SAND2017-1296C

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

Daniel Sunderland *, Brad Peterson †, John Schmidt‡, Alan Humphrey§, Jeremy Thornock¶, Martin Berzins‖
*Sandia National Laboratories, Albuquerque, NM 87175 USA
† ‡§‖Scientific Computing and Imaging Institute, ¶Institute for Clean and Secure Energy
University of Utah, Salt Lake City Ut 84112
Email: * dsunder@sandia.gov, † bpeterson@sci.utah.edu, ‡ jas@sci.utah.edu, § ahumphrey@sci.utah.edu,
¶ Jeremy.thornock@utah.edu , ‖ mb@sci.utah.edu

***Abstract*—The current diversity in nodal parallel computer architectures is seen in machines based upon multicore CPUs, GPUs and the Intel Xeon Phi's. A class of approaches for enabling scalability of complex applications on such architectures is based upon Asynchronous Many Task software architectures such as that in the Uintah framework used for the parallel solution of solid and fluid mechanics problems. Uintah has both an applications 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 for it to continue to do. Incrementally modifying Uintah to use the Kokkos performance portability library through prototyping experiments results in improved kernel performance by more than a factor of two.**

***Index Terms*—Uintah, Kokkos, hybrid parallelism, performance portability**

I. 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 dynamically executed as soon as their dependencies are met, as in Charm++, Legion and Uintah, see [1], and many other codes under development.

The Uintah software (http://www.uintah.utah.edu) [4] 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. 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. 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 means that Uintah needs to use is to use a performance portability layer based upon a many-core parallel programming model (see [3]), such as OpenMP, OpenACC, RAJA, Kokkos or OpenCL. In this work we have chosen to use Kokkos [2] 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 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 aim of this paper is to show how the Uintah's application programming model and its runtime system may be modified to use the Kokkos performance portability layer. Results from experiments demonstrate that Uintah applications kernels rewritten to conform to the Kokkos programming model improves in performance, with result seen up to a factor of at least two. This paper is a shortened form of a more detailed technical report [6].

II. Uintah and Arches Overview

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. As described in [4] and the references therein, problems are either initially laid out on a structured grid as shown in with the multi-material ICE code for both low and high-speed compressible flows, or by using particles on that grid with the multi-material, particle-based code MPM for structural mechanics or by combining the two in the fluid-structure interaction (FSI) algorithm MPM-ICE. The ARCHES turbulent reacting CFD component [5] is designed for simulating turbulent reacting flows with participating media radiation.

Simulation data is managed by a distributed data store known as a *Data Warehouse*, an object containing metadata for simulation variables. 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-centered, 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 two Data Warehouses available to the simulation, 1.) the *Old Data*

ESPM2 2016; Salt Lake City, Utah, USA; November 2016 978-1-5090-3858-9/16/$31.00

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. In the case of on-node GPUs, Data Warehouses specific to GPUs are used.

Parallelism within Uintah is achieved in three ways by using: domain decomposition to assign each MPI rank its own region of the computational domain; task level parallelism within an MPI rank to allow each task to run independently on node or thread level parallelism within a node. Uintah maintains a clear separation between applications code and its runtime system, and hence the details of the runtime system are hidden from the application developer. The task developer must supply entry functions to the task code, and write serial C++ code for CPU and Xeon Phi tasks and CUDA parallel code for GPU tasks. This model for CPU, GPU or Xeon Phi tasks currently requires that three versions of the task code be maintained. The use of Kokkos enables a move to a single code and allows users to exploit data parallelism within all Uintah tasks.

The primary motivation is to extend Uintah to emerging exascale problems with important commercial ramifications and benefits for improving coal combustion efficiency. For example the Arches component is being used to predict capabilities for a commercial, 1000 MW coal fired boiler. Given this challenging application, the Kokkos performance and portability improvements will be illustrated through the Arches component.

Arches is a finite volume combustion code that has been developed over a number of years [5]. 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.

III. Kokkos

Kokkos is a C++11 library for implementing portable thread-parallel codes on various HPC architectures [2]. 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 (open) source code is 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

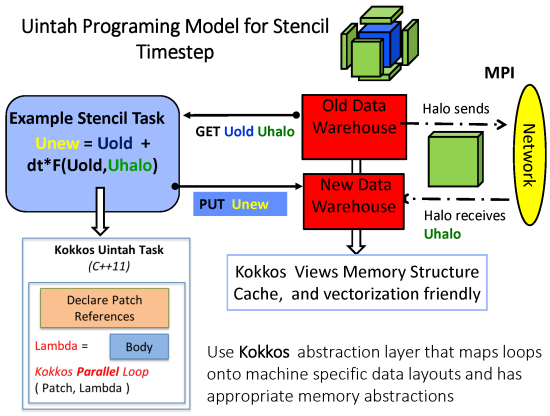

Fig. 1. Modified Uintah Programming Model

of kernel 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 CPU. GPU or Xeon Phi device as 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. 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 kernel of computation and data. 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 V.

IV. Modifying Uintah to Use Kokkos

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 substantial effort. Most of the work involves refactoring loops into parallel kernels and converting existing array data types into Kokkos views. Figure 1 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. 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 V.

When replacing the array data used by Uintah with Kokkos views. the runtime system needs to be extended to return Kokkos views in place of the current Uintah array data structures. 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 the user must incrementally verify the portability of kernels to other devices without waiting for the entire code base to be refactored. This is achieved here by extracting 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.

Uintah tasks declare and initialize 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 incremental, but allows for performance and portability verification at each step.

V. Results

Two examples are used to test the performance of the runtime system using Kokkos on key Uintah algorithms. The first example is is that of a nonlinear advection scheme and the second is a 3D loop in the Arches code [5]. 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 using 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 I. The upwind and the van Leer flux limiter show significant 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 significant speedups that are shown are a result of two complementary changes. These are 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 reimplementation of the computational kernel to perform better. This example suggests that careful rewrites of key computational kernels in conjunction with Kokkos can offer significant performance improvements. In

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 I
KOKKOS SPEEDUP ON ARCHES ADVECTION

porting the Arches 3D stencil example we needed a way to avoid porting the whole of Arches. 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 and were able to optimize to ensure that it used SIMD vectorization. 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. 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 II 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 in [6].

```
parallel_for(range,[=](int i,int j,range k_range){
auto r=subview(rhs,i,j,ALL());
auto x0=subview(X,i,j,ALL());
// Other subviews follow similarly
 . . . .
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 2. SIMD Kokkos 3D Stencil Kernel

Optimizing this kernel as in Codelisting 2 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 seen in Table II. These experiments were run on an 16 core Intel Xeon with a SIMD vector length of 2 yielding an ideal speedup of 2X of the Kokkos SIMD kernel over the Kokkos standard kernel. The results in Table II 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 speedups over standard Uintah code reflect the relative inefficiency of that user-friendly code. 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 affect the vectorization that CUDA already achieves.

VI. Conclusions

We have shown how it is possible to introduce the Kokkos performance portability layer into a sophisticated AMT runtime in the Uintah software. This involved rethinking the design of the Uintah nodal data warehouse and changing loops in the applications model. The initial experiments conducted show the promise of Kokkos as a means of providing present and future performance portability for the Uintah software. The incorporation of Kokkos on a standard cpu core offers anywhere from 2X speedups, to upwards to 12X speedups. 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 and portability.

	32^3		64^3		128^3	
	ms	**x**	ms	**x**	ms	**x**
Serial Uintah	1.06	1.0	8.04	1.0	64.9	1.0
Kokkos 1 core	0.65	1.6	4.30	1.9	36.1	1.8
Serial SIMD	0.31	3.4	2.47	3.3	20.2	3.2
Kokkos 2 cores	0.17	6.4	1.16	6.9	8.94	7.3
4 Threads SIMD	0.08	13	0.58	14	5.27	12
Kokkos 8 cores	0.07	16	0.54	15	4.51	14
16 Threads SIMD	0.04	24	0.31	26	2.54	25
Kokkos 16 cores	0.04	29	0.28	29	3.52	18
32 Threads SIMD	0.02	43	0.16	49	3.42	19
Kokkos GPU	0.09	12	0.21	38	0.61	105
CUDA SIMD	0.09	12	0.21	38	0.63	103

TABLE II
Results on 3D stencil Example. 2 sockets/16 cores/32 threads, avx, Intel Xeon CPU E5-2660 0 @ 2.20GHz 32 GB GeForce GTX TITAN X capability 5.2, 12 GB

VII. Acknowledgments

DOE NNSA funding under Award Number(s) DE-NA0002375 is gratefully acknowledged for the work of Peterson, Berzins, Humphrey, Schmidt, and Thornock. Erik Lindstrom is thanked for the results in Section 5.1. 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.

References

[1] J. Bennett, R. Clay, and et al. ASC ATDM level 2 milestone 5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical report, Sandia National Laboratories, 2015.

[2] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos. *J. Parallel Distrib. Comput.*, 74(12):3202–3216, December 2014.

[3] Matt Martineau, Simon McIntosh-Smith, Mike Boulton, and Wayne 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.

[4] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin 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.

[5] 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.

[6] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins. Performance Portability in the Uintah Runtime System Through the Use of Kokkos. Technical Report UUSCI-2016-001, Scientific Computing and Imaging Institute, University of Utah, 2016.