Ph.D. Research Proposal

Doctoral Program in Scientific Computing

Portable and Performant GPU/Heterogeneous Asynchronous Many-Task Runtime System

Bradley Ryan Peterson

bradpeterson@gmail.com

Advisor:

Dr. Martin Berzins

School of Computing

The University of Utah

TABLE OF CONTENTS

CHAPTER 1 – INTRODUCTION

Asynchronous many-task (AMT) frameworks 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. This model has enabled task developers to focus on writing task code while minimizing their interaction with MPI transfers, automatic halo gathering, data stores, concurrency of simulation variables, and proper ordering of task execution. Uintah is also exploring portability through task code written using Kokkos constructs and a generalized Uintah API.

Nvidia GPUs introduce numerous challenges in maintaining this clear separation between the application and runtime layer. Specifically, Nvidia GPUs require code 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. Abstracting these GPU features into an application layer while maintaining both portability and performance requires numerous novel solutions to both Uintah and Kokkos. The focus of this research covers heterogenous task scheduling, concurrent data stores which share data objects and data dependencies, and enabling both GPU portability and performance using Kokkos. The high-level target goal of this research is demonstrating that production quality tasks previously written with Kokkos constructs and demonstrated on CPUs and Xeon Phis can also be performantly executed on GPUs with minimal additional application layer modification.

This proposal demonstrates the uniqueness of this work in Chapter 2 through a comparison of other AMT runtimes and parallel tools. Chapters 3 describes the prior state of Uintah's GPU engine. Chapter 4 outlines work completed to date. Chapter 5 provides remaining work required to meet the full goal of this thesis. Chapter 6 outlines the proposed thesis format. The remainder of this document contains the Conclusion, references, and a list of my publications.

CHAPTER 2 – EXISTING RUNTIME SYSTEMS AND PARALLEL TOOLS

AMT runtimes that have demonstrated scalability at large-scale or plan to reach that goal use varied approaches to aid application developers in their GPU implementations. No clear dominant pattern has emerged. Rather, these projects are motivated both by target problems and intended audiences which largely drive their priorities and accompanying abstractions. Likewise, many parallel tools supporting portability also take varied approaches to support targeted applications. This chapter provides an overview of Uintah and its comparisons with other related AMT runtimes and tools.

Uintah

The Uintah software suite [1], [2] is an AMT runtime designed to support multiphysics simulations for a broad range of problems involving fluids, solids, and fluid-structure interaction problems.

Uintah supplies concurrent and heterogenous data stores, heterogenous task schedulers, automatic internode and intranode memory management, automatic halo scattering and gathering, out-of-order execution of tasks, overlapping and asynchrony of GPU memory copies and execution units. No other mature AMT contains this full set of features. Uintah tasks define where it will execute (CPU or GPU), what simulation variables it computes and requires, and the halo requirements of these simulation variables. The runtime gleans all necessary information during task graph creation and task scheduling.

Uintah is in the early stages of supporting code portability of CPUs, Xeon Phis (specifically the Xeon KNL), and Nvidia GPUs through Kokkos [3]. Many simulation tasks have been or are undergoing rewrites using Kokkos parallel constructs. Task schedulers and data warehouses also require modifications to support Kokkos data objects.

Uintah is the most specialized of all the runtimes and tools listed in this chapter. This is largely due to Uintah's focus on specific target problems on a structured grid of hexahedral cells. However, the concepts and proposed ideas of this thesis can be extended to other AMT runtimes using unstructured grids.

Other AMT Projects and Parallel Tools

Kokkos

Kokkos [3] 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 number of threads desired, execution patterns, and targeted execution space. The architecture's compiler then compiles the functors and lambda expressions for the target architecture, and Kokkos will execute the functors or lambda expressions in the manner specified.

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-device and device-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.

Kokkos does not have any support for a concurrent data store interface, internode memory movement, automatic data gathering, asynchrony in data movement, heterogeneity in task scheduling, or overlapping of GPU execution.

Charm++

Likely the closest to Uintah in terms of features and functionality, Charm++ is designed for a wide audience as a large, monolithic tool aiding developers requiring a prebuilt, mature, AMT runtime. Charm++ has extensive support for internode data movement and task schedulers for execution. Charm++ does not explicitly define tasks, 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 to the runtime providing the next function to invoke. This contrasts with Uintah where runtime is responsible for invoking tasks and tracking when tasks complete.

Data movement to GPU memory and GPU code execution can be realized through their GPU Manager [4]. While it is automatic in the sense that the GPU Manager will allocate GPU memory and perform host-to-device and device-to-host copies, the amount of development steps required to perform these steps are effectively equivalent to performing them through native CUDA code. The GPU Manager requires the user provide their own CUDA kernels, amount of GPU memory buffers, and size of each buffer. The user is also responsible for providing a callback functions when a GPU kernel completes. Data copies and kernel execution can be realized asynchronously to support overlapping kernels. The Accel framework [5] works on top of the 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.

The combined Charm++ with the GPU Manager and the ACCEL framework does not automatically support sharing of data variables or data dependencies at a runtime level. Similarly, halo gathering is a not automatically managed. Charm++ has no data stores, explicit listing of tasks, task graphs, automatic data dependency analysis, or automatic halo scattering and gathering.

Legion

The Legion [5] runtime system handles automatic dependency management and concurrency by first requiring the application developer supply many more characteristics of a data structure's data dependencies. Legion shares many similarities with Uintah's goal of becoming architecture portable in a heterogenous environment, however, 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. For example, the rules for describing dependency movement of one layer of halo cells in a structured grid is surprisingly complex [6] compared to Uintah's method of simply specifying a single integer for halo layers of ghost cells. Two additional notable differences with Uintah is that the Legion runtime requires manually launching tasks and no concurrent data store is provided.

HPX

HPX [7] is a runtime system which recently reached version 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. At the moment, HPX has no support for GPUs, data stores, automatic data dependency analysis, halo scattering and gathering, etc. Internodal memory movement would be achieved through a global address space [8].

OpenACC

OpenACC [9] is largely founded upon using `parallel_for` constructs to express both parallelism and portability. Support for GPU asynchrony and host-to-device and device-to-host copies is also provided. OpenACC supports CPUs and Nvidia GPUs, but not yet Xeon Phis [10]. Compiler support is largely limited to the PGI compiler with the gcc compiler supporting some, but not all, OpenACC features [11]. OpenACC is not a runtime system, and does not aid the developer in any forms of data management beyond copying data between memory spaces.

OpenCL

The Kronos Group's OpenCL [12] has a design similar to abstracted CUDA code, in that device space is allocated, data copied, and kernels are executed by subdividing them into execution blocks. Out-of-order asynchrony is supported for both data execution and memory copies. OpenCL is supported by many architectures, including Nvidia GPUs and Intel Xeon Phis. OpenCL's performance frequently lags behind CUDA code [13]. OpenCL has likely reached its last specification version, as the Kronos Group recently announced, "We are also working to converge with, and leverage, the Khronos Vulkan API" [13]. OpenCL is not a runtime system, and does not aid the developer in any forms of data management beyond copying data.

CUDA Unified Memory

Unified Memory [14] aids portability by allowing a developer to treat all data in one address space with the goal of not burdening the developer with CUDA specific host-to-GPU and GPU-to-host manual copies. Unified Memory will perform automatic memory copies as needed to support executing kernels. Asynchrony is allowed until device data copies back into host memory, in which point a synchronization barrier is enforced. Pascal GPUs running CUDA 8.0 support an address space larger than available GPU memory and performs page-faulting operations to evict existing data in GPU memory to bring in requested data. This page-faulting is very expensive and leads to severe performance degradation unless prefetching hints are supplied to CUDA ahead of the kernel execution, or data is simply manually moved into device memory [15].

Unified Memory is an attractive option in that CUDA-aware MPI can be employed allowing MPI to copy data directly in and out of GPU memory through Remote Direct Memory Access (RDMA), avoiding temporary host buffers [16]. The two largest downsides of Unified Memory in relation to Uintah's goals are 1) Performance degradation in kernels [17] and 2) CUDA 8.0's Unified Memory effectively requires managing a second memory data store to track and avoid costly page-faults, which is what Uintah already does without Unified Memory's assistance.

Novelty of Proposed Research

No other mature AMT or tool shields the application developer from parallelization complexities at Uintah's level. This research seeks to extend this runtime separation to GPU tasks while obtaining good performance and portability using Kokkos. The success of this research can be measured by demonstrating a very small list of GPU specific features that application developers must provide to performantly run tasks on GPUs.

A second novel feature of this work is an asynchronous and concurrent data store for both host memory and GPU memory. The data store is maintained and managed through the task scheduler. No other AMT runtime provides a combination of asynchrony, concurrency, data stores, simulation variable sharing among tasks, data dependency sharing of simulation variables among tasks, and automatic halo cell scattering and gathering.

CHAPTER 3 – UINTAH GPU SUPPORT PRIOR TO THIS RESEARCH

Prior Uintah work [18] provided a simplistic model for GPU task execution. First, a rudimentary GPU data store (hereafter referred to as a **Basic GPU Data Store**) was created to enable `get()` API calls from within CUDA code. Second, the task scheduler was modified by implementing four new steps. 1) Prepare the task by synchronously copying every task simulation variable from host memory to device memory. 2) Synchronously copy the entire Basic GPU Data Store managed in host memory into GPU memory. 3) Synchronously execute the GPU kernel found within the task. 4) Perform a cleanup phase where all computed simulation variables are synchronously copied back to host.

This prior GPU execution model had numerous deficiencies for both portability and performance, and was only suitable for large monolithic problems executing properly blocked GPU kernels which computed in the order of seconds. For all other simulations using Uintah, the combination of synchronization barriers, PCIe bus contention, and duplicated simulation variables meant that simulations utilizing GPU tasks either ran slower than their CPU counterparts, or they could not fit within GPU memory.

CHAPTER 4 – CURRENT WORK AND PRELIMINARY RESULTS

GPU/Heterogenous Data Warehouses and Task Scheduler

The Wasatch project led by James Sutherland and Tony Saad utilized Uintah with numerous short-lived GPU tasks which exposed performance flaws in the runtime. The movement of many simulation variables into GPU memory and back to host memory took longer than the task execution. However, most of this data movement was unnecessary as the only time simulation variables were required in host memory was to perform MPI sends and receives. In response, a new GPU data store (**GPU Data Warehouse**) was implemented to give the Uintah runtime an expanded API to manage simulation variables in GPU memory. Simulation variables were also still tracked in the Basic GPU Data Store in host memory which would then be copied into GPU memory.

An expanded task scheduler was paired with the GPU Data Warehouse to verify the status of any simulation variable in GPU memory. The task scheduler prepared simulation variables prior to GPU task execution through allocation, host-to-device copies, and halo copies. The halo management was the most complicated feature. Halo copying metadata was placed into the Basic GPU Data Store to ensure all halo copies occurred in GPU memory, not host memory. The cumulative effects of these changes for Wasatch tasks is seen in Figure 1.

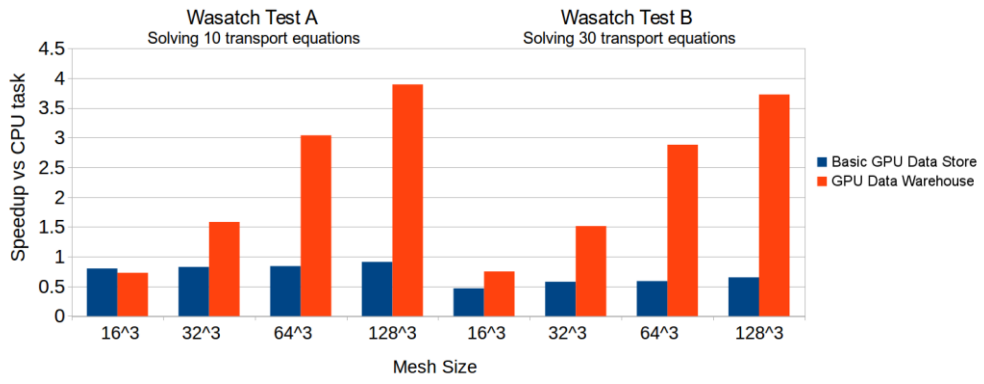

Figure 1: Short-lived GPU tasks were most susceptible to runtime overhead. Performing halo gathers entirely in GPU memory helped make total GPU simulation wall times tasks faster than CPU simulations. Computations performed on an Nvidia GTX 680 GPU with CUDA 6.5 and an Intel Xeon E5-2620.

Another GPU runtime enhancement reduced the size and enabled heterogenous concurrency of the Basic GPU Data Store. The GPU Data Warehouse generated a Basic GPU Data Store giving each task its own version of the data store containing only the simulation variables it requires. These data stores are termed **Task Data Stores.** Task scheduler threads could copy Task Data Stores into GPU memory without interfering with each other. As shown in Figure 2, the time spent copying data into GPU memory has been drastically reduced.

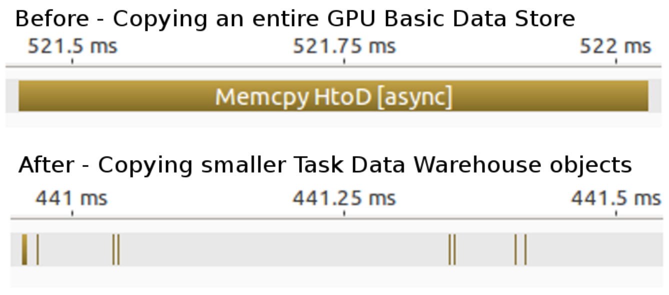
Before - Copying an entire GPU Basic Data Store
521.5 ms
521.75 ms
522 ms
Memcpy HtoD [async]
After - Copying smaller Task Data Warehouse objects
441 ms
441.25 ms
441.5 ms

Figure 2: Significant size reduction of Basic GPU Data Store objects better supports GPU tasks which execute on the order of milliseconds. The Task Data Stores are not shared between tasks, enabling GPU task asynchrony and overlapping.

An atomic bitset was added to each simulation variable's entry in the GPU Data Warehouse to track a simulation variable's lifetime. Task scheduler threads coordinated with one another by atomically querying these bitsets to ensure no two threads prepared the same data action. This change allowed the runtime to 1) simultaneously prepare two or more tasks sharing the same simulation variable 2) reduced memory usage in GPU memory and 3) enable full concurrency of GPU tasks by allowing kernel overlapping.

Figure 3 demonstrates this overlapping while running long-lived Reverse Monte-Carlo Ray Tracing (RMCRT) tasks.

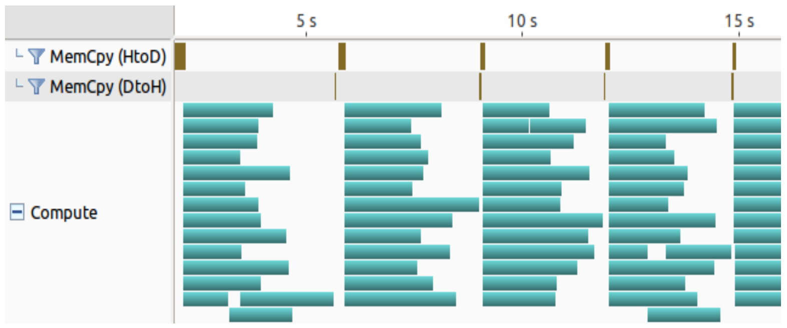

Figure 3: Data sharing and among tasks and GPU Task Data Stores allows the Uintah runtime to asynchronously invokes 16 sets of host-to-device copies followed by GPU task kernel executions. Computations performed on 16 RMCRT tasks per time step on an Nvidia K20c GPU.

The GPU Data Warehouse was further modified to avoid data duplication in the presence of global data dependencies. This change was motivated by a recent production run on the DOE Titan supercomputer for a proposed high efficiency coal boiler. The GPU Data Warehouse and accompanying task scheduler only supported tasks sharing simulation variables, but not sharing of halo data. The GPU Data Warehouse was modified to allow multiple data warehouse entries to create and use a shared data object. Likewise, task scheduler threads coordinated to ensure no two threads created duplicates of the shared data object. The result of these changes reduced memory usage allowing the problem to fit into GPU memory as shown in Table 1.

Memory Overhead Improvements		
Simulation patch layout	**Host memory usage before (MB)**	**Host memory usage after (MB)**
Coarse: 32^3 cells, 4^3 patches Fine: 64^3 cells, 4^3 patches	3073	65
Coarse: 32^3 cells, 4^3 patches Fine: 128^3 cells, 4^3 patches	23229	279
Coarse: 64^3 cells, 4^3 patches Fine: 128^3 cells, 4^3 patches	Exceeded memory	311

Table 1: Memory reductions from sharing halo data among simulation variables. Global data dependencies previously required each simulation variable to have its own copy of the entire domain.

Figure 4 demonstrates that the combined data warehouse and task scheduler work completed so far has enabled substantial speedups.

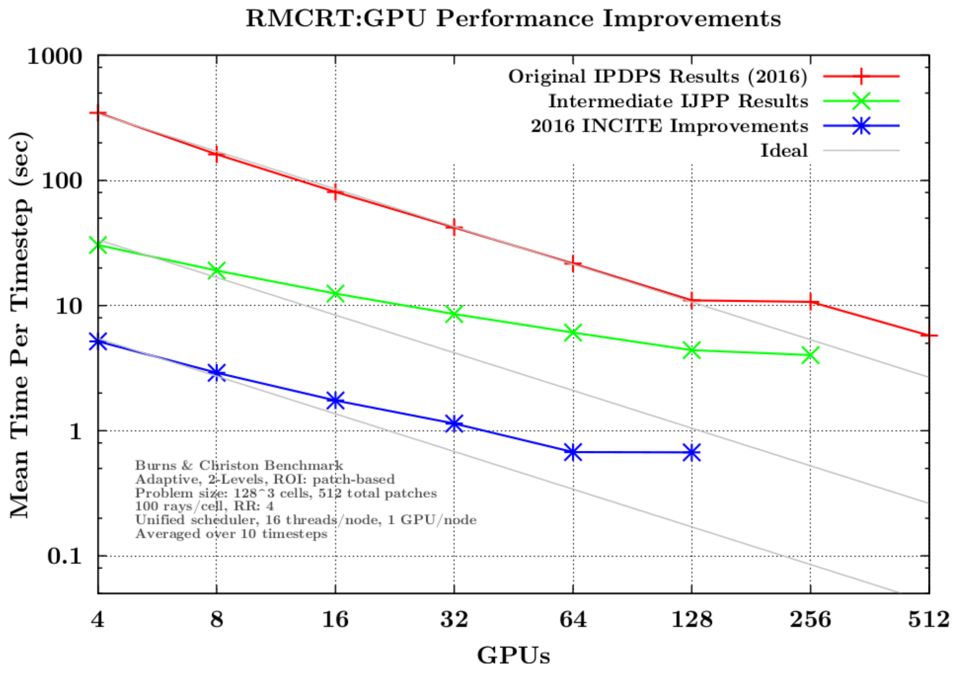

Figure 4: Speedups of one to two orders of magnitude of the Uintah GPU runtime from before this work until December 2016.

Generalizing Task Execution

Obtaining performance on GPUs is frequently achieved through many small blocks of kernel execution rather than large monolithic kernels. The GPU can distribute many small kernels among all of its streaming multiprocessors (**SMXs**). Using Uintah to over-decompose a problem into finer execution units is difficult due to finding a suitable decomposition geometry and increased dependency analysis among many more tasks. A solution was found where a task could split itself into many kernels, with each kernel executing on a different stream. This modification enables Uintah to saturate a GPU no matter how many SMXs it has. Figure 5 demonstrates that more kernels for a task do a better job of keeping a GPU occupied during a time step.

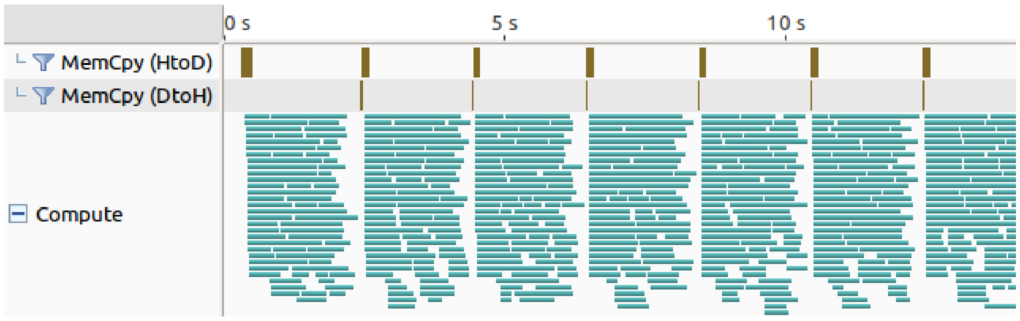

Figure 5: Splitting GPU task into many smaller kernels each with its own stream allows Uintah to occupy all GPU SMXs during the majority of a timestep. This configuration allows Uintah to run on GPUs with varying SMX amounts, freeing the application developer from retooling the decomposition to fit the machine. Computations performed on 16 RMCRT tasks per time step on an Nvidia K20c GPU.

Reduction of Dependency Analysis

In preparation for the recent production coal boiler run, the Uintah runtime had major performance deficiencies analyzing all dependencies among nodes in the presence of global data dependencies. Each task was responsible for determining what messages it must send out and what messages it must receive. The first trial run of the production problem performed this analysis phase in 4.5 hours. A dependency search algorithm was implemented to restrict dependency searches among tasks by finding maximum extents each task has within the simulation. For example, a task that simply computes simulation variables does not specify any halo extents. However, by analyzing how that simulation variable is used throughout other tasks, a vast swath of potential dependencies can be ruled out. In the production problem, dependency analysis was reduced 93% to 20 minutes, with most of the remaining time due to an unrelated problem finding cell extents among multiple adaptive mesh refinement layers.

Modifying Kokkos for Asynchrony

Kokkos's memory movement and kernel execution for Nvidia GPUs is similar to Uintah's GPU runtime two years ago. CUDA barriers are used extensively to avoid concurrency issues. Users using Kokkos must work with large, monolithic GPU kernels, which requires subdiving a functor into enough threads to properly fit within all GPU SMXs. This approach puts a difficult burden on application developers trying to properly size and partition problems. Further, with Uintah usually running each task on a 16^3 region of cells, it is difficult to subdivide such small regions to distribute throughout the GPU.

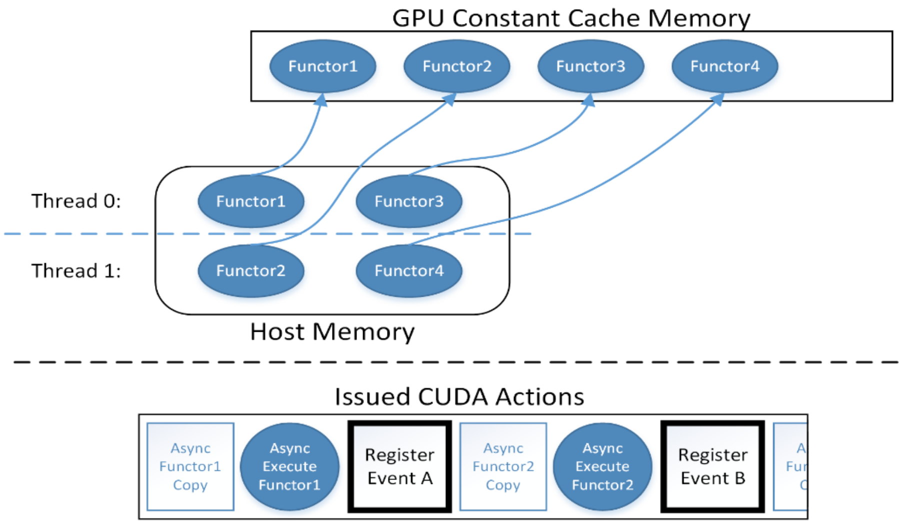

Figure 6: Multiple functors can be staged into GPU Constant memory, executed, and tracked with CUDA Events.

Recent work modified Kokkos to support asynchrony of Kokkos `parallel_for` code. Under-the-hood the Kokkos engine now allows multiple threads to claim a region of GPU constant cache memory, asynchronously copy the functor's information into that space, then launch the kernel to execute that functor. A visual representation of this is given in Figure 6. CUDA streams themselves are encapsulated into `Kokkos::Cuda` objects, and reference counting ensures that when all instances of the object are destructed the stream itself is reclaimed. This allows any developer using Kokkos `parallel_for` constructs to overlap kernels with only one minor modification to their existing code.

CHAPTER 5 – WORK PLAN AND IMPLICATIONS

Performance

Task Scheduler Modifications

Currently the OnDemand Data Warehouse allocates space on-the-fly during task execution for host memory simulation variables. To support sharing data objects and a unified code base, this model must change so that that simulation variables are allocated prior to task execution. Allocations should not happen through an `allocateAndPut()`, but rather before the task executes. This change requires that `allocateAndPut()` will simply act as a `get()` API call.

Any task simulation variables identified as `computes` should be pre-sized for upcoming halo cells based on the what can be inferred. Determining a proper size is a challenge as the initialization time step has no information as to what halo cells will soon be found in a normal time step. Resizing variables in GPU memory may also be a messy procedure.

Halo Gathering

Device halo cell gathering likely should be invoked host side through numerous device-to-device copies. Currently both James Sutherland and I have recognized unwanted performance deficiencies in the current approach of utilizing a GPU kernel to gather in halo cells.

Portability

Unifying the Data Warehouses into One Codebase

The OnDemand Data Warehouse and the GPU Data Warehouse have very different philosophies which need to be merged into one codebase. The OnDemand Data Warehouse largely avoids concurrency issues by duplicating simulation variables any time halo copying occurs or when two or more tasks share a simulation variable.

As Uintah is heavily implemented with OnDemand Data Warehouse code throughout, future work will preserve this class and merge GPU Data Warehouse concurrency and data sharing functionality into it. This may require making the underlying map data structure lock free. This work may be troublesome as the Uintah codebase is large and complex, with numerous code hacks for various edge cases.

Kokkos Modfications

Uintah utilizes the Kokkos `parallel_for` and `parallel_reduce` constructs. The work achieved for the `parallel_for` and for CUDA GPU tasks needs further investigation and implementation into Kokkos. The overall goal here is to deliver updates useful to the Kokkos project, and not just Uintah.

Uintah Integration with Kokkos

Results will be limited to existing tasks already written using CUDA code or Kokkos constructs. This work plan will not attempt to rewrite any existing task into GPU or Kokkos code. RMCRT is an excellent target task. We have tuned CPU code, Kokkos code, and GPU code. Arches has tasks written with CPU code and Kokkos code.

Some limitations will likely occur here as occasionally developers write tasks that conflict with Uintah philosophy. Five examples are 1) developers have been manually allocating their own simulation variables and performing their own halo cell gathers using a `getRegion()` call. 2) Application developers sometimes perform a Data Warehouse `get()` API call despite the task not previously listing that simulation variable. 3) A task may attempt to obtain both a CPU and GPU data pointer. 4) Developers utilize Uintah templated simulation variables with non-standard types (e.g. a CCVariable with a custom Vector object). 5) CPU and Xeon Phi tasks make OS API calls which are unavailable to

GPUs. For these reasons, it is very unlikely this work can achieve a full production run of the recent coal boiler multiphysics simulation on GPUs. This work instead will provide the mechanism so Kokkos enabled tasks can run on GPUs if properly designed.

CHAPTER 6 – THESIS FORMAT

My proposed thesis structure is as follows:

ABSTRACT

CHAPTER 1 – Introduction

A description for the motivation of this project, past contributions, and the prior state of Uintah's GPU functionality.

CHAPTER 2 – Related Frameworks and Tools

I foresee this chapter being like this document's overview of related tools, with the purpose of demonstrating the novelty of this research.

CHAPTER 3 – Uintah Overview and Application Developer API

Chapter 3 will contain an overview of the many moving parts that enable Uintah to function. A description will be given of all steps required for an application developer to create and execute CPU and GPU tasks.

CHAPTER 4 -- A Host and GPU Data Store Enabling Concurrency, Asynchrony, and Data Sharing

This chapter will contain an overview of the existing OnDemand Data Warehouse, including its strengths and limitations. One section will cover the GPU Data Warehouse including its iterations, with a demonstration at each iteration what new characteristics it achieved. This chapter should conclude with a data warehouse design using a unified codebase managing data in both host memory and GPU memory.

CHAPTER 5 – Uintah Heterogenous/GPU Task Scheduling and Execution

Task schedulers are the glue enabling the Uintah data store to achieve concurrency, asynchrony, and sharing of data. This chapter will analyze and report the many steps of a task lifetime: simulation variable preparation, halo cell gathering, task execution, and task cleanup.

CHAPTER 6 – Kokkos Code Modifications for Asynchrony of Data Movement and Execution

Kokkos has strong potential as a portable code solution for Uintah, but its GPU performance is currently basic and limited. In particular, Kokkos frequently utilizes synchronization barriers and has no mechanisms to support computation. This chapter will describe how Kokkos itself was modified to achieve full asynchrony.

CHAPTER 7 – Kokkos and GPU Integration Into Uintah

I desire this chapter to be the capstone of this work, demonstrating both performance and portability using production scale multiphysics tasks, while simultaneously minimizing application developer interaction with specific architectures. This chapter will highlight the ease which an application developer can create a task and performantly execute it on a GPU, CPU, or Xeon Phi. Some specific integration details are needed, such as transposing simulation variables and utilizing Kokkos Unmanaged Views.

CHAPTER 8 – Conclusions and Future work

APPENDIX

REFERENCES

CHAPTER 7 – CONCLUSIONS

The high-level goal of past and ongoing research is demonstrating both portability and performance of GPU enabled tasks for a production problem in the Uintah AMT runtime system. The portability goal should preserve Uintah's philosophy of requiring minimal runtime API interaction with the application developer while allowing tasks written with Kokkos to run on multiple architectures. The performance goal minimizes runtime wall time overhead and simulation variable duplication in memory.   When completed, this will make Uintah the first mature AMT runtime that has a combination of simulation variable management, GPU portability, performance, and ease of application task development.

REFERENCES

[1] M. Berzins, "Status of Release of the Uintah Computational Framework," Scientific Computing and Imaging Institute, Salt Lake City, Tech. Report UUSCI-2012-001, 2012.

[2] "Uintah Web Page," Scientific Computing and Imaging Institute, 2017. [Online]. Available: http://www.uintah.utah.edu.

[3] Computer Science Research Institute, Sandia National Laboratories, [Online]. Available: https://github.com/kokkos/kokkos. [Accessed July 2017].

[4] 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.shtm.

[5] D. Kuzman, "Runtime support for object-based message-driven parallel applications on heterogeneous clusters," Ph.D. Thesis, University of Illinois, Chicago, IL, 2012.

[6] "Legion Overview," Legion, [Online]. Available: http://legion.stanford.edu/overview/index.html. [Accessed 30 July 2017].

[7] "Explicit Ghost Regions," Legion, [Online]. Available: http://legion.stanford.edu/tutorial/ghost.html. [Accessed 30 July 2017].

[8] "HPX," [Online]. Available: https://github.com/STEllAR-GROUP/hpx. [Accessed 25 July 2017].

[9] H. Kaiser et al, "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 (PGAS '14)*, New York, NY, USA, 2014.

[10] "The OpenACC Application Programming Interface Version 2.5," OpenACC-standard.org, October 2015. [Online]. Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5.pdf. [Accessed 25 July 2017].

[11] "OpenACC," OpenACC-standard.org, [Online]. Available: https://www.openacc.org/about. [Accessed 25 July 2017].

[12] "GCC Wiki - OpenACC," [Online]. Available: https://gcc.gnu.org/wiki/OpenACC. [Accessed 25 July 2017].

[13] "The OpenCL Specification," Khronos OpenCL Working Group, [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf. [Accessed 25 July 2017].

[14] T. Sörman, "Comparison of Technologies for General-Purpose Computing on Graphics Processing Units," Master's Thesis, Linköping University, Linköping, Sweden, 2016.

[15] "Khronos Releases OpenCL 2.2 With SPIR-V 1.2," The Khronos Group Inc., 16 May 2017. [Online]. Available: https://www.khronos.org/news/press/khronos-releases-opencl-2.2-with-spir-v-1.2. [Accessed 25 July 2017].

[16] "CUDA C Programming Guide," Nvidia, 23 June 2017. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd. [Accessed 25 July 2017].

[17] M. Harris, "Unified Memory for CUDA Beginners," Nvidia, June 19 2017. [Online]. Available: https://devblogs.nvidia.com/parallelforall/unified-memory-cuda-beginners/. [Accessed 24 July 2017].

[18] J. Kraus, "An Introduction to CUDA-Aware MPI," NVidia, 23 March 2013. [Online]. Available: https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/. [Accessed 25 July 2017].

[19] R. Landaverde et al, "An Investigation of Unified Memory Access Performance in CUDA," in *IEEE conference on high performance extreme computing,* Waltham, MA, USA, 2014.

[20] Q. Meng, "Large-Scale Distributed Runtime System for DAG-Based Computational Framework," Ph.D. Thesis, University of Utah, Salt Lake City, UT, 2014.

PUBLICATIONS

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. 2016.

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

B. Peterson, H. K. Dasari, A. Humphrey, J.C. Sutherland, T. Saad, M. Berzins. "Reducing overhead in the Uintah framework to support short-lived tasks on GPU-heterogeneous architectures," In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC'15),* ACM, pp. 4:1-4:8. 2015.

B, Peterson, M. Datar, M. Hall, R. Whitaker. "GPU accelerated particle system for triangulated surface meshes," In *Proceedings of the Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, 2010.

Publications in Progress

B. Peterson, H. K. Dasari, A. Humphrey, J.C. Sutherland, T. Saad, M. Berzins. "Reducing overhead in the Uintah framework to support short-lived tasks on GPU-heterogeneous architectures," Submitted *International Journal of Parallel Programming*, 2016.

B. Peterson, A. Humphrey, J. Schmidt, M. Berzins. "Addressing Global Data Dependencies in Heterogenous Asynchronous Runtime Systems on GPUs", To be submitted to *Third International Workshop on Extreme Scale Programming Models and Middleware (ESPM2'17)*, 2017.