SAND2016-8097C

Performance Portability in the Uintah Runtime System Through the Use of Kokkos

Daniel Sunderland
Sandia National Laboratories,
Albuquerque, NM, 87175 USA
dsunder@sandia.gov

Brad Peterson
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
bpeterson@sci.utah.edu

John Schmidt
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
jas@sci.utah.edu

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

Jeremy Thornock
Institute for Clean and Secure Energy
University of Utah
Salt Lake City, UT 84112 USA
jeremy.thornock@utah.edu

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

ABSTRACT

The current trend towards diversity in nodal parallel computer architectures is seen in recent and proposed machines based upon multicore CPUs, GPUs and the Intel Xeon Phi as well as a number of emerging designs with large core counts per node. A class of approaches for enabling scalability of complex applications on such a broad range of architectures is based upon Asynchronous Many Task software architectures such as that in the Uintah computational framework used for the parallel solution of solid and fluid mechanics problems. Uintah is structured with an application layer with its own programming model and a separate runtime system. While Uintah scales well today, it is necessary to address nodal performance portability in order to continue to do so on future architectures. The principal contribution of this work is to show how both Uintah's runtime system and its programming model may be incrementally modified to use the Kokkos performance portability layer. Results from experiments shows that in the parts of Uintah that were refactored to conform to the Kokkos programming model, performance improved by more than a factor of two.

Keywords

Uintah, Kokkos, hybrid parallelism, performance portability, parallel

1. INTRODUCTION

A current trend in large scale computing is towards larger core counts per compute node. Whether this is through the use of GPUs, Xeon Phis or through standard/lightweight cores. One software approach that helps in the scaling of complex applications codes on such diverse architectures is based upon an Asynchronous Many Task (AMT) approach in which tasks are executed as soon as their dependencies are met. A number of such approaches Charm++, Legion and Uintah are compared by [2], but there are many other AMT codes under development.

Uintah enforces separation between the applications' tasks and the runtime system which executes them. This allows applications developers to focus on writing tasks for discretizing the partial differential equations of solid and fluid mechanics on a local set of block-structured, adaptive mesh patches without needing to specify how those tasks are to be executed. When the runtime system executes the applications' task it resolves details such as automatic MPI message generation, management of halo information (*ghost cells*) and the life cycle of data variables, and other details inherent in large scale, heterogeneous, parallel computer systems. A key feature of Uintah is that variables are stored in a part of the runtime system known as the data warehouse, which is accessed by each task to provide the data that it needs to work on. Tasks access data from the previous timestep using the *old data warehouse* and store data computed during the current timestep in the *new data warehouse*.

The Uintah software is described by [3, 6, 20, 22, 23]. Examples of the scalability of Uintah are given by [4] and the open-source software itself may be obtained from [30]. Uintah currently scales complex applications on a variety of CPU core based architectures up to about 700K cores. However a challenge of porting over 1M lines of highly templated C++ to either GPU or Xeon Phi architectures has meant that until recently only a straightforward port has been done [21]. Or that computationally intensive kernels, such as the Reverse Monte Carlo Raytracing approach for thermal radiation, have been bifurcated to support multiple architecture like GPUs and Xeon Phis [11, 13].

One way to address the portability challenge for code such as Uintah is to use a performance portability layer based upon a many-core parallel programming model [18], such as OpenMP, OpenACC, RAJA, Kokkos or OpenCL. A detailed comparison by [18] shows the strengths and weaknesses of these approaches. In this work we have chosen to use Kokkos [5,7,8] as it fits most easily with the underlying code philosophy of Uintah. In using Kokkos it is necessary to rewrite tasks into a form that allows Kokkos to map the computation and data in the most appropriate way to achieve performance

ESPM2 '16 November 13–18, 2016, Salt Lake City, UT, USA

DOI: **.***/***_*

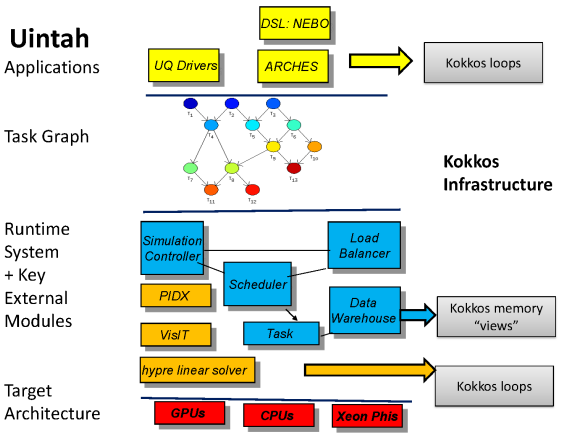

Figure 1: Uintah Structure with Kokkos

on the target architecture. Kokkos does this mapping at compile time through use of C++ template meta programming. The challenge in using Kokkos in Uintah is that both the user code through modified loop structures and the data warehouse through changed data structures must be refactored. The broad approach to changing the Uintah framework is shown in Figure 1. This figure shows that the applications' codes (primarily the Arches combustion code for this paper) are required to rewrite their loops to conform to the Kokkos programming model. The Uintah data warehouse also needs to change to support the use of Kokkos data structures. We note that Uintah makes extensive use of the hypre linear solver [9] and that it too (as far as we know) is in the process of being modified to be compatible with the Kokkos programming model. The aim of this paper is thus to show how the Uintah's runtime system and application programming model may both be modified to use the Kokkos performance portability layer. Results from experiments demonstrate that Uintah code rewritten to conform to the Kokkos programming model improves in performance, with result seen up to a factor of five.

We begin by giving an overview of the Uintah framework in Section 2. The Arches applications code driving a major part of our present development is described in Section 3 and Kokkos itself is described in Section 4 The precise details of how Uintah is modified, both at the applications and at the runtime system level with the work done to modify data warehouse to work with Kokkos are given in Section 5. The results from these improvements are shown in Section 6 and the paper concludes in Section 7 with discussion on future work.

2. UINTAH OVERVIEW

This description of the open source Uintah framework [3], [30], follow that in [27]. Uintah is used to solve problems involving fluids, solids, combined fluid-structure interaction problems, and turbulent combustion on multi-core and accelerator based supercomputer architectures. Problems are either initially laid out on a structured grid as shown in [15] with the multi-material ICE code for both low and high-speed compressible flows, or by using particles on that grid as shown in [1] with the multi-material, particle-based code MPM for structural mechanics. Uintah also provides the combined fluid-structure interaction (FSI) algorithm MPM-ICE [10], the ARCHES turbulent reacting CFD component [14] designed for simulating turbulent reacting flows with participating media radiation, and Wasatch, a general multiphysics solver for turbulent reacting flows. For the work described, the Arches component is of primray interest here.

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

With the availability of on-node GPUs, Data Warehouses specific to GPUs are used. Uintah task schedulers are responsible for scheduling and executing both CPU and GPU tasks, memory management of data variables, and invoking MPI communication. Parallelism within Uintah is achieved in three ways. First, by using domain decomposition to assign each MPI rank its own region of the computational domain, e.g. a set of hexahedral patches, usually with spatial contiguity. Second, by using task level parallelism within an MPI rank to allow each task to run independently on a CPU (or Xeon Phi) core or available GPU, and third, by utilizing thread level parallelism within a GPU.

Uintah maintains a clear separation between applications code and its runtime system, and hence the details of the parallelism Uintah provides through its runtime system are hidden from the application developer. A developer need only supply Uintah with a description of the task which would run serially on a single patch, namely what variables it will compute, what variables are required from the previous time step, and how many layers of ghost cell data are needed for a variable. The task developer must supply entry functions to his or her task code, and writes serial C++ code for CPU and Xeon Phi tasks and CUDA parallel code for GPU tasks. The present model for GPU-enabled tasks currently requires that two versions of the task code be maintained, one for CPU code and one for GPU code with possibly even a third version being needed to fully exploit the new Intel Knights Landing architecture. As outlined above, the use of Kokkos enables a move to a single code and allows users to exploit data parallelism within all Uintah tasks. Examples of this is shown in Section 6, where we focus on stencil computation examples related to the Arches code [14].

3. ARCHES COMBUSTION SOLVER

The primary motivation for much of this work is to develop and improve upon our existing Uintah computational framework for solving emerging exoscale problems with important commercial ramifications and benefits for improving coal combustion efficiency. This work has been the basis for a 2015-2016 INCITE Award which is to use computational science to predict capabilities for a commercial, 1000 MW coal fired boiler. The goals of the project include the assessment of mixing and combustion efficiency, minimize energy imbalances and predict heat flux patterns to the furnace walls. Given this challenging application, the improvements to the Uintah programming model and runtime system will be demonstrated through the Arches combustion component.

Arches is a finite volume combustion code that has been developed over a number of years [14,28,31]. Traditional Lagrangian/RANS approaches do not address very well the particle effects that are necessary for the simulations with strong particle effects. The use of the Large Eddy Simulation (LES) approach of Arches has potential to be an important design and prediction tool.

The approach used in Arches is that of a structured, high order finite-volume mass, momentum, energy conservation discretization method for the gas and solid phase with combustion [14, 24, 25]. The modeling of the all-important soot particles in Arches is done via DQMOM (which utilize many small linear solves) [25]. Using the low Mach number approximation results in the formulation of the Pressure Poisson equation that requires the solution of a matrix with up to 10^{12} variables. The approach adopted here is to use the hypre code with geometric multigrid preconditioning conjugate gradients with a red-black Gauss Seidel Smoother, which has shown excellent scalability [29]. Momentum and species transport is handled by a dynamic Smagorinksy closure model and accounts for approximately 30% of the overall computational time in a typical timestep. Focusing on the scalar transport equations as an easy adoption point for our Kokkos work since it has relatively straightforward data structures and loops that allowed us to characterize and better understand the development issues for wider Kokkos adoption. Later sections of this paper describe the challenges and accomplishments we faced in obtaining performance for key loops using Kokkos.

The energy balance includes the effect of radiative heat-loss/gains in the IR spectra. Initially, the radiative intensity equations are solved using a Discrete-Ordinates solver [16] that involved solving a linear system of equations using hypre [17]. This involves very large and time consuming solves every 10-25 timesteps with the hypre code requiring upwards to 50% of the runtime for the radiation timestep. To mitigate these time consuming solves and to take advantage of GPU accelerators on current petascale and future exoscale systems, we are adopting the Reverse Monte Carlo ray tracing-based radiation model (RMCRT) which uses both CPUs and GPUs for the radiation calculation in lieu of the discrete ordinates solver [12, 13]. Both Discrete Ordinates and RMCRT are currently being used or will be used shortly in design production runs on the large scale coal boiler as part of ongoing work on our INCITE Award.

4. KOKKOS AND PERFORMANCE PORTABILITY

Kokkos is a C++11 library for implementing portable thread-parallel codes on various HPC architectures [5, 7, 8]. Kokkos is used to optimize single-node performance, since most HPC codes already have strategies to optimize their intra-node performance. It currently supports CPU, GPU, Intel Xeon Phi and IBM Power 8 architectures. The source code is open source and is freely available at https://github.com/kokkos/kokkos.

Kokkos allows users' to encapsulate their code into computational kernels, and uses template meta-programming to optimize their kernels at compile time for the given device. Kokkos is able to optimize users kernels because it requires them to conform to abstractions provided by the Kokkos API. The main abstractions within Kokkos are *Parallel Patterns*, *Execution Space*, *Execution Policy*, *Views*, *Memory Space*, *Memory Layout* and *Memory Traits*.

The user can specify a kernel which only uses a subset of these abstractions, and the others will default to optimal values for the current device. The *Parallel Pattern* describe what type of kernel

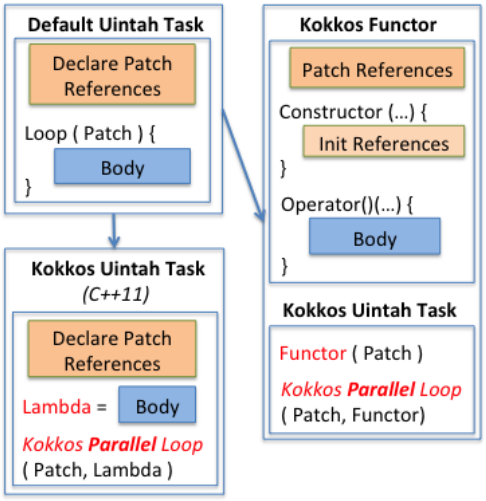

Figure 2: Kokkos Paths for Uintah Code

the user wishes to execute be it a **parallel_for**, **parallel_reduce** or **parallel_scan**. The *Execution Space* informs the compiler about where the kernel is to be run, i.e., GPU or CPU cores, and the *Execution Policy* dictates how a kernel should be executed in the given Execution Space.

Since most scientific codes store data in multi-dimensional arrays, Kokkos provides *Views*, which are light-weight, reference counted multi-dimensional arrays. Emerging HPC architecture have deep memory hierarchies so Kokkos Views allow the user to specify in which *Memory Space* the array exists. *Memory Layout* dictates how the array is mapped to memory (row-major, column-major, tiled, etc), and it is critical for performance that the memory layout is suitable for the given device. For example, CUDA prefers arrays that use a column-major layout to optimize memory access, while most CPU code use a row-major layout to obtain a better SIMD vectorization. Using the wrong layout can have significant performance penalties. *Memory Traits* provides additional information about how the views are allocated or used and can enable other compile-time optimizations. By using views, Kokkos is able to separate the data locality and layout from the computational code. This avoids architecture-specific code implementations, since Kokkos is then able to select the best memory layout and execution policy at compile time for the given architecture.

To use Kokkos a user identifies a parallelizable grain of computation and data. This grain of code is called the compute kernel. A user can used C++11 lambdas or create function objects (functors) to encapsulate a kernel. Kokkos then maps the computations onto cores and the data onto memory using the execution and memory spaces. The user is responsible for writing thread-scalable, high-performance kernels. Carefully written kernels can obtain portable SIMD auto vectorization, as is shown in Section 6.

5. MODIFYING UINTAH TO USE KOKKOS

While the Uintah AMT runtime system hides latency and maximizes network throughput, emerging architectures require leveraging even more on-node parallelism to exploit as much of their performance as is realistic.

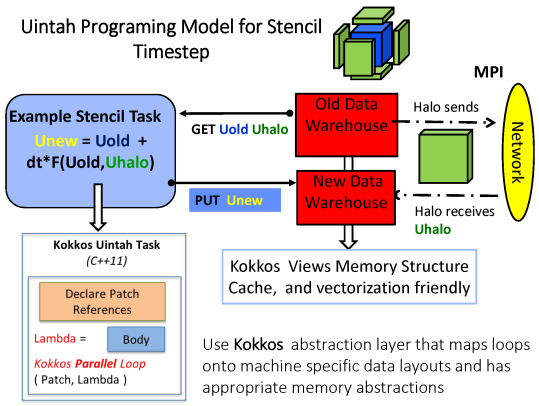

Figure 3: Modified Uintah Programming Model

5.1 Overview of Uintah Modifications

Uintah, like many HPC codes, has a large legacy code base with limited support and development resources. To refactor Uintah to fully utilize Kokkos kernels is a multi-year effort that involves rewriting significant parts of the runtime system, and refactoring component codes to encapsulate work within computation kernels. Most of the work involves refactoring loops into parallel kernels and converting existing array data types into Kokkos views. Figure 2 shows several options for refactoring loops within Uintah tasks, and Figure 3 shows how Uintah is modified overall to use Kokkos at both the data warehouse and user task level. It is desirable to do this refactor incrementally without further bifurcating the Uintah code base. Also, when refactoring Uintah component codes we have been able to take advantage of new and experimental, but planned future Kokkos features. For example we have used an experimental planned Kokkos parallel three-level loop for the final example in Section 6. It is important to stress that for a large code such as Uintah these changes must be introduced incrementally. This requires significant planning to enable migration the current Uintah APIs to a subset that will be portable between the various device architectures.

To achieve device portability, the array data used by Uintah is to be replaced with Kokkos views. Since the Uintah runtime system is responsible for maintaining the data warehouses, the runtime system needs to be extended to return Kokkos views in place of the current Uintah array data structures. Most codes are hard coded to specific data structures, and changing them takes significant effort. Using the *Unmanaged* memory trait, the runtime system can wrap the existing data structures with Kokkos unmanaged views. Unmanaged views do not include reference counting, and must be supplied a layout and memory space. Unmanaged views allow codes to incrementally adopt Kokkos APIs without requiring a massive upfront rewrite. The runtime system and component codes can then incrementally track down instances where non-view APIs are being used to refactor them individually to remove the assumptions that make them incompatible with pure Kokkos views. After these incompatibilities are removed the code should then be portable to other architectures.

However, codes which use unmanaged views are not portable to different devices, so it is necessary for the user to incrementally verify the portability of kernels to other devices without waiting for the entire code base to be refactored. One technique that we have used successfully is to extract kernels into stand-alone executables with mock inputs. The kernels can then be compiled for various devices and optimized to run better on those devices. When doing this for a diffusion kernel within ARCHES, we were able to obtain good SIMD vectorization on CPUs and better caching effects on GPUs. It is desirable that these stand-alone executables uses the same code as the production executable.

By leveraging C++11 features like "auto", component codes can be written without explicitly specifying the array data types. An example of this can be seen in Code Listing 3. This allows components to assume that the array type conforms to a standard API without needing to know the exact type. By using these features, the runtime system will be able to change the array data types without requiring that the component codes be updated again.

The process of converting a component task into a Kokkos compatible computational kernel is shown in Figure 2. For example, Uintah tasks declares and initializes mesh patch array variables which are then used within one or more loops throughout the execution of the task. Using C++11 the loop bodies of the tasks are encapsulated in lambdas, which are then invoked with the appropriate parallel pattern. The loop bodies could also be extracted into a function object, and then invoked by the parallel pattern. This entire process is extremely incremental, but allows for performance and portability verification at each step. Furthermore, this approach has yielded some performance improvements on existing architecture for production runs. Code examples and results are shown in Section 6.

5.2 DataWarehouse Changes

Uintah currently has two separate data warehouses which have effectively become bifurcated, and work has commenced to unify these back into one. Uintah uses a host memory data store known as the *On-Demand Data Warehouse* [19]. A task developer simply requests simulation variables from the data store using a get() and put() API interface. A task developer is not involved with the underlying preparation of simulation variables or halo transfer both on-node and MPI. Recently a GPU memory data store known as the *GPU Data Warehouse* [22] was created to allow task developers to write CUDA enabled tasks which could request simulation variables from a data store while in CUDA kernel code. It followed a similar philosophy of data management, removing the task developer from the runtime system details.

The GPU Data Warehouse was initially created only to provide limited features as a proof of concept supporting only a handful of simulations. In the past two years the capabilities of the GPU Data Warehouse have significantly increased to support a wide variety of simulation problems and simulation variables on both host and GPU memory. As a result, overhead wall clock times have been significantly reduced [26]. The result of these changes increased the GPU Data Warehouse's codebase and API to the point where its concepts have become similar to that as the On-Demand Data Warehouse. Despite surface similarities between the two data stores, the underlying code utilizes distinctly different designs. Now when new API features are requested, we often must implement the functionality twice, once for each data store, with each one needing a platform-specific implementation. This is not sustainable.

Fortunately this bifurcation of the data stores has allowed us to identify good design patterns and to use these insights to begin implementing a *Unified Data Warehouse*, which seeks to be a single data store for not only host and GPU memory but also other memory hierarchies such as NVRAM. This approach additionally allows multiple options for memory layouts, storage locations, and padding configurations. The remainder of this section outlines how this new data store is implemented, and what influenced these design decisions.

The Unified Data Warehouse opts for an internally simpler design, using a simple array of metadata, and a single atomic bitset per variable to track a variable's status. While Uintah has many sections of code which still utilize mutex protected data structures, their frequency of usage has become problematic. It is often difficult to extend the code without carefully analyzing how the locks are used. In regions where we have removed mutex locks and replaced them with data structures allowing for atomic reads and compare-and-swap writes, we generally experience fewer issues in development, and code refactoring proceeds more quickly. For this reason, the Unified Data Warehouse is designed to not require any mutex locks. Associative arrays are dropped in favor of regular arrays. Instead of multiple booleans or multiple bitsets, the Unified Data Warehouse uses a single bitset per simulation variable to track the variable's status.

At present, the On-Demand Data Warehouse stores a collection of polymorphic objects instead of a collection of meta data. Successive developers over the years felt it natural to extend these objects to perform multiple duties. From an Object-Oriented Programming perspective, these concepts seem sensible. For example, instead of having the Uintah runtime system perform MPI sends for simulation variables, the objects themselves perform all the logic instead. A challenge in moving away from this design is that Uintah's automated MPI halo transfer engine is deeply intertwined with these polymorphic simulation variable objects, where each simulation variable creates a buffer and deep copies halos into it. This is then sent to the On Demand Data Warehouse in host memory. With our Unified Data Warehouse approach, no deep copies to a buffer are made and the Uintah runtime system manages MPI completely, not the variable objects. These changes enable the implementation of CUDA-Aware MPI.

With the original data store approach, halo cells are gathered into the simulation variable on the CPU which happens on the fly as part of get() API call. Larger variables are then created and the deep copies, with halo information, commence. With the GPU data store, the Uintah task scheduler initiates halo gathering prior to task execution and the task developer needs to pre-size variable to ensure it has room. With our Unified data store approach, Uintah automatically accounts for padding to prepare for the halo data.

The Unified Data Warehouse allows for easier transition to Kokkos parallel code. Instead of writing a CPU task and also a similar GPU task, now a developer can write only one task containing Kokkos code. In contrast to requesting data from the On-Demand Data Warehouse or the GPU Data Warehouse, the developer need only to request data from the Unified Data Warehouse using a get() interface, which retrieves objects containing Kokkos Views, as shown in Figure 3. From here the task developer can proceed in writing Kokkos parallel code with a minimal need to know about where that code will be executed. Uintah will know the execution space requested for that task, and prior to task execution Uintah will prepare the Kokkos Views in those simulation variables to have the necessary data in the desired memory spaces. Thus a task developer can simply request that task to run on CPU cores, or on Nvidia GPUs, or on Intel Xeon Phis, etc., and Uintah manages the remaining details.

6. RESULTS

Three examples are used to test the performance of the runtime system using Kokkos on key Uintah algorithms. The first example is a simple Laplace's equation example while the second is that of a nonlinear advection scheme and finally the third is a 3D loop in the Arches code [14].

6.1 Laplace's Equation Example

Using experimental features from the Kokkos pthread execution space we were able to utilize data parallelism within asynchronous Uintah tasks. A simple Laplace equation with a seven point finite difference stencil example showed speedups around a factor of two as seen in Table 1. The conversion of this code was extremely simple, and only required encapsulating the loop body within a lambda and calling a modified version of the Kokkos **parallel_for** which iterates over multiple indices. Promising results such as these have encouraged us to extend our use of Kokkos into the full Arches code.

Domain size	128^3	256^3	512^3
Original(ns)	1.521	1.978	1.651
Kokkos (ns)	0.723	0.840	0.833

Table 1: Kokkos timings on Laplace's equation

6.2 Arches Advection Example

The Arches code is the computational component of the large scale DOE coal boiler simulation that is a primary driver for moving Uintah beyond petascale. In Arches 30-40% of the code is spent on model evaluation, discretization of transport and other flow components. Kokkos is a natural fit for Arches because it is possible to achieve lamda/functorization of existing code with relatively little work. Fast initial adoption is very helpful for our engineering developers. This process is illustrated by the discretization of the simple advection component corresponding to transport of a simple scaler. In experiments we looked at many different, but standard, approaches such as upwinding and flux limiting. In this case speed-up measured for a standard upwinding discretization from existing baseline code against the Kokkos code, using unmanaged views. The speedup for different patch sizes are shown in Table 2. The upwind and the van Leer flux limiter show signficant speedups over the original Uintah implementation. The van Leer result speedup is not as large as the upwind result due to the number of branches (1 versus 5) in the computational kernel. The very significant speedups that are shown are a result of two complementary changes, the first being the use of the Kokkos **parallel_for** and the improved way in which Kokkos iterates through the memory space as compared to the original Uintah implementation. And the second significant reason is the reimplementation of the computational kernel to be more performant. This example suggests that careful rewrites of key computational kernels in conjunction with Kokkos can offer significant performance improvements. Profiling and understanding data layouts are significant factors towards improving performance in computationally intense routines.

Patch size	8^3	16^3	32^3	64^3	128^3
Upwind Kokkos Speedup	4.6	10.0	10.7	12.9	12.7
van Leer Kokkos Speedup	2.76	4.05	4.04	5.01	6.37

Table 2: Kokkos speedup on Arches advection

6.3 Arches 3D Stencil Example

In testing parts of the Uintah code we note that the conversion would need to be entirely completed for all the code in a physics

component before Uintah could run those tasks on other HPC architectures like a GPU. Developing a simple mock runtime system which allows individual tasks to be tested independently is vital for ensuring that the compute kernels are portable to other devices, without waiting for a full code refactor.

Using the technique of creating a simple mock runtime system, we were able to verify that the diffusion kernel in Arches is portable between GPU, CPU, and Xeon Phi devices. With minor modifications to the kernel we were able to optimize it in the mock system to ensure that it leverage SIMD vectorization. With a slight change to our build system we could have done that optimization on the production code, so that we can test portability for each kernel independently, while ensuring that it still runs in the production environment.

The loop used is a simple diffusion kernel which amounts to the convolution of 1D stencils for 3 face centered variables X, Y, Z with 3D stencils of 2 cell centered variables D, phi. The initial Uintah code for this loop uses Uintah arrays and iterators. Uintah arrays are indexed with an IntVector representing an (i, j, k) tuple. Uintah Iterators are initialized with low and high IntVectors and will iterate over the indicated range in a column-major order. The initial Uintah code is show in Code Listing 1. The Uintah framework used the concept of a single loop iteration with IntVectors as an aid to the development of the computational algorithms for the application developers. These techniques were optimized to assist in the development and debugging of application algorithms. The indirection and pointer hops that occur in the IntVector and loop traversal are non-ideal from a performance standpoint, but offer significant benefits to initial algorithm development. With the advent and incorporation of the Kokkos library, the transition from easy development using Uintah loop iteration and IntVectors is ongoing. While the benefits of the Uintah constructs are numerous from an algorithm development point of view, the drawbacks to raw performance are reflected in Table 3 and show that rewriting the kernels with the Kokkos constructs and using techniques to promote SIMD vectorization can offer significant performance improvements.

```
typedef IntVector IV;
for( Iterator itr(low,high); !itr.done(); ++itr)
    {
  IV c=*itr;
  IV xp=c+IV(1,0,0), xm=c+IV(-1,0,0);
  IV yp=c+IV(0,1,0), ym=c+IV(0,-1,0);
  IV zp=c+IV(0,0,1), zm=c+IV(0,0,-1);
  rhs[c] +=
     ax*(X[xp]*(D[xp]+D[c]) *(phi[xp]-phi[c])
        -X[c] *(D[c] +D[xm])*(phi[c] -phi[xm]))
   +ay*(Y[yp]*(D[yp]+D[c]) *(phi[yp]-phi[c])
        -Y[c] *(D[c] +D[ym])*(phi[c] -phi[ym]))
   +az*(Z[zp]*(D[zp]+D[c]) *(phi[zp]-phi[c])
        -Z[c] *(D[c] +D[zm])*(phi[c]-phi[zm]));
}
```

Code Listing 1: Uintah 3D Stencil Kernel

There are three step to naively convert a Uintah kernel to Kokkos. First, the iterators loops are replaced with a parallel algorithms over the same range. Second, IntVector indexing is replaced with direct i, j, k lookups. Lastly, Uintah arrays are wrapped and replaced with unmanaged Kokkos views. Using unmanaged views allow for an incremental transition to Kokkos, though to achieve performance portability these views will need to become managed Kokkos views in the future. The naive Kokkos loop is shown in Code Listing 2.

```
parallel_for( range,[=](int i,int j,int k){
  rhs(i,j,k) +=
    ax*(X(i+1,j,k)
       *(D(i+1,j,k)+D(i,j,k))
       *(phi(i+1,j,k)-phi(i,j,k))
     -X(i,j,k)
       *(D(i,j,k)+D(i-1,j,k))
       *(phi(i,j,k)-phi(i-1,j,k)))
   +ay*(Y(i,j+1,k)
       *(D(i,j+1,k)+D(i,j,k))
       *(phi(i,j+1,k)-phi(i,j,k))
     -Y(i,j,k)
       *(D(i,j,k)+D(i,j-1,k))
       *(phi(i,j,k)-phi(i,j-1,k)))
   +az*(Z(i,j,k+1)
       *(D(i,j,k+1)+D(i,j,k))
       *(phi(i,j,k+1)-phi(i,j,k))
     -Z(i,j,k)
       *(D(i,j,k)+D(i,j,k-1))
       *(phi(i,j,k)-phi(i,j,k-1)));
});
```

Code Listing 2: Naive Kokkos 3D Stencil Kernel

```
parallel_for( range, [=](int i, int j, pair<int,
    int> k_range) {

auto r=subview(rhs,i,j,ALL());
auto x0=subview(X,i,j,ALL());
auto xp=subview(X,i+1,j,ALL());
auto y0=subview(Y,i,j,ALL());
auto yp=subview(Y,i,j+1,ALL());
auto z=subview(Z,i,j,ALL());
auto d00=subview(D,i,j,ALL());
auto dm0=subview(D,i-1,j,ALL());
auto dp0=subview(D,i+1,j,ALL());
auto d0m=subview(D,i,j-1,ALL());
auto d0p=subview(D,i,j+1,ALL());
auto p00=subview(phi,i,j,ALL());
auto pm0=subview(phi,i-1,j,ALL());
auto pp0=subview(phi,i+1,j,ALL());
auto p0m=subview(phi,i,j-1,ALL());
auto p0p=subview(phi,i,j+1,ALL());

 parallel_for( krange, [&](int k) {
  r(k) +=
   ax*(xp(k)*(dp0(k)+d00(k))*(pp0(k)-p00(k))
     - x0(k)*(d00(k)+dm0(k))*(p00(k)-pm0(k)))
  +ay*(yp(k)*(d0p(k)+d00(k))*(p0p(k)-p00(k))
     - y0(k)*(d00(k)+d0m(k))*(p00(k)-p0m(k)))
  +az*(z(k+1)*(d00(k+1)+d00(k))*(p00(k+1)-p00(k))
     - z(k)*(d00(k)+d00(k-1))*(p00(k)-p00(k-1)));
 });
});
```

Code Listing 3: SIMD Kokkos 3D Stencil Kernel

Optimizing this kernel to allow SIMD auto vectorization requires extracting 1D subviews from the 3D arrays views. The Kokkos subview function creates a new view from an existing view given ranges of indices, similar to subview operations on Matlab arrays. Using C++11 *auto* we are able to represent these subviews without needing to know the exact type of view that Kokkos returns, this allows Kokkos to optimize the resulting view for the given context. It is important to extract these 1D subviews so that the compiler knows that we are using a stride-one memory access pattern on the CPU in the inner loop so that it can correctly identify the loop as a candidate for vectorization (assuming that the arrays are laid out in row-major order on the CPU). The inner array is then implemented with another **parallel_for** loop which only depends on the k_{th} index. The user is responsible for verifying that there are no loop carry dependencies in the inner loop. The speedups of the SIMD kernel over the initial Uintah kernel can be

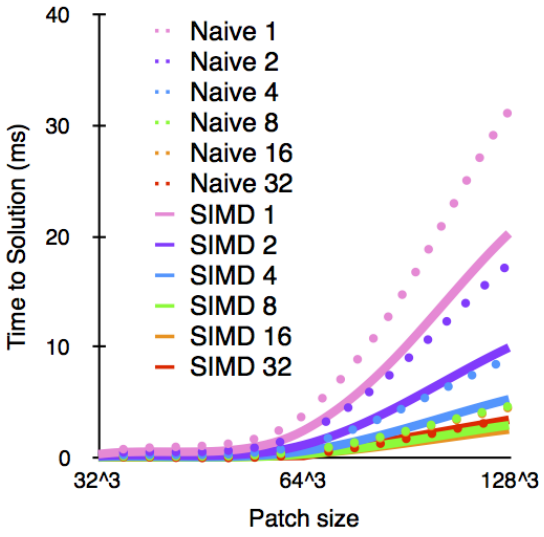

Figure 4: Results on 3D stencil Example. Kokkos kernels with given number of CPU Threads

		32^3		64^3		128^3	
		ms	**x**	*ms*	**x**	*ms*	**x**
Serial Uintah		1.063	1.0	8.041	1.0	64.86	1.0
Kokkos	Naive	0.649	1.6	4.30	1.9	36.06	1.8
Serial	SIMD	0.313	3.4	2.47	3.3	20.21	3.2
Kokkos	Naive	0.166	6.4	1.165	6.9	8.943	7.3
4 Threads	SIMD	0.081	13	0.583	14	5.275	12
Kokkos	Naive	0.068	16	0.545	15	4.508	14
16 Threads	SIMD	0.044	24	0.313	26	2.542	25
Kokkos	Naive	0.037	29	0.279	29	3.517	18
32 Threads	SIMD	0.025	43	0.163	49	3.421	19
Kokkos	Naive	0.090	12	0.210	38	0.614	105
CUDA	SIMD	0.090	12	0.210	38	0.628	103

Table 3: Results on 3D stencil Example. Albion: 2 numa/16 cores/32 threads, avx, Intel Xeon CPU E5-2660 0 @ 2.20GHz 32 GB Main Memory GeForce GTX TITAN X capability 5.2, Total Global Memory: 12 GB

seen in Table 3. These experiments were run on an Intel Xeon with a SIMD vector length of 2 yielding an ideal speedup of 2X of the Kokkos SIMD kernel over the Kokkos naive kernel. The results in Table 3 demonstrate that with careful rewrites of computational kernels with techniques that promote vectorization, it is possible to achieve the ideal speedup of 2X (1.8X- 2.3X) for sufficient workloads. We believe that the caching effects contributed to the speedup of 2.3X. The CUDA results shown in the table are present to show that the changes required to the diffusion kernel to get SIMD vectorization do not negatively impact the vectorization that CUDA already achieves.

Since this kernel is relatively light in computation compared to the number of memory load it requires, it is limited by memory bandwidth. Once the memory bandwidth is saturated, the time to solution of the SIMD kernel converges to the naive kernel as seen in Figure 4. So the naive and SIMD kernels converge to the same solution time as number of threads increase.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have shown how it is possible to introduce the Kokkos performance portability layer into a sophisticated AMT runtime in the Uintah software. The introduction of Kokkos not only involved rethinking the design of the Uintah nodal data warehouse but also of changing loops in the applications model, albeit in a straightforward way. While this process is still ongoing the initial experiments conducted show the promise of Kokkos as a means of providing present and future performance portability for the Uintah software. We have shown that with a naive incorporation of Kokkos on a standard cpu core offers anywhere from 2X speedups, but with careful rewrites of key computational kernels combined with Kokkos can yield upwards to 12X speedups. We have also shown that the portability features of Kokos enable speedups of up 30x to 50x using multiple cores and threads or GPUs. The process of adopting Kokkos into the Uintah framework offers an iterative path forward for improved performance that begins with an adoption of various Kokkos constructs initially with an ongoing process of rewriting key Arches computational kernels in a more performant manner to achieve the significant performance improvements that we have seen thus far.

8. ACKNOWLEDGMENTS

Funding from DOE NNSA is gratefully acknowledged. The work of Peterson, Berzins, Humphrey, Schmidt, and Thornock is supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. We would also like to thank all those involved with Uintah past and present. John Holmen is thanked for the Laplace results in Section 6.1 and Erik Lindstrom is thanked for Arches Advection results in Section 6.2. Dan Sunderland was supported by Sandia National Laboratories on a Ph.D. studentship at the University of Utah.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

9. REFERENCES

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

[2] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer, and T. Gamblin. Asc atdm level 2 milestone 5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical report, Sandia National Laboratories, 2015.

[3] M. Berzins. Status of release of the uintah computational framework. SCI Technical Report UUSCI-2012-001, SCI Institute, University of Utah, 2012.

[4] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, , and C. Wight. Extending the uintah framework through the petascale

modeling of detonation in arrays of high explosive devices. *SIAM Journal on Scientific Computing (Accepted)*, 2016.

[5] H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos. *J. Parallel Distrib. Comput.*, 74(12):3202–3216, Dec. 2014.

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

[7] 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*, PMAM '12, pages 1–10, New York, NY, USA, 2012. ACM.

[8] H. C. Edwards and C. R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Proceedings of the 2013 Extreme Scaling Workshop (Xsw 2013)*, XSW '13, pages 18–24, Washington, DC, USA, 2013. IEEE Computer Society.

[9] R. Falgout, J. Jones, and U. Yang. *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume UCRL-JRNL-205459, chapter The Design and Implementation of Hypre, a Library of Parallel High Performance Preconditioners, pages 267–294. Springer-Verlag, 51, 2006.

[10] J. E. Guilkey, T. B. Harman, A. Xia, B. A. Kashiwa, and P. A. McMurtry. An Eulerian-Lagrangian approach for large deformation fluid-structure interaction problems, part 1: Algorithm development. In *Fluid Structure Interaction II*, Cadiz, Spain, 2003. WIT Press.

[11] J. K. Holmen, A. Humphrey, and M. Berzins. Exploring use of the reserved core. In J. Reinders and J. Jeffers, editors, *High Performance Parallelism Pearls*, pages 229–242. Elsevier, 2015.

[12] A. Humphrey, T. Harman, M. Berzins, and P. Smith. A scalable algorithm for radiative heat transfer using reverse monte carlo ray tracing. In J. M. Kunkel and T. Ludwig, editors, *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 212–230. Springer International Publishing, 2015.

[13] 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 *Accepted - The 17th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2016)*, 2016.

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

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

[16] G. Krishnamoorthy. *Predicting Radiative Heat Transfer in Parallel Computations of Combustion*. PhD thesis, University of Utah, Salt Lake City, UT, December 2005.

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

[18] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin. An evaluation of emerging many-core parallel programming models. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'16, pages 1–10, New York, NY, USA, 2016. ACM.

[19] Q. Meng. *Large-Scale Distributed Runtime System for DAG-Based Computational Framework*. PhD thesis, School of Computing, University of Utah, August 2014. Ph.D. in Computer Science, advisor Martin Berzins.

[20] Q. Meng and M. Berzins. Scalable large-scale fluid-structure interaction solvers in the uintah framework via hybrid task-based parallelism algorithms. *Concurrency and Computation: Practice and Experience*, 26(7):1388–1407, May 2014.

[21] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede. In *The 2nd Conference of the Extreme Science and Engineering Discovery Environment (XSEDE 2013)*. ACM, 2013.

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

[23] S. G. Parker, J. Guilkey, and T. Harman. A component-based parallel infrastructure for the simulation of fluid-structure interaction. *Engineering with Computers*, 22:277–292, 2006.

[24] J. Pedel, J. Thornock, and P. Smith. Ignition of co-axial oxy-coal jet flames: data collaboration of experiments and simulations. *Combustion and Flame*, Accepted for publication, 2012.

[25] J. Pedel, J. N. Thornock, and P. J. Smith. Large eddy simulation of pulverized coal jet flame ignition using the direct quadrature method of moments. *Energy & Fuels*, 26(11):6686–6694, 2012.

[26] 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, pages 4:1–4:8, New York, NY, USA, 2015. ACM.

[27] 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. Technical report, SCI Institute, 2015.

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

[29] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland. Large scale parallel solution of incompressible flow problems using uintah and hypre. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 458–465, 2013.

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

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