU memory, and `Kokkos::Vector` and `Kokkos::UnorderedMap` classes for portable versions of the popular `std::vector` and `std::unordered_map` containers.

The example code below demonstrates creating a Kokkos View object to hold a 3D integer array, creating a random number generator pool, and filling that view with random integers between 0 and 100. This code is portable for both CPUs and GPUs and compiles for the target back-end depending on prior Kokkos build configuration.

```
Kokkos::View< int*** > rand_nums_arr( "RandNums", 64, 64, 64 );
Kokkos::Random_XorShift64_Pool<> rand_pool64( 1791619 );
Kokkos::fill_random( rand_nums_arr, rand_pool64, 100 );
```

6.3 Limitations of Kokkos's Synchronous GPU Execution Model

This section describes three limitations of Kokkos's CUDA abstraction. 1) Kokkos invokes synchronization barriers for CUDA actions, which limits simultaneous processing of these actions (Section 6.3.1). 2) Bulk-synchronous execution limits how the application developer partitions problems (Section 6.3.2). 3) Kokkos's API limits Uintah's ability to change CUDA block partition parameters (Section 6.3.3).

6.3.1 Bulk-Synchronous Execution

Kokkos makes use of its internal `Kokkos::fence()` calls after it invokes a CUDA kernel, and `Kokkos::fence()` in turn invokes CUDA's `cudaDeviceSynchronize()`, which "waits

until all preceding commands in all streams of all host threads have completed." [96]. This pattern ultimately translates to a bulk-synchronous parallel model, which typically results in wall time performance loss [97].

Figure 6.1 depicts this problem. The first group of kernels, those in the default stream, are Kokkos `parallel_for` loops. The second group of kernels, those operating in non-default streams, contain the same code placed in regular CUDA kernels. Both sets use simple arithmetic code designed execute for hundreds of milliseconds per kernel. Both sets are launched on 256 threads in one CUDA block. Streams are an effective mechanism to avoid problems associated with bulk-synchronous execution [98].

Similarly, Kokkos has the same issue with data copies in and out of the GPU memory. All Kokkos GPU copies are synchronous, and so no kernels can execute while copies occur, even if the data is unrelated to the executing kernel. Streams again would be an effective mechanism to avoid memory copy problems.

6.3.2 CUDA Kernel Partitioning Into Blocks

The second challenge with bulk-synchronous GPU execution is ensuring a loop is partitioned into enough CUDA blocks to use all GPU SMs. Each CUDA kernel can be partitioned between one to many blocks, with each block having its own group of CUDA threads. Blocks are then distributed among a GPU's SMs.

Block sizing and placement is a strategic decision. As shown in Figure 6.2, a single CUDA block resides within an SM, and a block cannot span multiple SMs. A kernel can utilize one block within one SM, or several blocks within one SM (provided there are enough registers for each block), or multiple blocks spanning multiple SMs. A kernel can have more blocks than GPU SMs. The GPU will process these blocks as work units, and when a block completes, another block's data is loaded into that SM to continue execution.

Some bulk-synchronous computations can efficiently use the GPU, provided the problem size is large enough. The user can 1) split a kernel into many blocks, (Section 6.3.2.1) or 2) split a kernel into blocks exactly equal to the number of SMs (Section 6.3.2.2). If too few blocks are used, then not all SMs are utilized, as demonstrated in Figure 6.3. Likewise, if blocks slightly exceed the number of SMs, GPU starvation will likely occur.

6.3.2.1 Partitioning Many Blocks

Suppose a kernel is split into many blocks far greater than the GPU's number of SMs, then because the GPU places blocks into SMs as SMs becomes available, the GPU is fully occupied with work for most of the computation. This approach typically requires a parallel loop contain at least tens of thousands of iterations. Upcoming results in Section 6.8 and Section 7.8 demonstrate and analyze minimum task sizes and task work desired to hide latency costs.

6.3.2.2 Partitioning Blocks to Match SMs

An alternative approach is splitting a kernel into enough blocks exactly equal to the number of SMs on the GPU. While effective, this alternative approach requires architecture specific knowledge and may not work across a range of GPUs, such as the Nvidia K20X GPU (used on the DOE Titan supercomputer) which has 14 SMs [99], or the Nvidia P100 which uses 56 SMs [100], or the Nvidia V100 GPU (used in the DOE Summit and DOE Sierra supercomputers) which has 84 SMs [101].

6.3.3 Kokkos's GPU Block Partitioning

Kokkos block partitioning presents a problem for Uintah. Each Kokkos Execution Policy (Section 6.2.2) has its own block partitioning logic. The *RangePolicy* and *MDRangePolicy* take the loop's iteration range and assigns a CUDA block to each group of 256 iterations. For example, suppose a parallel loop using a *RangePolicy* has an iteration range of [0, 2000). Here Kokkos will partition that into eight blocks. The *TeamPolicy* gives the application developer direct access to CUDA block partitioning parameters through a nested loop design pattern. The outer loop corresponds to the number of CUDA blocks, and the inner loop corresponds to the number of threads per block.

Uintah desires both efficient 3D data layout iteration strategies and also direct control of CUDA block partitioning. However, the *MDRangePolicy* and *TeamPolicy* each provide only one of these two needed features. Kokkos has no execution policy for proper 3D data layout iteration with custom block partitioning. This problem is solved in (Section 7.5.4.4).

6.4 Modifying Kokkos for GPU Asynchrony

Kokkos executes functor objects (or lambda closures which effectively are compiled into functor objects) on the GPU. The process used to execute functor objects is to first launch a CUDA kernel, and within the kernel code, invoke the functor object (see Section 6.1). Kokkos executes CUDA capable functors objects using two execution mechanisms: 1) passing in the functor object as a CUDA kernel argument and relying on the GPU to automatically copy it into the constant chance memory, 2) manually placing the functor object in the constant cache memory with a CUDA memory copy API invocation, then running that functor within a kernel.

In both execution mechanisms, the functor object will reside in a GPUs constant cache memory. This memory space is beneficial as it acts like read-only registers. Functor data is fetched in a single clock cycle when all executing threads in a warp of 32 threads request the same constant cache memory address. In this manner, a GPU's other fast memory spaces, such as registers, are left free for normal code execution.

The first execution mechanism, passing the functor object as a CUDA kernel object, has a limitation of particularly large functor objects, as CUDA kernel arguments "are passed to the device via constant memory and are limited to 4 KB." [96] Thus the second mechanism can work with functor objects that are much larger than 4 KB. Most functor objects observed both by Uintah developers and Kokkos developers have observed sizes ranging in the low hundreds of bytes. The largest functor encountered by Uintah is roughly 3.5KB, but this could easily grow if more array data is needed for that particular functor. This work provides an asynchronous solution for functors smaller and larger than 4KB.

Functor object size determines whether to allow CUDA to automatically copy the functor into the constant cache or to manually copy the functor. Based on prior extensive benchmarking, the Kokkos development team determined that functor objects less than 512 bytes are best executed as kernel parameters, while larger functor objects should utilize a manual copy into constant cache memory [102]. Sections 6.4.1 through 6.7 provide this work's modifications to Kokkos for asynchronous GPU support. These sections cover asynchronously launching functor objects passed in as an argument (Section 6.4.1), asynchronously launching functors objects manually copied into the constant cache memory (Section 6.4.2), allowing Kokkos to either manage CUDA streams or have the application

developer manage CUDA streams (Section 6.5.1), additional work needed for Kokkos deep copy mechanisms (Section 6.5.2), and additional work needed for `parallel_reduce` loops (Section 6.7).

6.4.1 Asynchronous Functors Using Kernel Parameters

Extending Kokkos for asynchronous execution when functors are passed in as a parameter requires only minor Kokkos modifications. A CUDA stream must be attached to the CUDA kernel that invokes the functor. The Kokkos code which launches the functor object is shown below:

```
// Invoke the driver function on the device
  cuda_parallel_launch_local_memory
    < DriverType , /* ... */ >
    <<< grid , block , shmem , stream >>>( driver );
```

The `driver` is a Kokkos object which encapsulates a functor object. The CUDA kernel launch parameters are shown between the triple chevron syntax. For asynchrony, a non-default CUDA stream is now supplied within those CUDA kernel parameters. This work also modified Kokkos to not invoke its internal `Kokkos::fence()` call if the non-default stream is used.

The application developer is responsible for either invoking a CUDA execution barrier or checking the stream for completion. This is further detailed in Section 6.5.

6.4.2 Asynchronous Functors Using Constant Cache Memory

A new approach is needed to manually copy functors into constant cache memory and do so asynchronously [16]. Currently, Kokkos utilizes the constant cache memory by 1) copying the functor into CUDA constant cache memory, instead of regular global memory, 2) invoking a launching kernel, then 3) assuming the functor is always found at the 0th byte in the constant cache memory and invoking that functor. Figure 6.4 depicts this process. A single functor generates two synchronization points, one for the copy, and another to wait for the functor object's execution completion so a subsequent functor object can be placed into the constant cache memory.

Figure 6.5 depicts this work's new Kokkos asynchrony implementation. The Nvidia GPU's constant cache space is treated as a shared pool usable by many concurrent host threads, and CUDA events must be employed to track the completion of functor object

execution. The constant cache memory capacity of all CUDA capable Nvidia GPUs is 64 KB. Of that, Kokkos sets aside 32 KB to store functor objects. Within Kokkos a bitset is now used (see Figure 6.6) to represent blocks of that 32 KB of the constant cache memory. When a CPU thread initiates a Kokkos asynchronous parallel loop, Kokkos can determine the size of the functor object, and then determines how many data blocks that functor requires. Kokkos then attempts to locate and atomically claim a contiguous bitset region which corresponds to constant cache memory that can fit the functor object. Upon atomically claiming a region, the functor object is asynchronously copied into the constant cache memory, and the functor is executed after supplying the correct constant cache memory offset.

An additional array of structs, also shown in Figure 6.6 associates the bits to the associated CUDA stream or event. When a stream or event indicates a functor has completed execution, the struct item uses its bitset mask to unset the set bits, enabling reuse of that region of GPU constant cache memory.

As an example, suppose a functor object requires 4640 bytes, and the constant cache memory is split into 512-byte data blocks. This functor requires 10 data blocks, and Kokkos will atomically attempt to set 11 contiguous unclaimed bits in the bitset. When more than one functor is considered, then each will utilize its regions of the bitset. Suppose a maximum of 64 blocks are used, then in the prior example of a 4640-byte functor requiring 11 blocks each, then Kokkos could concurrently execute a maximum of 64 / 10 = 6 kernels. While this does limit concurrent kernel usage, this approach allows for functor objects larger than 4KB to be executed, which would otherwise be unable to execute at all through the method outlined in Section 6.4.1. For an analysis of how to distribute work throughout a GPU with limited kernels, see Section 6.4.4.

6.4.3 Recognizing Completion of Executed Functors

Kokkos must know when an asynchronous functor completes so Kokkos can mark that region of constant cache memory as reusable. CUDA does not provide any native API designed explicitly to send an asynchronous signal or set a flag to indicate when a kernel completes. This section describes three approaches to identify kernel completion and which one was adopted for Kokkos.

The first approach to identify kernel completion is by limiting all kernels to their unique stream, and then using native CUDA API such as `cudaStreamQuery` or `cudaStreamSynchronize` to check for successful completion of the kernel assigned to that stream. This approach is easier to implement within Kokkos, but it offloads additional work to the application developer by requiring end-user management of every CUDA stream or every Kokkos object which internally manages a stream. Kokkos design philosophy avoids burdening end users with additional requirements, and so this first approach was not adopted.

The second approach attempted to use CUDA callback functions. As described in the CUDA C Programming Guide [96] "The runtime provides a way to insert a callback at any point into a stream via `cudaStreamAddCallback()`. A callback is a function that is executed on the host once all commands issued to the stream before the callback have completed." When this approach was attempted in Kokkos to manage functor object completion, a major drawback not mentioned in the CUDA C Programming Cuda was observed. When callback functions were used among many asynchronously queued streams, significant synchronization occurred among CUDA kernels which increased wall time by several factors.

The adopted approach utilizes CUDA events. CUDA allows these lightweight event objects to be asynchronously inserted and interspersed in a stream alongside other CUDA actions such as kernels or memory copies, and the event is updated when the stream arrives at that event. While CUDA events are often portrayed as a tool for wall timing, they can also be used to track a GPU's progress. CUDA events are not perfectly efficient at identifying kernel completion, as the event is a wholly separate item in a stream apart from the preceding CUDA actions. CUDA events also do not actively signal a CPU host thread or launch a callback function when an event is triggered.

In the context of Kokkos, CUDA events provide a useful tool to allow an application developer to asynchronously determine if a Kokkos loop has completed, regardless of whether the application developer places loops on the same stream or different streams. Kokkos itself was modified so that immediately after a kernel containing a functor is invoked in a stream, a CUDA event is placed in the same stream (see Figure 6.5), and Kokkos internally associates that functor with that stream.

Kokkos does not have a runtime thread to actively monitor and test for event completion. Instead, Kokkos only is concerned with reusing the bitset slots to store and track upcoming functors. Kokkos does not search for any events for completion unless the constant cache memory is full with no room for an additional functor. At this point, Kokkos will search through all events and atomically unset bits associated with completed functor objects.

6.4.4 Analysis of Functors in Constant Cache Memory

The sections below provide analysis for three questions. 1) What kinds of unused constant cache space limitations occur by using a bitset (Section 6.4.4.1). 2) What limitations occur by limiting the number of tracked functors (Section 6.4.4.2). 3) What limitations exist when functor object size is large, such as 4KB or larger (Section 6.4.4.3). This analysis is done in the context of varying GPU characteristics, such as those found in Table 6.2. These GPU characteristics affect decisions of how to utilize functors in constant cache memory.

This analysis demonstrates that for most Kokkos loop scenarios, all SMs of a GPU can be kept occupied with CUDA blocks. For other scenarios, especially those with finer-grained tasks operating on many data variables, an application developer using Kokkos should be aware of new limitations.

6.4.4.1 Unused Constant Cache Space

One limitation of manual functor copies is unused constant cache memory. Each GPU in Table 6.2 shows hat every GPU has 64KB of constant cache memory and Kokkos splits that into 32 KB regions, one for automatic parameter data and another for Kokkos's manual functor copies. This work added a 64-bit integer to represent chunks of constant cache memory storing functor objects, with each bit representing a 512-byte chunk.

Functor objects that are not exactly a multiple of 512 bytes will waste space. This problem is mitigated through two mechanisms. 1) Use a larger bitset to track smaller chunks of constant cache memory. This approach may be slower due to atomic operations on bitsets larger than 64 bits. 2) Set a higher threshold for functors to be managed by the manual copy method. For example, suppose only functors larger than 4 KB are manually copied rather than the current 512 byte threshold. The worst case here is a functor at 4097 bytes, which uses nine chunks of 512 bytes, or 4097 / (9 * 512) = 89% used space and 11%

unused space.

6.4.4.2 Limiting Number of Functors

A 64-bit bitset to track functors means that it can track at most 64 functors. For the Pascal or Kepler GPUs in Table 6.2, this is short of the 128 simultaneous possible kernels.

A solution to both preceding limitations is ensuring these functor objects span multiple CUDA blocks. For example, if only 64 kernels can be executed at a given time, then by ensuring each kernel can run 2 or more blocks per kernel, all SMs of all the GPUs listed in Table 6.2 can be filled with work. Results in Section 7.8 demonstrate that utilizing between 4 to 16 blocks per kernel provides better wall timings over one block per kernel.

6.4.4.3 Limitations Due to Large Functor Objects

This work allows execution of functor objects greater than 4 KB on the GPU, which is larger than what is allowed under CUDA's automatic copy approach. These larger functor objects increase the lower limit of task granularity, even more than in the prior section. The application developer will need to ensure the functor object uses many CUDA blocks to fully utilize the GPU.

The Volta V100 GPU list in Table 6.2 has the most SMs, at 84. Suppose a functor object requires 4.1KB, just past the 4 KB threshold. Further, suppose that Kokkos is configured to manually manage the constant cache memory for functor objects larger than 4KB and that Kokkos set aside 32 KB of the constant cache memory for this purpose. Here, only 32 KB / 4.1 KB = 7 functor objects at a time could concurrently execute. With 84 SMs, then each functor object would need to occupy 12 blocks to at least provide one block per SM. Suppose the kernel uses 256 threads per block, then the kernel requires 12 blocks * 256 threads per block = 3072.

Relating this to Uintah's task sizes, a task should not operate on a patch of fewer than 3072 cells. This limit suggests that the smallest patch size for large functors is $15x15x15 = 3375$ cells, which fits with Uintah minimum task sizes described by Meng [83].

6.5 Kokkos API Modifications

This work made several smaller modifications to other parts of Kokkos. These modifications include instance object API to track streams (Section 6.5.1), Kokkos managed

streams and unmanaged streams (Section 6.5.1.1 and Section 6.5.1.2), and minor Kokkos deep copy modifications (Section 6.5.2). These modifications only touched Kokkos::Cuda implementations, and were not carried elsewhere into other Kokkos parallel back-ends.

6.5.1 Kokkos Instance Objects and Streams

This work allows an application developer using Kokkos to supply existing CUDA stream (unmanaged streams) or let Kokkos create and destroy streams as needed (managed streams). This approach is analogous to a Kokkos Managed View and Unmanaged View (Section 6.2.3), where either Kokkos or the user manages all allocation, reference counting, and deallocation.

Before this work, a user would invoke a Kokkos loop by creating a Kokkos execution policy (*RangePolicy*, *MDRangePolicy*, or *TeamPolicy*, as described in Table 6.1), and then pass the policy object into a `parallel_for`, `parallel_reduce`, or `parallel_scan`:

```
Kokkos::RangePolicy myRange(0, 100);
Kokkos::parallel_for(myRange, DemonstrationFunctor());
```

Alternatively, a user could create a specific `Kokkos::Cuda` instance object and pass that object into a policy object:

```
Kokkos::Cuda myInstance("Some instance description");
Kokkos::RangePolicy myRange(myInstance 0, 100);
Kokkos::parallel_for(myRange, DemonstrationFunctor());
```

The Kokkos team recommended the `Kokkos::Cuda` instance object encapsulate a CUDA stream. The user would be responsible for triggering whether Kokkos should manage a CUDA stream, or the user should manage a stream. These two options are given below.

6.5.1.1 Managed Streams

A managed CUDA stream is created according to specific constructor arguments. Internally Kokkos now creates a CUDA stream and provides reference counting on that instance object. Thus if a `Kokkos::Cuda` instance object is copied, both objects share the same stream, and the stream will not be reclaimed until both run the object's destructor code. Overall, the application developer doesn't need to provide any explicit CUDA code, and thus retains Kokkos's portability theme.

The application developer still requires interaction with asynchrony to know when a parallel loop has completed. An undesirable option is to employ `Kokkos::fence()`, which

in turn signals `cudaDeviceSynchronize()` to create a synchronization point. A second unimplemented option was an instance object `getCudaStream()` method and letting the user manage asynchronous status directly through Kokkos API.

The implemented asynchrony interaction is an instance object `getStatus()` method that returns a new Kokkos C++ enum. The `getStatus()` performs the CUDA stream queries, then returns whether items on the stream have completed, not completed, or errored. In this manner, the application developer can utilize full asynchrony without interacting with any CUDA API.

6.5.1.2 Unmanaged Streams

To utilize unmanaged streams, an application developer simply passes the stream into a `Kokkos::Cuda` instance object on construction. An example is shown below:

```
cudaStream_t *myStream = new cudaStream_t;
Kokkos::Cuda instanceObject(myStream);
Kokkos::RangePolicy myRange(instanceObject 0, 100);
Kokkos::parallel_for(myRange, DemonstrationFunctor());
```

Because the user manages the stream allocation and deallocation, the stream object should persist beyond the local block of code. The application developer can now use the CUDA stream directly to query the stream's status. Uintah itself has adopted this route to seamlessly fit into its task scheduler (Chapter 5).

6.5.2 Kokkos Deep Copies

Kokkos employs a `deep_copy()` API to transfer data from one memory space into another, such as from host memory into GPU memory. The Kokkos team previously implemented an overloaded method of `deep_copy()` which accepts a `Kokkos::Cuda` instance object as an argument, and correctly invoked an asynchronous memory copy. The instance object work for streams (Section 6.5.1.1) and (Section 6.5.1.2) now allows the application developer to asynchronously utilize `deep_copy()`.

Internally, no other Kokkos code utilized this asynchronous overloaded method. Instead, Kokkos used the `deep_copy()` method that immediately synchronized with `cudaDeviceSynchronize()`. This work made the necessary changes throughout Kokkos's CUDA back-end implementation to use the appropriate `deep_copy()` method depending on whether a non-default stream was utilized.

6.6 Observed Overhead

These Kokkos modifications were tested and profiled to measure two overhead items: 1) launch latency when invoking kernels through different approaches, and 2) any difference in execution times between native CUDA code and Kokkos `parallel_for` code.

6.6.1 Initialization Latency

This section measures initialization latency and compares functor objects copied automatically using CUDA through a kernel parameter with those copied manually through a CUDA API host-to-GPU copy call. First, two sets of codes were created, a non-Kokkos CUDA kernel which performs a single instruction per thread, and a Kokkos `parallel_for` loop performing the same single instruction.

The parameter size was varied to measure automatic and explicit parameter copies into GPU constant cache memory. The Kokkos `parallel_for` loop used 256 iterations. Block partitioning was kept consistent throughout. Both codes were repeated 64 times to diminish any potential warm-up initialization found on the first few runs. Finally, Kokkos itself was modified for this test to either always copy functor objects or never copy functor objects. Both codes were first executed on the default stream so that no loops would overlap, and then executed and interleaved among eight streams for concurrent kernel execution. All tests were performed on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 8.0 and on an Intel Xeon E5-2660 CPU.

The launch latency overhead times are shown in Table 6.3. All CUDA kernel invocations occur some latency, with the non-Kokkos kernels ranging between 11.17 and 14.33 μs of added time. Using functors with Kokkos adds additional latency over non-Kokkos runs, ranging between 8.84 and 22.65 μs. This work's manual functor copy mechanism has an additional 10.77 to 15.7 μs on top of the automatic functor copy.

Uintah executes tasks on many streams, and their associated parameter sizes are usually less than 1 KB. Therefore, the 20.75 μs should be treated as the expected latency overhead. This latency also implies a lower limit on task granularity. Previous work by Meng [83] recommended an 8^3 patch minimum task size. This work likewise does not recommend smaller task granularity. For example, Table 5.3's current GPU runtime performance time for a 256^3 patch problem has a throughput of 14.84 cells per μs. An 8^3

patch would be expected to compute in 34.8 μs with a 20.75 μs launch overhead, or in other words, 37% of this Kokkos loop's time would be spent in launch overhead.

6.6.2 Effect on Loops With Many Instructions

Next, an additional test was created where the kernel times were varied, and enough work provided to keep the kernels busy between 3-25 ms. The purpose of this test is to simulate the kind of short-lived tasks typically found in AMT runtimes.

The test code computed simple addition operations on values in global memory, and both codes were written using identical logic. Each pair of codes was executed multiple times on eight streams, with the number of iterations varying each invocation. In total three versions of the code were executed: 1) a non-Kokkos CUDA kernel, 2) a Kokkos `parallel_for` using automated functor object copies, and 3) a Kokkos `parallel_for` using manual functor object copies All tests were performed on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 8.0 and on an Intel Xeon E5-2660 CPU.

Figure 6.7 shows an Nvidia Visual Profiler output of a set of `parallel_for` loops with functor objects automatically copied into the constant cache memory, and another set with functor manually copied. These two approaches do an excellent job overlapping computations and avoiding synchronization. Overall wall times indicate the automatic copy memory functors approach was only 0.1% to 1% slower than using native CUDA code. The manual copies into constant cache memory functors were roughly 2% slower, which is somewhat surprising as the prior initialization latency analysis suggested the slowdowns would have been closer to 0.2% The average measured additional slowdown per `parallel_for` loop in the manual functor object copies approach is roughly 0.22 milliseconds compared to native CUDA code.

6.7 Parallel_Reduce Asynchrony

RMCRT utilizes `Kokkos::parallel_reduce`, and this section describes work to make the corresponding GPU execution of this parallel pattern asynchronous. Extending the `Kokkos::parallel_for` modifications to `Kokkos::parallel_reduce` required addressing the challenge of copying a reduction result in GPU memory to host memory while avoiding synchronization of other kernels currently executing or queued for execution. Further,

the Kokkos API was modified to allow testing if a reduction value is ready without utilizing CUDA synchronization. When implementing these modifications, care was taken to allow future work where a reduction value from a prior loop is used as input into an upcoming loop.

Most of the prior Kokkos `parallel_for` modifications were carried forward into `parallel_reduce` logic. Kokkos was further modified to asynchronously copy the reduction value from device-to-host if a stream was used to invoke the reduction kernel. Using `cudaMemcpyAsync()` to copy this data may incur a synchronization point. The CUDA documentation explains that even though the function call contains the `Async` suffix, "this [Async] is a misnomer as each function may exhibit synchronous or asynchronous behavior depending on the arguments passed to the function." [96] The documentation further states that when using asynchronous copies to transfer "from device memory to pageable host memory, the function will return only once the copy has completed" [96]. For this reason, the buffer used to receive the reduction value should be stored in pinned host memory.

6.7.1 Parallel_Reduce Results

Test code similar to that used in Section 6.6 was used to measure many quickly executing reduction loops operating on eight streams. Figure 6.8 shows the synchronization incurred by using non-pinned memory for the reduction value. Figure 6.9 demonstrates the desired overlapping by using pinned memory. Detailed timing indicated minimal differences in using either manual or automatic copies of functor objects into GPU constant cache memory, with the latter computing these test problems roughly 2% slower and requiring an additional overhead of 0.21 milliseconds per kernel compared to the automatic copy approach.

To determine if the reduction value is ready, the `Kokkos::Cuda` instance object described in Section 6.5 now offers a `getStatus()` method which indicates if all operations on the stream are ongoing or complete. In future work, the reduction value can be encapsulated inside a specialized Kokkos View object capable of receiving future data. This approach would allow Kokkos to recognize whether the reduction value can stay resident in GPU data if necessary, thus better facilitating nested loops of reductions.

6.8 Results

The Poisson and RMCRT codes were rewritten for Kokkos CUDA portable lopps. Section 6.8.1 gives Poisson results and analyzes how to hide launch latency. Section 6.8.2 gives RMCRT results.

6.8.1 Poisson

The purpose of this section is understanding overheads, latencies, and performance when maximizing all CPU or GPU hardware cores on various back-ends. Table 6.4 shows the Poisson problem (Section 1.6.1) running on a 192^3 cell domain, decomposed into tasks operating on patches of different sizes. Table 6.4 measures how frequently a task's parallel codes complete. All times were computed by obtaining the difference of two timestamps, the first being the timestamp before the first loop, and the second being the timestamp after the last loop.

CPU-based results were gathered on an Intel Xeon E5-2660 CPU @ 2.2 GHz with two sockets, eight physical cores per socket, and two hyperthreads per core. 16 threads were used for the column labeled CPU Pthreads. GPU-based results were gathered on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 8.0.

The CPU Pthreads test ensures there are at least as many patches as threads available, so no thread is starved. The column labeled "GPU Raw CUDA" used asynchronously executing kernels on different streams. The Kokkos OpenMP column computed one task at a time in all available CPU cores. Kokkos CUDA original codes all execute parallel loops synchronously. The *RangePolicy* highlights Kokkos's approach of subdividing problems into blocks and allowing the GPU to distribute those blocks. The *TeamPolicy* highlights manually subdividing problems into a fixed number of blocks, with one test using 24 blocks to correspond with this GPU's 24 SMs. The work from this chapter is shown in the last column, using 16 blocks to encourage kernel overlap.

The main takeaway from Table 6.4 is that this work's modifications to Kokkos allow configurations that are better than the original Kokkos CUDA implementation, with speedups between 1.54x to 2.54x observed. (The exception being the 192^3 patch case, which does not benefit from asynchrony as it only has one task.) Table 6.4 also demonstrates the application developer now has new flexibility by not having to precisely fine-tune the

number of CUDA blocks to match the number of GPU streaming multiprocessor. The application developer can instead use a smaller block number and let many asynchronously executing loops fill the GPU. (Table 6.4 does not show running this work's Kokkos CUDA implementation against the Kokkos *RangePolicy* because the *RangePolicy* acts somewhat like a synchronous execution model by filling the entire GPU with CUDA blocks.)

This work's CUDA implementation is faster for all patches compared to the Kokkos OpenMP implementation for this machine. The original Kokkos CUDA implementation is slower compared to the Kokkos OpenMP implementation for patch sizes between 16^3 through 48^3 cells.

The non-Kokkos GPU code performs faster than this work's Kokkos CUDA implementation for patch sizes between 16^3 cells through 64^3 cells. Analysis indicates two reasons for this performance. First, the raw CUDA implementation uses only one kernel, while the Kokkos implementation has one loop to initialize a data variable and a second loop for the computation. The Kokkos implementation incurs more launch latency. Second, Section 6.6.1 demonstrated that loops launched through Kokkos compatible functors have larger latency compared to normal CUDA kernel invocations. Note that later Nvidia GPU architectures have reduced kernel launch times, and this should mean that Kokkos CUDA and raw CUDA results shouldn't be as disparate between 16^3 cells through 64^3 cells.

Of the three columns for Kokkos CUDA original tests, the *TeamPolicy* approach using all SMs on the GPU performs best. When fewer blocks (16) than SMs (24 on this GPU) are used, some cores are starved. When the *RangePolicy* is used, performance is much worse, which is likely due to Kokkos oversupplying CUDA blocks to the kernel, which in turn inefficiently uses CUDA registers. The non-Kokkos CPU code performs faster than the Kokkos OpenMP implementation for patch sizes between 16^3 cells through 32^3 cells, and then the Kokkos OpenMP is more performant. The faster performance of non-Kokkos CPU code for smaller patch sizes is likely due to the OpenMP static scheduling employed, where each thread has a fixed amount of work, but some threads are delayed compared to others. The faster performance for Kokkos OpenMP for larger patch sizes is likely due to Kokkos having more efficient (i, j, k) access to its variables compared to Uintah's approach.

6.8.1.1 Latency Hiding Analysis

This analysis describes what is required to hide 90% of the launch latency for *synchronous* execution of this memory bounded problem. Starting with Table 6.3, assume each loop has 20.75 μs of launch latency with multiple CUDA streams. This GPU has a combined read/write bandwidth of 336.5 GB/s [103], and this Poisson problem is read bandwidth bounded. This launch latency represents a loss of $(336.5/2) * 20.75\ \mu\text{s} = 3.75$ MB of data could have been read in that time frame. To hide 90% of the launch latency, a task requires reading 37.5 MB of data. Suppose the Poisson problem reads roughly five float value reads per cell (two of cells in the stencil computation will likely be found in cache). Thus, 37.5 MB $/(5$ float values $*$ 4 bytes per float$) = 1,875,500$ cells, which corresponds to roughly a 124^3 patch.

Launch latency can be avoided in *asynchronous* execution. A GPU can maximize its bandwidth without having all cores performing work. Thus, supposing a scenario where the GPU sets aside some cores for initializing kernels, other cores can still maximize the bandwidth so long as they were previously computing work.

Compute bounded problems can hide launch latency in *synchronous* execution with more flops. This GPU can compute 6.144 gigaflops per second [103], and spending only 10% of the time in launch latency requires a task have $6.144/20.75\ \mu\text{s} * 10 = 1,369,000$ flops. Compute latency can be avoided in *asynchronous* execution by utilizing kernels small enough such that at least two kernels or CUDA blocks can reside within a SM [90].

6.8.2 RMCRT

Both the RMCRT single-layer and multilayer task codes (Section 1.6.2) were ported into Kokkos compatible functors as part of a joint effort with John Holmen. Table 6.5 and Table 6.6 presents results [16] comparing mean times per timestep for two problem sizes (64^3 cells and 128^3 cells). All three RMCRT code implementations (RMCRT:CPU, RMCRT:GPU, RMCRT:Kokkos) were used for the single-level approach and multilevel approach. Codes were executed single-nodes using CPUs, GPUs, and Intel Xeon Phis.

The absorption coefficient was initialized according to the benchmark [5] with a uniform temperature field and 100 rays per cell used to compute the radiative-flux divergence for each cell. Tasks are each assigned to compute on patch size of 16^3 cells. CPU-based re-

sults were gathered on an Intel Xeon E5-2660 CPU @ 2.2 GHz with two sockets, eight physical cores per socket, and two hyperthreads per core. For RMCRT:CPU, Uintah's scheduler utilized 32 threads. GPU-based results were gathered on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 7.5. Xeon Phi KNL results were gathered by John Holmen on an Intel Xeon Phi 7230 Knights Landing processor @ 1.30 GHz with 64 physical cores and four hyperthreads per core. Xeon Phi KNL simulations were launched using 1 MPI process and 256 OpenMP threads with the *OMP_PLACES=threads* and *OMP_PROC_BIND=spread* affinity settings.

The results are very promising. RMCRT loop code was successfully consolidated into one codebase. GPU codes now run significantly faster due to this work's Kokkos modifications enabling asynchrony. Further, rewriting code into Kokkos compatible portable code required the code be simpler (such as fewer API calls or indirection), and this led to Kokkos RCMRT codes performing faster than the non-Kokkos equivalent. These changes are summarized in the next chapter in Section 7.7.

6.9 Kokkos Modification Summary

Uintah can now successfully use Kokkos to write portable task loops and efficiently execute those loops on various architectures, including GPUs. This work required numerous Kokkos modifications to enable asynchronous execution of parallel loop code without any CUDA synchronization barrier. These modifications were demonstrated within the Uintah AMT runtime on a lightweight problem and a complex multiphysics problem.

Unfortunately, the combined modifications to Uintah and Kokkos up to this point do not satisfy the thesis goal (Section 1.2). Specifically, the thesis goal desires an application developer using an AMT runtime to "write portable task code capable of compilation and execution on both CPUs and GPUs as *easily* as they currently do for CPU-only task code." The process still is not easy. While Kokkos allows portable loops, the surrounding task code before and after the loops contained significant nonportable branching statements for both the preprocessor and C++ code itself. The next chapter completes the thesis goal by enabling fully portable tasks that can be *easily* written.

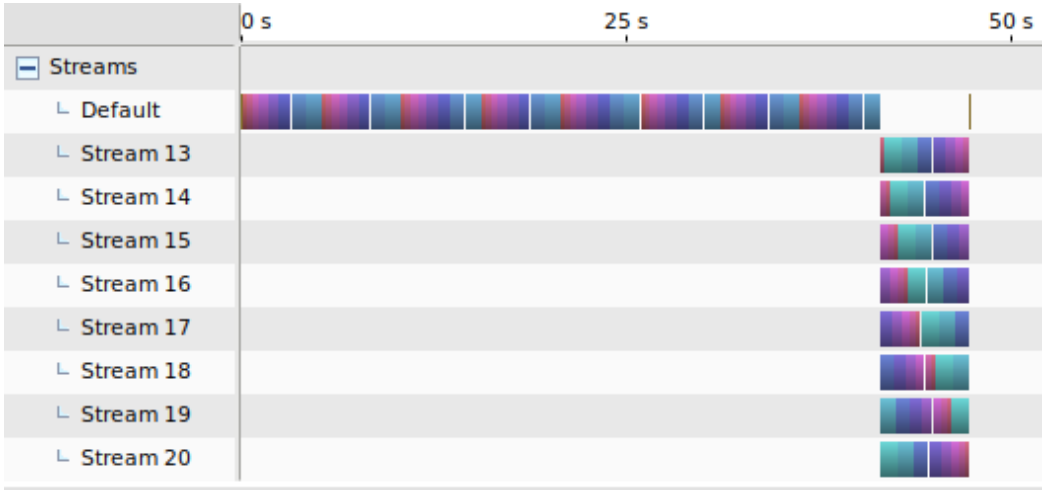

Figure 6.1: A profiled application run using the Nvidia Visual Profiler demonstrating the need to run CUDA code on multiple streams. Each colored rectangle represents the execution of a single CUDA kernel, either executed through a Kokkos `parallel_for` or through native CUDA code.

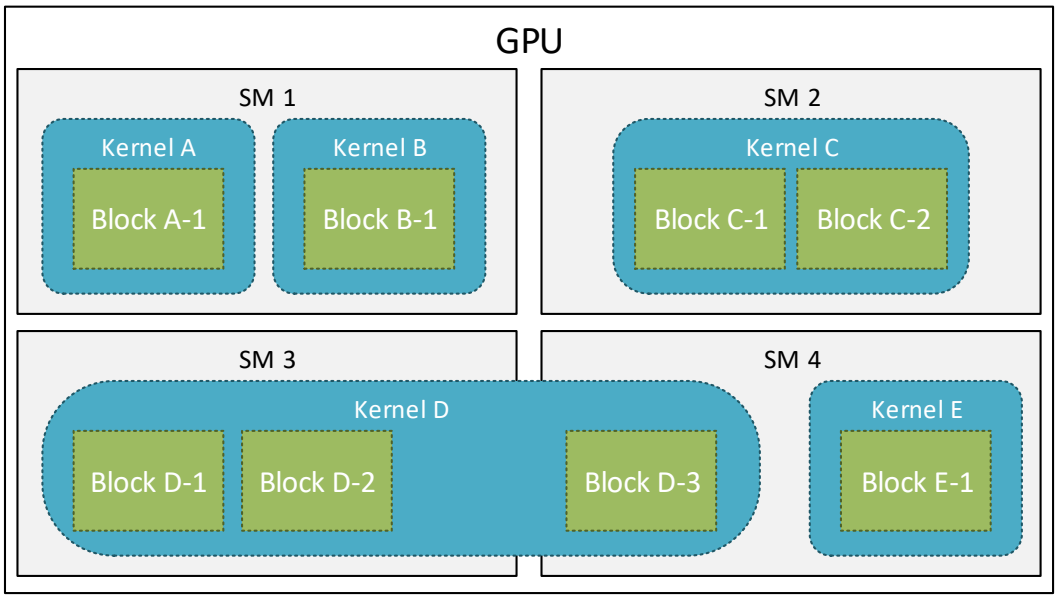

Figure 6.2: A GPU will distribute blocks throughout its streaming multiprocessors (SMs). Blocks cannot span multiple SMs, but kernels can have multiple blocks and also span multiple SMs.

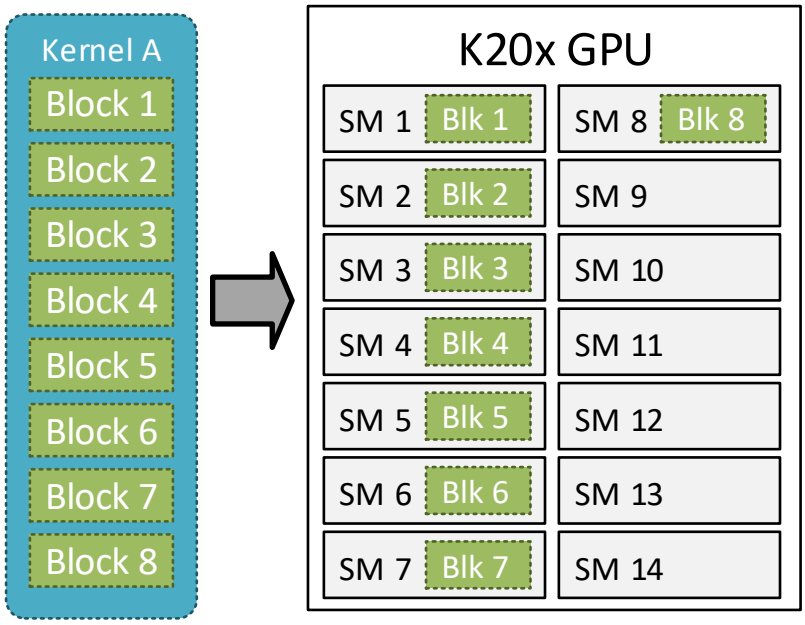

Figure 6.3: Kernels themselves are not equally distributed throughout a GPU. Instead, blocks are distributed to SMs according to available SM resources. In the above example, an Nvidia K20x GPU (which is used on the Titan supercomputer) must leave some SMs idle to process a kernel with 8 blocks. If synchronization is used, no other blocks from other kernels can be placed into SMs until all 8 blocks finish execution. The Summit supercomputer's GPUs have 84 SMs, further exasperating the synchronization problem.

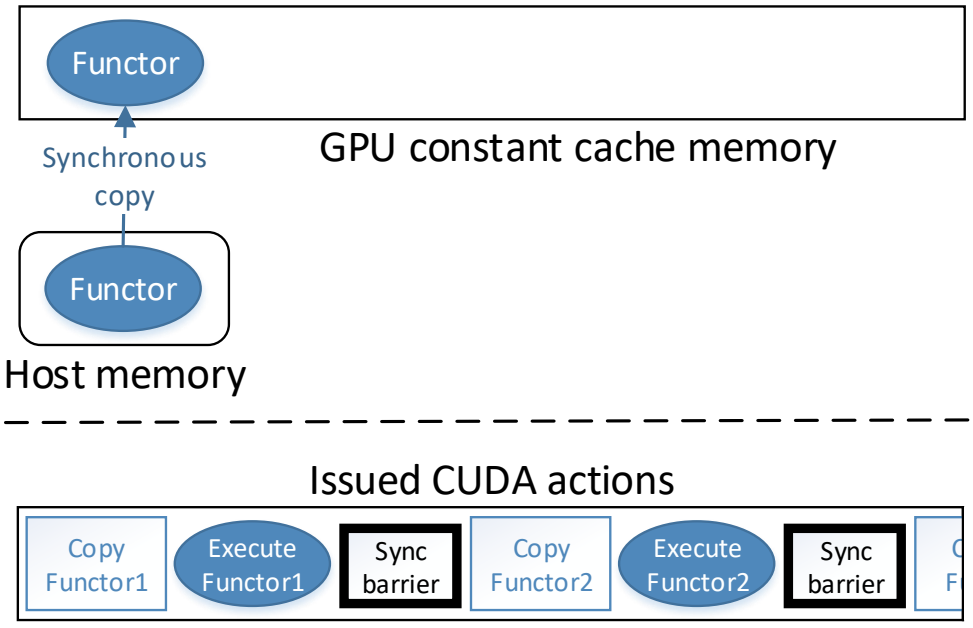

Figure 6.4: Current Kokkos GPU execution model using the constant cache memory.

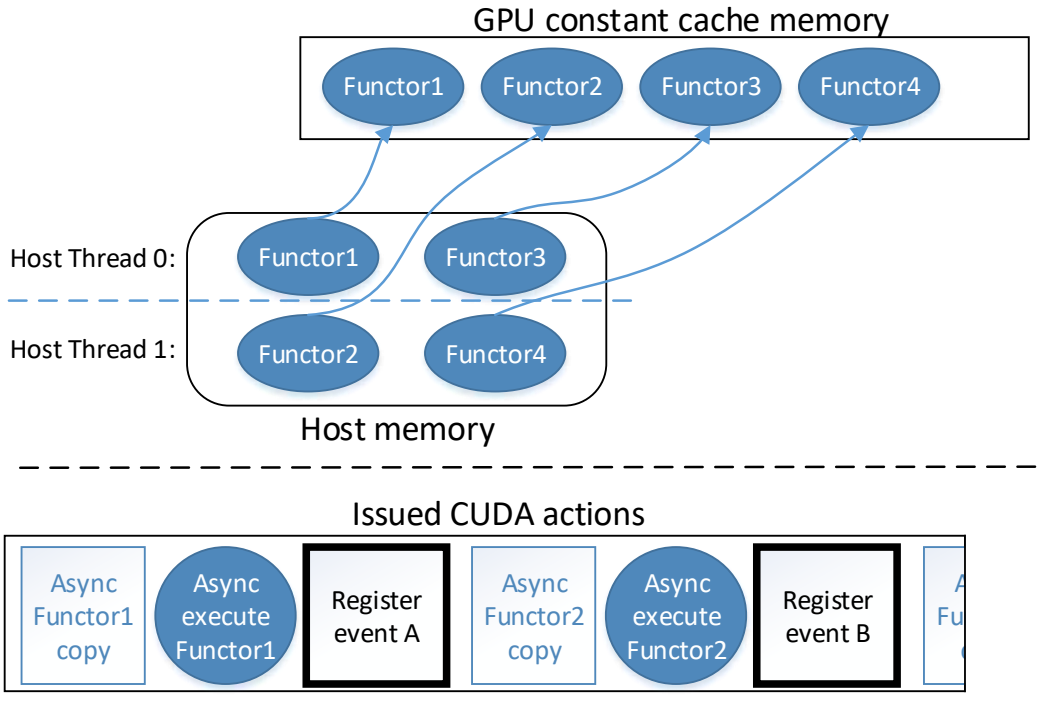

Figure 6.5: A desired solution so that functors can be asynchronously copied into GPU constant cache memory.

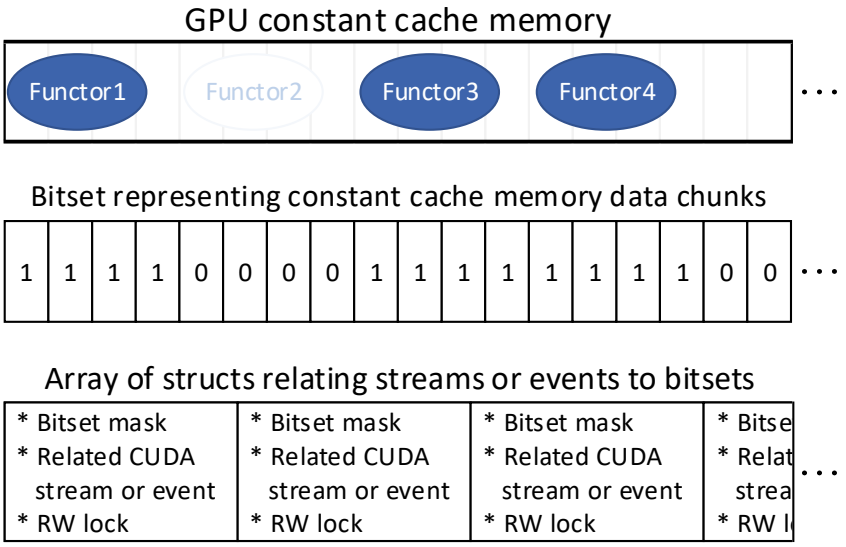

Figure 6.6: A bitset array and an array of structs are used to track functor objects and their associated CUDA streams or events. Bits set to 1 indicate the above data chunk contains data for a functor object, while bits set to 0 indicate the associated data chunk is unused or marked for reuse. In this example, Functor2 has completed while the other functor objects haven't yet completed execution.

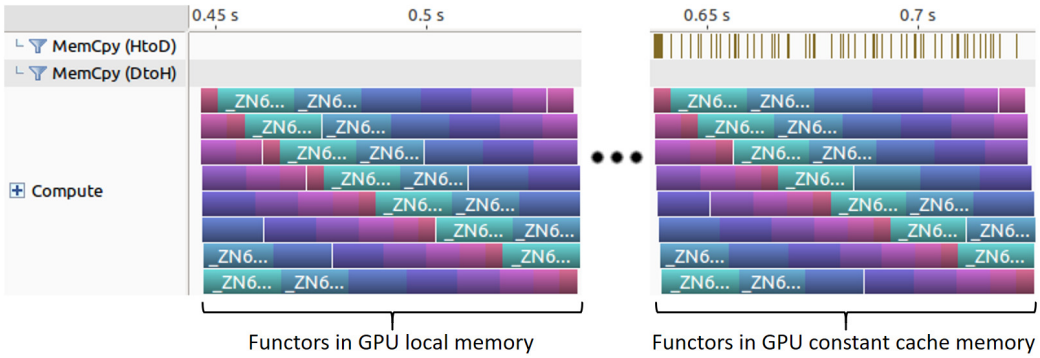

Figure 6.7: Profile of executing many `parallel_for` loops on 8 streams. The left side is when CUDA automatically copies the functor and the right side is when functors are manually copied through this work. Similar behavior is observed in both functor approaches.

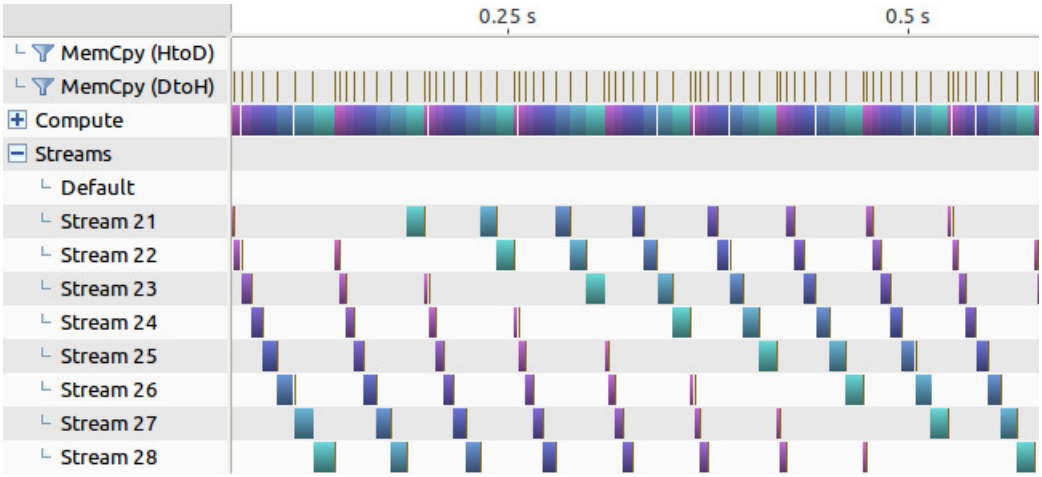

Figure 6.8: Profile of executing many `parallel_reduce` loops on 8 streams. Reduction values in pageable host memory invoked repeated synchronization and caused delays as shown by the gaps between executing kernels.

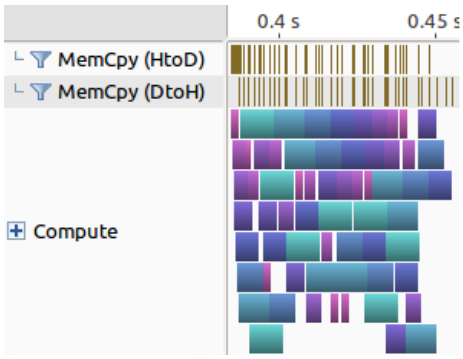

Figure 6.9: Using pinned host memory for reduction value buffers enables concurrent execution of reduction kernels. Synchronization is avoided even when pageable host memory is used to copy the functor data host-to-device.

Table 6.1: The three Kokkos loop iteration policies.

Kokkos Loop Pattern	Purpose	Example
RangePolicy	1D iteration patterns	Loop from 0 to 4096.
MDRangePolicy	2D or 3D iteration into (i,j) or (i,j,k) indexes. Supports different iteration patterns based on variable memory alignment.	Loop from (0,0,0) to (16,32,10), and assume row-major alignment.
TeamPolicy	Teams of threads, with each team using a 1D iteration pattern. Also supports hierarchical parallel loops. Inner loops use teams of threads declared in the outer loop.	Create 4 teams of 256 threads. Have each individual team of threads cooperate in an inner loop of 1200 iterations.

Table 6.2: Characteristics of various high performance Nvidia GPUs across four architectures.

	K20x (Kepler)	GTX Titan X (Maxwell)	P100 (Pascal)	V100 (Volta)
Constant Cache Size	64KB	64KB	64KB	64KB
Max number of concurrently executing kernels	16-32	32	128	128
Streaming multiprocessors (SMs)	14	24	56	84
Max number of resident blocks per SM	16	32	32	32

Table 6.3: Average launch latency measured when running a group of 64 CUDA capable kernels with varying parameter size. A kernel's execution consisted of a single instruction per iteration and 256 parallel iterations total. Cells denoted as (-) indicate no results as CUDA is limited to functors up to 4 KB.

Kernel Launch Latency Overhead				
	# of streams	Non-Kokkos CUDA kernel (μs)	Kokkos parallel_for loop, automatic functor copy (μs)	Kokkos parallel_for loop, manaul functor copy (μs)
4 byte parameter	1	11.17	33.82	49.52
	8	11.91	20.75	34.82
4 KB parameters	1	13.51	23.87	36.55
	8	14.33	25.81	36.58
8 KB parameters	1	(-)	(-)	40.46
	8	(-)	(-)	41.36

Table 6.4: A comparison of the Poisson problem (Section 1.6.1) on a 192^3 domain on various back-ends.

Poisson per-Loop Throughput Utilizing Various Back-Ends - (Times in Milliseconds)							
	Non-Kokkos Codes		Kokkos OpenMP	Kokkos CUDA Original			Kokkos CUDA This Work
Path Size (cells)	CPU Pthreads	GPU Raw CUDA		RangePolicy	TeamPolicy 16 blocks, 512 threads per block	TeamPolicy 24 blocks, 512 threads per block	TeamPolicy 16 blocks, 512 threads per block
16^3	0.039	0.027	0.133	0.189	0.187	0.183	0.072
24^3	0.035	0.043	0.13	0.171	0.17	0.174	0.08
32^3	0.051	0.033	0.128	0.147	0.147	0.14	0.083
48^3	0.179	0.052	0.134	0.166	0.141	0.138	0.088
64^3	0.548	0.103	0.162	0.375	0.212	0.208	0.113
96^3	1.57	0.368	0.251	1.71	0.463	0.35	0.222
192^3	18.09	3.33	1.95	31.3	3.11	2.15	2.16

Table 6.5: Single-node experiments demonstrating one single-level codebase executed on the CPU, GPU, and Intel Xeon Phi Knights Landing (KNL) processors. The preexisting C++ and CUDA implementations are also given for comparison purposes. The RMCRT:Kokkos on GPUs shows results before and after the work given in this chapter.

Comparison in RMCRT Single-Level Mean Times per Timestep (s)			
		Problem Size	
Processor/Accelerator	Implementation	64^3	128^3
CPU	RMCRT:CPU	14.34	302.02
	RMCRT:Kokkos	7.41	182.81
GPU	RMCRT:GPU	5.94	76.31
	RMCRT:Kokkos this work	3.71	64.72
	RMCRT:Kokkos original	16.82	274.57
Xeon Phi KNL	RMCRT:Kokkos	4.84	106.89

Table 6.6: Single-node experiments demonstrating a multilevel codebase executed on the CPU, GPU, and Intel Xeon Phi Knights Landing (KNL) processors. The preexisting C++ and CUDA implementations are also given for comparison purposes. The RMCRT:Kokkos on GPUs shows results before and after the work given in this chapter.

Comparison in RMCRT 2-Level Mean Times per Timestep (s)			
		Problem Size	
Processor/Accelerator	Implementation	64^3	128^3
CPU	RMCRT-DO:CPU	5.34	66.59
	RMCRT-DO:Kokkos	4.16	52.21
GPU	RMCRT-DO:GPU	3.47	45.30
	RMCRT-DO:Kokkos this work	2.49	26.72
	RMCRT-DO:Kokkos original	41.77	466.30
Xeon Phi KNL	RMCRT-DO:Kokkos	2.87	31.67

CHAPTER 7

KOKKOS AND GPU INTEGRATION INTO UINTAH

Kokkos provides significant portability support for application developer code within a parallel loop, but Kokkos provides little additional portability for task code surrounding the parallel loops. Uintah's application developers desired a fully portable solution for all Uintah task code. This chapter new Uintah API needed for fully portable tasks capable of compiling and running in three different portability modes, namely 1) Uintah's legacy task system using Pthreads, 2) CPU cores using a Kokkos OpenMP backend, and 3) GPU cores using a Kokkos CUDA backend. A key focus of this chapter is enabling the possibility of trivially implementing additional portability modes.

In the simplest form, Kokkos integration into Uintah would require Uintah application developers place `Kokkos::parallel_for` loops or `Kokkos::parallel_reduce` loops containing portable C++ code, and then compiling Uintah for a single given architecture. However, this simplistic approach is largely unusable by Uintah developers, as the following issues existed preventing feasible portability. 1) Uintah needed to support a heterogeneous mixture of tasks in the same simulation [15]. 2) Data access from a data store was still nonportable, requiring numerous preprocessor directives and duplicated code. 3) Data layout of simulation variables required different access patterns for GPUs compared to CPUs [104, 105]. 4) Some tasks required additional portable API needed across architectures (e.g., random number generation). 5) CUDA stream management was ill-fitting for developers not accustomed to developing for Nvidia GPUs. 6) No mechanism is given for fine tuning of architecture specific looping parameters at a Uintah level and a task level.

Initial attempts at using Kokkos without addressing these issues resulted in unmaintainable code. For example, data warehouse calls required preprocessor and C++ code

before and after the parallel loop sections, and often this resulted in duplicated logic. Occasionally branching logic was needed for system calls or third party API library tools. The number of total lines of task code to support portable Kokkos loops was often larger than had two different architecture specific tasks been coded instead. The resulting development and management load placed on the application developer was unwieldy. This approach simply couldn't scale up thousands of complex multiphysics tasks per node.

This chapter focuses on portability changes in four key areas of the Uintah runtime to support using many Uintah Kokkos-enabled tasks on many nodes at large scale. The major focus is keeping the application developer removed from as many runtime and portable decisions as possible. The four key areas of focus are 1) task code, 2) task declaration, 3) Uintah compilation, and 4) runtime configuration.

This chapter first reviews prior and ongoing Kokkos OpenMP work in Section 7.1. Section 7.2 describes current and new Uintah scheduler and execution models. Section 7.3 describes why this work merges together different Uintah code branches targeting backend models. Section 7.4 covers new changes needed to the task declaration phase to allow application developers an easy mechanism for defining all portable modes. Portable task code integration into Uintah is covered throughout Section 7.5. Section 7.6 covers configuration and runtime parameters now available for choosing portability modes and options. Section 7.7 covers lessons learned form refactoring tasks into portable code. Section 7.8 gives results after porting RMCRT and ARCHES tasks into fully Uintah task portable code.

7.1 Uintah Integration With Kokkos OpenMP

While this chapter's focus is achieving Uintah portability to Nvidia GPUs via Kokkos, solutions presented are coordinated with ongoing work by John Holmen which targets Uintah's Kokkos OpenMP integration for Intel-based processors. In particular, the Intel Xeon Phi KNL processor can employ up to 288 threads per node [106]. The prior Uintah model of one thread per task [9] was inefficient due to a combination of Uintah scheduler latency, CPU hardware limitations, and issues with domain decomposition. The Kokkos OpenMP backend offers an alternative of multiple CPU threads executing a single Uintah task.

Initial work by Sunderland et al. [11] provided a foundational strategy to incrementally add Kokkos loops to Uintah. Additionally, a simple mock runtime patterned after Uintah demonstrated the viability of vectorization for a stencil computation on CPUs, Nvidia GPUs, and Intel Xeon Phis. Follow-up work [13] implemented Kokkos into Uintah for the Intel Xeon Phi and demonstrated substantial speedups by allowing Kokkos to use its OpenMP backend to run all available hardware threads in a task.

The OpenMP model was demonstrated on RMCRT benchmark problem, while also demonstrating CUDA loop portability [16] (Section 6.8.2). Here, Uintah's portability was still in its early stages. The main loop's code was made architecture portable, but nothing else in Uintah was yet made portable.

Uintah integration with Kokkos OpenMP is still ongoing. Efforts are currently aimed at assigning each task a subset of a node's CPU threads, instead of either assigning only one thread per task or all threads per task. Future work will investigate ways of using OpenMP dynamic scheduling of worker threads in a task to provide more adaptive and efficient execution [107].

7.2 Two New Task Execution Modes

As described in Chapter 5, Uintah has two task execution modes, one for CPU tasks and another for GPU tasks. Integrating Kokkos into Uintah required adding two new execution mode, one each for Kokkos OpenMP and Kokkos CUDA.

A summary of the four execution mode is shown in Table 7.1. Different Uintah task schedulers are required for different execution modes. The CPU task execution mode used Uintah's Unified Scheduler [9] and executed one CPU thread per task. The GPU task execution mode extended the Unified Scheduler so that one CPU thread can asynchronously launch a GPU kernel [2, 3, 10]. John Holmen is currently leading an effort to implement a Uintah-OpenMP scheduler [13] to support Kokkos OpenMP tasks on architectures like the Intel Xeon Phi KNL.

This work in this chapter supports not only the Kokkos CUDA mode but creates a Uintah design model encompassing all modes. As the next section describes, supporting four separate modes presented challenges, and a new overarching new design was desired.

7.3 Merging Uintah Execution Models Into Compiled Builds

Initially, the Uintah integration work proceeded separately. The Kokkos OpenMP work was kept separate from Kokkos CUDA work, and both of these were kept separate from the prior CPU and GPU work. Developers were expected to compile a separate Uintah build for different modes listed in Table 7.1. Four use cases persuaded Uintah developers to merge these efforts so that two or more modes could exist in the same Uintah compiled build: 1) supporting heterogeneity debugging, 2) regression testing, 3) debugging, and 4) future work.

7.3.1 Supporting Heterogeneity

Several use cases demonstrated a need for a task that cannot or should not execute on GPUs, but must execute on CPUs. The application developer would desire running as many tasks as possible on the GPU and would have tasks fall back to OpenMP or Pthread-based CPU modes when CUDA execution is not available. The Uintah ARCHES component, for example, has numerous tasks already written using functors for CPU execution only. Application developers cannot make all tasks CUDA portable simultaneously, so tasks are made GPU portable one-by-one. A similar use case occurs when some application tasks do not need GPU portability, such as Uintah tasks managing a CPU linear solver library, but do desire an OpenMP backend for task execution.

7.3.2 Regression Testing

The second use case is regression testing. Both application developers and runtime developers desire to ensure any code changes do not wrongly alter computed results. For simplicity, the Uintah development team desired a single Uintah build that could test both Kokkos OpenMP and Kokkos CUDA modes, rather than recompiling Uintah separately for each.

7.3.3 Debugging

Debugging code presents a problem. Kokkos created an *nvcc_wrapper* script for use in place of Nvidia's *nvcc* compiler. The *nvcc_wrapper* script is a two-stage process, where the first stage uses multiple preprocessor steps to dramatically reorganize a cpp file's C++

code, and the second stage compiles the resulting .cpp and .cu files. Any application developer attempting to debug portable code on the GPU will likely not recognize the resulting code generated from *nvcc_wrapper*'s first stage. As an alternative, the application developer can debug portable code in a CPU portable mode, as this code will be recognizable. The simulation can then run in both CPU and GPU modes to compare outputs for any differences. In this manner, the application developer does not need to compile and debug in one mode and then compile again in a second mode to compare. A single build suffices.

7.3.4 Executing Same Tasks in Multiple Modes

A fourth use case allows for future work. Uintah may desire running the same tasks on two modes simultaneously. For example, suppose an application developer desires to utilize both the CPU memory bus and GPU memory bus for the same task. Latency bounded tasks, in particular, would benefit most from this configuration. By allowing one portable task to compile for both CPU and GPU modes, Uintah becomes well-suited to implement this feature.

7.3.5 Difficulty Having One Single Build

Unfortunately, compiling all four modes into a single build is not currently possible. Numerous difficulties were observed attempting to combine both the Pthreads and OpenMP modes. The root causes of these issues were not fully discovered, but many seemed related to difficulties of mixing libraries at compile time on various machines, and would only manifest at runtime. Kokkos itself also quietly disables all Pthread execution if an application developer requested their Kokkos build support both OpenMP and Pthreads.

The implemented approach is shown in three parts in Figure 7.1. The first part is covered in Section 7.4 and Section 7.4.1 by describing how tasks can be declared for multiple modes. The second part is covered in Section 7.4.2 through Section 7.5 by describing how tasks can be designed and compiled for all the modes the application developer desired. The third part is covered in Section 7.6 by describing how the application developer can choose the modes at runtime.

The integration of the Unified Scheduler (supporting GPUs) and the Uintah-OpenMP

scheduler (supporting Kokkos OpenMP) is part of future work led by John Holmen. The work in this chapter keeps the Unified Scheduler and Uintah-OpenMP schedulers separate, and instead focuses on providing a Uintah design allowing application developers to write fully portable with no CUDA or OpenMP specific logic within the task itself.

7.4 Task Declaration

Previous to this work, an application developer would declare a Uintah task by creating a task object, supplying the function pointer of the task, and then add that task object to a Uintah scheduler object. For this example and the upcoming sections, Poisson tasks are used (Section 1.6.1). The original mechanism for declaring a Poisson task is shown below in Figure 7.2.

This work added a portable version of task declaration which can co-exist with the mechanism shown above. The portable version relies on supplying all possible architecture modes the task may support. This new mechanism is shown below in Figure 7.3. Section 7.4.1 and Section 7.4.3 will describe new features in more detail.

7.4.1 Task Tagging

A new task declaration mechanism, shown in yellow in Figure 7.3, declares all possible portability modes by supplying one tag per desired compiled portability mode. This mechanism marks the start of Uintah task template metaprogramming. The tagged task template parameters propagate through several layers of Uintah, with propagation ending at the task code itself.

The task tagging approach solves two core needs. First, it allows the application developer to specify all possible modes the task supports. Second, it helps ensure the compiler only compile the desired modes. The key mechanism is the preprocessor defines for `UINTAH_CPU_TAG`, `KOKKOS_OPENMP_TAG`, and `KOKKOS_CUDA_TAG`. These three tags are defined in the Uintah runtime code as shown below:

```
#define COMMA ,
#define UINTAH_CPU_TAG UintahSpaces::CPU COMMA UintahSpaces::HostSpace
#define KOKKOS_OPENMP_TAG Kokkos::OpenMP COMMA Kokkos::HostSpace
#define KOKKOS_CUDA_TAG Kokkos::Cuda COMMA Kokkos::CudaSpace
```

The C++ preprocessor expands the relevant portion of the Poisson code example into the following:

```
create_portable_tasks( . . . ,
  &Poisson1::timeAdvance<UintahSpaces::CPU, UintahSpaces::HostSpace>,
  &Poisson1::timeAdvance<Kokkos::OpenMP, Kokkos::HostSpace>,
  &Poisson1::timeAdvance<Kokkos::Cuda, Kokkos::CudaSpace>,
  . . .)
```

That code would then attempt to compile three different flavors of the templated function `Poisson1::timeAdvance<>()`.

7.4.2 Controlling Compilation Modes

Suppose the application developer desired to compile CPU tasks for Pthreads and GPU tasks for Kokkos CUDA but wishes to avoid OpenMP as the target computer does not support it. This solution is accomplished by ensuring no OpenMP related template parameters compile at all. The application developer supplies configuration arguments, which Uintah uses to select the tags. So in this example, the C++ tag defines have the `KOKKOS_OPENMP_TAG` fall back to a Pthread mode:

```
#define COMMA ,
#define UINTAH_CPU_TAG UintahSpaces::CPU COMMA UintahSpaces::HostSpace
#define KOKKOS_OPENMP_TAG UintahSpaces::CPU COMMA UintahSpaces::HostSpace
#define KOKKOS_CUDA_TAG Kokkos::Cuda COMMA Kokkos::CudaSpace
```

The preprocessor generated code shown below demonstrates why this works. The Poisson1's `timeAdvance<>()` method cannot compile for Kokkos OpenMP, as it is never listed. Further, the compiler will only compile the legacy CPU mode code once, as duplicates are ignored.

```
create_portable_tasks( . . . ,
&Poisson1::timeAdvance<UintahSpaces::CPU, UintahSpaces::HostSpace>,
&Poisson1::timeAdvance<UintahSpaces::CPU, UintahSpaces::HostSpace>,
&Poisson1::timeAdvance<Kokkos::Cuda, Kokkos::CudaSpace>,
. . .)
```

In another example, suppose the user wanted to only compile for Kokkos Cuda and nothing else, then the tags `UINTAH_CPU_TAG` and `KOKKOS_OPENMP_TAG` would be updated to support only the template parameters for Kokkos Cuda compilation. These template parameters would then propagate through Uintah down to the task level. If that task's code can be compiled by Nvidia's nvcc compiler, then the task would be managed by Uintah as if its simulation variables and execution will occur in a GPU.

In Figure 7.3, three portability modes are specified, but this could easily be only one, two, four, or more modes. The application developer can use this to limit modes as needed. Suppose a task is not ready for CUDA execution, such that it uses heavy use of operating

system calls not available to the GPU. The application developer would simply leave off the `KOKKOS_CUDA_TAG`. If another user attempted to compile Uintah for Kokkos Cuda execution only, the compilation would correctly fail.

7.4.3 Task Declaration Functor

The purple section in Figure 7.3 includes a new functor to manage additional Uintah task parameters. This functor is then passed into a Uintah API `create_portable_tasks()` to help create the task. This functor's purpose supports Uintah backwards compatibility of declaring additional task options (see the similarity of the purple section in Figure 7.2), as well as supporting future work of running a task in more than one mode in the same timestep.

For example, the task declaration functor allows Uintah to create two tasks objects for this task, one for the CPU and another for the GPU. Next, Uintah can load specific task parameters by running this task declaration functor twice, once for each task object. An alternative solution is to create a task object once for one portability mode, and then use a copy constructor to clone a second task object for the other mode. However, cloning task objects proved impractical in Uintah's case as the task object itself heavily utilizes pointers to data members, thus complicating object cloning process.

7.5 Task Code

After the application developer informs Uintah of a task and what modes it supports, the actual task code itself can be defined. This work's portability necessitated several changes to traditional Uintah tasks. A before and after of these changes can be seen in Figure 7.4 and Figure 7.5. Specifically, the changes are 1) template parameters, 2) new function parameters, 3) a templated *Execution Object*, which encapsulates many architecture-specific items, 4) portable simulation variable retrieval from a data store, 5) using functors or lambda expressions, 6) Uintah's 3D lambda indexing code pattern, and 7) other new portable API. Each of these are covered in sections 7.5.1 through 7.5.5.

7.5.1 Task Parameters

Every portable Uintah task must be prefixed with two template parameters, one describing the execution space, another describing the memory space. These template pa-

rameters are useful for any Uintah API requiring knowledge of where a task's computations occur or where the data is located. For example, in Figure 7.5, the data warehouse API shown in green requires the memory space template parameter.

The task parameter list is also updated from the prior task model. The largely unused `ProcessorGroup` parameter was removed. The `UintahParams` object allows for future Uintah changes by encapsulating any additional lesser used parameters. The `ExecObject` (*Execution Object*) solves a handful of crucial portability questions and is discussed in Section 7.5.2. Tasks are not limited to only the six task parameters shown in Figure 7.5 parameters, Uintah retains its support for additional task specific parameters by utilizing C++ variadic arguments.

7.5.2 Template Metaprogramming via a Uintah Execution Object

Kokkos focuses on providing code portability within a parallel loop but leaves all architecture and portable decisions outside the portable loop up to the application developer. The `ExecObject` task parameter greatly aids in task portability for two major purposes. 1) The object is templated on both the execution space and the memory space for runtime and API usage. 2) The object encapsulates architecture-specific information, such as CUDA streams. The former need is covered in this section. The architecture specific parameters are covered in Section 7.5.4.3.

The templated *Execution Object* was fundamental in achieving a full template metaprogramming solution. The overall need is propagating template parameters for the execution space and memory space from the task declaration phase (Section 7.4), through the Uintah runtime, and then into the task code itself. A full flow of this propagation can be seen in Figure 7.6.

The Uintah runtime heavily leverages function pointers as well as polymorphism for its task objects. This work now requires templates to be utilized throughout the runtime. Unfortunately, mixing both C++ polymorphism and templates is rather challenging, as templates are known at compile time and use early binding, while polymorphism is determined at runtime through late binding. Mixing these function pointers and templates is not as challenging, and can be accomplished through template deduction via a templated function parameter.

Therefore, the cleanest solution had the template parameters for the execution space and memory space carried through a parameter. The *Execution Object* serves this role. C++11's template deduction mechanisms allowed the template metaprogramming to propagate throughout the runtime from task declaration to task code.

7.5.3 Simulation Variable Retrieval

Application developers routinely utilize Uintah task API to retrieve simulation variables from a data store. The underlying API for these data store accesses is not portable. Previous to this work, Uintah had a single API to retrieve simulation variables [9]. This work extended [2, 4, 15] a nonportable second API [3, 10] for simulation variables in GPU memory. These two data store API are described in Chapter 4. For portability, Uintah required two more APIs, one to work with Kokkos Views in host memory, and another for Kokkos Views in GPU memory. Altogether, this resulted in four total sets of API.

A summary of the four sets of API is given in Table 7.2. Three of these four (OnDemand Data Warehouse API, host memory Kokkos Views, and GPU memory Kokkos Views) contain similar syntax and design patterns which can be unified into a common Uintah interface. These API employed a design pattern where simulation variables were obtained in host code *prior* to the parallel loop. One of these four (the original GPU Data Warehouse API), cannot be made portable, as it uses a different design pattern where simulation variables are obtained in CUDA code and *within* the parallel loop itself.

Unifying the approaches using the OnDemand Data Warehouse, host memory Kokkos Views, and GPU memory Kokkos was challenging. An example of these three nonportable modes is shown in the code below.

```
// Uintah legacy and corresponding Uintah grid variable objects
constNCVariable<double> phi;
old_dw->get(phi, phi_label, matl, patch, Ghost::AroundNodes, 1);
NCVariable<double> newphi;
new_dw->allocateAndPut(newphi, phi_label, matl, patch);

// Kokkos Views in host memory
constNCVariable<double> phi_cNC_var;
old_dw->get(phi_cNC_var, phi_label, matl, patch, Ghost::AroundNodes, 1);
KokkosView3<const double, Kokkos::HostSpace> phi;
phi = phi_constNC_var.getKokkosView();
NCVariable<double> phi_constNC_var;
new_dw->allocateAndPut(phi_constNC_var, phi_label, matl, patch);
KokkosView3<const double, Kokkos::HostSpace> newphi;
newphi = phi_constNC_var.getKokkosView();
```

```
// Kokkos Views in GPU memory
KokkosView3<const double, Kokkos::CudaSpace> phi;
phi    = old_dw->getGPUDW()->getKokkosView<const double>(
           phi_label->getName().c_str(), patch->getID(), matl, 0);
KokkosView3<double, Kokkos::CudaSpace> newphi;
newphi = new_dw->getGPUDW()->getKokkosView<double>(
           phi_label->getName().c_str(), patch->getID(), matl, 0);
```

Each of these three take different approaches. The Uintah legacy API utilizes specific Uintah grid variable objects (in this case, node centered grid variables). The Kokkos Views in host memory extract a Kokkos View from these grid variable objects. Note that the `KokkosView3` is simply a C++ type alias of a three-dimensional Kokkos View. The Kokkos Views in GPU memory retrieve from the GPU data warehouse directly, not from the grid variable objects.

Suppose Uintah opted for a very basic form of task portability, where the parallel loops were portable, but all other task code was not portable, including data store variable retrieval. Here, the application developer would be required to write three different sets of preprocessor `#ifdef` statements, to inform the compiler which mode should be compiled and which should be ignored. Further, the task would be limited to compiling for only one portable mode.

This work opted to avoid portability specific preprocessor `#ifdef` statements entirely. The portable version of the above code is shown below:

```
auto phi    = old_dw->getConstNCVariable<double, MemSpace> (
                phi_label, matl, patch, Ghost::AroundNodes, 1);
auto newphi = new_dw->getNCVariable<double, MemSpace> (
                phi_label, matl, patch);
```

The Uintah API for the methods `getConstNCVaraible()` and `getNCVariable()` act as an API layer, using C++ template specialization to invoke one of the three nonportable API previously shown. Further, by using C++ templates, Uintah can utilize C++ template SFINAE (substation failure is not an error) to effectively overload the return type. Thus the `phi` and `newphi` simulation variables will be completely different data types depending on which portable mode is selected.

Future work will have a major code refactor of all data store API into one single data store codebase. For now, the approach explained in this section enabled a temporary, portable solution.

7.5.4 Loop Iterations in Serial and Parallel

Kokkos's strongest feature is portable, parallel loop code through functors or lambda expressions. Many other codebases which adopted Kokkos rely on application developers directly using Kokkos API and invoking Kokkos parallel loops themselves. However, Uintah legacy CPU task loops execute serially. A solution was desired which retained non-Kokkos serial execution and Kokkos parallel execution in a single portable loop design pattern. Upcoming sections 7.5.4.1 through 7.5.4.6 describe both the prior serial API model and features of the Uintah parallel API model.

7.5.4.1 The Prior Nonportable Loop Iteration API

Application developers targeting nonportable CPU tasks use Uintah API to serially loop over every cell in a patch. This process is kept simple, so that the application developer is not required to write a triply-nested loop over a 3D Cartesian range, or managing particulars such as managing additional cells on some geometric boundary such as a physical wall, or managing data padding for architecture vector lengths. Previously [9], Uintah application developers would employ custom C++ iterators as shown in Figure 7.7

Utilizing Kokkos for cell iteration poses three major challenges. 1) Mapping patch cells to architecture threads require different strategies depending on the architecture, 2) Uintah's simulation variables require `(i,j,k)` indexing over three dimensions, and 3) CPUs best process multidimensional arrays through row-major iteration while GPUs best process data using column-major iteration. The following three subsections demonstrate a mechanism for hiding the details of all three challenges.

7.5.4.2 Uintah API for Functors and Lambda Expressions

Uintah offers its own parallel API for functors and lambda expressions. The runtime developers desired to hide from the application developer the Kokkos API for five major reasons. 1) Providing support for a backward compatible model where Uintah could execute parallel loop code had Kokkos not been enabled at all. 2) The Uintah design team desired more control over architecture specific execution parameters, so that a Kokkos *TeamPolicy* could be applied to CUDA compilations and a *RangePolicy* to OpenMP compilations. 3) Uintah strongly supports a 3D iteration pattern, and the existing Kokkos *MDRangePolicy* pattern to support 3D iteration did not contain specific desired features

for GPUs. 4) Application developers could be given more prebuilt API choices, which avoids burdening the application developer from repeatedly implementing the same code patterns manually. 5) Experience has shown that application developers often inefficiently use generalized API and create unmaintainable code, and specialized API helps avoid these problems.

A rudimentary form of this Uintah API layer was created previously by Sunderland [11, 13] which executed a functor either without Kokkos using a triply-nested loop or with Kokkos using basic settings. Application developers would invoke a `Uintah::parallel_for` (not templated) rather than a `Kokkos::parallel_for`. This work greatly expands this API layer to enable a production ready and performant design model.

This work originally attempted to support a template argument model. The example below shows supplying a `EXEC_SPACE` template parameter representing one of Uintah's portable execution back-end modes, and the `uintah3DRangeObject` function parameter stored the low and high (i, j, k) indexes:

```
Uintah::parallel_for< EXEC_SPACE >( uintah3DRangeObject,
                     [](const int i, const int j, const int k) {
                     /* loop body */});
```

However, this design model was not optimal. Application developers felt it cumbersome to type `EXEC_SPACE` as a template argument for every `Uintah::parallel_for` invocation. Further, this model does not give an intuitive location to place architecture specific parameters.

The *Execution Object* (Section 7.5.2) provided a solution by both storing these architecture specific parameters and holding portable template information. The design pattern now used by Uintah is given below:

```
Uintah::parallel_for( execObject, uintah3DRangeObject,
                     [](const int i, const int j, const int k) {
                     /* loop body */});
```

The `Uintah::parallel_for` retains a similar pattern to a `Kokkos::parallel_for`. Both use template deduction, and both support lambdas which supply the current iteration index or indexes. The `Uintah::parallel_for` differs by splitting apart the loop iteration range into one object, and the architecture logic into a separate object. Further, the Uintah runtime itself supplies the Uintah task a premade `execObject`, which has additional features described in the next section.

7.5.4.3 Architecture Parameters via the Execution Object

The *Execution Object* further enhances Uintah portable task design by storing non-portable parameters in portable code. Without such an object, the application developer is routinely exposed to supplying architecture specific information. An earlier iteration for Uintah task portability demonstrates this problem. Previously, a Uintah task received from the runtime a CUDA stream object as a `void*` parameter. The application developer was responsible for passing this CUDA stream into other Uintah API. Application developers with little familiarity using CUDA were confused as they had no prior exposure or concept of using asynchronous streams. They reported a frustrating disconnect in supplying architecture specific information into portable task code.

The solution encapsulates a CUDA stream inside the *Execution Object*. The application developer need only pass in the *Execution Object* into Uintah portable API, and the Uintah runtime retrieves the CUDA stream and uses it as needed. This approach was successful, application developers with no experience writing CUDA code created CUDA portable code and executed tasks asynchronously without ever being exposed to streams.

The *Execution Object* also encapsulates runtime architecture arguments. For example, CUDA allows its kernels to vary the number of CUDA threads per block and number of blocks per kernel. Uintah has default values for these, and they can be overwritten in task code (see Section 7.5.4.5 for more details).

The *Execution Object* is not limited to just runtime architecture parameters. Compile time template parameters may be implemented as well. For example, Kokkos supports utilizing CUDA's `__launch_bounds__` parameters through `Kokkos::Cuda` instance objects. These parameters signal Nvidia's nvcc compiler to override default compilation behavior in assigning a total number of GPU registers per CUDA thread. Manually adjusting this setting may result in more blocks per GPU streaming multiprocessor, and can obtain performance increases. (The `__launch_bounds__` settings were utilized in prior work demonstrating the GPU data warehouse and heterogeneous scheduler for the CUDA implementation of the RMCRT algorithm [15].) The *Execution Object* can simply carry and supply these parameters to later Kokkos parallel loop invocations, and is part of future work.

7.5.4.4 3D Indexing

Kokkos supplies three execution policies (*RangePolicy*, *MDRangePolicy*, and *TeamPolicy*) as described in Section 6.2.2 and summarized in Table 6.1. Uintah attempted several approaches using these policies to find a satisfactory 3D iteration scheme, and ultimately adopted a mixture of Kokkos policies hidden from the application developer's perspective. This section describes three failed attempts and then the adopted solution.

The first attempt desired a 1D parallel pattern using either the *RangePolicy* or *TeamPolicy* and applied them to dimensional slices. For example, application developers naturally gravitated to thinking of parallelism in terms of iteration over successive z-slices of a Cartesian (x,y,z) domain. Suppose a Uintah task and its corresponding patch used `16x16x16` cells. Then the application developer would supply a 1D parallel loop for 16 iterations in the z dimension, and then manually iterate in the x and y dimension.

Dimensional slicing as a parallel pattern is limiting. Suppose a patch has `15x15x15` cells, and CUDA is used for parallelism. CUDA tasks typically utilize some multiple of 128 threads, and attempting to partition these threads while preserving dimensional slices results in unused threads. This prior problem implies that patches should be sized to fit the architecture (which in itself is a very limiting requirement), but even then dimensional slicing is not free from problems. Uintah tasks operate on patches of slightly differing sizes, (e.g., patches on physical walls may get an extra layer of additional cells to help with stencil computations). A domain with walls partitioned into `16x16x16` cell patches may find some patches requiring 17 cells on some dimensions. When CUDA threads are applied at some multiple of 64 threads, these patches with 17 cells in a dimension require an imbalance of CUDA threads to work of cells. For these reasons, parallel patterns using dimensional slicing was abandoned.

The second attempt used a 1D parallel pattern supplied by either the *RangePolicy* or *TeamPolicy* and mapped those back to 3D. The application developer was given a 1D iteration index and converted it to a 3D `(i,j,k)` index. This approach avoided dimensional slicing entirely, and also avoided requiring sizing Uintah patches relative to the number of parallel threads. A major drawback was shifting a burden that should be the responsibility of the runtime to the application developer, and assuming the application developer wouldn't create any mistakes in the process. This drawback was most evident

for simple loops which simply required initializing data values in a cell, as the application developer sought one line of code (the initialization), but required multiple lines of additional support code to reach it (the manual 1D to the 3D index to cell mapping).

The third attempt sought to use the experimental Kokkos *MDRangePolicy*. This policy was introduced for users desiring to portably iterate over a 2D or 3D collection of data. The policy also gives control over iteration patterns to target row aligned or column aligned data. A major downside of the *MDRangePolicy* is that it does not allow control over CUDA blocks. For example, *MDRangePolicy* will automatically assign each loop some amount of threads per block and blocks per loop, and these parameters are not configurable (a typical *MDRangePolicy* CUDA configuration is 256 threads per block). Suppose an application developer recognizes that a particular loop dominates most of a simulation's wall time, and verifies there are enough free CUDA registers to run this loop at 384 CUDA threads per block. The *MDRangePolicy* Policy does not allow this without modifying Kokkos source code itself.

The adopted solution came from joint work with John Holmen and mixes concepts from the second and third attempts, as well as introducing new features [17]. The distinctions are 1) the iteration from the application developer's perspective is an `(i, j, k)` index, 2) internally the iteration may be 1D or 3D depending on the portability mode, and 3) different portable modes could utilize completely different parallel loop approaches to maximize application developer control. The adopted solution would be like a Kokkos *MDTeamPolicy*, if any such Kokkos concept existed.

A simple summary shown in Figure 7.8 shows how a *Uintah::parallel_<pattern>* invokes different kinds of loops, depending on the portability mode. Figure 7.9 depicts the core code for a *Uintah::parallel_<pattern>* loop when Uintah is compiled without Kokkos. Figure 7.10 depicts the core code for a *Uintah::parallel_<pattern>* loop when Uintah is compiled with Kokkos::OpenMP support. Figure 7.11 depicts the core code for a *Uintah::parallel_<pattern>* loop when Uintah is compiled with Kokkos::Cuda support.

7.5.4.5 Application Developer Control of CUDA Blocks and Threads

Referring again to the Uintah CUDA loop (Figure 7.11), the outer loop corresponds to the number of CUDA blocks desired. The inner loop's range corresponds to the number

of CUDA threads desired. This approach allows the application developer to tune these parameters through new command line arguments:

```
-cuda_threads_per_block <#> : Number of threads per CUDA block
-cuda_blocks_per_loop <#>   : Number of CUDA blocks per loop
```

The values entered in at the command line are used in this CUDA loop (Figure 7.11). Default values are used if the application developer doesn't specify either of these two values.

Note that the CUDA loop (Figure 7.11) reuses CUDA threads in the inner loop. Each CUDA thread requires its own unique set of GPU registers, and so thread reuse enables fewer registers per block. An alternative strategy uses each CUDA thread only once in a block, and maps all iterations into as many CUDA blocks as needed, so when a block completes its work, its registers can be assigned to an upcoming block. Uintah enables both strategies by giving the application developer control over blocks and threads per block.

Not all parallel loops require the same resources. Resource heavy loops tend to be limited to 256 threads per CUDA block. Resource-light loops, such as those for data initialization, tend to perform better with more 512 threads per CUDA block. The Execution Object approach allows for encapsulating not only a CUDA stream pointer but also loop specific information, as Figure 7.12 demonstrates:

7.5.4.6 Supporting Row-Major or Column-Major Iteration

The memory layout is important when a group of parallel threads accesses an array of at least two dimensions. Portable code should be performant on CPUs, GPUs, and Intel Xeon Phi KNLs. However, row-major and column-major alignments perform differently on different architectures given exactly similar loop code. This difference is due to CPU and KNL cache locality and GPU coalesced memory accesses. When one alignment is chosen for CPUs and KNLs, the other alignment would be needed for GPUs. To demonstrate why, see Figure 7.13 from a recent Kokkos tutorial [6] :

When using the *RangePolicy* or *TeamPolicy*, Kokkos developers recommended keeping code GPU portable by transposing 2D arrays in memory and sending the transposed copies into GPU memory. The *MDRangePolicy* partially alleviates the problem of manual

transposing by having the dimensions needed to perform architecture optimized iteration. As noted previously, the *MDRangePolicy* was not adopted by Uintah as it doesn't give finer-grained control of CUDA threads per block or blocks per loop.

Figures 7.9 through 7.11 in the preceding section demonstrate architecture compatible iteration schemes. Uintah simulation variables are stored in row-major format in both CPU and GPU memory. The non-Kokkos loop (Figure 7.9) correctly iterates down the first dimension. Thus, the loop utilizes the optimal cache iteration.

The OpenMP loop (Figure 7.10) utilizes the *chunk_size* OpenMP feature [69], which assigns each thread n contiguous chunks of iterations. Uintah set $n = 1$, giving each CPU thread one contiguous set of iterations. For example, a node assigning four threads to a task with 1000 iterations total and *chunk_size* set to one results in the first thread receiving iterations 0 through 249, the second thread receives iterations 250 through 499, etc. Within each thread and each chunk, Uintah correctly iterates down the first dimension. Thus, the OpenMP loop maintains the optimal cache iteration for all threads.

The CUDA loop (Figure 7.11) utilizes the Kokkos *TeamPolicy*. Within the inner loop, Uintah correctly iterates down the first dimension. For example, suppose CUDA thread 0 reads index (0, 3, 5), then CUDA thread 1 reads index (1, 3, 5), etc. These consecutive reads will be proper coalesced reads, as CUDA threads all perform the same read instruction on a warp of threads.

This design enables future support for column-major simulation variables and tile-based [108] approaches. For example, column-major data layouts would simply require iterating down the 2nd dimension. This implementation requires Uintah's runtime to track the data layout mode for a given memory space. So suppose a future application developer wished to explore column-major and tile-based layouts. The *Uintah::parallel_<pattern>* code simply requires a logical branching, either at runtime using if statements or at compile time through template metaprogramming. Within each logical code branch, different iterating patterns are supplied for each set of for loops.

7.5.5 Additional Portable API

The API described up to this point has been adopted for use among Uintah application developers. They reported success in most cases of refactoring legacy codes into Uintah

and Kokkos portable code. However, simulation variable initializations and boundary conditions posed difficulties. This section describes additional API for these use cases.

To highlight the problem with simulation variable initialization, consider a task from the ARCHES codebase that must initialize six variables:

```
Uintah::parallel_for(execObj, rangeBoundary, KOKKOS_LAMBDA(int i, int j, int
    k){
  char_rate(i, j, k)          = 0.0;
  gas_char_rate(i, j, k)      = 0.0;
  particle_temp_rate(i, j, k) = 0.0;
  particle_Size_rate(i, j, k) = 0.0;
  surface_rate(i, j, k)       = 0.0;
  reaction_rate(i, j, k)      = 0.0;
});
```

The application developer simply wishes to initialize these variables to zero, yet the code above requires supplying a `0.0` value six times, and writing the `(i, j, k)` indexes seven times (once for the parameter and six for the variables). Not shown is another challenge when some simulation variables have slightly larger dimensions. The solution to this challenge is a `Uintah::parallel_for` loop for those grid variables having larger dimensions. The ARCHES developers suggested a `Uintah::parallel_initialize` API instead:

```
  parallel_initialize(executionObject, 0.0,
                      char_rate, gas_char_rate, particle_temp_rate,
                      particle_Size_rate, surface_rate, reaction_rate);
```

This API is feasible. The simulation variables know their own bounds, and thus variables of different sizes can be supplied. C++ variadic arguments allow varying the number of Uintah simulation variables arguments supplied. For CUDA, the challenge is launching a single kernel which has the knowledge needed to initialize all variables.

The runtime's `parallel_initialize` starts by creating a 1D array of structs in host memory. Each element of the struct holds a simulation variable metadata, namely the pointer address and its dimension sizes. After each simulation variable is loaded into the 1D array of structs, a Kokkos CUDA loop is utilized. The 1D array of structs is copied into GPU memory as a normal kernel parameter via C++ lambda capture. From here, the loop simply proceeds using the supplied metadata to initialize the variables to the given value.

One limitation of `parallel_initialize` is that each simulation variable must be of the same type. For example, it cannot mix initializing both integers and floats. The application

developer can always fall back to using a `Uintah::parallel_for` loop as shown at the start of this section.

A second needed API handles boundary conditions, and by extension, unstructured grids. Both boundary conditions and unstructured grids utilize the same API logic, as both require that Uintah know exactly which cells require iteration. This API is part of future work and will likely be called `Uintah::parallel_unstructured_for`.

7.6 Runtime Configuration

Uintah's main executable, `sus`, has runtime arguments exposing choices for both the portability modes to use and specific portability mode parameters. Some of these arguments existed from prior work (e.g., the `-nthreads` option to specify the number of Pthreads available). Other arguments are implemented as part of other research (e.g., `-omp_threads_per_executor` and `-omp_task_executors`). This work modifies the `-gpu` option from the legacy GPU mode to the current Kokkos CUDA mode. This work introduces the `-cuda_threads_per_block` and `-cuda_blocks_per_loop` arguments. For these two new CUDA argments, default values are used if they are not explicitly specified.

Examples of command line arguments from before this work are shown below:

Use 16 CPU Pthreads (1 Pthread per CPU task)

```
sus -nthreads 16
```

Use 16 CPU Pthreads (1 Pthread per CPU task), and CUDA threads for GPU tasks.

```
sus -nthreads 16 -gpu arches_charox.ups
```

Examples of command line arguments from this work and Holmen's ongoing work are shown below:

Use 16 CPU Pthreads (1 Pthread per CPU task), CUDA threads for GPU tasks, and customized GPU thread/block options.

```
sus -nthreads 16 -cuda_threads_per_block 256 -cuda_blocks_per_loop 4
-gpu arches_charox.ups
```

1 OpenMP thread per task

```
sus arches_charox.ups
```

8 OpenMP threads per task, 256 threads total (targets the Intel KNL)

```
sus -omp_threads_per_executor 8 -omp_task_executors 16 arches_charox.ups
```

7.7 Task Refactoring Lessons Learned

The Poisson (Section 1.6.1), RMCRT (Section 1.6.2), and ARCHES (Section 1.6.4) codes were refactored through joint work with Holmen to support full OpenMP and CUDA portability. This process was challenging and required creating new design patterns and identifying best practices. Below is a collection of lessons learned from refactoring several tasks to support the Kokkos::Cuda back-end [17]:

1. Favor using lambdas, instead of functors, for parallel patterns. Functors require duplication of parameter lists across multiple locations. This duplication can be problematic for large parameter lists.

2. Favor use of plain-old-data to avoid temporary object construction and deep object hierarchies.

3. Copy object data members into local variables to pass them into a lambda. Passing members accessed via a C++ *this* pointer into a lambda captures the host memory *this* pointer address, which is not accessible from a GPU. (Note, C++17 allows a lambda capture of `*this`, which can alleviate this issue [109].)

4. Replace use of a collection containing a *few* items (such as an array of *std::vector*) created outside a functor or lambda and later passed into the parallel loop. Replace with prebuilt Uintah arrays of plain-old-data or structs of plain-old-data. Passing an array itself into a lambda captures the host memory pointer address, which is not accessible from within a GPU. The Uintah arrays of structs approach passes in the entire array as a parameter.

5. Replace use of a collection containing a *many* items (such as an array of *std::vector*) created outside a functor or lambda and later passed into the parallel loop. Replace with a data buffer obtained from a Uintah memory pool, and ensure the data is written or copied into that buffer.

6. Eliminate the allocation of memory within parallel patterns.

7. Eliminate the use of C++ standard library classes and functions that do not have CUDA equivalents. Examples encountered include replacing use of *std::cout* with

printf, replacing use of *std::string* with null-terminated arrays of characters, and hard-coding *std::accumulate*.

8. Replace use of *std::vector* created inside a functor or lambda with either arrays of plain-old-data or *Kokkos::vector*.

7.7.1 Favoring Lambda Expressions

The prior list's item #1 specified to use lambda expressions over functors. When application developers used functors instead of lambda expressions, far more work was required. Specifically, application developers had to 1) create the functor's struct, 2) specify data members, 3) specify the constructor arguments, 4) assign constructor arguments into the data members, and 5) specify the `operator()` code. This entire process was laborious and time-consuming, as each simulation variable was listed four additional times. Further, experience demonstrated that functors gave perhaps too much flexibility to application developers, as they began implementing unmaintainable ideas in the struct's constructor.

Lambda expressions simplified the code and were more popular with application developers overall. For example, the Poisson code using a lambda expression utilized 79 fewer lines of code compared to the functor approach. The joint work with Holmen implementing the ARCHES char oxidation into a lambda expression code utilized 1010 fewer lines [17].

7.8 Results

Results for portable RMCRT and ARCHES tasks are given below. This section expands on results given previously [17].

7.8.1 RMCRT Results

Section 6.8.2 previously gave Kokkos loop portable RMCRT results, but those were limited to nonportable tasks and only utilized Kokkos's *RangePolicy*. This section demonstrates the flexibility of Uintah's new GPU runtime features allowing the application developer to vary the number of CUDA blocks per loop number of threads per loop.

Table 7.3 shows single-node results using CPUs, GPUs, and Intel Xeon Phis for the Burns and Christen [5] benchmark problem using all three RMCRT code implementations

of the single-level approach and multilevel approach (RMCRT:CPU, RMCRT:GPU, RMCRT:Kokkos). The absorption coefficient was initialized according to the benchmark [5] with a uniform temperature field and 100 rays per cell used to compute the radiative-flux divergence for each cell. CPU and Xeon Phi KNL results were obtained by Holmen. CPU-based results were gathered on an Intel Xeon E5-2660 CPU @ 2.2 GHz with two sockets, eight physical cores per socket, and two hyperthreads per core. For RMCRT:CPU, Uintah's scheduler utilized 32 threads. Xeon Phi KNL results were gathered by Holmen on an Intel Xeon Phi 7230 Knights Landing processor @ 1.30 GHz with 64 physical cores and four hyperthreads per core. Xeon Phi KNL simulations were launched using 1 MPI process and 256 OpenMP threads with the *OMP_PLACES=threads* and *OMP_PROC_BIND=spread* affinity settings. GPU-based results were gathered on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 7.5.

As Table 7.3 demonstrates the Kokkos implementation demonstrated speedup over the non-Kokkos implementation in all architecture modes. Various Uintah runtime arguments were sampled, with the best configurations given. This work demonstrates that for the GPU, the best configuration speedup was 1.61x and 1.82x compared to the non-Kokkos code, while the default configuration sepedup was 1.24x and 1.27x.

7.8.2 ARCHES Results

The ARCHES char oxidation problem (Section 1.6.4) was ported into Uintah portable tasks. Results below demonstrate the ability for application developers to vary Uintah command line arguments to achieve optimal throughput. Timing results for Intel CPU and Xeon Phi KNL processors are also given for comparison and to demonstrate the success of writing portable code on a complex multiphysics code (Section 1.6.4). Many of these timings can be found in a recent technical report [17]. CPU and Intel Xeon Phi KNL results were obtained by Holmen. CPU-based results were gathered on an Intel Xeon E5-2660 CPU @ 2.2 GHz with two sockets, eight physical cores per socket, and two hyperthreads per core. Xeon Phi KNL results were gathered on an Intel Xeon Phi 7230 Knights Landing processor @ 1.30 GHz with 64 physical cores and four hyperthreads per core. All GPU-based results were gathered on an Nvidia GPU GeForce GTX TITAN X with 12GB of RAM with CUDA version 7.5.

Figure 7.14 depicts visual profiling of a problem that maximizes the GPU's capabilities using Uintah. A $128x128x64$ cell domain char oxidation problem is partitioned into 256 16^3 cell patches and executed on one compute node. The Uintah runtime processed 1280 tasks on 1280 CUDA streams. Each task has its own initialization loop and char oxidation loop, and thus Figure 7.14 shows 2560 kernels. Each char oxidation loop executed on 16 CUDA blocks. The average task loop times and per-loop throughput had similar values to those found in upcoming Table 7.4 and Table 7.5, indicating these loops executed efficiently. A follow-up test doubled the size and number of tasks to a 512 16^3 cell patches. Uintah processed one timestep before crashing due to the ARCHES char oxidation problem's memory requirements exceeding the GPU's DRAM capacity.

Table 7.4 depicts GPU performance when varying the number of CUDA blocks used per loop. This simulated problem spans 16 patches of data, each patch requires 5 ARCHES char oxidation tasks, and each task operates on its own CUDA stream. With the blocks varying from 1 to 24 per loop, the GPU is processing 80 to 1920 blocks per timestep. Additionally, each char oxidation loop has an initializer kernel to set allocated data to zero. The values in Table 7.4 and subsequent tables consider one ARCHES char oxidation loop to be the combined average of the initialization kernel and the main char oxidation kernel. As shown in Table 7.4, individual loops continue to demonstrate speedup as more blocks are used, though the strong scaling impact drops off between four and eight blocks. Table 7.4 should be viewed in context of Table 7.5 which gives actual throughput of loops.

Table 7.5 depicts loop throughput in a particular timestep when varying the number of CUDA blocks used per loop for char oxidation modeling. Times are gathered by measuring the timestamp of the first executed loop against the end timestamp of the last executed loop. The GPU can simultaneously execute an indeterminate number of a kernel's blocks, and so loop throughput gives a better indication of total GPU processing capability. For example, in the four-block 16^3 patch simulation, the GPU completes a char oxidation loop every 0.141 milliseconds. The speedups tend to peak near 3x, indicating that either the problem has reached a memory-bounded state or that the GPU issues a maximum number of blocks per kernel. Overall, the data suggest the application developer run this simulation with at least four blocks per loop.

Table 7.6 and Table 7.7 depicts this work's ability to vary the number of threads per

loop through a Uintah runtime parameter. Table 7.6 depicts loop level performance and Table 7.7 depicts total GPU loop throughput. The char oxidation simulation cannot run at more than 256 threads per loop due to significant register pressure. The results here indicate 256 threads per loop is ideal.

Table 7.8 depicts the key goal of this work, enabling full portability on complex problems and performant execution on multiple architectures. This table [17] presents CPU-, GPU-, and KNL-based results gathered using the Kokkos char oxidation implementation with the Kokkos::OpenMP, Kokkos::Cuda, and Kokkos::OpenMP back-ends, respectively. CPU and KNL results were obtained by Holmen. For CPU-based results, tasks were executed using 16 task executors with one thread per task executor via 1 MPI process and 16 OpenMP threads. (See the related technical report for further description of task executors [17].) For KNL-based results, tasks were executed using 64 task executors with four threads per task executor via 1 MPI process and 256 OpenMP threads. For GPU-based results, tasks were executed using 1 CUDA stream and 16 CUDA blocks per loop with 256 CUDA threads per block and 255 registers per thread.

The 64 patch row in Table 7.8 demonstrates the char oxidation problem shows a GPU speedup of 3.31x over the CPU computation and a 2.80x speedup over the Intel Xeon Phi KNL computation. The 128 patch case gives a GPU speedup of 3.61x and 3.01x speedup over the CPU and KNL times, respectively. The 128 patch case gives a GPU speedup of 3.67x and 3.06x speedup over the CPU and KNL times, respectively.

7.9 Uintah Task Portability Summary

The work in this chapter allows application developers the means to easily write portable task code. Uintah can configure, compile, and run tasks in various modes. New API was introduced, such as portable data store retrieval methods and a `parallel_initialize()` method. The *Execution Object* serves numerous runtime and application developer roles of enabling template metaprogramming and exposing architecture specified runtime parameters to the application developer. These portable changes were demonstrated on two complex multiphysics examples while also demonstrating speedup over prior execution models.

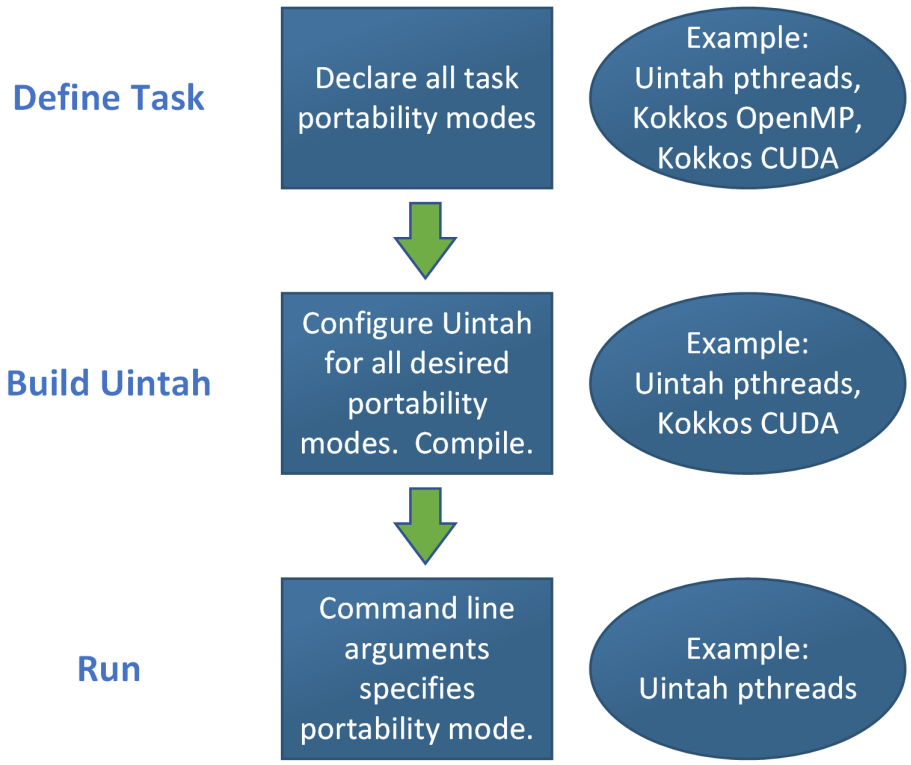

Figure 7.1: A task is declared for all possible portability modes it supports. The Uintah configure settings specifies which of these modes are allowed at compilation. Uintah runtime arguments determines which of these compiled modes are used during execution.

```
void Poisson1::scheduleTimeAdvance(const LevelP& level,
                                   SchedulerP& sched)
{
  Task* task = scinew Task("Poisson1::timeAdvance",
                           this,
                           &Poisson1::timeAdvance);

#if defined(HAVE_CUDA)
  if (Uintah::Parallel::usingDevice()) {
    task->usesDevice(true);
  }
#endif

  task->requires(Task::OldDW, phi_label,
                 Ghost::AroundNodes, 1);
  task->computesWithScratchGhost(phi_label, nullptr,
                                 Uintah::Task::NormalDomain,
                                 Ghost::AroundNodes, 1);
  task->computes(residual_label);

  sched->addTask(task,
                 level->eachPatch(),
                 m_sharedState->allMaterials());
}
```

Figure 7.2: The key parts of the Poisson task declaration are highlighted. Yellow provides the task object the entry function address. Green indicates the task has a GPU kernel available. Purple gives the new task object to Uintah's runtime. The yellow, green, and purple sections would become more cumbersome with Kokkos portability if no new API changes are made.

```
void Poisson1::scheduleTimeAdvance( const LevelP     & level
                                    ,       SchedulerP & sched
                                    )
{

  auto TaskDependencies = [&](Task* task) {
    task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
    task->computesWithScratchGhost(phi_label, nullptr,
                                   Uintah::Task::NormalDomain,
                                   Ghost::AroundNodes, 1);
    task->computes(residual_label);
  };

  create_portable_tasks(TaskDependencies, this,
                        "Poisson1::timeAdvance",
                        &Poisson1::timeAdvance<UINTAH_CPU_TAG>,
                        &Poisson1::timeAdvance<KOKKOS_OPENMP_TAG>,
                        &Poisson1::timeAdvance<KOKKOS_CUDA_TAG>,
                        sched, level->eachPatch(),
                        m_sharedState->allMaterials(),
                        TASKGRAPH::DEFAULT);

}
```

Figure 7.3: Task declaration with Uintah's new portable mechanisms. Similar regions as Figure 7.2 are also highlighted.

```
void Poisson1::timeAdvance( const ProcessorGroup * pg,
                            const PatchSubset    * patches,
                            const MaterialSubset * matls,
                                  DataWarehouse  * old_dw,
                                  DataWarehouse  * new_dw )
{
  int matl = 0;
  for (int p = 0; p < patches->size(); p++) {
    const Patch* patch = patches->get(p);

    constNCVariable<double> phi;
    old_dw->get(phi, phi_label, matl, patch, Ghost::AroundNodes, 1);
    NCVariable<double> newphi;
    new_dw->allocateAndPut(newphi, phi_label, matl, patch);

    newphi.copyPatch(phi, newphi.getLowIndex(), newphi.getHighIndex());

    double residual = 0;
    IntVector l = patch->getNodeLowIndex();
    IntVector h = patch->getNodeHighIndex();

    for(NodeIterator iter(l, h);!iter.done(); iter++){
      newphi[*iter]=(1./6)*(
        phi[*iter + IntVector(1,0,0)] + phi[*iter + IntVector(-1,0,0)]+
        phi[*iter + IntVector(0,1,0)] + phi[*iter + IntVector(0,-1,0)]+
        phi[*iter + IntVector(0,0,1)] + phi[*iter + IntVector(0,0,-1)]);
      double diff = newphi[*iter] - phi[*iter];
      residual += diff*diff;
    }

    new_dw->put(sum_vartype(residual), residual_label);
  }
}
```

Figure 7.4: The original Uintah CPU Poisson task code. This task's basic structure is similar to other Uintah tasks. The yellow area covers task parameters, green is data store variable retrieval, pink is initializing a data variable, and purple is the Poisson code's parallel loop.

```
template <typename ExecSpace, typename MemSpace>
void Poisson1::timeAdvance( const PatchSubset* patches,
                            const MaterialSubset* matls,
                            OnDemandDataWarehouse* old_dw,
                            OnDemandDataWarehouse* new_dw,
                            UintahParams& uintahParams,
                            ExecObject<ExecSpace, MemSpace>& executionObject )
{
  int matl = 0;
  for (int p = 0; p < patches->size(); p++) {
    const Patch* patch = patches->get(p);

    auto phi = old_dw->getConstNCVariable<double, MemSpace> (phi_label, matl, patch,
                                                             Ghost::AroundNodes, 1);
    auto newphi = new_dw->getNCVariable<double, MemSpace> (phi_label, matl, patch);

    double residual = 0;
    IntVector l = patch->getNodeLowIndex();
    IntVector h = patch->getNodeHighIndex();

    Uintah::BlockRange range( l, h );

    Uintah::parallel_for(executionObject, range, KOKKOS_LAMBDA(int i, int j, int k){
        newphi(i, j, k) = phi(i,j,k);
    });

    Uintah::parallel_reduce_sum(executionObject, range,
                                KOKKOS_LAMBDA (int i, int j, int k, double& residual){
      newphi(i, j, k) = (1. / 6) * (
          phi(i + 1, j, k) + phi(i - 1, j, k) +
          phi(i, j + 1, k) + phi(i, j - 1, k) +
          phi(i, j, k + 1) + phi(i, j, k - 1));
      double diff = newphi(i, j, k) - phi(i, j, k);
      residual += diff * diff;
    }, residual);
  }
}
```

Figure 7.5: The portable Uintah Poisson task code capable of CPU and GPU compilation and execution. The colored regions match those of Figure 7.4.

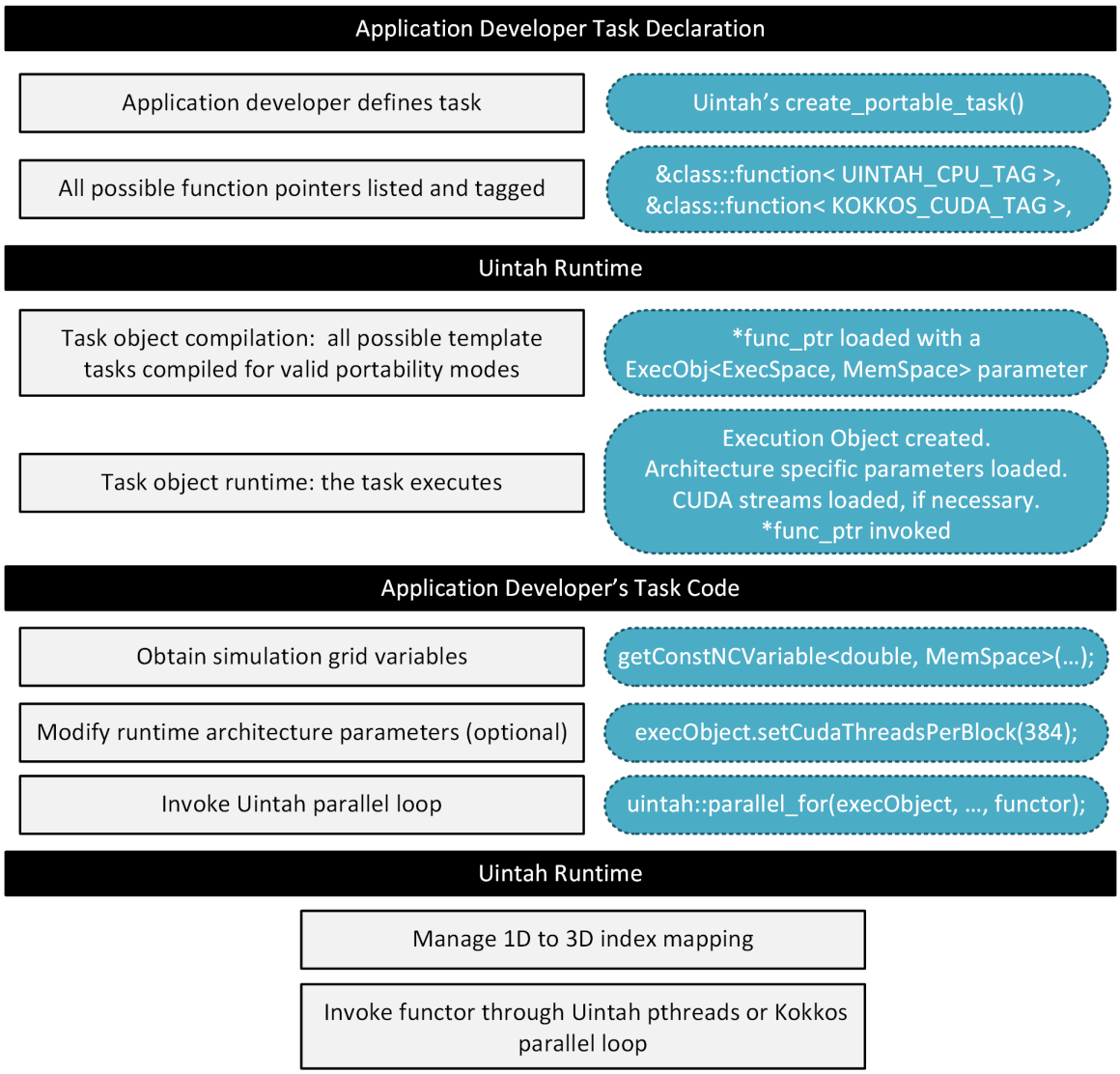

Figure 7.6: Template metaprogramming of Uintah portable code starts at the task declaration phase, propagates into the runtime, then back into task code, then into Uintah's parallel API, then into Kokkos's parallel API. The *Execution Object* is a central piece of this template metaprogramming.

```
double residual = 0;
IntVector l = patch->getNodeLowIndex();
IntVector h = patch->getNodeHighIndex();
for (NodeIterator iter(l, h); !iter.done(); iter++) {
  IntVector n = *iter;

  newphi[n] = (1. / 6)
          * (phi[n + IntVector(1, 0, 0)] + phi[n + IntVector(-1, 0, 0)]
           + phi[n + IntVector(0, 1, 0)] + phi[n + IntVector(0, -1, 0)]
           + phi[n + IntVector(0, 0, 1)] + phi[n + IntVector(0, 0, -1)]);
  double diff = newphi[n] - phi[n];
  residual += diff * diff;
}
```

Figure 7.7: The Poisson task using the previous Uintah API to iterate over cells. These iterators were intuitive and highly successful from an application developer standpoint. However, they are not portable as they process cells sequentially.

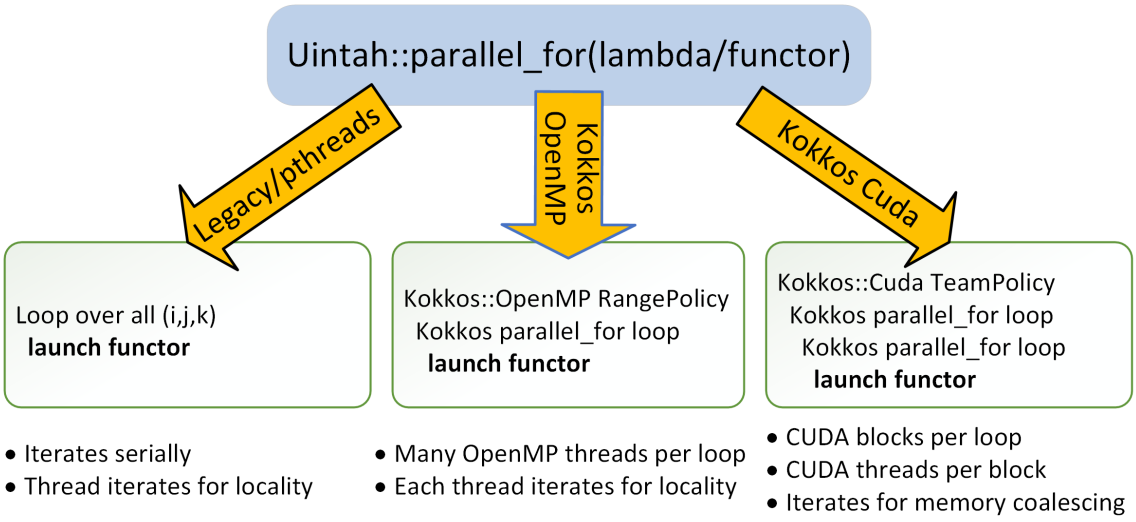

Figure 7.8: Uintah's `parallel_for` supports three portable execution modes.

```
for ( int k = kb; k < ke; ++k ) {
for ( int j = jb; j < je; ++j ) {
for ( int i = ib; i < ie; ++i ) {
functor( i, j, k );
}}}
```

Figure 7.9: Code illustrating how Uintah executes a functor on CPUs when Uintah is built without Kokkos.

```
Kokkos::parallel_for( Kokkos::RangePolicy<Kokkos::OpenMP, int>(0, numItems).
    set_chunk_size(1), [&, i_size, j_size, k_size, rbegin0, rbegin1, rbegin2
    ](int n) {
const int k = ( n / ( j_size * i_size ) ) + rbegin2;
const int j = ( n / i_size ) % j_size + rbegin1;
const int i = ( n ) % i_size + rbegin0;
functor( i, j, k );
});
```

Figure 7.10: Code illustrating how Uintah executes a functor when Uintah is built with Kokkos::OpenMP support.

```
Kokkos::TeamPolicy< Kokkos::Cuda > tp( cuda_blocks_per_loop,
    cuda_threads_per_block );
Kokkos::parallel_for ( tp, [=] __device__ ( typename policy_type::member_type
     thread ) {
const unsigned int startingN = /* calculate starting iteration for this block
    */
const unsigned int totalN = /* calculate iterations for this block */
Kokkos::parallel_for (Kokkos::TeamThreadRange(thread, totalN), [&, startingN,
     i_size, j_size, k_size, rbegin0, rbegin1, rbegin2] (const int& N) {
const int k = ( startingN + N ) / (j_size * i_size) + rbegin2;
const int j = ( ( startingN + N ) / i_size) % j_size + rbegin1;
const int i = ( startingN + N ) % i_size + rbegin0;
functor( i, j, k );
});
}
```

Figure 7.11: Code illustrating how Uintah executes a functor when Uintah is built with Kokkos::Cuda support.

```
executionObject.setCudaThreadsPerBlock(4);
executionObject.setCudaThreadsPerBlock(512);
Uintah::parallel_for(executionObject, rangeBoundary, KOKKOS_LAMBDA(int i, int j, int k){
    newphi(i, j, k) = phi(i,j,k);
});

executionObject.setCudaThreadsPerBlock(4);
executionObject.setCudaThreadsPerBlock(256);
Uintah::parallel_reduce_sum(executionObject,
                            range, KOKKOS_LAMBDA (int i, int j, int k, double& residual){
```

Figure 7.12: Application developers can supply architecture specific portable options through a Uintah `ExecObject`. Uintah command line defaults are used instead if these options are not provided prior to a parallel loop. In this example, these CUDA options would be ignored when CUDA is not used for portability mode this task.

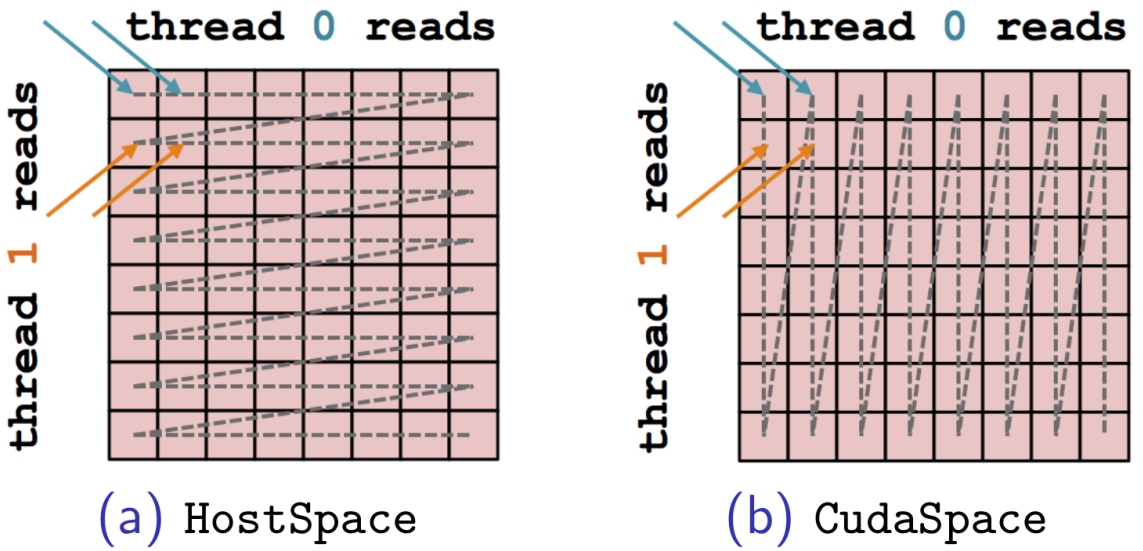

Figure 7.13: A visualization of memory access patterns in parallel [6]. The host memory 2D array (on the left) is in row-major format. The GPU memory 2D array (on the right) is in column-major format. In both figures, threads are sequentially assigned to the first index of the array's two indexes, and each thread iterates down the second index. CPUs and Xeon Phis perform better in the row-major format, as each thread utilizes locality by obtaining a cache line of subsequent array values. GPUs perform better in the column-major format as the columns coalesce in memory. Thus the warp of threads can perform one coalesced memory read for all threads instead of many reads for each thread in the warp.

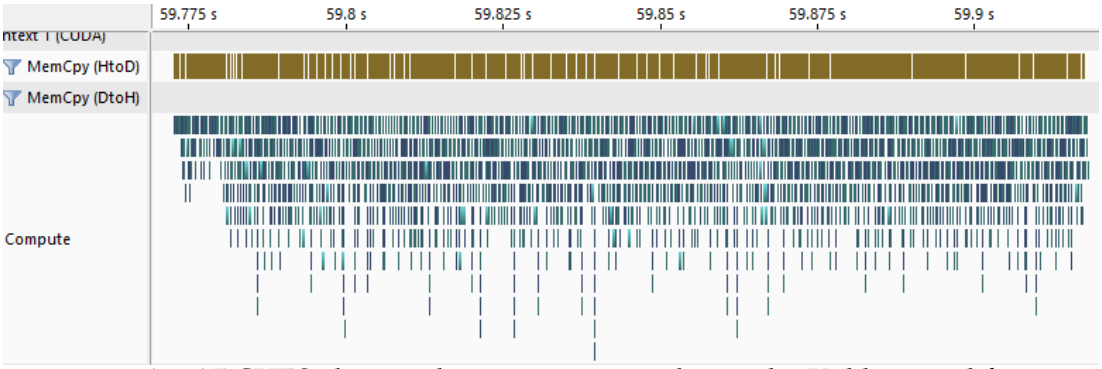

Figure 7.14: An ARCHES char oxidation timestep utilizing the Kokkos modifications in Chapter 6 and Uintah portable tasks and lambda expressions as described in this chapter. The timestep has 1280 GPU tasks on 1280 CUDA streams and 2560 individual loops. The brown blocks in the MemCpy row represents data copies into the GPU while the blue blocks in the Compute row represent initialization and computation kernels. Full asynchrony is realized as no parallel blocking operations occur.

Table 7.1: The four supported Uintah modes and their accompanying thread execution models. Normal CPU tasks are serially executed with 1 thread per task, while the others are executed in parallel with many threads participating in the execution of a single task. The Unified Scheduler can execute Kokkos-enabled OpenMP tasks, but with the limitation that all CPU threads (and not a subset of them) must execute the Kokkos parallel loop.

Uintah Mode	# of Threads Executing a Single Task	Max # of Concurrently Executing Tasks	Supporting Schedulers
Normal CPU Task	1 CPU thread	Number of CPU threads within the MPI rank	All
Normal GPU Task	Many CUDA threads	Many	Unified
Task using Kokkos OpenMP back-end	1 thread for task code outside parallel loops. Many to all available threads for task code in parallel loops.	Ranges from 1 to many	Uintah-OpenMP Unified*
Task using Kokkos CUDA back-end	Many CUDA threads	Many	Unified

Table 7.2: The four API sets to retrieve Uintah simulation variables. The first two are covered in Chapter 4. The last two are described in this chapter.

Data Store API for simulation variable retrieval	Where simulation variables are retrieved	Object storing simulation variable data	Supported for this work's portability
Uintah OnDemand Data Warehouse	C++ host code	Uintah specific types: CCVariabe, NCVariable, SFCXVariable, PerPatch, Reduction, etc.	Yes
Uintah GPU Data Warehouse	CUDA GPU code	Uintah specific types: GPUGridVariable, GPUPerPatch, GPUReduction, etc.	No
Kokkos host memory view	C++ host code	Kokkos View	Yes
Kokkos GPU memory view	C++ host code	Kokkos View	Yes

Table 7.3: Single-node per-timestep timings comparing 2-level RMCRT performance across Intel Sandy Bridge, NVIDIA GTX Titan X, and Intel Knights Landing. Same Configuration indicates use of the same run configuration as the existing non-Kokkos implementation. Best Configuration indicates use of the best run configuration enabled by additional flexibility introduced when adopting Kokkos. (X) indicates an impractical patch count for a run configuration using the full node. (*) indicates use of 2 threads per core. (**) indicates use of 4 threads per core. Holmen obtained the CPU and Intel Xeon Phi KNL results.

RMCRT Per-Timestep Timings - in seconds (x speedup) - CPU/GPU/KNL				
Architecture	Implementation	512 - 16^3 Patches	64 - 32^3 Patches	8 - 64^3 Patches
CPU	2L-RMCRT:CPU	51.57* (-)	71.69 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	36.30* (1.42x)	55.49 (1.29x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	34.96* (1.48x)	42.03* (1.71x)	60.55* (-)
GPU	2L-RMCRT:GPU	32.08 (-)	46.58 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	25.88 (1.24x)	36.66 (1.27x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	19.96 (1.61x)	25.60 (1.82x)	43.63 (-)
KNL	2L-RMCRT:CPU	57.93** (-)	102.11 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	43.82** (1.32x)	80.99 (1.26x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	29.17** (1.99x)	38.78** (2.63x)	60.45** (-)

Table 7.4: Single-GPU loop level performance when varying quantities of CUDA blocks per loop for the Kokkos implementation of the ARCHES char oxidation problem using 256 threads per block. All speedups are referenced against 1 block per loop timings.

Per-Loop Scalability - in milliseconds (x speedup) - GPU			
CUDA Blocks per Loop	16^3 Patch	32^3 Patch	64^3 Patch
1	2.80 (-)	18.57 (-)	147.59 (-)
2	1.47 (1.91x)	9.59 (1.94x)	77.58 (1.90x)
4	0.80 (3.49x)	5.50 (3.38x)	43.99 (3.36x)
8	0.48 (5.78x)	3.17 (5.85x)	25.57 (5.77x)
16	0.36 (7.76x)	2.29 (8.09x)	18.99 (7.77x)
24	0.28 (10.07x)	1.92 (9.68x)	13.88 (10.63x)

Table 7.5: Single-GPU throughput performance when varying quantities of CUDA blocks per loop for the Kokkos implementation of the ARCHES char oxidation problem using 256 threads per block. All speedups are referenced against 1 block per loop timings.

Per-Timestep Loop Throughput - in milliseconds (x speedup) - GPU			
CUDA Blocks per Loop	16^3 Patch	32^3 Patch	64^3 Patch
1	0.406 (-)	2.59 (-)	21.25 (-)
2	0.226 (1.80x)	1.39 (1.86x)	11.19 (1.90x)
4	0.141 (2.89x)	0.89 (2.92x)	7.37 (2.88x)
8	0.138 (2.93x)	0.83 (3.14x)	7.30 (2.91x)
16	0.144 (2.81x)	1.08 (2.40x)	6.97 (3.05x)
24	0.138 (2.95x)	0.98 (2.63x)	7.08 (3.00x)

Table 7.6: Single-GPU throughput performance when varying quantities of CUDA threads per CUDA block per loop for the Kokkos implementation of the ARCHES char oxidation problem using 4 blocks per loop. All speedups are referenced against 128 threads per block timings.

Per-Loop Scalability - in milliseconds (x speedup) - GPU			
CUDA Threads per CUDA Block	16^3 Patch	32^3 Patch	64^3 Patch
128	1.346 (-)	9.09 (-)	71.91 (-)
160	1.154 (1.16x)	7.37 (1.23x)	58.50 (1.23x)
192	1.140 (1.18x)	7.12 (1.28x)	55.24 (1.30x)
224	0.954 (1.41x)	6.27 (1.45x)	49.18 (1.46x)
256	0.802 (1.68x)	5.50 (1.65x)	43.99 (1.64x)

Table 7.7: Single-GPU throughput performance when varying quantities of CUDA threads per CUDA block per loop for the Kokkos implementation of the ARCHES char oxidation problem using 4 blocks per loop. All speedups are referenced against 128 threads per block timings.

Per-Loop Throughput - in milliseconds (x speedup) - GPU			
CUDA Threads per CUDA Block	16^3 Patch	32^3 Patch	64^3 Patch
128	0.190 (-)	1.24 (-)	10.07 (-)
160	0.187 (1.02x)	1.12 (1.11x)	9.26 (1.09x)
192	0.184 (1.03x)	1.09 (1.14x)	8.56 (1.18x)
224	0.164 (1.16x)	1.03 (1.21x)	7.76 (1.30x)
256	0.150 (1.27x)	0.89 (1.40x)	7.37 (1.37x)

Table 7.8: Single-node per-timestep loop throughput timings comparing CharOx:Kokkos performance across Intel Sandy Bridge, NVIDIA GTX Titan X, and Intel Knights Landing. (X) indicates an impractical patch count for a run configuration using the full node. (-) indicates a problem size that does not fit on the node.

Per-Timestep Loop Throughput - in milliseconds - CPU/GPU/KNL			
16^3 Patches per Node	CPU - Kokkos::OpenMP	GPU - Kokkos::Cuda	KNL - Kokkos::OpenMP
16	34.60	9.76	X
32	69.29	20.71	X
64	138.49	41.79	117.41
128	277.08	76.69	230.96
256	554.89	150.93	461.85
512	1108.88	-	915.99
1024	2219.71	-	1878.02
2048	4444.84	-	3706.70
4096	-	-	7356.10

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This dissertation has demonstrated that it is possible to construct a portable and performant GPU/heterogeneous asynchronous many-task (AMT) runtime system. All work was accomplished within the context of the Uintah AMT runtime [1, 9] and the Kokkos portability framework [12]. Results were demonstrated on four separate computational problems from Section 1.6 using Uintah and Kokkos modifications described throughout.

The modified AMT runtime remains consistent with Uintah separating the application developer's task code from the details of the underlying runtime system. As discussed in Chapter 1, Nvidia GPUs cannot be easily added into an AMT runtime. Nvidia GPUs add several challenges beyond normal CPU task execution, in particular, a proprietary programming model, separate high capacity memory, asynchrony of data movement and execution, and partitioning work among its many streaming multiprocessors. Previous work implemented mechanisms to execute GPU tasks in Uintah [3, 9, 10]. However, several difficulties remained. 1) Application developers wrote and maintained separate codes for CPUs and GPUs for the same tasks. 2) GPU code development required different strategies, particularly with different API tools. 3) GPU code execution parameters were hard-coded and often inefficient or device specific. 4) GPU tasks using Uintah required large task sizes so that each task utilized all GPU cores. 5) Debugging GPU task code took more time and effort than their CPU task counterparts. 6) Maintaining and unit testing all GPU task codes in the months or years afterward.

Additionally, the application developers using Uintah routinely had backgrounds outside of software development or computer science. The reasons outlined so far in this chapter motivated this work's thesis statement of providing new solutions "so that application developers can write portable task code capable of compilation and execution on both CPUs and GPUs as easily as they currently do for CPU-only task code" (Section 1.2).

Chapter 2 overviews related frameworks, AMT runtimes, libraries, portability layers, and software tools which help assist and abstract parallel program development. While many AMT runtimes support GPUs, they currently implement different methodologies affecting development effort and code performance. Chapter 2 also demonstrates a increasing trend toward portability frameworks, with some supporting compiler-level portability (OpenMP [19] and OpenACC [20]), while others supporting code-level portability (Kokkos, RAJA [18], Hemi [64], and OCCA [65]). Of these, Kokkos has an enticing combination of matured portability features, code flexibility, and low overhead performance costs. Unfortunately, Kokkos's Nvidia GPU implementation lacked some needed features, namely asynchronous execution of loops, efficient execution of loops that alone don't utilize all GPU cores, and custom problem partitioning while maintaining three-dimensional iteration corresponding to data layouts.

Chapter 3 provides an overview of Uintah and explains how its approach is unique among AMT runtimes due to its focus on three high-level goals. 1) Strong separation of concerns between the application developer and the runtime. 2) Strong support for automated halo management, including local halos, global or nearly global halos, transfers within a compute node, transfers among many compute nodes, and transfers across multiple adaptive mesh refinement levels. 3) Ability to reach full scale on current large parallel machines like DOE Titan and DOE Mira.

This work retains these three high-level goals and accomplishes the thesis statement through overarching work covering four main areas. 1) A modified GPU data store for simulation variable and task concurrency [2, 4, 14, 15]. 2) A modified GPU task scheduler enabling concurrency and cooperation among scheduler threads [2, 4, 14, 15]. 3) Modifying Kokkos for asynchronous GPU loops needed for tasks with smaller workloads [11, 16]. 4) Implementing Uintah full task portability [16, 17]. These four areas are now considered in turn.

8.1 GPU Data Stores

Chapter 4 describes the prior GPU data store, its mechanism of performing halo transfers in host memory, and why the overall data store design cannot work in an asynchronous GPU task environment. A specific contribution of this chapter is providing each

data store entry an atomic bitset [4, 15] which combined two lock-free design patterns to 1) track each data store entry's current states, using a model similar to the MSI cache coherency protocol, and 2) atomically declare intended action which precedes the actual action.

This chapter also describes task data stores where instances are exclusive for each task [2, 4]. This task data store model contains only data store entries needed for that task, and nothing more. This model provided numerous benefits to Uintah. 1) Avoiding AMT runtime race conditions of attempting to retrieve simulation variables that may not yet be available. 2) Providing a natural mechanism to avoid concurrency issues when copying data stores across memory spaces. 3) Minimize data store byte size and associated copy times. 4) Assisting the AMT runtime in properly reclaiming a simulation variable's allocated space when it will no longer be used. 5) Providing a way to supply all halo transfer instructions which should occur within GPU memory. Results shown in Figure 4.7 show that task data warehouses significantly reduced launch latency overhead by over 1000 us per task.

8.2 GPU Task Scheduler

Chapter 5 describes the prior GPU task scheduler [9, 10], how it performed actions synchronously, and problems that occur when it attempts to process GPU tasks asynchronously. A fundamental concurrency challenge of the prior GPU task scheduler is any task scenarios where two or more tasks share the same simulation variable. Uintah had no mechanism to detect and avoid the resulting race conditions.

This work resolved these simulation variable concurrency issues by changing Uintah's task scheduler approach from independent task preparation to cooperative task preparation [2, 4]. The atomic bitset described in the previous chapter is employed in combination with new scheduler queue phases. All task preparation work is now guaranteed to be performed only once, and tasks cannot execute until all needed preparation work has completed. Results given in Chapter 5 demonstrate that the combined work up to this point enable Wasatch tasks to execute faster in most observed scenarios. Compared to the prior GPU execution model [9, 10] observed speedups were 1.92x and 2.62x for 32^3 cell task sizes and 4.27x and 5.71x for 128^3 cell task sizes. Compared to equivalent CPU tasks,

observed speedups were 1.51x and 1.58x for 32^3 cell task sizes and 3.73x and 3.89x for 128^3 cell task sizes.

This work also described an AMT runtime methodology of assigning unique CUDA streams to each task, both CPU and GPU [2, 4]. This approach enabled the GPU task scheduler to overlap data copies in and out of GPU memory while GPU tasks executed. Further, streams allowed GPU kernels to run simultaneously, enabling performant execution of Uintah tasks that alone would not fill all GPU cores. Results given Chapter 5 demonstrate GPU task execution using 32^3 and 64^3 task sizes on the RMCRT benchmark problem [5] yield order of magnitude speedups (see Figure 5.12).

Additionally, this chapter added a new concurrency mechanism for sharing large halo data among simulation variables [15]. For example, the RMCRT computation problem required simulation variables which have global or nearly global halos, and a GPU's memory capacity is not large enough for each simulation variable to have a copy of the halo data. This work enabled coal boiler production runs on DOE Titan to proceed [15].

8.3 Kokkos Modifications

Chapter 6 reviewed the portability design pattern by Kokkos which enables code in functors and lambdas to be compiled to desired architecture compilers. Kokkos's portability features and execution methods are described in more detail. Chapter 6 also describes why Kokkos's current synchronous GPU execution model results in starved GPU cores unless the loops iterate over ranges much larger than those typically found within typical Uintah tasks.

This work modifies Kokkos to enable asynchronous execution of Kokkos loops [16]. This approach went far beyond simply assigning CUDA streams to each loop. Kokkos previously placed its functor objects in the GPU's constant cache memory and executed each serially assuming a functor will always be in the 0th byte. This work created a lock-free mechanism enabling multiple parallel CPU threads to place functors into any free region of the GPU's constant cache memory, execute the functor, and asynchronously verify when the functor completed. The end result preserves a functor execution design pattern preferred by Kokkos [102] and enables Kokkos to also exceed the standard 4 KB functor size limit.

Additional Kokkos API modifications were added. Application developers now have an option to use Kokkos for asynchronous loops while never encountering a CUDA stream. Additionally, streams can be manually provided to Kokkos when desired. Launch latency analysis indicates explicitly copying functors for later `parallel_for` execution has a 35-36 μs delay. Functors themselves execute between only 0.1% to 2% slower than had native CUDA code been used instead.

This work was demonstrated on the Poisson and RMCRT problems using Uintah's standard task granularity sizes. The Poisson tasks were demonstrated on various backends. Table 6.4 demonstrated that this work's modifications to Kokkos executed GPU loops faster than prior Kokkos CUDA configurations, with speedups between 1.54x to 2.54x observed compared to the best original Kokkos CUDA parameters. RMCRT results were computed as a joint effort with John Holmen. RMCRT was demonstrated on an Intel CPU, Intel Xeon Phi KNL, and an Nvidia GPU [16]. The results demonstrate clearly that Kokkos's new GPU asynchronous execution model enables significant speedup for loops that previously starved GPU cores. For RMCRT, this speedup was measured between 4.24x and 4.53x. Additional speedups were measured (1.17x and 1.6x on the GPU and 1.65x and 1.94x on the CPU) compared to non-Kokkos code equivalents due to implementing cleaner code patterns needed for portable loops.

8.4 Full Task Portability

Chapter 7 implements the remaining work needed to complete the thesis goal [11, 17]. Three major Uintah AMT runtime problems remained. 1) The application developer code utilizing Kokkos was unwieldy due to significant non portable logic needed to support portable loop code. 2) The need to expose architecture runtime launch parameters to the application developer. 3) Uintah needed a mechanism to correctly iterate in parallel corresponding to the data layout of a 3D array.

Chapter 7 solves the first problem by implementing full task portability. Kokkos provides loop portability, but all other task code was not portable. This work implements a task tagging system enabling application developers a mechanism to easily define all possible portability modes that task can support. Challenges mixing existing Uintah polymorphism and template metaprogramming were solved by implementing a templated

Execution Object parameter. This *Execution Object* then propagated at compile time throughout Uintah and to the Kokkos parallel loops themselves.

The Uintah tasks themselves were made portable by implementing new data store API capable of retrieving simulation variables as either Uintah grid variables or as Kokkos View objects. Uintah simulation variable syntax was extended to mirror Kokkos View syntax, enabling portable data store code. Joint work with John Holmen supported enabled supporting lambda expressions over functors, which reduced Poisson task code by 79 fewer lines of code ARCHES char oxidation code by 1010 fewer lines.

The second problem was solved by again leveraging the *Execution Object*. Both CUDA streams and CUDA launch parameters were encapsulated in that object. Application developers unfamiliar with Nvidia GPUs now no longer have to manually pass in CUDA streams. CUDA kernel block partitioning parameters are supplied default values and can be overridden by the application developer before each loop.

The third problem was solved by implementing expanded logic into Uintah's `paralle_for` loop and `parallel_reduce` loops. These Uintah API call their corresponding Kokkos API but do so differently for each architecture mode. The expanded logic maps a 3D iteration range to a 1D iteration range while also ensuring that both CPU tasks retain proper cache lines reads per thread and GPU tasks correctly iterate for coalesced memory accesses. The OpenMP mode uses the Kokkos *RangePolicy*, while the CUDA mode uses the *TeamPolicy* mode, which enables application developers to have full control over threads per CUDA block and the number of CUDA blocks. This work was also completed jointly with John Holmen who worked on the CPU and Intel Xeon Phi KNL aspects.

Chapter 7 concludes by demonstrating all work achieved in this dissertation on two complex multiphysics computations, RMCRT and ARCHES. For RMCRT tasks on 16^3 cells patches, the default Uintah CUDA kernel parameters yielded speedups of 1.24x over the non-Kokkos CUDA code, and 1.61x speedup with optimized block partitioning parameters. For ARCHES, these partition parameters were shown in more detail, indicating that each loop should utilize between 4-16 CUDA blocks and 256 CUDA threads per block, and achieves roughly 3x total throughput over using one CUDA block on 256 threads. Table 7.3 and Table 7.8 in particular demonstrate a key goal of this work by implementing production multiphysics problems in code portable tasks capable of performant execution

on GPUs and other architectures.

8.5 Lessons Learned

Asynchronous many-task runtimes can allow application developers to develop task code on both CPUs and GPUs as easily as they currently do for CPU-only task code. Application developers need not be limited to parallel tools incurring substantial overhead costs, utilizing custom domain-specific languages, utilizing custom compilers, or employing costly abstractions. Application developers with little background in computer science or software development can write C++ code capable of performant execution on multiple architectures.

Kokkos provides an excellent framework for achieving loop portability. While the functor portable design pattern can be implemented relatively quickly without Kokkos, its broad support of additional memory tools and API libraries justify its use. Kokkos can also be extended into an asynchronous loop execution model, as this work demonstrated.

Data stores and their associated task schedulers should enable lock-free cooperation for tasks sharing simulation variables or sharing halo data within simulation variables. Uintah's overarching design themes continue to correctly influence runtime development, namely 1) enforcing strict separation between the application developer and the runtime, 2) declaring tasks and its simulation variables in a *Computes*, *Modifies*, *Requires* model, and 3) automatically managing all halos transfers among tasks.

Implementing portable AMT runtime API beyond Kokkos portable API is vital for ease of application development, code maintenance, debugging, and preparing for future architectures. Experience demonstrated that when application developers write nonportable architecture specific logic, they often implement the same general concept in dozens of different and unmaintainable ways. A code portable methodology simply avoids numerous future problems.

8.6 Future Work

The following is future work related to this dissertation. Some of these are under development or planned, while others require further study.

8.6.1 Decoupling Dependencies From Data Structures

Uintah is fundamentally built upon a concept of patches operating on a structured grid. Uintah's data warehouse, task schedulers, automated halo processing, and automated MPI message generation likewise are hard-coded for structured grid halo extents. Moving forward, the Uintah team desires a runtime where the domain's data structure is largely unknown to most Uintah components. The end goal would enable Uintah to support architecture heterogeneity in structured grids, particles, unstructured grids, and other future unknown domain layouts. The underlying concept is that the data structures managing the domain data inform Uintah as to its data dependency needs. For example, instead of Uintah knowing that a task needs a grid variable with one layer of halo cell data, Uintah instead would query the grid variable's data structure logic and receive instructions how its data dependencies must be processed. This querying process can be used by multiple Uintah runtime in its dependency analysis.

8.6.2 Generalizing the Data Store and Task Scheduler

Uintah currently has a hard-coded data store for host memory, and another for GPU memory. A new data store should be written and generalized to support not just these two memory locations, but also future potential accelerators or discrete GPUs. Likewise, a generalized task scheduler should assist in preparing simulation variables in these memory locations and executing tasks.

Data store unification work is already underway. Both current data stores (the host memory data store and GPU memory data store) each have features desired for the unified data store. Work is also underway by John Holmen to unify Uintah's OpenMP based and Pthread-based GPU task scheduler allowing for an efficient heterogeneous mixture of Kokkos tasks executed through OpenMP and CUDA back-ends.

8.6.3 Event Driven Dependencies

The current task scheduler queue model incurs unnecessary delays and does not efficiently overlap GPU memory copies with GPU computation. Currently, when the scheduler finishes preparing a task's simulation variables in a given memory location, the scheduler places the task in the next work queue. The result is that tasks tend to process all host-to-device copies before any GPU kernels execute, and this incurs unwanted Uintah

launch latency delays because work is not overlapped.

This process can be improved by having tasks invoke events when its dependencies are met. For example, when a task's data variables are ready in GPU memory, tasks dependent on those data variables should immediately check if they are ready to execute, and if so, be placed in upcoming CUDA streams. Such an event-driven model would allow much better overlapping of the GPU's copy queues while simultaneously executing kernels.

8.6.4 Vectorization and Instruction Level Parallelism

Detailed profiling of the RMCRT problem and ARCHES char oxidation problem indicated both the CPU and GPU are latency bounded. For GPUs, this can be mitigated by employing instruction level parallelism (ILP). For CPUs, this can be mitigated by employing vectorized instructions. The challenges here are finding a mechanism so that application developers can write and compile such vectorized or ILP-oriented codes while providing a portable solution.

Damodar Sahasrabudhe is exploring utilizing Uintah and Kokkos so that portable C++ code can compile as CPU SIMD intrinsic instructions [32] or CUDA ILP instructions by utilizing overloaded operators and functions. Initial results are very promising for CPU-based vector execution.

Much ILP work remains, but the potential is promising. ILP would require more than simply overloading operators. Numerous tools such as math functions would require their own ILP versions. For example, the ARCHES char oxidation computation spent 90% of its time in math library calls, and thus ILP efforts should target these.

8.6.5 Executing Same Tasks on CPUs and GPUs

Memory bounded tasks, specifically, those that are latency bounded may benefit from executing the same tasks on both CPUs and GPUs simultaneously. The work so far allows Uintah to compile one task for multiple architecture modes in the same build, but no mechanism has yet been proposed to allow Uintah to efficiently execute these task on multiple nodes. Such future work would likely require novel modifications to the task scheduler to decide when it's appropriate to execute tasks on a different architecture back-end.

APPENDIX

RELATED PUBLICATIONS

1. **B. Peterson**, A. Humphrey, D. Sunderland, J.C. Sutherland, T. Saad, H. K. Dasari, and M. Berzins, "Automatic Halo Management for the Uintah GPU-Heterogeneous Asynchronous Many-Task Runtime," in *International Journal of Parallel Programming, 2018.* DOI: 10.1007/s10766-018-0619-1

2. **B. Peterson**, A. Humphrey, D. Sunderland, T. Harman, H. C. Edwards, and M. Berzins, "Demonstrating GPU Code Portability Through Kokkos on an Asynchronous Many-Task Runtime on 16384 GPUs," in *Journal of Computational Science and Engineering, Volume 27, pp 303 – 319, 2018.* DOI: 10.1016/j.jocs.2018.06.005

3. S. Kumar, A. Humphrey, W. Usher, S. Petruzza, **B. Peterson**, J. Schmidt, D. Harris, B. Isaac, J. Thornoc, T. Harman, V. Pascussi, and M. Berzins, "Scalable Data Management of the Uintah Simulation Framework for Next-Generation Engineering Problems with Radiation," in *Supercomputing Frontiers, Springer International Publishing, pp. 219–240, 2018.* DOI: 10.1007/978-3-319-69953-0_13

4. ***Awarded Best Paper*** **B. Peterson**, A. Humphrey, J. Schmidt, and M. Berzins, "Addressing Global Data Dependencies in Heterogeneous Asynchronous Runtime Systems on GPUs," in *Third International IEEE Workshop on Extreme Scale Programming Models and Middleware, held in conjunction with SC17: The International Conference on High Performance Computing, Networking, Storage and Analysis, 2017.*
   ISBN: 978-3-319-69953-0 DOI: 10.1145/3152041.315208

5. D. Sunderland, **B. Peterson**, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, "An Overview of Performance Portability in the Uintah Runtime System Through the Use of Kokkos," in *Proceedings of the Second International Workshop on Extreme*

Scale Programming Models and Middleware (ESPM2). IEEE Press, Piscataway, NJ, USA, 44-47. DOI: 10.1109/ESPM2.2016.012

6. **B. Peterson**, H. Dasari, A. Humphrey, J. Sutherland, T. Saad, and M. Berzins, "Reducing Overhead in the Uintah Framework to Support Short-Lived Tasks on GPU-Heterogeneous Architectures," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC '15)*. ACM, New York, NY, USA, Article 4 , 8 pages. DOI: 10.1145/2830018.2830023

7. **B. Peterson**, N. Xiao, J. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. Humphrey, and M. Berzins, "Developing Uintah's Runtime System For Forthcoming Architectures," Refereed paper presented at *the RESPA 15 Workshop at Supercomputing 2015 Austin Texas, SCI Institute, 2015.*

8. **B. Peterson**, M. Datar, M. Hall, and R. Whitaker, "GPU Accelerated Particle System for Triangulated Surface Meshes," in *Proceedings of the Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*.

REFERENCES

[1] M. Berzins, "Status of release of the Uintah computational framework," Scientific Computing and Imaging Institute, Salt Lake City, UT, USA, Technical Report UUSCI-2012-001, 2012. [Online]. Available: http://www.sci.utah.edu/publications/SCITechReports/UUSCI-2012-001.pdf

[2] B. Peterson, H. Dasari, A. Humphrey, J. Sutherland, T. Saad, and M. Berzins, "Reducing overhead in the Uintah framework to support short-lived tasks on GPU-heterogeneous architectures," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:8. [Online]. Available: http://doi.acm.org/10.1145/2830018.2830023

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

[4] B. Peterson, A. Humphrey, D. Sunderland, J. Sutherland, T. Saad, H. Dasari, and M. Berzins, "Automatic halo management for the Uintah GPU-heterogeneous asynchronous many-task runtime," *International Journal of Parallel Programming*, Dec 2018. [Online]. Available: https://doi.org/10.1007/s10766-018-0619-1

[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*, vol. 31, no. 4, pp. 401–421, 1997.

[6] H. C. Edwards, C. R. Trott, and J. Amelang, *Kokkos Tutorial*, 2015. [Online]. Available: https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/KokkosTutorialPresentation_Aug10_2015.pdf

[7] J. Peddie and R. Dow, "Add-in Board report," 2019. [Online]. Available: https://www.jonpeddie.com/store/add-in-board-report

[8] E. Strohmaier, J. Dongarra, H. Simonm, and M. Meuer, "Top 500 Supercomputer Sites," Nov. 2018. [Online]. Available: https://www.top500.org/lists/2018/11/highs/

[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] 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.

[11] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, "An overview of performance portability in the Uintah runtime system through the use of Kokkos," in *Proceedings of the Second International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2. Piscataway, NJ, USA: IEEE Press, 2016, pp. 44–47. [Online]. Available: https://doi.org/10.1109/ESPM2.2016.10

[12] H. C. Edwards and D. Sunderland, "Kokkos array performance-portable manycore programming model," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, 2012, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141703

[13] J. K. Holmen, A. Humphrey, D. Sunderland, and M. Berzins, "Improving Uintah's scalability through the use of portable Kokkos-based data parallel tasks," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, ser. PEARC17. New York, NY, USA: ACM, 2017, pp. 27:1–27:8.

[14] B. Peterson, N. Xiao, J. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. Humphrey, and M. Berzins", "Developing Uintah's runtime system for forthcoming architectures - refereed paper presented at the RESPA 15 Workshop at SuperComputing 2015 Austin Texas," SCI Institute, Austin, Texas, USA, Technical Report, 2015. [Online]. Available: http://www.sci.utah.edu/publications/Pet2015f/9-2csbfkt.pdf

[15] B. Peterson, A. Humphrey, J. Schmidt, and M. Berzins, "Addressing global data dependencies in heterogeneous asynchronous runtime systems on GPUs," in *Third International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2. IEEE Press, 2017.

[16] B. Peterson, A. Humphrey, J. H. T. Harman, M. Berzins, D. Sunderland, and H. Edwards, "Demonstrating GPU code portability and scalability for radiative heat transfer computations," *Journal of Computational Science*, vol. 27, pp. 303–319, July 2018. [Online]. Available: http://www.sci.utah.edu/publications/Pet2018a/jocs17-2.pdf

[17] J. K. Holmen, B. Peterson, A. Humphrey, D. Sunderland, O. H. Dia-Ibarra, J. N. Thornock, and M. Berzins, "Portably improving Uintah's readiness for exascale systems through the use of Kokkos," Salt Lake City, UT, USA, Technical Report UUSCI-2019-001, 2019.

[18] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: overview and status," Lawrence Livermore National Laboratory, Livermore, CA, USA, Technical Report LLNL-TR-661403, 2014.

[19] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: https://doi.org/10.1109/99.660313

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

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

[22] T. Saad and J. C. Sutherland, "Wasatch: an architecture-proof multiphysics development environment using a domain specific language and graph theory," *Journal of Computational Science*, 2015.

[23] 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.: WIT Press, 2008.

[24] S. Kumar, A. Humphrey, W. Usher, S. Petruzza, B. Peterson, J. A. Schmidt, D. Harris, B. Isaac, J. Thornock, T. Harman, V. Pascucci, , and M. Berzins, "Scalable data management of the Uintah simulation framework for next-generation engineering problems with radiation," in *Supercomputing Frontiers*. Springer International Publishing, 2018, pp. 219–240. [Online]. Available: http://www.sci.utah.edu/publications/Kum2018a/Scalable_Data_Management_of_the_Uintah_Simulation_.pdf

[25] 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*, vol. 125, pp. 389–400, 2017.

[26] 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*, Honolulu, Hawaii, USA, Jun. 2011, pp. AIAA–2011–3377.

[27] 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)*, vol. 39, no. 1, p. 1, 2012.

[28] R. Falgout, J. Jones, and U. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, ser. Lecture Notes in Computational Science and Engineering, A. Bruaset and A. Tveito, Eds. Springer Berlin Heidelberg, 2006, vol. 51, pp. 267–294.

[29] W. P. Adamczyk, B. Isaac, J. Parra-Alvarez, S. T. Smith, D. Harris, J. N. T. amd Minmin Zhoub, P. J. Smith, and R. Zmuda, "Application of LES-CFD for predicting pulverized-coal working conditions after installation of NOx control system," *Energy*, vol. 160, pp. 693–709, 2018.

[30] J. Pedel, J. N. Thornock, S. T. Smith, and P. J. Smith, "Large eddy simulation of polydisperse particles in turbulent coaxial jets using the direct quadrature method of moments," *International Journal of Multiphase Flow*, vol. 63, pp. 23–38, 2014.

[31] A. Humphrey, "Scalable asynchronous many-task runtime solutions to globally coupled problems," Ph.D. dissertation, Dept. of Computer Science, University of Utah, Salt Lake City, UT, USA, 2019.

[32] D. Sahasrabudhe, E. T. Phipps, S. Rajamanickam, and M. Berzins, "Adding a performance portable SIMD primitive in kokkos for effective vectorization across heterogeneous architectures," accepted to the International Conference of Parallel Computing, ICPP'19.

[33] D. Sahasrabudhe, M. Berzins, and J. Schmidt, "Node failure resiliency for Uintah without checkpointing," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 20, p. e5340, 2019.

[34] J. C. Bennett, R. Clay *et al.*, "ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms," Sandia National Laboratories, Albuquerque, New Mexico, USA, Technical Report SAND2015-8312, 2015.

[35] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993. [Online]. Available: http://doi.acm.org/10.1145/167962.165874

[36] L. Wesolowski, "An application programming interface for general purpose graphics processing units in an asynchronous runtime system," Master's thesis, Dept. of Computer Science, University of Illinois, 2008, http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

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

[38] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[39] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for HPC with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 81:1–81:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807629

[40] E. Slaughter, "Regent: A high-productivity programming language for HPC with logical regions," Ph.D. dissertation, Dept. of Computer Science, Stanford University, Stanford, CA, USA, 2017, http://purl.stanford.edu/mw768zz0480.

[41] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[42] S. Treichler, "Realm: Performance portability through composable asynchrony," Ph.D. dissertation, Dept. of Computer Science, Stanford University, Stanford, CA, USA, 2016, stanford University.

[43] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: A multi-stage language for high-performance computing," *SIGPLAN Not.*, vol. 48, no. 6, pp. 105–116, Jun. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462166

[44] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[45] M. Copik and H. Kaiser, "Using SYCL as an implementation framework for HPX.Compute," in *Proceedings of the 5th International Workshop on OpenCL*, ser. IWOCL 2017. New York, NY, USA: ACM, 2017, pp. 30:1–30:7. [Online]. Available: http://doi.acm.org/10.1145/3078155.3078187

[46] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.10.003

[47] G. Bosilca, A. Bouteiller, T. Hérault, P. Lemarinier, N. O. Saengpatsa, S. Tomov, and J. J. Dongarra, "Performance portability of a GPU enabled factorization with the DAGuE framework," *2011 IEEE International Conference on Cluster Computing*, pp. 395–402, 2011.

[48] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra, "Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach," *Scalable Computing and Communications: Theory and Practice*, pp. 699–735, 03-2013 2013.

[49] B. Haugen, S. Richmond, J. Kurzak, C. A. Steed, and J. Dongarra, "Visualizing execution traces with task dependencies," in *Proceedings of the 2Nd Workshop on Visual Performance Analysis*, ser. VPA '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/2835238.2835240

[50] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631

[51] V. G. Pinto, L. M. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A visual performance analysis framework for task-based parallel applications running on hybrid clusters," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 18, 2018.

[52] J. C. Bennett, J. Wilke *et al.*, "The DARMA approach to asynchronous many-task programming," in *Presented at ECP Review 2016, Sandia National Laboratories*, Albuquerque, New Mexico, USA, 2016.

[53] D. S. Hollman, J. C. Bennett, H. Kolla, J. Lifflander, N. Slattengren, and J. Wilke, "Metaprogramming-enabled parallel execution of apparently sequential C++

code," in *2016 Second International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, vol. 00, Nov. 2017, pp. 24–31. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ESPM2.2016.009

[54] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard templates adaptive parallel library (STAPL)," in *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, ser. LCR '98. London, UK, UK: Springer-Verlag, 1998, pp. 402–409. [Online]. Available: http://dl.acm.org/citation.cfm?id=648048.745869

[55] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065981

[56] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlic, S. Chatterjee, J. B. Fryman, I. Ganev, R. Knauerhase, M. Lee *et al.*, "The Open Community Runtime: A runtime system for extreme scale computing." in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, ser. HPEC, 2016, pp. 1–7.

[57] J. Dokulil and S. Benkner, "Retargeting of the Open Community Runtime to Intel Xeon Phi," *Procedia Computer Science*, vol. 51, no. C, pp. 1453–1462, Sep. 2015. [Online]. Available: https://doi.org/10.1016/j.procs.2015.05.335

[58] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight, "Extending the Uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S101–S122, 2016.

[59] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, Berkeley, California, USA, 2014.

[60] A. Bhatele, J.-S. Yeom, N. Jain, C. J. Kuhlman, Y. Livnat, K. R. Bisset, L. V. Kale, and M. V. Marathe, "Massively parallel simulations of spread of infectious diseases over realistic social networks," in *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, ser. CCGrid '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 689–694. [Online]. Available: https://doi.org/10.1109/CCGRID.2017.141

[61] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," in *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 21–30. [Online]. Available: https://doi.org/10.1109/WOLFHPC.2014.8

[62] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017. [Online]. Available: https://hal.inria.fr/hal-01618526

[63] Lawrence Livermore National Labs, "CHAI web page," 2017, https://github.com/LLNL/CHAI.

[64] M. Harris, "Hemi web page," 2017, http://harrism.github.io/hemi.

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

[66] D. Medina and T. Warburton, "OCCA," 2018. [Online]. Available: https://libocca.org/\#/

[67] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems Jade Edition*, vol. 2, pp. 359–371, 2011.

[68] "Thrust v1.8.0 release," 2014. [Online]. Available: https://thrust.github.io/01-12-2015/Thrust-v1.8.0-release.html

[69] *OpenMP Application Program Interface Version 5.0*, 2018, OpenMP Architecture Review Board.

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

[71] M. Martineau, C. Bertolli, and S. McIntosh-Smith, *Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support*. Institute of Electrical and Electronics Engineers (IEEE), 3 2017, pp. 54–64.

[72] A.Bourd, "The OpenCL specification," 2017. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf

[73] J. Kessenich, B. Ouriel, and R. Krisch, "SPIR-V specification, version 1.3, revision 6," 2018, https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf.

[74] T. Sörman, "Comparison of technologies for general-purpose computing on graphics processing units," Master's thesis, Department of Electrical Engineering, Linköping University, 2016.

[75] R. Keryell, M. Rovatsou, and L. Howes, "SYCL specification, version 1.2.1, revision 3," 2018, https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf.

[76] R. Landaverde, T. Zhang, A. K. Coskun, and M. C. Herbordt, "An investigation of Unified Memory Access performance in CUDA," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, 2014, pp. 1–6.

[77] *CUDA Runtime API Guide*, July 2017.

[78] NVIDIA, *Developing a Linux Kernel Module Using RDMA for GPUDirect*, Oct. 2018.

[79] T. O. M. Project, "FAQ: Running CUDA-aware Open MPI," 2018. [Online]. Available: https://www.open-mpi.org/faq/?category=runcuda

[80] B. A. Kashiwa and R. M. Rauenzahn, "A cell-centered ICE method for multiphase flow simulations," Los Alamos National Laboratory, Technical Report LA-UR-93-3922, 1994.

[81] 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*, vol. 2, pp. 509–522, 2001.

[82] 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: WIT Press, 2003.

[83] Q. Meng, "Large-scale distributed runtime system for DAG-based computational framework," Ph.D. dissertation, University of Utah, Salt Lake City, UT, USA, 2014.

[84] NVIDIA, "NVLink web page," 2015, http://www.nvidia.com/object/nvlink.html.

[85] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, and J. Dongarra, "GPU-aware non-contiguous data movement in Open MPI," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '16. New York, NY, USA: ACM, 2016, pp. 231–242. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907317

[86] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Computer Architure News*, vol. 12, no. 3, pp. 348–354, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/773453.808204

[87] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. London, UK, UK: Springer-Verlag, 2001, pp. 300–314. [Online]. Available: http://dl.acm.org/citation.cfm?id=645958.676105

[88] B. Ren, N. Ravi, Y. Yang, M. Feng, G. Agrawal, and S. Chakradhar, "Automatic and efficient data host-device communication for many-core coprocessors," in *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, ser. LCPC 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 173–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-29778-1_11

[89] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[90] V. Volkov, "Understanding latency hiding on gpus," Ph.D. dissertation, EECS Department, University of California, Berkeley, Berkeley, California, USA, Aug

2016, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.pdf. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[91] J. D. C. Little, "A proof for the queuing formula: L = λw," *Oper. Res.*, vol. 9, no. 3, pp. 383–387, Jun. 1961. [Online]. Available: http://dx.doi.org/10.1287/opre.9.3.383

[92] 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, pp. 1–10. [Online]. Available: http://www.sci.utah.edu/publications/Men2010b/Meng_TaskSchedulingUintah2010.pdf

[93] J. Jarvi, J. Freeman, and L. Crowl, "Lambda expressions and closures: Wording for monomorphic lambdas (revision 4)," Technical Report N2550=08-0060, 2008.

[94] *The Kokkos Programming Guide*, Oct 2018. [Online]. Available: https://github.com/kokkos/kokkos/wiki/The-Kokkos-Programming-Guide

[95] S.Vigna, "An experimental exploration of Marsaglia's xorshift generators, scrambled," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, pp. 30:1–30:23, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/2845077

[96] *CUDA C Programming Guide - PG-02829-001_v9.1*, Jan. 2018.

[97] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 134–144.

[98] S. Venkatasubramanian and R. W. Vuduc, "Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 244–255. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542312

[99] "NVIDIA K20X GPU accelerator," NVIDIA, Technical Report BD-06397-001_v05, 2012.

[100] "NVIDIA Tesla P100," NVIDIA, Technical Report WP-08019-001_v01.1, 2016.

[101] "NVIDIA Tesla V100 GPU architecture," NVIDIA, Technical Report WP-08608-001_v1.1, 2017.

[102] C. Trott, private communication, 2017.

[103] Nvidia, "GeForce GTX TITAN X Specifications," 2019. [Online]. Available: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications

[104] I.-J. Sung, J. A. Stratton, and W.-M. W. Hwu, "Data layout transformation exploiting memory-level parallelism in structured grid many-core applications," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 513–522. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854336

[105] I. Sung, G. D. Liu, and W. W. Hwu, "Dl: A data layout transformation system for heterogeneous computing," in *Innovative Parallel Computing - Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*, vol. 00, May 2012, pp. 1–11. [Online]. Available: doi.ieeecomputersociety.org/10.1109/InPar.2012.6339606

[106] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2Nd Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[107] M. Durand, F. Broquedis, T. Gautier, and B. Raffin, "An efficient OpenMP loop scheduler for irregular applications on large-scale NUMA machines," in *OpenMP in the Era of Low Power Devices and Accelerators*, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 141–155.

[108] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1989, pp. 357–361. [Online]. Available: http://dl.acm.org/citation.cfm?id=645818.669220

[109] H. C. Edwards, D. Vandevoorde, C. Trott, H. Finkel, J. Reus, R. Maffeo, and B. Sander, "Lambda capture of *this by value as [=,*this]," Open Standards, Technical Report P0018R3, 2016. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0018r3.html