LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-677453

ASC Tri-lab Co-design Level 2 Milestone Report 2015

R. Hornung, H. Jones, J. Keasler, R. Neely, O. Pearce, S. Hammond, C. Trott, P. Lin , C. Vaughan, J. Cook, R. Hoekstra, B. Bergen, J. Payne, G. Womeldorff

September 23, 2015

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

ASC Tri-lab Co-design Level 2 Milestone Report 2015

Rich Hornung, Holger Jones, Jeff Keasler, Rob Neely, Olga Pearce, LLNL
Si Hammond, Christian Trott, Paul Lin, Courtenay Vaughan, Jeanine Cook, Mahesh Rajan, Rob Hoekstra, SNL
Ben Bergen, Josh Payne, Geoff Womeldorff, LANL

Lawrence Livermore National Laboratory
Sandia National Laboratories
Los Alamos National Laboratory

LLNL-TR-677453

U.S. DEPARTMENT OF ENERGY NNSA

ASC Tri-lab Co-design Level 2 Milestone Report 2015

Rich Hornung, Holger Jones, Jeff Keasler, Rob Neely, Olga Pearce
Lawrence Livermore National Laboratory

Si Hammond, Christian Trott, Paul Lin, Courtenay Vaughan, Jeanine Cook, Mahesh Rajan, Rob Hoekstra
Sandia National Laboratories

Ben Bergen, Josh Payne, Geoff Womeldorff
Los Alamos National Laboratory

LLNL-TR-677453

In 2015, the three Department of Energy (DOE) National Laboratories that make up the Advanced Scientific Computing (ASC) Program (Sandia, Lawrence Livermore, and Los Alamos) collaboratively explored performance portability programming environments in the context of several ASC co-design proxy applications as part of a tri-lab L2 milestone executed by the co-design teams at each laboratory. The programming environments that were studied included *Kokkos* (developed at Sandia), *RAJA* (LLNL), and *Legion* (Stanford University). The proxy apps studied included: *miniAero*, *LULESH*, *CoMD*, *Kripke*, and *SNAP*. These programming models and proxy-apps are described herein. Each lab focused on a particular combination of abstractions and proxy apps, with the goal of assessing performance portability using those. Performance portability was determined by: a) the ability to run a single application source code on multiple advanced architectures, b) comparing runtime performance between "native" and "portable" implementations, and c) the degree to which these abstractions can improve programmer productivity by allowing non-portable implementation details to be hidden from the application developer. This report captures the work that was completed for this milestone, and outlines future co-design work to be performed by application developers, programming environment developers, compiler writers, and hardware vendors.

Auspices

Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344.

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.

Los Alamos National Laboratory is managed and operated by Los Alamos National Security, LLC (LANS), under contract number DE-AC52-06NA25396 for the Department of Energys National Nuclear Security Administration (NNSA).

Contents

Chapter 1

Introduction

Over the past couple of decades, HPC application performance has improved dramatically with advances in computer architectures and CPU clock rates. During this period, hardware platforms have remained relatively homogeneous and consistent. Thus, application scientists have been able to focus on algorithm development and coarse-grained parallelism (mostly MPI) with little concern for fine-grained parallelism and a detailed understanding of hardware variations across platforms. The performance and portability challenges now facing ASC codes are rooted in recent, disruptive changes to HPC node architectures. Exploiting the full range of hardware capabilities forces developers to express ample fine-grained parallelism in varied forms, such as SMT (Simultaneous Multithreading), SIMT (Single Instruction, Multiple Threads), and SIMD (Single Instruction, Multiple Data). Also, careful management of data locality and memory access is becoming paramount with the emergence of deeper memory and cache hierarchies with different sizes and access timing rules. To achieve high performance portably, ASC codes must adopt algorithms and programming styles that can express various forms of parallelism and help manage data motion, without overburdening code maintainability.

In 2015, the three DOE ASC National Laboratories (Sandia, Lawrence Livermore, and Los Alamos) collaborated on an effort to explore performance portability programming environments in the context of several ASC proxy applications as part of a tri-lab L2 milestone executed by the co-design teams at each laboratory. The programming environments that were studied included: *Kokkos* (developed at Sandia), *RAJA* (LLNL), and *Legion* (Stanford University). These models and proxy-app assessments are described herein. Kokkos and RAJA are C++ programming environments that encapsulate hardware machine/compiler specific details and programming models, such as OpenMP and CUDA. This insulates application developers from non-portable implementation details, and allows an application to target multiple architectures with a single source code base. RAJA and Kokkos both focus on on-node challenges, and rely on other other programming models (e.g., MPI) to manage inter-node communication. Legion takes a system view of the programming challenges, and uses a tasking model to map work to resources - potentially allowing much greater scalability (particularly for unbalanced computations) and a unified programming model for inter-node and intra-node parallelism.

The proxy apps studied in this effort were developed at multiple ASC laboratories. They are: *miniAero*, *LULESH*, *CoMD*, *Kripke*, and *SNAP*. Together, these proxies embody a broad cross-section of algorithms and computational patterns found in scientific applications, including structured and unstructured mesh-based methods, particle computations, and regular nearest-neighbor and wave-front sweep communication patterns.

Each lab focused on a particular combination of abstractions and proxy applications and assessed performance portability using those. Performance portability was measured in terms of:

- the ability to run a single source code on multiple advanced architectures,
- runtime comparisons between “native” and “portable” implementations, and
- the degree to which abstractions can improve programmer productivity by insulating application developers from non-portable implementation details.

This report captures the work that was completed as part of this milestone, and outlines a set of future work to be performed in co-design between application developers, programming environment developers, compiler writers, and hardware vendors.

In the rest of this paper, each lab presents an overview of the programming models(s) they studied, and a detailed description of how it was applied to each of one or two proxy applications. Metrics on performance, portability, and productivity are presented. The goal was not to directly compare one approach to another since each environment is a work-in-progress and trade-offs are not always objective. Instead, the aim was to enumerate lessons learned in developing each approach, and to present an honest assessment of the performance portability each offers and the amount of work needed to transform an application to use it. We conclude with a set of common lessons learned, and a set of recommendations on a path forward for the ASC Co-design Project.

Chapter 2

Lawrence Livermore National Laboratory

2.1 Introduction

Many science and engineering HPC applications, including ASC codes, have used the portable and standard MPI (Message Passing Interface) library [30] successfully for over two decades to achieve highly scalable distributed memory parallel execution. Typically, such codes encapsulate MPI data structures and library calls in routines that are called to perform application-specific *inter-process* data communication and synchronization at points as needed for each numerical algorithm. Using this approach, domain scientists focus on writing serial code that runs on each MPI process without needing detailed knowledge of the underlying parallel operations. In recent years, the diversity and magnitude of fine-grained *intra-node* parallelism available on HPC hardware has grown substantially. All indications suggest that this trend will continue into the foreseeable future.

Ideally, applications will be able to continue to follow a similar encapsulation approach that would allow domain scientists to profitably exploit the large amounts of available fine-grained hardware parallelism while maintaining, in large part, the "look and feel" of serial code. This would enable applications to run on and be tuned for different advanced architectures with an acceptable level of software disruption. Preserving the advantages and hugely successful dynamics of MPI usage would substantially benefit large multiphysics applications where software maintenance and the ability to quickly develop new algorithms and models is as important as performance and platform portability.

The approach we are pursuing at LLNL toward this "ideal" is called RAJA. The overarching goals of RAJA are to make existing production codes *portable with minimal disruption* and to provide a model for new application development so that they are portable from inception. RAJA uses standard C++11 – C++ is the predominant programming language used in LLNL ASC codes. RAJA shares goals and concepts found in other C++ portability abstraction approaches, such as Kokkos [18], Thrust [10], Bolt [1], etc. However, RAJA provides constructs that are absent in other models and which are used heavily in LLNL ASC multiphysics codes. Also, a goal of RAJA is to adapt concepts and specialize them for different code implementation patterns and C++ usage, since data structures and algorithms vary widely across applications.

In the next section, we present an overview of the RAJA abstraction model and describe various portability and performance features it supports. After that, we discuss integration of RAJA into two proxy applications, CoMD and Kripke. In the context of those proxy apps, we evaluate RAJA in terms of ease of integration/level of code disruption, architecture portability, algorithm flexibility enabled, and performance. Finally, based on these assessments, we describe lessons learned, issues encountered, and future work.

2.1.1 RAJA Overview

Loops are the main conceptual abstraction in RAJA. A typical multiphysics code contains O(10K) loops in which most computational work is performed and where most fine-grained parallelism is available. RAJA defines a systematic loop encapsulation paradigm that helps insulate application developers from implemen-

tation details associated with software and hardware platform choices. Such details include: non-portable compiler and platform-specific directives, parallel programming model usage and constraints, and hardware-specific data management. RAJA can be used incrementally and selectively in an application and works with the native data structures and loop traversal patterns in a code.

A typical RAJA integration approach is a multi-step process involving steps of increasing complexity. First, basic transformation of loops to RAJA is reasonably straightforward and can be even mundane in many cases. The initial transformation makes the code portable by enabling it to execute on both CPU and GPU hardware by choosing various parallel programming model back-ends at compile-time. Second, loop *execution policy* choices can be refined based on an analysis of loops. Careful categorization of loop patterns and workloads is key to selecting the best choices for mapping loop execution to available hardware resources for high performance. Important considerations include: the relationship between data movement and computation (operations performed per byte moved), control flow branching intensity, and available parallelism. Earlier, we stated that ASC multiphysics codes contains O(10K) loops. However, such codes typically contains only O(10) *loop patterns*. Thus, a goal of transforming a code to RAJA should be to assign loops to execution policy equivalence classes so that similar loops may be mapped to particular parallel execution choices or hardware resources in a consistent fashion.

Lastly, more rigorous algorithm, performance, and memory analysis can help to determine whether employing more advanced RAJA features may further benefit performance. Some performance-critical algorithms may require deeper restructuring, or even versions tuned for particular hardware architectures. It is important to note that no programming model can fully obviate the need for in depth analysis if the goal is to achieve the highest possible performance. Also, no programming model allows sophisticated algorithms to be cast in a single form that yields the best performance across diverse hardware. RAJA can simplify the ability to act based on sound analysis by enabling platform-specific choices to be localized in header files. When this is done, such choices can be propagated throughout a large application code base without needing to modify many sites within the code.

In the next several sections, we describe the encapsulation features of RAJA and how they cooperate to manage architecture-specific concerns.

2.1.2 Decoupling Loop Body from Loop Traversal

Like other C++-based portability layer abstraction approaches, RAJA relies on decoupling the body of a loop from the mechanism that executes the loop; i.e., its *traversal*. This allows the same traversal method to be applied to many different loop bodies and different traversals to be applied to the same loop body for different execution scenarios. In RAJA, the decoupling is achieved by re-casting a loop into the generally-accepted "parallel for" idiom. To illustrate, consider a C-style for-loop containing a "daxpy" operation and two reductions:

```
double* x; double* y;
double a, tsum = 0.0, tmin = MYMAX;
// ...
for ( int i = begin; i < end; ++i ) {
   y[i] += a * x[i] ;
   tsum += y[i] ;
   if ( y[i] < tmin ) tmin = y[i] ;
}
```

```
double* x; double* y;
RAJA::SumReduction< reduce_policy, double > tsum(0.0);
RAJA::MinReduction< reduce_policy, double > tmin(MYMAX);
// ...
RAJA::forall< exec_policy >( begin, end, [=] (int i) {
   y[i] += a * x[i] ;
   tsum += y[i] ;
   tmin.min( y[i] );
}
```

Listing 2.1: Basic RAJA loop transformation

There are several key differences to note in the RAJA loop in Listing 2.1:

- The for-loop construct is replaced by a call to a traversal template method (RAJA::forall), where the template parameter is the loop *execution policy*.
- The loop-body is passed to the traversal template as a *C++ lambda function* [2].
- The reduction variables are converted to RAJA objects templated on a *reduction policy* and reduction data type.

It is important to note that in the original C-style for-loop, all loop execution details, such as iteration order and data array element access, are expressed explicitly in the source code. Changing any aspect of execution requires changes to this source code. Decoupling the loop body and traversal as in the RAJA version, iteration orders, data layout and access patterns, parallel execution strategies, etc. can altered without changing the way the loop is written. Apart from the slight difference in syntax for the min reduction, the loop body is that same as the C-style version. The C++ 11 lambda function capability undergirds a key RAJA design goal – to enable portability with minimal disruption to application source code.

2.1.3 RAJA Encapsulation Model

In this section, we briefly describe four main encapsulation features in RAJA that can be used to manage architecture-specific concerns. They are illustrated in Figure 2.1 using our earlier example loop with different colors.

```
RAJA::Real_ptr x, RAJA::Real_ptr y ;
RAJA::Real_type a ;
RAJA::SumReduction<…, Real_type> tsum(0);
RAJA::MinReduction<…, Real_type> tmin(MYMAX);
…
RAJA::forall< exec_policy >( IndexSet, [=] (Index_type i)  {
     y[ i ] += a * x[ i ] ;
     tsum += y[i];
     tmin.min( y[i] );
}  );
```

Figure 2.1: The four primary cooperating encapsulation features of RAJA: traversal methods and execution policies (blue), index sets (purple), data types (red), and C++ lambda loop body (green).

- **Traversals and execution policies [blue]**. A traversal method specialized with an execution policy template parameter defines how the loop will be executed. For example, a traversal may run the loop sequentially, as multithreaded parallel loop using OpenMP [13] or CilkPlus [7], or may launch the loop iterations as a CUDA kernel to run on a GPU [8]. We describe how this works in more detail in Section 2.1.4.
- **IndexSets [purple]**. In Listing 2.1, "begin" and "end" loop bounds are passed as arguments to the traversal method. While RAJA can process explicitly bounded loop iterations in various execution schemes that are transparent to the source code, the RAJA *IndexSet* abstraction in Figure 2.1 enables much more flexible and powerful ways to control loop iterations. IndexSets allow loop iteration order to be changed in ways which can, for example, enable parallel execution of a non-data-parallel loop without rewriting it. Typically, an IndexSet is used to partition an iteration space into *Segments*; i.e., "chunks" of iterations. Then, different subsets of iterations may be launched in parallel or run on different hardware resources. IndexSets also provide the ability to manage dependencies among Segments to resolve thread safety issues, such as data races. In addition, IndexSet Segments enable

coordination of iteration and data placement; specifically, chunks of data and iterations can be mapped to individual cores on a multi-core architecture. While IndexSets provide the flexibility to be defined at runtime, compilers can optimize execution of kernels for different *Segment type* implementation at compile-time. For example, a Segment type implementation that processes contiguous index ranges can be optimized in ways that an indirection Segment type implementation cannot. In Section 2.1.5, we elaborate on a variety of key IndexSet features.

- **Data type encapsulation [red]**. RAJA provides data and pointer types, seen here as *Real_type* and *Real_ptr*, that can be used to hide non-portable compiler directives and data attributes, such as alignment, restrict, etc. These compiler-specific data decorations often enhance a compiler's ability to optimize. While these types are not required to use RAJA, they are a good idea in general for HPC codes. They eliminate the need to litter a code with detailed, non-portable syntax and enable architecture or compiler-specific information to be propagated throughout an application code with localized changes in a header file. For any parallel reduction operation, RAJA does require a reduction class template to be used. Template specialization of a reduction in a manner similar to the way a traversal method is specialized on an execution policy type enables a portable reduction operation while hiding programming model-specific reduction constructs from application code. We describe implementation issues associated with reduction objects in Section A.1

- **C++ lambda functions [green]**. The standard C++11 lambda feature captures all variables used in the loop body which allows the loop construct to be transformed with minimal modification, if any is required, to the original code.

The RAJA encapsulation features described here can be used in part or combined based on application portability and performance needs. They may also be combined with application-specific implementations. This allows a *multi-tiered approach* approach to performance tuning for a particular architecture. Most loops in a typical HPC application can be parameterized using basic RAJA encapsulation features. Other kernels may require a combination of RAJA entities and customized implementations suited to a particular algorithm. A specific example of this is discussed in Section 2.3.

2.1.4 Basic Traversal Methods and Execution Policies

In this section, we make loop traversal specialization and execution policy concepts more concrete by showing how a loop may be executed in different ways depending on the specialization. Recall the initial loop example we described earlier.

```
double* x; double* y;
RAJA::SumReduction< reduce_policy, double > tsum(0.0);
RAJA::MinReduction< reduce_policy, double > tmin(MYMAX);
// ...
RAJA::forall< exec_policy >( begin, end, [=] (int i) {
   y[i] += a * x[i] ;
   tsum += y[i] ;
   tmin.min( y[i] );
}
```

Suppose that the policy template parameters are defined as *typedefs* in a header file. By changing the parameter definitions, we can run the loop with different programming model backends or on different hardware resources *without changing the code in this example.*

CPU Execution

RAJA provides a variety of traversal and execution policy options for CPU loop execution. The simplest option is to execute a loop sequentially. To do this, we could define the execution policy template parameters as:

```
typedef RAJA::sequential    exec_policy ;
typedef RAJA::seq_reduce    reduce_policy ;
```

When this is done, the traversal template that runs the loop is:

```
template< typename LB >
void forall(sequential, Index_type begin, Index_type end, LB body) {
#pragma novector
    for ( int i = begin; i < end; ++i ) body( i );
}
```

Note that we use the "novector" pragma to prevent the compiler from generating SIMD vectorization optimizations for this case. Changing exec_policy to RAJA::simd allows the compiler to generate SIMD optimizations if it decides to do so.

For OpenMP multithreaded parallel CPU execution, we could define the template parameters as:

```
typedef RAJA::omp_parallel_for  exec_policy ;
typedef RAJA::omp_reduce        reduce_policy ;
```

The traversal method that runs the loop in this case is:

```
template< typename LB >
void forall(omp_parallel_for, Index_type begin, Index_type end, LB body) {
#pragma omp parallel for
    for ( int i = begin; i < end; ++i ) body( i );
}
```

Accelerator Execution

There are multiple programming model options for executing code on an accelerator device, such as a GPU. To offload the loop to run on a GPU using the OpenMP 4.1 accelerator model, we would define the template parameters as:

```
typedef RAJA::omp_parallel_for_acc  exec_policy ;
typedef RAJA::omp_acc_reduce        reduce_policy ;
```

Now, the traversal method that runs the loop is:

```
template< typename LB >
void forall(omp_parallel_for_acc, Index_type begin, Index_type end, LB body) {
#pragma omp target
#pragma omp parallel for
    for ( int i = begin; i < end; ++i ) body( i );
}
```

Note that the RAJA template cannot explicitly setup the GPU device data environment with an OpenMP map clause. Map clauses are used to specify how storage associated with specific named variables is moved between host and device memories. Since a RAJA traversal is generic with respect to the loop body, it knows nothing about the data used in the loop. The OpenMP 4.1 standard [4] fills gaps to support "unstructured data mapping" that will allow one to set up the proper device data environment before offloading via a RAJA traversal. We expect to manage such host-device data transfers in real application codes using a

similar encapsulation approach to the way MPI communication is typically hidden, and which we briefly discussed earlier.

In the NVIDIA CUDA model, the notion of a loop is fundamentally absent. Instead, the algorithm for a single "loop iteration" is executed as a CUDA kernel function that is launched over a thread block on a CUDA-enabled GPU device. Each iteration executes on a different CUDA thread. To launch the loop as a CUDA kernel, we define the template parameters as:

```
typedef RAJA::cuda_acc      exec_policy ;
typedef RAJA::cuda_reduce   reduce_policy ;
```

The following code snippets illustrate RAJA backend code for CUDA. So that the loop code continues to look like a loop, we pass the loop body to the traversal template (B), which has the same arguments as other traversals. methods. This template launches a GPU kernel template (A) that executes each loop iteration on a separate GPU thread.

```
// (A) kernel function template
template< typename LB >
__global__ void forall_cuda_kernel(Index_type begin, Index_type len, LB body) {
   Index_type i = blockIdx.x * blockDim.x + threadIdx.x ;
   if ( i < len ) {
      body( begin + i ) ;
   }
}

// (B) traversal template that launches CUDA GPU kernel
template< typename LB >
void forall(cuda_acc, int begin, int end, LB body) {
   size_t blockSize = THREADS_PER_BLOCK;
   size_t gridSize = (end - begin + blockSize - 1) / blockSize;
   Index_type len = end - begin;
   forall_cuda_kernel<<<gridSize, blockSize>>>(body, begin, len);
   // ...
}
```

To manage data transfers between host and device when using CUDA, we have multiple options. Using CUDA UM, or *Unified Memory*, is the simplest and least intrusive method. With UM, memory allocations are replaced with calls to *cudaMallocManaged*(), which allows data to be accessed in the same way on either the host or device with no explicit transfer operations. However, this may not yield desired performance in many situations. When this is the case, we can encapsulate CUDA memory copy routines in a manner similar to how we would use OpenMP unstructured data mapping.

It is important to note that CUDA support for passing C++ lambda functions to code that executes on a device is a new nvcc capability and is still under development. Currently, it is required to attach the CUDA *__device__* qualifier to the lambda where it is defined. So the code in our example actually looks as follows:

```
RAJA::forall< exec_policy >( begin, end, [=] RAJA_LAMBDA (int i) {
   // ...
} );
```

where RAJA_LAMBDA is a macro defined in a header file as:

```
#if defined(RAJA_USE_CUDA)
#define RAJA_LAMBDA __device__
#else
#define RAJA_LAMBDA
#endif
```

This macro usage makes the code somewhat clumsy and forces an application to make a *compile-time* choice as to how the loop will execute. This limits RAJA flexibility and our ability to choose at runtime how the loop will run or to run different IndexSet Segments on a CPU and a GPU in parallel, for example. We hope this constraint will be removed in the future. NVIDIA developers used a RAJA variant of the LULESH proxy-app as the first somewhat comprehensive DOE application to test the device lambda capability. We are directly encouraging NVIDIA compiler developers to improve lambda support.

2.1.5 IndexSets and Additional Traversal Features

Mesh-based multiphysics applications contain loops that iterate over mesh elements, and thus data arrays representing fields on a mesh, in a variety of ways. Some operations involve stride-1 array data access while others involve unstructured accesses using indirection arrays. Often, these different access patterns occur in the same physics operation. For code maintenance, such loop iterations are usually coded using indirection arrays since this makes the code flexible and relatively simple. In this section, we describe some key features of RAJA IndexSets and how they can be used to manage complex loop iteration patterns and address a variety of performance concerns. In particular, IndexSets provide a powerful mechanism to balance runtime iteration space definition with compile-time optimizations.

A RAJA IndexSet is an object that encapsulates a complete loop iteration space that is partitioned into a collection of Segments, of the same or different *Segment types*. Figure 2.2 shows two different types of simple Segments, a "range" and a "list" that may be used to iterate over different portions of an array. A RAJA *RangeSegment* object defines a contiguous set of iteration indices with constraints applied to the iteration bounds and to the alignment of data arrays with memory constructs. For example, range Segments can be aligned multiples of a SIMD or a SIMT width to help compilers generate more efficient code. A RAJA *ListSegment* is a chunk of iterations that do not meet range Segment criteria. It is important to note, that, with RAJA, we emphasize the tight association between a loop iteration and a "footprint" of data array elements in memory.

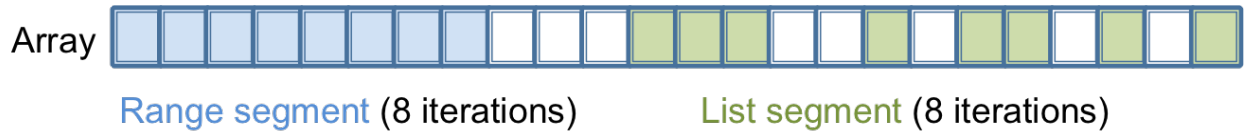


Figure 2.2: RAJA "range" and "list" Segments iterate over subsets of array indices using different loop constructs.

To illustrate some simple IndexSet mechanics, consider the following set of array indices to process.

```
int num_elems = 21;
int elems[] = {0, 1, 2, 3, 4, 5, 6, 7, 14, 27, 36,
               40, 41, 42, 43, 44, 45, 46, 47, 87, 117};
```

Such a set of indices may enumerate elements on a mesh containing a particular material in a multi-material simulation, for example. The indices may be assembled at runtime into an IndexSet object by manually creating and adding Segments to an IndexSet object. A more powerful alternative is to use one of several parameterized RAJA *IndexSet builder methods* to partition an iteration space into a collection of "work Segments" according to some architecture-specific constraints. For example,

```
RAJA::Indexset segments = RAJA::createIndexset( elems, num_elems );
```

might generate an IndexSet object containing two range Segments ($\{0, ..., 7\}, \{40, ..., 47\}$) and two list Segments ($\{14, 27, 36\}, \{87, 117\}$).

When the IndexSet object is passed along with a loop body (lambda function) to a RAJA iteration template, the operation will be dispatched automatically to execute each of the Segments:

```
RAJA::forall< exec_policy >( Segments, [=] (...) {
   // loop_body
} );
```

That is, a specialized iteration template will be generated at compile-time for each Segment type. Iteration over the range Segments may involve a simple for-loop such as:

```
for ( int i = begin; i < end; ++i) loop_body( i );
```

Iteration over the list Segments may be involve a for-loop, with indirection applied:

```
for ( int i = 0; i < seglen; ++i ) loop_body( Segment[i] );
```

IndexSet builder methods can be customized to tailor Segments to hardware features and execution patterns to balance compile-time and runtime considerations. Presently, IndexSets enable a two level hierarchy of scheduling and execution. A dispatch policy is applied to the collection of Segments. An execution policy is applied to the iterations within each Segment. Examples include:

- Dispatch each Segment to a CPU thread so Segments run in parallel and execute range Segments using SIMD vectorization.
- Dispatch Segments sequentially and use OpenMP within each Segment to execute iterations in parallel.
- Dispatch Segments in parallel and launch each Segment on either a CPU or GPU as appropriate.

In Place SIMD

As we noted earlier, runtime Segment construction can impose constraints that complement compile-time pragmas and optimizations, which can be hidden in RAJA traversals. Multiphysics codes often use indirection arrays to iterate over: elements of an unstructured mesh, element subsets that contain a particular material on a mesh, etc. For example, in a *real* ASC code running a large hydrodynamics problem containing ten materials and over 16 millions elements (many multi-material), we have observed that most loops traverse "long" stride-1 ranges. Figure 2.3 summarizes this particular case.

Range length	% loop its.	Range length	% loop its.
>= 16	84%	>= 128	69%
>= 32	74%	>= 256	67%
>= 64	70%	>= 512	64%

Figure 2.3: A summary of the percentage of total loop iterations performed within stride-1 ranges of various lengths. Data from an LLNL ASC code running a large multi-material hydrodynamics problem.

Here, we see that 84% of loop iterations occur in stride-1 ranges of length 16 or more and 64% of iterations occur in stride-1 ranges longer than 512. Since the complete iteration over the elements containing each material are not contiguous, relatively expensive gather/scatter operations are needed to pack and unpack element data to expose stride-1 iterations to a compiler. Using RAJA IndexSets, we can partition the iterations into range and list Segments to expose the SIMD-vectorizable loop portions while leaving the data arrays *in place*. This approach has the potential to extract substantial performance gains from a variety of numerical operations while obviating the need for additional temporary arrays and data motion.

Loop Reordering and Tiling

RAJA IndexSets can expose available parallelism in loops that are not parallelizable as written. For example, a common operation in a staggered-mesh code, sums zonal values to surrounding nodes as is illustrated in the left image in Figure 2.4. IndexSets can be be used to reorder loop iterations to to achieve "data parallel" execution without modifying the loop body code. Figure 2.4 shows two different ordering possibilities, (A) and (B), in the center and right images. Different colors indicate independent groups of computation, which can be represented as Segments in indexSets. For option A, we iterate over groups (Segments) sequentially (group 1 completes, then group 2, etc.) and operations within a group (Segment) can be run in parallel. For option B, we process zones in each group (row) sequentially and dispatch rows of each color in parallel. For a 3D problem in a production LLNL ASC code run on BG/Q, option A gives 8x speedup with 16 threads over the original serial implementation. Option B provides 17% speedup over option A at 16 threads. It is worth reiterating that no source code modifications are required to switch between these parallel iteration patterns once RAJA in place.

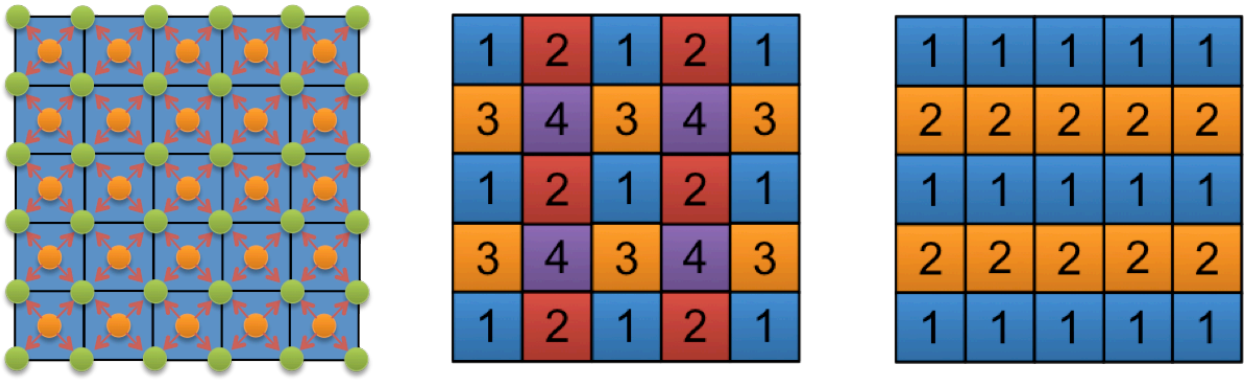


Figure 2.4: Zone-to-node sum operations (left), ordering option A (center), and ordering option B (right).

RAJA Segments can also represent arbitrary tilings of loop iterations that can be tuned and sized for specific architecture and memory configurations. Figure 2.5 shows two different element "tilings" that represent different iteration and data orderings on a portion of a mesh. When loop iterations are encapsulated in IndexSet Segments, instead of hard-coded in an application, data arrays can be permuted for locality and cache reuse. For example, the canonical tiling in the upper part of Figure 2.5 can be transformed into the "compact" tiling in the lower part of the figure.

Segments can also work together with data allocation to further enhance optimization and performance. A typical ASC code centralizes data allocation in macros or functions for consistent usage throughout a code. Data allocation can be based on IndexSet configurations to apply optimization-enhancing allocation techniques, such as "first-touch", which can result in improved NUMA behavior during multithreaded execution.

Dependence Scheduling

It is important to emphasize that once RAJA is in place, many aspects of execution may be tailored and optimized by application developers. They can modify Segment dispatch and execution mechanisms in traversal methods, or build custom IndexSets to explore alternative "work-around" implementations that may solve problems when execution performance does not match expectations. Such complete control is not found in monolithic programming models, typically.

To demonstrate the range of flexibility the that RAJA provides, we have developed a version of the LULESH proxy-app that can be run in numerous ways using different IndexSet configurations and RAJA execution policies without changing the application source code. Specifically, by changing one use case parameter in a single header file, the code can be run in ten different ways. Each value of the parameter triggers a different IndexSet construction process (as described earlier) and set of RAJA execution policies.

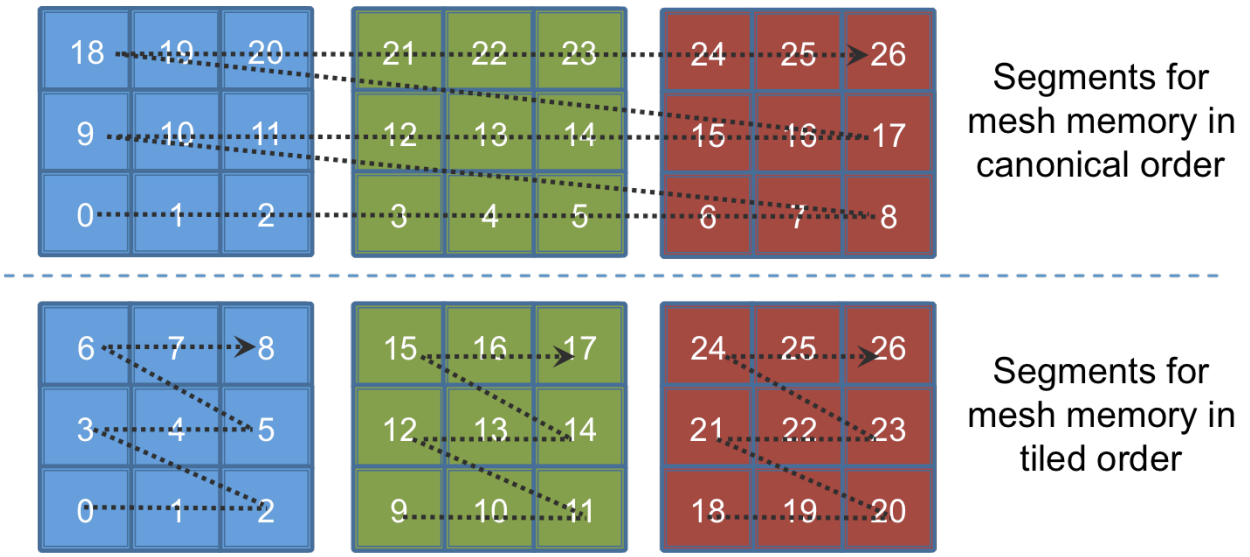


Figure 2.5: RAJA Segments represent arbitrary "tilings" of loop iterations that in turn touch data on a mesh. Colors represent Segments within an IndexSet. Numbers represent 1d array indices.

The execution modes include: different OpenMP variants using different data/loop iteration tilings and parallel execution strategies, CilkPlus parallelism, and two different CUDA-based GPU variants.

The RAJA IndexSet-traversal model supports advanced features, such as task dependence scheduling, that we demonstrate in one of the OpenMP variants of this code. In particular, IndexSet Segments can be defined and arranged to encode dependence scheduling patterns that enable more efficient parallelism. A particularly interesting performance comparison of three different variants is shown in Figure 2.6. Here, we compare strong-scaling of each variant relative to the baseline (non-RAJA) OpenMP version using 1 to 16 threads on a single node of an ASC TLCC2 Linux cluster (two-sockets, each with an 8 core Intel Xeon Sandy Bridge CPU). The blue curve shows that the basic RAJA OpenMP variant has a $10-15\%$ overhead at small core counts compared to the baseline. An analysis of the Intel compiler assembly code reveals that, when OpenMP is combined with the RAJA C++ abstraction layer, certain optimizations are not performed. Beyond 4 threads, the RAJA version scales well and outperforms the baseline due to some memory usage optimizations we have done. While these optimizations are not directly due to RAJA, without it parts of the code would have to be rewritten to enable the same improvements. A CilkPlus variant (red curve) does not have the same low thread count overhead, but does not scale as well as the OpenMP variant – the OpenMP and CilkPlus variants are identical except for the RAJA execution policy. Finally, the purple curve shows the performance achieved when RAJA "lock-free" Segment dependence scheduling is used. Effectively, this mechanism allows us to replace fine-grained gather-sum synchronization with coarser-grained semaphore synchronization between tiles that eliminates unnecessary memory movement.

The dependence scheduling is controlled by a simple semaphore mechanism applied per Segment, as shown in the code sample below. To manage Segment dependencies, three pieces of information are required. First, a "reload" value defines the number of dependencies that must be satisfied before a Segment can execute. As each dependency is satisfied the semaphore value is decremented by one; when it reaches zero, the Segment can execute. Until then, the thread "yields" the CPU resource. After a Segment is dispatched to execute, its semaphore value is reset to the reload value. Second, an "initial" value is an override for the reload value. A subset of Segments must be "primed" to execute; ideally, a number of Segments at least as large as the maximum number of threads available should be able to execute immediately. The semaphore value for such Segments is initialized to zero to indicate that they can execute immediately. Semaphore values for all other Segments are initialized to their reload values. Third, "forward dependencies" are the set of Segments that must be notified when a Segment execution completes. Here, notification means that the semaphore value in each forward Segment is decremented by one.

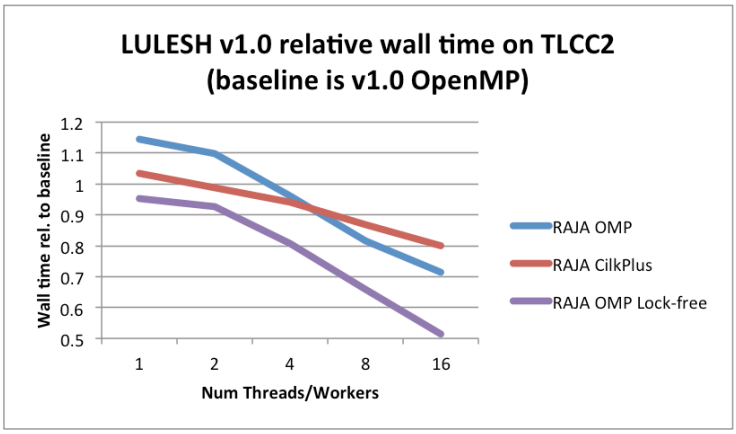


Figure 2.6: A strong-scaling runtime comparison of three different LULESH variants to the baseline (non-RAJA) OpenMP version.

```
#pragma omp parallel for schedule(static, 1)
for (int i = 0; i < num_seg; ++i) {

   IndexSetSegInfo* seg_info = iset.getSegmentInfo(i);
   DepGraphNode* task  = seg_info->getDepGraphNode();

   while ( task->semaphoreValue() != 0 ) {
      sched_yield();
   }

   //
   // execute segment i ...
   //

   if ( task->semaphoreReloadValue() != 0 ) {
      task->semaphoreValue() = task->semaphoreReloadValue() ;
   }

   if ( task->numDepTasks() != 0 ) {
      for (int ii = 0; ii < task->numDepTasks(); ++ii) {
         int seg = task->depTaskNum(ii) ;
         DepGraphNode* dep  = iset.getSegmentInfo(seg)->getDepGraphNode();
         __sync_fetch_and_sub(&(dep->semaphoreValue()), 1) ;
      }
   }

} // end loop over index set segments
```

In Section 2.2, we will show the performance benefit that can be achieved using similar Segment scheduling mechanisms in the CoMD proxy-app. There, we create Segment task graphs to define dependency scheduling for tile Segments in a wave-front algorithm.

Fault Tolerance

Although fault tolerance was not part of this L2, we have explored encapsulating a fine-grained, transient fault recovery mechanism in RAJA traversal templates. The ability to support such a capability in a portable and transparent manner is potentially very powerful and could help to resolve critical usability/productivity issues on future platforms.

The simplified code example below shows the basic elements of the idea:

```
void forall(policy, Index_type begin, Index_type end, LB body) {
   bool repeat;
   do {
      repeat = false;

      // Execute loop
      for (Index_type ii = begin ; ii < end ; ++ii ) {
         body( ii );
      }

      if (Transient_Fault) {
         cache_invalidate();
         repeat = true;
      }
   } while (repeat);
}
```

Here, the data cache is invalidated and reloaded from memory and the loop is re-executed whenever a transient fault is signaled. This requires that each loop in a code where this mechanism is applied be *idempotent*; that is, it can be run any number of times and produce the same result. This requires read-only and write-only arrays (no read-write arrays), which can increase memory usage and bandwidth requirements slightly. The effort to achieve a full level of idempotence depends on how a code is implemented.

Note that this mechanism cannot handle hard faults or those that occur between loops. But, the vast majority of work in a typically multiphysics code occurs in the loops. So we believe an approach like this would allow a code to recover from most transient fault conditions in a very localized way. In particular, the recovery cost for a fault addressed by this method is commensurate with the scope of the fault. That is, a code can recover with minimal localized disruption without needing to coordinate with other loops or code operations or needing a full restart.

A complete, robust implementation of this approach would benefit from additional hardware and O/S support. Specifically, processors could emit signals for specific fault conditions (as some Intel chips currently do) and the O/S could be specialized to process them in a way that an application could respond. Nevertheless, we have simulated this mechanism in LULESH and have found that the software and performance impact of the method to be acceptable for that case. Specifically, making the code idempotent required minor changes to a small fraction of loops and these changes added a small runtime overhead, $2-5\%$ depending on thread count. In addition, when manually injecting a large number of faults (50 in a 120 second run), total runtime was increased by roughly half a percent when compared to no faults.

In the next two sections, we discuss transforming the CoMD and Kripke proxy applications to use RAJA. We evaluate RAJA in terms of ease of integration/level of code disruption, architecture portability, algorithm flexibility enabled, and performance.

2.2 CoMD

We ported CoMD to RAJA to evaluate the following:

- The amount of effort and code changes needed for the port;
- The performance of the initial (no tuning) port;
- The ability to add new schedules;
- Portability.

2.2.1 CoMD Description

CoMD is a proxy-app for a broad class of molecular dynamics simulations developed through the ASCR ExMatEx [3] co-design center led by Los Alamos. Figure 2.7 shows two large-scale MD simulations. In

particular, CoMD considers the simulation of materials where the interatomic potentials are short range (e.g., uncharged metallic materials). In that case the simulation requires the evaluation of all forces between atom pairs within the cutoff distance. The performance of the simulation is then dependent on the efficiency of 1) identifying all atom pairs within the interaction radius and 2) computing the force between the atom pairs. For the proxy app, only the simple Lennard-Jones (LJ) and Embedded Atom Method (EAM) potentials are considered.

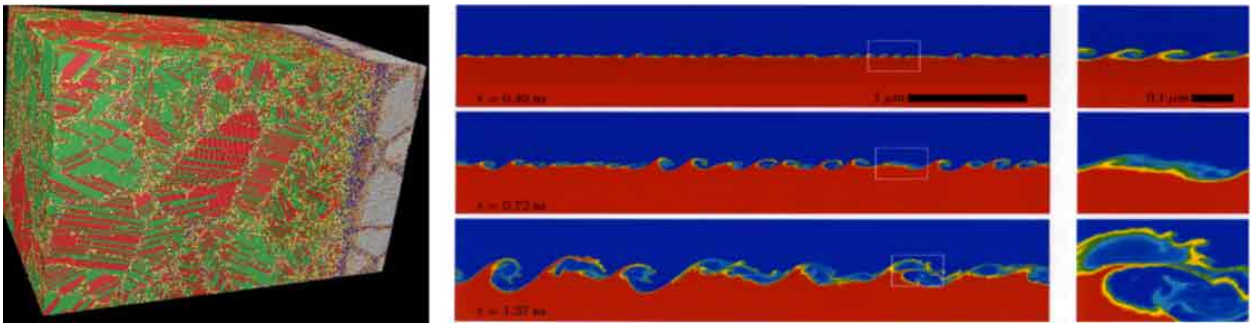


Figure 2.7: Two examples of such large-scale MD simulations, SPaSM and ddcMD, simulating solid and liquid problems.

Figure 2.8 shows a sketch of a set of atoms, with a typical interatomic potential with a finite cutoff radius. The problem is then reduced to computing the interactions of the atom in question with all the other atoms within the shaded circle, rather than with all the other atoms in the system.

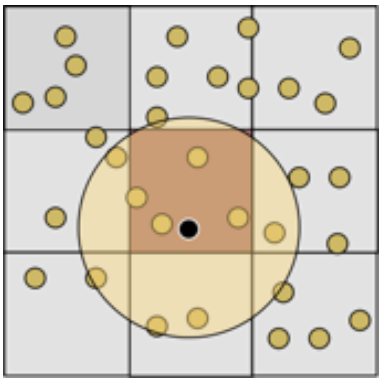


Figure 2.8: Cut-off distance in CoMD.

Inter-node Work Decomposition

CoMD utilizes a Cartesian spatial decomposition of atoms across nodes, with each node responsible for computing forces and evolving positions of all atoms within its domain boundaries. As atoms move across domain boundaries they are handed off from one node to the next.

Intra-node Decomposition

CoMD assumes a cutoff distance for the interatomic potentials which is much smaller than the characteristic size of the spatial domain of the node. To allow efficient identification of the atoms pairs which are within the cutoff distance, the atoms on a node are assigned to cubic *link cells* which are slightly larger than the cutoff distance. Thus, an atom within a given link cell only needs to test all the atoms within its own cell and the 26 neighboring link cells in order to guarantee that it has found all possible interacting atoms. In

contrast to some codes where the atoms are assigned to link cells only during the force computation, CoMD uses the link cells as the basic data structure for storing the atoms.

Figure 2.8 shows a sketch of the link cell decomposition of atoms with a finite cutoff. Searching all the neighboring link cells guarantees that all atoms within the shaded circle are checked.

Inter-node Communication

Since atoms interact with all atoms within the cutoff radius, atoms near the domain boundaries need information from adjacent nodes. The atoms state data for these atoms is placed in halo (ghost cell) regions around the local spatial region. These regions are decomposed into halo link cells, just as occurs in the interior domain. For the LJ potential, only the atom positions need to be communicated prior to force computation. For EAM, a partial force term also needs to be communicated, interleaved with the force computation step.

The halo exchange communication pattern takes advantage of the Cartesian domain decomposition and link cell structure. Each node sends to its 26 neighbors in 6 messages in 3 steps, first the x-faces, followed by the y-faces, then z-faces. This minimizes the number of messages and maximizes the size of the messages. This requires that the message traffic be serialized with the steps processed in order.

2.2.2 CoMD Implementations

The reference implementation of CoMD is coded in C. The atom data is stored in a partial array of structures format, utilizing separate 3-vectors for position, velocity and force, and flat arrays for most other variables such as energy. Internode decomposition is via encapsulated MPI communication. Since the local work is purely serial, we make use of force symmetry (F_{ij} = - F_{ji}) to reduce the amount of computation to the minimum. A pure serial code (without MPI) can be built by switching a flag in the makefile. In this case the halo regions are populated by copying data from the opposite boundary (assuming periodic boundary conditions).

OpenMP Implementation

The OpenMP implementation parallelizes the force loop, which is the majority of the computational work. In order to ensure thread safety, this version does not use force symmetry, but rather each atom in the pair computes F_{ij} separately and updates only their own force term. This results in 2x the amount of computation in exchange for speedups due to shared memory threading - an unfortunate tradeoff that we address with RAJA described later in this section.

The force computation loops pseudocode is described as follows:

```
forall local link cells
  foreach atom in local link cell
    forall neighbor link cells
      foreach atom in neighbor link cell
        ...
```

2.2.3 Porting to RAJA

Because RAJA requires the use of a C++ compiler, minor modifications to CoMD source code were necessary to enable compilation with the C++ compiler (mostly type casts).

There are two RAJA implementations of CoMD:

1. An implementation that is functionally equivalent to the OpenMP implementation of the mini-app;
2. An implementation that performs each force computation once, and thus has to ensure that there are no data races when updating atom values by using dependence checks.

Basic port of CoMD to RAJA

The general code structure of CoMD remains the same when the loops in CoMD are replaced with RAJA abstractions.

Original CoMD code:

```
//loop over local boxes
#pragma omp parallel
for (int iBox=0; iBox<s->boxes->nLocalBoxes; iBox++) {
  int nIBox = s->boxes->nAtoms[iBox];
  //loop over neighbor boxes of iBox
  for (int jTmp=0; jTmp<nNbrBoxes; jTmp++) {
    int jBox = s->boxes->nbrBozes[iBox][jTmp];
    int nJBox = s->boxes->nAtoms[jBox];
    //loop over atoms in iBox
    for (int iOff =MAXATOMS*iBox; iOff<(iBox*MAXATOMS+nIBox; iOff++) {
      //loop over itoms in jBox
      for (int jOff=MAXATOMS*jBox; jOff<(jBox*MAXATOMS+nJBox; jOff++) {
          //physics
```

RAJA version:

```
//loop over local boxes
RAJA::forall_segments<LinkCell>(*s->isLocal, [=] (RAJA::IndexSet *iBox) {
  //loop over neighbor boxes of iBox
  RAJA::IndexSet *iBoxNeighbors =
      static_cast<RAJA::IndexSet*>(iBox->getSegment(0)->getPrivate()) ;
  RAJA::forall<linkCellWork>(*iBoxNeighbors, [=] (int jOff) {
    //loop over atoms in iBox
    RAJA::forall<linkCellWork>(*iBox, [&] (int iOff) {
          //physics (minimal changes to code)
```

The main change to the source code was to replace OpenMP link cell traversal with RAJA::forall_segments. Once in place, the RAJA interface hides many low level implementation details that otherwise would need to be repeated for multiple loops in the code.

Once RAJA was inserted, it was easy to try different parallel algorithms in key kernels. We explored different IndexSets and schedules defining different policies, without any additional code changes.

Options for *LinkCells:*

- seq_segit (Sequential)
- omp_parallel_segit (Parallel OpenMP)
- omp_taskgraph_interval (Task graph with round robin)
- omp_taskgraph_segit (Task graph blocked)

Options for *IndexSets:*

- Linear (no dependencies)
- WaveFront (linkCell dependent waits)

LinkCellWork is a composed type: *ExecPolicy<seq_segit, simd_exec>*.

With some constraints, the developer can use a combination of execution options.

Reduction operations in RAJA are simple

Original CoMD:

```
real_t v0 = 0.0, v1 = 0.0, v2 = 0.0, v3 = 0.0;

// sum the momenta and particle masses
#pragma omp parallel for reduction(+:v0) reduction(+:v1) reduction(+:v2) reduction(+:v3)
for (int iBox=0; iBox<s->boxes->nLocalBoxes; ++iBox) {
  for (int iOff=MAXATOMS*iBox, ii=0; ii<s->boxes->nAtoms[iBox]; ++ii, ++iOff) {
    v0 += s->atoms->p[iOff][0];
    v1 += s->atoms->p[iOff][1];
    v2 += s->atoms->p[iOff][2];

    int iSpecies = s->atoms->iSpecies[iOff];
    v3 += s->species[iSpecies].mass;
  }
}
```

RAJA version:

```
ReduceSum<ReducePolicy, real_t> v0(0.0), v1(0.0), v2(0.0), v3(0.0);
// sum the momenta and particle masses
RAJA::forall<atomWork>(*s->isLocal, [&] (int iOff) {
  v0 += s->atoms->p[iOff][0];
  v1 += s->atoms->p[iOff][1];
  v2 += s->atoms->p[iOff][2];

  int iSpecies = s->atoms->iSpecies[iOff];
  v3 += s->species[iSpecies].mass;
});
```

ReducePolicy options:

- seq_reduce
- omp_reduce
- cuda_reduce

The algorithm in the loop body is the same for Original and RAJA code. Reductions are supported through a simple change to scalar type .

RAJA enables custom schedules and dependencies

We developed a custom schedule for CoMD that uses a dependence graph to guarantee that there will be no data races between the threads while computing forces once per pair of atoms, and avoiding data privatization required for the OpenMP version of CoMD.

The wavefront algorithm is described as follows:

1. Assign IndexSet segments to threads (spatially partition simulation space).
2. Define segment dependencies, including periodic boundaries, to guarantee no data races by neighboring threads.
3. Order segments as a wavefront to minimize waiting for dependencies to complete.

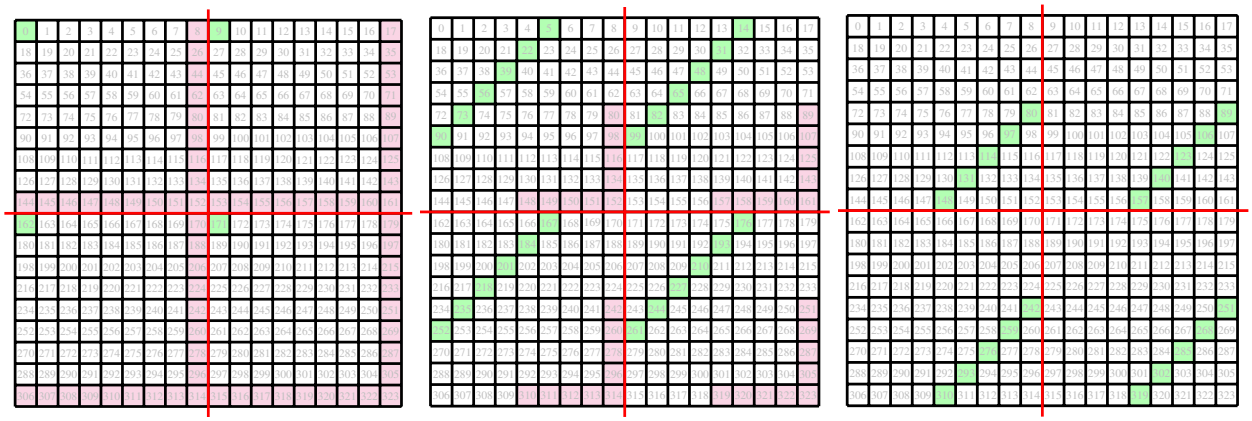


Figure 2.9: Wavefront Schedule

Figure 2.9 demonstrates the wavefront schedule for an iteration of CoMD. The red lines indicate how LinkCells are divided between threads. We show the currently processed LinkCells in green, and the LinkCells that are waiting on other LinkCells to be executed in red. As the threads process LinkCells, they write to their neighbors. Once the LinkCells are processed and no longer need to write to their neighbors' data, the dependencies are satisfied. When the iteration is finished, we reset the schedule and the dependencies to their initial state.

IndexSets support complicated schedules and dependency management. We use dependencies to remove data races.

2.2.4 Performance

Node-level performance data was collected using the following machine configurations:

- **Intel Xeon x86** experiments ran on a Linux cluster with nodes consisting of two Intel E5-2680 processors running at 2.8 GHz, each with 10 cores, with 24GB main memory per node. We used GCC 4.9.2 and Intel 15.0.133 compilers.
- **IBM Power8** results were run on POWER8 processor itself, manufactured on a 22 nm process, with 12 eight-way multithreaded cores running at 4 GHz. We used xlC and clang++ compilers.
- **Intel Xeon Phi** experiments ran on Compton Intel Sandy Bridge and Knights Corner. We utilized the 57-core 1.1GHz KNC-C0 with 6GB/RAM processes, using Intel 15.2.164 Compiler.
- **NVIDIA GPU** results ran on a 16 core Intel Xeon CPU with 4 Tesla K80 graphics boards, using a single Tesla graphics board.

The following figures compare performance of CoMD to performance of the two CoMD-RAJA versions on the tested platforms. Baseline implementation is the implementation of CoMD from GitHub, with the type casts to make it possible to compile it with a C++ compiler so that we are able to compare the results using the same compiler for all the versions. RAJA-reference is the initial RAJA port of CoMD, which is functionally equivalent to the Baseline implementation. RAJA-schedule is the wavefront schedule version of CoMD, performing force computations once; without RAJA, this new schedule would require a code rewrite of CoMD.

Figure 2.10 shows the figure of merit, microseconds per atom, for the LJ kernel on x86 platform with GNU and Intel compilers for a problem size of 131K atoms. Figure 2.11 shows the figure of merit, microseconds per atom, for the EAM kernel on x86 platform with GNU and Intel compilers for a problem size of 131K atoms. Execution time decreases up to 16 processes because there are only 20 physical processes. On a higher process count, hyperthreading does not hurt the performance significantly.

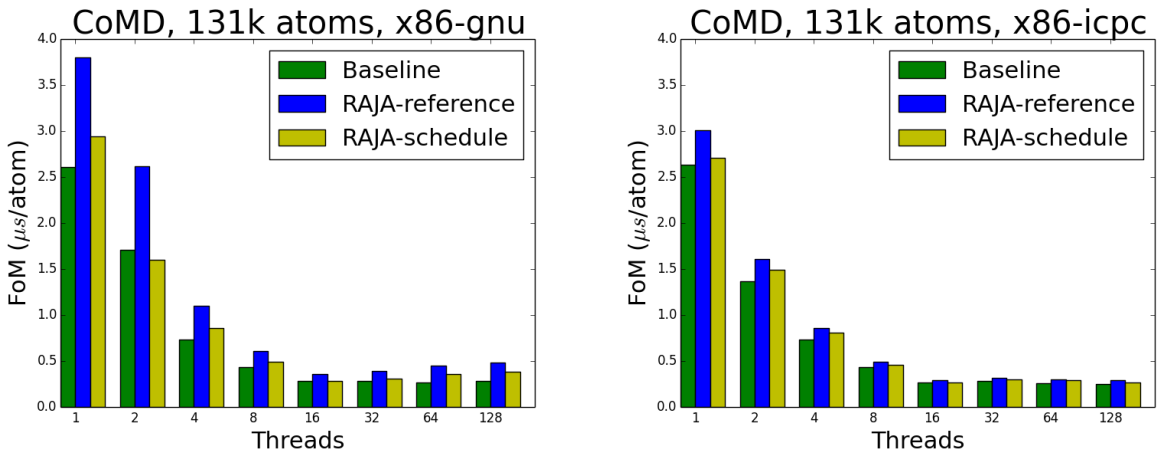


Figure 2.10: CoMD on x86, LJ force computation. Problem size 131K atoms.

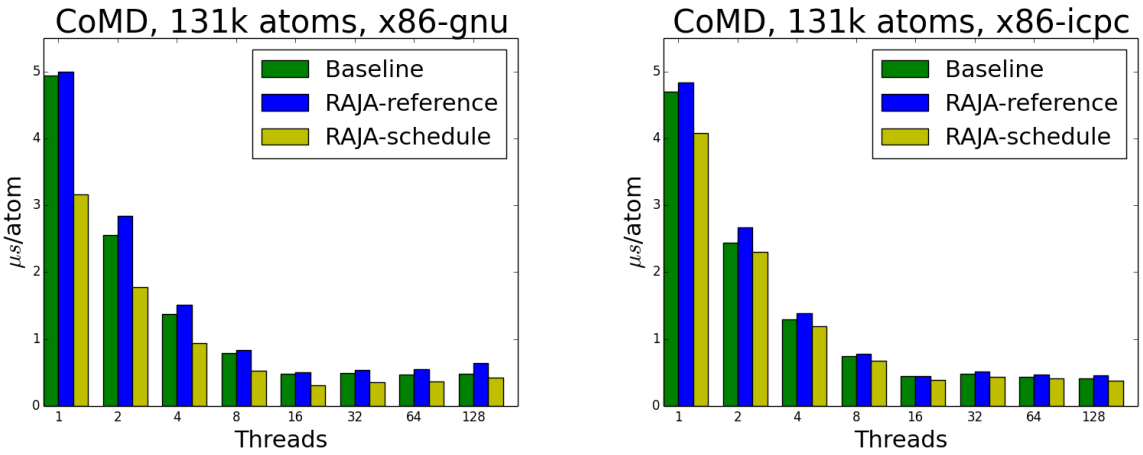


Figure 2.11: CoMD on x86, EAM force computation

Figure 2.12 shows the figure of merit on Power 8 with IBM and Clang compilers, for both the LJ and EAM kernels and a problem size of 1M atoms. Power 8 we ran on is early release hardware and software with a number of unresolved issues, e.g., OpenMP reductions do not currently work with the Clang compiler. Execution time decreases up to 16 processes because there are only 20 physical processes. Results show performance continues to improve with threads added as long as more physical processes are available.

Figure 2.13 shows the figure of merit on MIC with Intel compiler, for both the LJ and EAM kernels and a problem size of 1M atoms. Results show performance continues to improve with threads added as long as more physical processes are available.

Figure 2.14 shows the figure of merit of RAJA-reference on GPU with NVIDIA compiler relative to the baseline Intel compiler, for a problem size of 2K and 16K atoms. We were not yet able to run the main force loop on the GPU. However, RAJA allows the flexibility to use different execution policies for different loops within an application, so we ran the force loop on the CPU and the rest of the loops on the GPU. Running some loops on the CPU and others on the GPU resulted in data transfers between the CPU and the GPU, resulting in poor performance of the mini-app overall.

2.2.5 Productivity

We enabled compilation of CoMD with a C++ compiler, primarily by adding type casts. This change impacted 47 lines of code. Figure 2.15 shows the number of changes we made to the source code to convert it

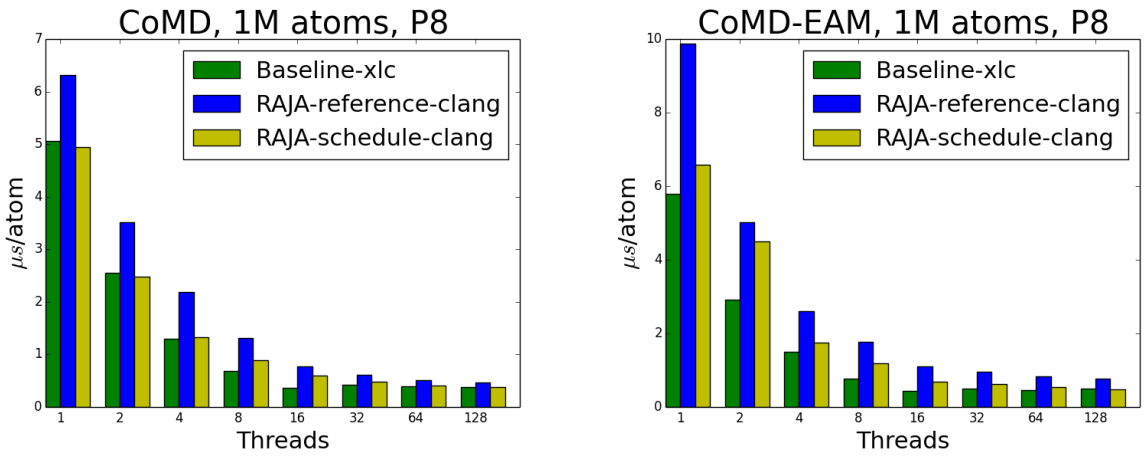


Figure 2.12: CoMD on Power8 with xlC and Clang compilers, LJ and EAM forces

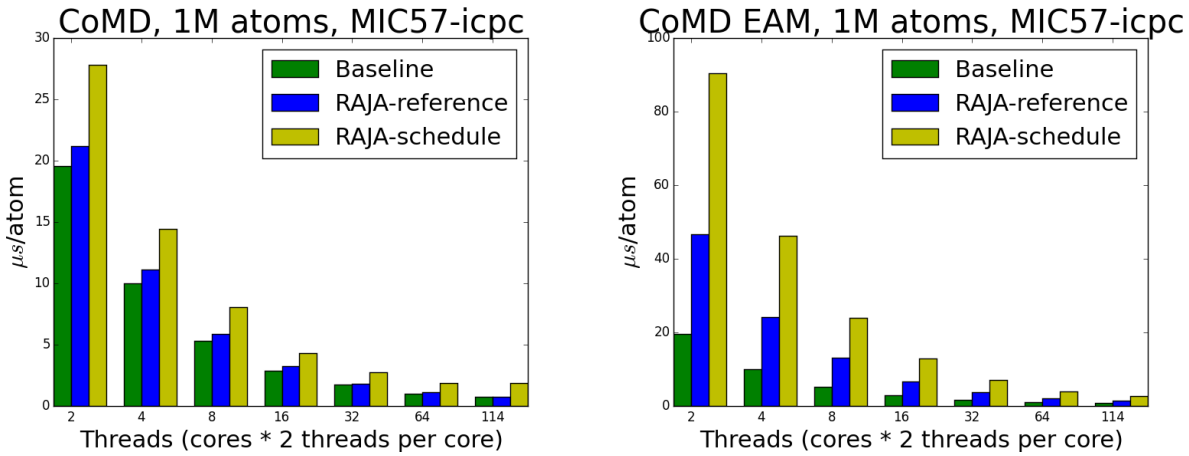


Figure 2.13: CoMD on MIC with Intel compiler, LJ and EAM forces. Problem size 1M atoms.

to RAJA, both in terms of the number of statements changed and the number of lines changed. The number of lines modified to support RAJA was 2%.

2.2.6 Summary

Overall, minimal changes to CoMD source code were necessary to make it run with RAJA (2% of the lines of code modified.). Without further source code changes, we were able to experiment with different schedules, including a schedule that allows performing half the computation of an OpenMP version by guaranteeing no data races, which without a RAJA abstraction layer would require a code rewrite. RAJA provides programming model specific reductions and the flexibility to easily run different loops in the code differently. Performance depends greatly on inlining.

2.3 Kripke

2.3.1 Kripke Description

Kripke is a simple, scalable, 3D Sn deterministic particle transport code. Its primary purpose is to research how data layout, programming paradigms and architectures effect the implementation and performance of

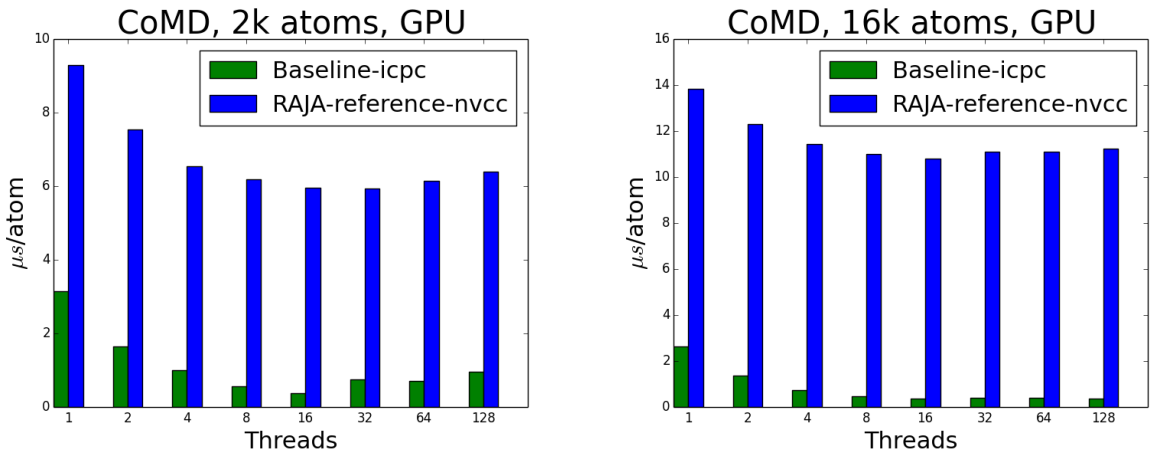


Figure 2.14: CoMD on GPU with NVIDIA compiler

File	Orig #lines	RAJA mod	RAJA add	RAJA subtr	C to C++*
CoMD.c	1075		15 (1.4%)		3 (0.3%)
eam.c	839	10 (1.2%)	16 (1.9%)	37 (4.4%)	8 (0.1%)
initAtoms.c	246	9 (0.4%)	5 (2.0%)	30 (12.2%)	1 (0.04%)
ljForce.c	232	6 (3.7%)	7 (3.0%)	18 (7.8%)	
timestep.c	155	11 (7.1%)	13 (8.4%)	15 (9.7%)	
timestep.h	16		1 (6.3%)		
CoMDTypes	85		9 (10.6%)		
Total	**5644**	**36 (0.6%)**	**66 (1.2%)**	**100 (1.8%)**	**47 (0.8%)**

Figure 2.15: CoMD productivity metrics.

Sn transport. A main goal of Kripke is investigating how different data-layouts affect instruction, thread and task level parallelism, and what the implications are on overall solver performance.

Kripke supports storage of angular fluxes (Ψ, or Psi) using all six striding orders (or "nestings") of Directions (D), Groups (G), and Zones (Z), and provides computational kernels specifically written for each of these nestings. Most Sn transport codes are designed around one of these nestings, which is an inflexibility that leads to software engineering compromises when porting to new architectures and programming paradigms.

Early research has found that the problem dimensions (zones, groups, directions, scattering order) and the scaling (number of threads and MPI tasks), can make a profound difference in the performance of each of these nestings. To our knowledge this is a capability unique to Kripke, and should provide key insight into how data-layout effects Sn solver performance. An asynchronous MPI-based parallel sweep algorithm is provided, which employs the concepts of Group Sets (GS) Zone Sets (ZS), and Direction Sets (DS), borrowed from the Texas A&M code PDT.

As we explore new architectures and programming paradigms with Kripke, we will be able to incorporate these findings and ideas into our larger codes. The main advantages of using Kripke for this exploration is that it's light-weight (ie. easily refactored and modified), and it gets us closer to the real question we want answered: *"What is the best way to **layout and implement an Sn code** on a given architecture+programming-model?"* instead of the more commonly asked question *"What is the best way to **map my existing Sn code** to a given architecture+programming-model?"*.

More info: `https://codesign.llnl.gov/kripke.php`

2.3.2 Evaluation of Various Programming Models

We evaluated the Sweep Kernel in Kripke using various different programming models and abstractions. We focused our attention on the Sweep Kernel because its deep loop nesting and semi-sequential nature makes achieving high performance difficult. For these reasons, the sweep kernel is a major concern when we examine writing discrete ordinates transport codes using programming model abstractions. We first provide an overview of the variants we evaluated, and then provide some more detailed descriptions.

- **Original hand coded serial code** The original version of Kripke uses explicity coded kernels for each of the six possible data layouts (permutations of directions, groups and zones).
- **TLoops** This variant uses C++ template metaprogramming to abstract the loop nesting order and the data layouts. Data layout and loop nestings are chosen by template parameters, so only one source code version of each kernel is needed to model all six permutations.
- **Sierra Center-of-Excellence (OpenMP, CUDA)** This variant developed within the *Sierra Center of Excellence*, and implements the "hyperplane sweep method". No programming abstractions were used, only hand tuned algorithms written directly in OpenMP and CUDA.
- **RAJA (Serial, OpenMP, CUDA)** Two approaches were taken with RAJA, but both allow a single kernel to be targeted for serial, OpenMP and CUDA execution. The data layout ZDG was selected for this work.

  The first approach used a vector of IndexSets, with one IndexSet per hyperplane. Each Segment represents a single zone, and the set of directions and groups in that zone. A custom "forall" execution policy was created which provided the nested loops over directions and groups.

  The second approach used a single IndexSet, with each Segment representing a hyperplane. RAJA supplied execution policies were used to iterate over zones, and the direction and group loops were not abstracted but rather provided within the kernel body.
- **OCCA DSL (OpenMP)** OCCA [9] is a programming model a Domain Specific Language which is an annotated extension of C. OCCA uses source-to-source translation, which hides the abstractions in the programming model from the underlying C compiler, which should reduce or eliminate the "overheads" that we see from models like RAJA.

2.3.3 Original Kripke Details

In the base version of Kripke, we explicitly write kernels for each of the 6 data layouts. Hand optimizations are limited to the use of the "restrict" keyword, and loop invariant hoisting transformations (discussed below). Since there is no programming model abstractions used (aside from OpenMP directives), there are no abstraction overheads. This version of Kripke *should* be the easiest for the compilers to optimize, and should therefore be the fastest serial variant.

Manual writing of kernels allows for inter-loop logic to be placed. This allows for fetching values from memory, array index calculations, and other floating point operations, to be "hoisted" to the outermost loop nesting in which they remain invariant.

For example, if we look at the loop:

```
double * __restrict__ a = ...;
double * __restrict__ b = ....;

for(int i = 0;i < ni;++ i){
  for(int j = 0;j < nj;++ j){
    a[i*nj+j] += b[i] * 2.0 * c;
  }
}
```

The value of $b[i]$ is invariant over the inner-loop, so it can be hoisted to before the j loop. The value of $2.0 * c$ is invariant over both loops, so it can be hoisted to before the i loop. And the calculation of $i + j * ni$ can be separated and partially hoisted. The result is:

```
double c2 = 2.0*c;
for(int i = 0;i < ni;++ i){
  double b_i = b[i];
  double * __restrict__ a_i = a + i*nj;
  for(int j = 0;j < nj;++ j){
    a_i[j] += b_i * c2;
  }
}
```

This manual hoisting is a common practice when writing kernels by hand, and often has a profound affect on performance. The above code will typically generate fewer instructions, and allow more compilers to apply SIMD instructions for the inner loop. Modern compilers can perform many of these hositing transformations themselves, but their ability to match a programmers manual hoisting varies greatly from compiler to compiler.

To maintain a consistent level of optimization across kernels in Kripke, we only employ loop-invariant hoisting transformations and the use of the "restrict" keyword. We never have aliased pointers in Kripke, so the liberal use of "restrict" is always safe, and is the only way to achieve consistent generation of SIMD instructions by the compilers.

2.3.4 CoE OpenMP and CUDA Variants

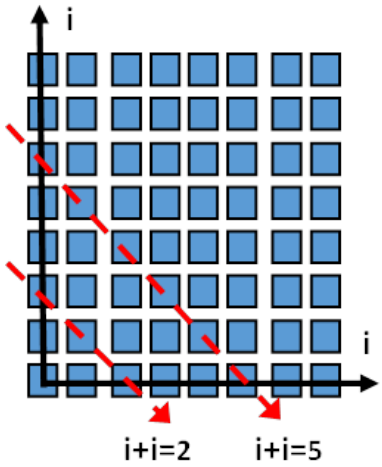


Figure 2.16: Hyperplane sweep method.

This variant is a cumulation of effort by Leopold Grinberg (IBM) and Steven Rennech (NVidia) under contract with the *Sierra Center of Excellence* at LLNL. They implemented a hyperplane sweep method, as seen in Figure 2.16, in both OpenMP and CUDA. Transport solves have "upwind" dependencies based on the direction of particle flow, which causes the sweep (or wavefront-like) pattern. Each hyperplane is an independent set of unknowns, which allows for threading over zones in a hyperplane. Hyperplanes have dependencies upon their upwind neighbor, requiring sequential solves of these hyperplanes.

For this variant, no programming abstractions were used, only and tuned algorithms written directly in OpenMP and CUDA. The goal of this work was work out a "lower-bound" of performance that we could expect when we start applying programming model abstractions to target platforms such as Sierra/CUDA.

For the CUDA implementation, a combination of explicit data movement and zero-copy are used between the host and accelerator. In both implementations, pinned memory is used on the CPU.

This CoE OpenMP hyperplane implementation is used as a starting point for our RAJA implementation.

2.3.5 TLoops Variant

We explored using C++ templates and lambda expressions to abstract multiply-nested loops, much in the way RAJA abstracts single loops. A template parameter is used to permute the loop-nesting order of 3 loop execution policies, which are described using functors. The loop body is passed in as a C++11 lambda expression, making the nested loop function look like a C++ loop:

```
...
SweepLoopVars vars;
LoopDGZ<NEST> loop;
loop(LoopLocalDirs(dims), LoopLocalGroups(dims), LoopSweepZones(grid_data, extent), vars,
[=](SweepLoopVars &v){

// Fetch Coefficients, and compute indices
double xcos_dxi = 2.0 * direction[d].xcos / dx[v.i + 1];
int psi_idx = layout.idxPsi(dims, v.dir, v.group, v.zone);
...

// Apply diamond-difference relationship for
// zone, direction and group
...

});
```

The data layout of each multi-dimensional variable is also abstracted by using "layout" objects, which perform array index calculations. These layout objects are also templated on the same parameter as the loop-permutation, which allows for the variables and loops to be permuted in unison.

Only one version of each kernel is needed to provide each of the six data layouts, but we lose the manual loop invariant optimizations and have to rely on compiler optimizers.

2.3.6 RAJA Variant

Two approaches examined with RAJA:

- One IndexSet per hyperplane, using a custom execution policy. The nested direction and group loops are handled by the execution policy, hiding all loops from the kernel writer.
- One IndexSet, with one Segment per hyperplane. Directions and group loop are inside the kernel body. This uses only built-in RAJA execution policies, and is the most straight forward to implement.

For RAJA targeting CUDA, memory movement is handled entirely through NVidia's Unified Memory (UM) system. The current support through UM is currently only though software, and is probably less efficient than what we will see on Sierra. In order to separate the cost of memory movement from the efficiency of the kernels, we have two data points for the RAJA+CUDA variant: One that keeps all of the data on the GPU (the "nomove" variant) and one that moves the data before and after the sweep kernel is called.

One IndexSet per Hyperplane

The RAJA approach defined one IndexSet per hyperplane slice. Each segment in an IndexSet represents one Zone, and each element in the segment represnts one (Group,Direction) tuple. The Lambda is then called for every (Zone, Group, Direction) tuple, allowing for the greatest flexibility in how the execution policy is implemented.

```
for(int slice = 0;slice < extent.hp_sets.size();++ slice) {

  IndexSet &slice_set = extent.hp_sets[slice];

  // Get pointers to data
  double * SRESTRICT psi_lf = i_plane.ptr();
```

```
  ...

  // Launch RAJA sweep kernel on slice
  forall<SweepExecPolicy>(slice_set,
    [=] KDEVICE (int i, int j, int k, int z, int d, int group, int offset) {

      // Fetch all constants (15 doubles)
      double xcos = direction[d].xcos;
      ...

      // Apply diamond-difference relationship for
      // zone, direction and group
      ...
    });
}
```

This approach has the advantage of allowing a custom execution policy to easily map the kernel to either OpenMP or CUDA in a native way. For OpenMP, threads are used across the zones in a hyperplane. For CUDA, a sequence of kernels are launched for each hyperplane, and the directions and groups are mapped to threads and blocks, giving each unknown it's own thread.

The drawbacks to this approach include:

1. All constant values are all fetched each time the inner loop-body is executed. For a CUDA execution policy, this may be unavoidable (without an explicit CUDA implementation). For a serial or OpenMP execution policy, the compiler should be able to expand the lambda expression, and determine which fetches should be factored out into outer loop nestings. However this is not happening and produces significant impact on performance.

2. Different data layouts require explicit re-writing of the kernel. This kernel assumes the data layout is psi[Z][D][G]. While it is possible to re-write the forall execution policy to support psi[Z][G][D], it is not possible to move Zones into an inner stride due to the structure of the outer for loop.

One IndexSet, Explicit Direction and group Loops

We explored an alternative implementation using RAJA in which the IndexSet described just the zone iteration pattern. We used each segment of the IndexSet to represent one hyper-plane, where each Segment contained the indices of each zone in the hyperplane. Since the "forall" only iterates over zones, we needed to explicity look over directions and groups inside of the lambda kernel.

```
  IndexSet &zone_iset = ...

  // Get pointers to data
  double * SRESTRICT psi_lf = i_plane.ptr();
  ...

  // Launch RAJA sweep kernel on slice
  forall<ZoneSweepExecPolicy>(zone_iset,
    [=] KDEVICE (int zone) {

      for(int d = 0;d < num_directions;++ d){
        // Fetch all constants (15 doubles)
        double xcos = direction[d].xcos;
        ...

        for(int group = 0;group < num_groups;++ group){
          // Apply diamond-difference relationship for
          // zone, direction and group
          ...
        }
      }
    });
}
```

This approach had the benefit of utilizing the built-in RAJA execution policies, using a sequential iteration over segments, and an OpenMP iteration within a segment. Performance of this approach was identical to the initial approach.

The main drawback is that it does not map well to CUDA, since it is impossible to map the direction and group loops to threads. This leads to a major loss of parallelism compared the the "one IndexSet per hyperplane" method.

Restrict and Device Keywords

Let's use the following example code to motivate the following points:

```
for(int slice = 0;slice < extent.hp_sets.size();++ slice) {
  IndexSet &slice_set = extent.hp_sets[slice];

  double * __restrict__ psi_ptr = sdom->psi->ptr();
  ...
  forall<SweepExecPolicy>(slice_set, [=] __device__ (...) {
        psi_ptr[i] = ...
      });
}
```

If we compile the code with nvcc, the `restrict` keyword causes a lambda template deduction error. This is most likely due to the beta-release status of NVIDIA's device lambda capture mechanism.

If we compile the code with a host compiler, such as icpc or g++, the `device` keyword is not recognized, and the code will not compile.

We can't have it both ways. We want restrict for good CPU performance (ie SIMD generation), but we need the use the `device` keyword to target GPUs with nvcc. The current solution is to use macros, which turn on *either* the `device` or the `restrict` keyword, depending on what device is being targeted. This forces a *compile* time choice of the targetd device, which precludes the use of runtime scheduling decisions. This is important because the choice of whether to run on the CPU or GPU may depend on parameters that cannot be determined at compile time, such as problem size.

2.3.7 OCCA

OCCA provides a domain-specific-language that is compiled (JIT) into C++, then compiled into machine code with a users compiler (GCC or ICC, etc). Porting of Kripke to OCCA was work completed by David Medina during a summer 2015 internship at LLNL, is the product his PhD thesus research and is available at `http://libocca.org/`

OCCA has the advantage of adding the abstractions we are looking at with RAJA, etc., but produces C++ code without complex template constructs. It has the potential to reduce the burden on the compiler, which may result in better performance.

The OCCA DSL is an extension of C which uses annotations to describe loop-nesting orders, data layouts, and how loops map to different threading models (such as OpenMP or CUDA):

```
kernel void sweep(...
  double * restrict rhs @(dim(PSI_DIM), idxOrder(PSI_IDX)),
  double * restrict psi @(dim(PSI_DIM), idxOrder(PSI_IDX)),
  double * restrict psi_lf @(dim(PSI_LF_DIM), idxOrder(PSI_IDX)),
  double * restrict psi_fr @(dim(PSI_FR_DIM), idxOrder(PSI_IDX)),
  double * restrict psi_bo @(dim(PSI_BO_DIM), idxOrder(PSI_IDX)),
  const double * restrict sigt @(dim(SIGT_DIM), idxOrder(SIGT_IDX))) {

  for(int k; ...; ...; loopOrder(sw_KOrder)) {
    for(int j; ...; ...; loopOrder(sw_JOrder)) {
      for(int i; ...; ...; loopOrder(sw_IOrder)) {
```

```
        for(int g; ...; ...; outer, loopOrder(sw_GOrder)) {
          for(int d; ...; ...; inner, loopOrder(sw_DOrder)) {
            // Fetch all constants (15 doubles)
            double xcos = direction[d].xcos;
            ...

            // Apply diamond-difference relationship for
            // zone, direction and group
            ...
          }
        }
      }
    }
  }
}
```

OCCA has the same "kernelization" issues that TLoops and RAJA have, forcing the compiler to do all of the loop-invariant optimizations. Since the OCCA runtime generates the intermediate C++ compiled kernel, we could "intercept" the C++ code and manually add back in the loop invariant optimizations. The results shown in Figure 2.17 show the original Kripke results, the OCCA results (without manual optimizations), and the OCCA results with these manual "prefetch" optimizations. From these results we can directly see

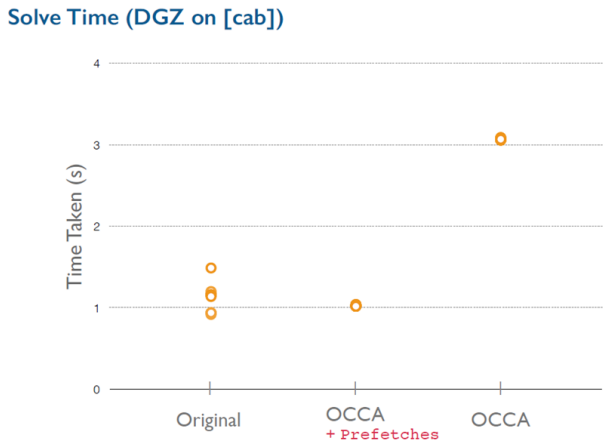


Figure 2.17: Performance of OCCA Kripke implementation, with and without manual "prefecth".

that adding these optimizations reclaim all of the lost performance incurred by moving from hand-coded C++ to the OCCA DSL. These results give us hope that either: the compiler vendors could add these optimizations to their compilers (which would benefit everyone), or that DSL's like OCCA could add these optimizations for the compilers. Furthermore, it shows that using and abstraction or DSL *could* have a zero-overhead impact at worst, and performance improvements if the DSL can give the compiler information that it might not otherwise be able to determine for optimization.

2.3.8 Performance Results

In Figure 2.18 we show the performance of the RAJA Kripke variant across several different architectures. The K20xm variant is using the "nomove" option to eliminate host-to-gpu memory movement. The Power8 is running 1 hardware thread per core, due to platform support issues on LLNL's rzmist machine.

Not suprisingly, we were able to achieve roughly the same performance between the CoE and RAJA OpenMP variants, as seen in Figure 2.19. Since the RAJA variant was based on the CoE OpenMP varient, the only code modifications were replacing the "for" loops with a "foreach" RAJA function call.

The serial performance of Kripke when switching from hand-coded loops, to TLoops abstraction and to RAJA demonstrates some of the performance issues we have run into (Figure 2.20. It's currently unclear

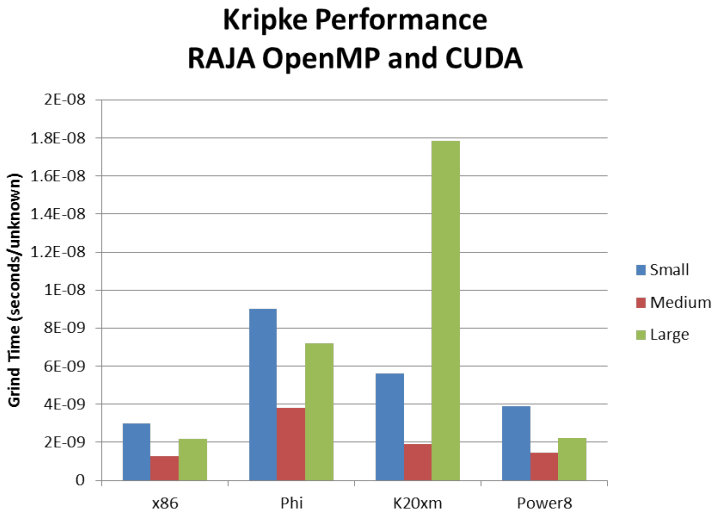


Figure 2.18: Performance of RAJA Kripke implementation across multiple architectures.

why the TLoops abstraction performs better than the RAJA abstraction, since they both have the same inner lambda expression, however the TLoops variant has a much more complex layer of abstraction on the loop execution policy.

Figure 2.21 shows that we can achieve almost the same performance as the CoE GPU code when we are not dealing with memory movement. The hand coded CoE CUDA makes heavy use of manual data movement and zero copy, while the RAJA version uses Unified Memory and relies on the runtime to perform all of the data movement. The CoE CUDA version of the sweep kernel took 3 months of effort, while the RAJA variant was implemented by someone (without prior RAJA experience) in 3 *hours*. Furthermore, the RAJA version uses the same code to be targeted for both serial, OpenMP and CUDA execution.

Not suprisingly, like on the x86-64 platform, were able to achieve roughly the same performance between the CoE and RAJA OpenMP variants, as seen in Figure 2.22.

2.3.9 Lessons Learned

Kernelization of nested loops pushes loop invariants into the inner-most loop body. From timing analysis, it appears that many compilers struggle to make the same loop invariant optimizations that humans can make, which causes excess memory and floating point operations. We believe that it should be possible to improve compiler optimizations to alleviate these inefficiencies.

RAJA has distinct productivity advantages over using programming models, like OpenMP and CUDA, directly. The CoE CUDA version of the sweep kernel took 3 months of effort, while the RAJA variant was implemented by someone (without prior RAJA experience) in 3 *hours*.

OpenMP and CUDA language incompatibilities make runtime CPU versus GPU scheduling choices impossible. This is specifically due to the non-standard restrict and device keywords causing compiler specific incompatibilities. We believe that simple language modifications could remedy this.

More investigation is needed to extend RAJA to support deeply nested kernels. Merging concepts from TLoops into RAJA is something we plan to explore in the future.

2.4 Conclusions

We assert that this report shows that RAJA has demonstrated significant progress toward several key design goals. Most notably, RAJA enables CPU-GPU portability in diverse applications with modest changes to the source code. Here, we have shown using CoMD (Section 2.2) and Kripke (Section 2.3) that basic RAJA transformations allow OpenMP CPU and CUDA GPU execution without further code changes. In particular,

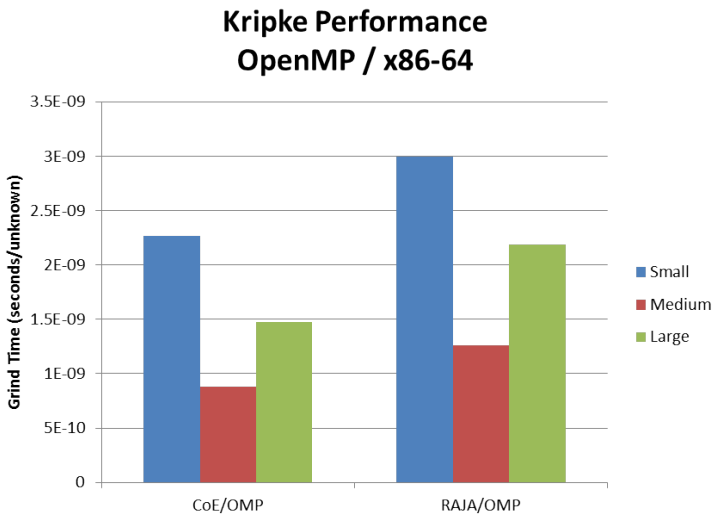


Figure 2.19: Comparison of CoE and RAJA OpenMP implementation on x86-64.

algorithm restructuring and multiple code versions are not required. In several cases, including other RAJA assessments not discussed in this report, we have found that RAJA actually can make application source code easier to understand, more flexible, and less error-prone.

RAJA also provides portable reduction operations that obviate the need for coding variations to work with different programming model back-ends (CoMD discussion in Section 2.2). In other contexts, we have shown that multiple different RAJA reductions can be combined in the same loop execution context along with other operations. Often, parallel programming models constrain programmers to write reductions as distinct parallel constructs.

RAJA IndexSets provide a powerful balance between runtime flexibility and compile-time optimizations. That is, traversal specializations for different segment types are generated and optimized at compile-time, while IndexSet segments can be defined and configured at runtime (i.e., code paths through compile-time specializations). In Section 2.1.5, we presented several examples that show additional capabilities of RAJA IndexSets and traversal methods and their potential to enable significant performance improvements. These included:

- Partitioning an iteration space into IndexSet segments can enable "in place" SIMD vectorization for unstructured mesh operations by exposing stride-1 index ranges without resorting to gather/scatter operations.
- Reordering loop iterations using an IndexSet can enable a non-thread safe algorithm to run in parallel without rewriting it (e.g., accumulating mesh element data to surrounding nodes). While rewriting such an algorithm will likely yield better performance, the performance gains resulting from basic parallel execution can be significant.
- IndexSet segments can represent arbitrary tilings of loop iterations that can work together with data allocation to improve NUMA (Non-Uniform Memory Access) behavior and increase performance (e.g., LULESH OpenMP variants).
- IndexSets support task dependency scheduling of segments which can be used to coarsen the granularity of multithreading fork-join operations (e.g., LULESH "lock free" dependence graph) or ensure that data races do not occur (e.g., CoMD wave-front schedule).

These items and others show that complex algorithm restructuring can be performed behind an abstraction layer like RAJA which simplifies the exploration of implementation alternatives without disrupting application source code.

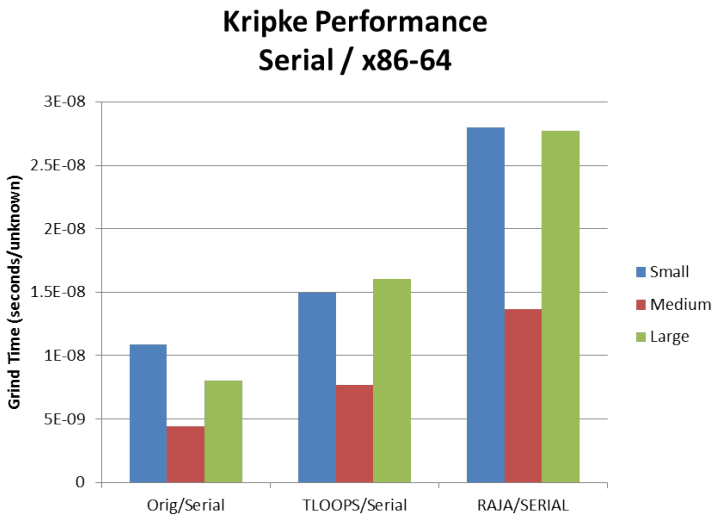


Figure 2.20: Comparison of serial performance of various Kripke variants.

2.4.1 Lessons Learned

A variety of valuable lessons were reinforced and/or learned as a result of this L2 effort. An especially important point is that it is difficult to draw firm conclusions about a "best approach" to the performance portability challenge for production ASC applications based on experimentation with proxy applications using early release hardware and software. Proxy apps are necessarily small and incomplete representations of real multiphysics applications. It is unclear how tuning a simplified algorithm in isolation will affect aggregate performance of a full application. We have found that test-bed platforms are sometimes unstable and have idiosyncrasies that make them hard to work with and which need to be understood. Also, compilers struggle to optimize well in when faced with newer language features, such as C++ lambda functions, and software abstractions. In addition, programming model implementations are immature (e.g., OpenMP 4.x, CUDA host-to-device lambda kernel launch, etc.). Lastly, good memory and performance analysis tools are essential. In general, tool support must improve to work more seamlessly with software abstractions to make it easier to analyze how codes run on machines with heterogeneous processors and complex memory hierarchies.

Although it is well-known, our studies for this L2 show that data placement and motion must be managed more carefully in HPC applications than ever before to achieve high performance. For example, memory pools are essential for handling temporary data, as exemplified in a number of studies of LULESH, which like many of our true applications - performs a number of dynamic memory allocations and frees each timestep. Due to memory constraints, a typical multiphysics code allocates and frees temporary arrays repeatedly during each timestep. We have observed that replacing this behavior with a memory pool in LULESH can more than halve the runtime. For all of our GPU runs in this effort, we relied exclusively on CUDA Unified Memory [23]. This greatly simplifies CPU-GPU programming by allowing a single pointer to access "managed" memory, pages of which are automatically transferred between host and device memory spaces as needed. The current software implementation of unified memory works pretty well, and we expect performance to improve with hardware support through NVIDIA's NVLINK architecture (e.g., in Pascal and Volta architectures). However, we expect that explicit data transfers done properly will almost always yield the best performance and so we do not believe we can avoid them entirely in the future. In Section A.1, we elaborate on additional issues and considerations for using CUDA Unified Memory.

Before we describe future work, we feel it is instructive to discuss some issues related to passing data objects, such as C++ class instances, to CUDA device kernels when they are created on the host. In particular, objects must be passed to the device by value and their state must be preserved across kernel invocations. The code snippet below shows a motivating use case. Here, we perform sum and min reductions on a GPU using a RAJA traversal. The reduction variables (i.e., class objects) are initialized on the host,

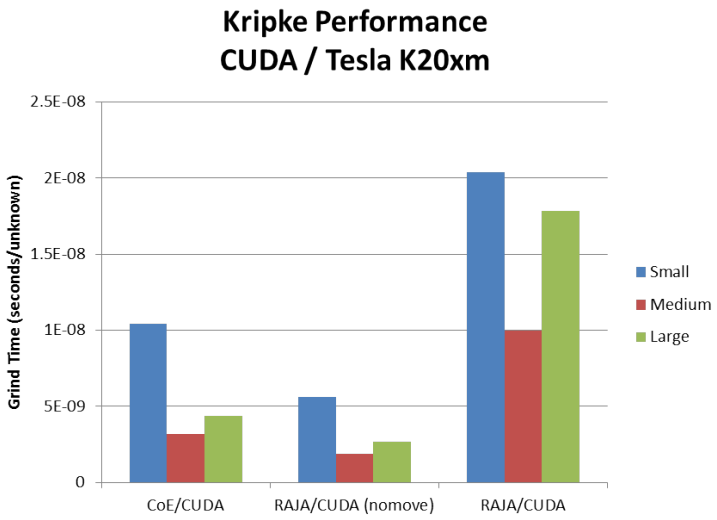


Figure 2.21: Comparison of CUDA (K20xm) performance of Kripke CoE and RAJA variants.

passed to a device kernel via the forall() method, and then used to retrieve the reduced values back on the host. Note that since there are two IndexSet segments and the loop contains over a million iterations, the kernel will be launched multiple times to execute the entire loop. The example is somewhat contrived, but nevertheless illustrates several salient points.

```
#define TEST_VEC_LEN  1024 * 1024

int main(int argc, char *argv[])
{
   double *dvalue ;

   //
   // Allocate a managed memory array and initialize all values to 0.0
   //
   cudaMallocManaged( (void **)&dvalue, sizeof(double)*TEST_VEC_LEN,
                      cudaMemAttachGlobal ) ) ;
   for (int i=0; i<TEST_VEC_LEN; ++i) {
      dvalue[i] = 0.0 ;
   }

   //
   // Create an index set with two range segments
   //
   RAJA::RangeSegment seg0(0, TEST_VEC_LEN/2);
   RAJA::RangeSegment seg1(TEST_VEC_LEN/2 + 1, TEST_VEC_LEN);

   RAJA::IndexSet iset;
   iset.push_back(seg0);
   iset.push_back(seg1);

   //
   // Set a random minimum value at a random location in the array
   //
   std::random_device rd;
   std::mt19937 mt(rd());
   std::uniform_real_distribution<double> dist(-10, 0);
   std::uniform_real_distribution<double> dist2(0, TEST_VEC_LEN-1);

   double min_value = dist(mt) ;
   int min_loc = int(dist2(mt));

   dvalue[min_loc] = min_value;
```

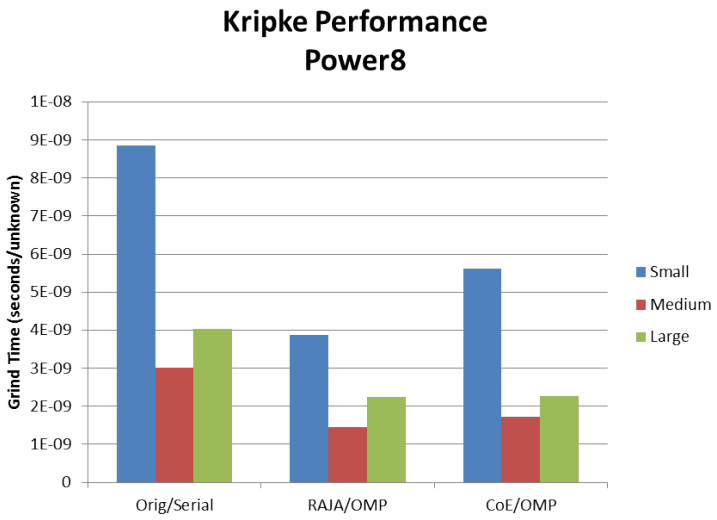


Figure 2.22: Comparison of Power8 performance of various Kripke variants.

```
    //
    // Create min and sum reduction objects and perform the reductions
    // on the GPU by launching kernels via a forall traversal
    //
    RAJA::ReduceMin<RAJA::cuda_reduce, double> dmin(DBL_MAX);
    RAJA::ReduceSum<RAJA::cuda_reduce, double> dsum(0.0);

    RAJA::forall<RAJA::cuda_exec>(iset, [=] __device__ (int i) {
       dmin.min(dvalue[i]) ;
       dsum += dvalue[i] ;
    } ) ;

    std::cout << "\n min, sum, value = "
              << dmin << " , " << dsum << " , " << min_value << std::endl;

    return 0;
}
```

There are several problems that could occur in this example if the reduction class initialization and copy operations are not implemented properly. First, passing the reduction objects by value to the traversal template and subsequently to the GPU kernel launch (which, again, is required) generates multiple object copies, some on the host and some on the GPU. Each function call that accepts an object by value is a potential place where object state could become out-of-sync. Also, kernel invocation against a class object could be pre-empted by the class destructor. So the original objects, or their copies, could fall out of scope completely and be destoyed. This could result in no GPU code being executed, or an inability to retrieve the reduction values back on the host. Second, when multiple kernels are run, one kernel could complete and call a destructor prematurely while another is running. This can happen even when a call to cudaDeviceSyncrhonize() is made in the destructor. Third, default copy constructors only support shallow copies. We need to maintain non-trivial state in the reduction objects to hold partially reduced values so the reduction can finalize properly across thread blocks. To avoid performance issues, we want to avoid deep memory copies and still make this work.

The key to making this all work properly is proper use of the C++ RAII (Resource Acquisition Is Initialization) idiom. This requires a custom copy constructor and a boolean class member to explicitly track whether an object has been created as a copy and only allow the original contructor and destructor to acquire and release resources [15]. The following code snippet shows the basic mechanics of this using the

RAJA CUDA min reduction class as an example.

The constructor executes only on the host; it records the fact that it is not a copy and sets some simple POD data members to managed state that is shared across reduction objects. The copy constructor and destructor can execute on either the host or device. The copy constructor, which is the only mechanism allowed to create object copies, copies the POD members and records the fact that each copy is indeed a copy. The destructor releases its hold on the shared data, but only when called on the host. For completeness, we note the basic properties of the reduction method accessor operator. The operator used to access the reduced value finalizes the reduction and executes only on the host. The reduction method operated only on the device and updates the reduced value in a unique location for each thread block. The basic elements of this implementation are important to keep in mind when implementing other code where objects are shared between host and device.

```
template <typename T>
class ReduceMin<cuda_reduce, T>
{
public:
   //
   // Constructor takes default value (default ctor is disabled).
   // Ctor only executes on the host.
   //
   explicit ReduceMin(T init_val)
   {
      // Constructed object is not a copy!
      m_is_copy = false;
      m_reduced_val = init_val;

      //
      // Set pointers and offsets into memory block shared by reduction objects
      //
      m_myID = getCudaReductionId();
      m_blockdata = getCudaReductionMemBlock() ;
      m_blockoffset = getCudaReductionMemBlockOffset(m_myID);

      cudaDeviceSynchronize();
   }

   //
   // Copy ctor executes on both host and device.
   //
   __host__ __device__ ReduceMin( const ReduceMin<cuda_reduce, T>& other )
   {
      *this = other;
       m_is_copy = true; // Flag object as a copy
   }

   //
   // Destructor executes on both host and device.
   // Destruction on host releases the unique id and shared memory block
   // for others to use.
   //
   __host__ __device__ ~ReduceMin<cuda_reduce, T>()
   {
      if (!m_is_copy) {
#if !defined( __CUDA_ARCH__ )
         releaseCudaReductionId(m_myID);
#endif
      }
   }

   //
   // Operator to retrieve reduced min value (before object is destroyed).
   // Accessor only operates on host.
   //
   operator T()
   {
      cudaDeviceSynchronize() ;
```

```
      // Finalize reduction and set m_reduced_val...

      return m_reduced_val;
   }

   //
   // Update reduced value into the proper shared memory block location.
   //
   __device__ ReduceMin<cuda_reduce, T> min(T val) const
   {
      // ...
   }

private:
   // ...
};
```

2.4.2 Future Work

Beyond completing the L2 milestone requirements, it is important that our work can make a a positive impact on real application codes, which is a key goal of co-design.

To make such an impact, the results, lessons learned, and issues encountered in our various studies must be propagated to application code teams and used to educate other developers. Specific issues we intend to focus on in discussions with code teams in the near term at LLNL include: addressing thread safety to enable fine-grained parallelism, general best practices for performance, proper usage of parallel programming models and software abstractions, and ways to bridge gaps between compiler and runtime deficiencies and application constraints.

Moving forward, we will continue to refine and extend RAJA concepts using additional proxy-app explorations as well as integration experiments with LLNL ASC codes. For example, we will explore ways to coordinate RAJA IndexSet configurations with centralized memory management routines in these codes. We will also continue working with compiler and hardware vendors and programming model standards committees to improve support for DOE HPC needs. We have been actively discussing issues discussed in A.1 and others with several vendors over the past couple of years issues and have been making steady progress toward solutions. Tri-lab co-design and Center-of-Excellence activities also play a central role. To be most effective, vendor engagement should be a *unified DOE effort*. We firmly believe could be resolved with proper investments (e.g., support contracts) and engagement with compiler vendors and programming model committees. Joint L2 milestone efforts such as this help increase inter-lab collaboration on common issues and move us toward common solutions.

Chapter 3

Sandia National Laboratories

For the FY15 ASC L2 Trilab Codesign milestone Sandia National Laboratories performed two main studies. The first study investigated three topics (performance, cross-platform portability and programmer productivity) when using OpenMP directives and the RAJA and Kokkos programming models available from LLNL and SNL respectively. The focus of this first study was the LULESH mini-application developed and maintained by LLNL. In the coming sections of the report the reader will find performance comparisons (and a demonstration of portability) for a variety of mini-application implementations produced during this study with varying levels of optimization. Of note is that the implementations utilized including optimizations across a number of programming models to help ensure claims that Kokkos can provide native-class application performance are valid. The second study performed during FY15 is a performance assessment of the MiniAero mini-application developed by Sandia. This mini-application was developed by the SIERRA Thermal-Fluid team at Sandia for the purposes of learning the Kokkos programming model and so is available in only a single implementation. For this report we studied its performance and scaling on a number of machines with the intent of providing insight into potential performance issues that may be experienced when similar algorithms are deployed on the forthcoming Trinity ASC ATS platform.

3.1 Kokkos Programming Model

The Kokkos programming model [17, 18, 16] is designed to act as an abstraction layer to address three important concerns for cross platform application development: (1) where computation is performed; (2), where data is allocated and, finally, (3) how data is accessed. By abstracting these concerns the mapping of each of these to modern high-performance computing architectures can be precisely remapped to provide strong performance. Thus the abstraction layers provide insulation for application developers from the details of rapidly changing hardware architectures.

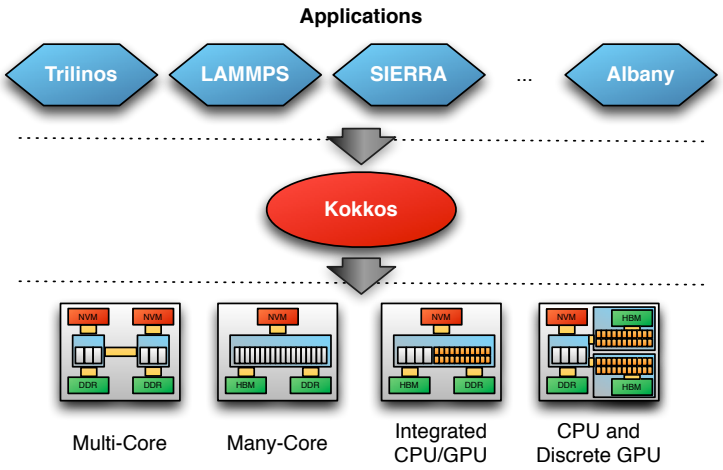
3.1.1 Abstraction Concepts in the Kokkos Model

Kokkos provides six key abstractions which are designed to transparently manage thread parallel computations and their data access patterns. This capability enables users' thread parallel computations to be both portable and performant across diverse and heterogeneous manycore architectures such as multicore CPUs, the Intel Xeon Phi, and NVIDIA GPUs. The six key abstractions are as follows:

1. Users' express parallel computations with **parallel patterns**; e.g., for-each, reduce, scan, and directed acyclic graph (DAG) of tasks.

2. Parallel computations occur within **execution spaces** of a heterogeneous architecture; e.g., latency-optimized CPU cores and throughput optimized GPU cores.

3. Parallel computations are scheduled according to **execution policies**; e.g., statically scheduled range [0..N) and dynamically scheduled thread teams.



4. Data are allocated within **memory spaces** of a heterogeneous architecture; e.g., in CPU main memory and GPU memory.

5. Data are allocated through multidimensional arrays with **polymorphic layout** that specifies how an arrays multi-index domain space is mapped to an allocation within a memory space.

6. Arrays may be annotated with **access intent traits** such as "random acces" or "atomic access". Kokkos may use these traits to map array entry access to architecture-specific mechanisms such as GPU texture cache or atomic instructions.

The Kokkos library API has been designed and improved through many development iterations. Given an understanding of the Kokkos abstractions listed previously and an intermediate knowledge of C++ (2011 standard), this library API is concise and intuitive for users. Performance portability has been demonstrated across architectures using mini applications including MiniFE, MiniMD, MiniAero, and LULESH.

Kokkos is the main performance portable programing model in a number of projects at Sandia National Laboratories and continues to be a key component of the strategy for next-generation computing platforms. The three projects with the most components written to use Kokkos are the Trilinos solver library, the LAMMPS molecular dynamics simulation code and Albany, an open source multi-physics research code. We are also writing research prototypes for important linear algebra kernels, solvers, graph analytics problems and during the next year, engineering contact problems, finite element code and fluid-dynamics simulations.

In the ASC FY15 Kokkos formally became an independent open source project with development now publicly available on the github source code repository site (github.com/kokkos/kokkos). The project is available under a BSD license.

3.2 Advanced Architecture Test Beds and Benchmarking Resources

3.2.1 Shepard: Intel Haswell Architecture Testbed

Shepard contains the latest HPC variant of Intel's Haswell server class processor. Each socket has 16 dual-threaded SMT cores and 40MB of level-3 cache. The clock rate is 2.3GHz continuing the trend of server processor clocks decreasing over previous generations when new cores are added. Haswell processors introduce the AVX-2 instruction set from Intel which extends the floating point capabilities to include fused multiply-add instructions (effectively doubling peak floating point SIMD throughput), gather/scatter memory operations and vectorized integer operations which may be useful in the generation of address offsets. For Shepard all benchmark runs utilize Intel's 15.2 Composer-XE compiler environment.

3.2.2 Compton: Intel Sandy Bridge and Knights Corner Architecture Testbed

Compton is a well established test-bed at Sandia providing access to Intel's Xeon Phi Knights Corner many-core architecture. The cards feature 57-cores and 6GB of GDDR-5 system memory. Each core is clocked at 1.1GHz and provides a single 8 double-precision operand wide vector unit, delivering greater than 1TFLOP of peak computation performance. For this platform all benchmarks utilize the Intel 15.2 Composer-XE compiler environment.

3.2.3 Shannon: Intel Sandy Bridge and NVIDIA Kepler Testbed

The Shannon test bed provides a mixture of NVIDIA GPU hardware connected to Intel Sandy Bridge processors. GPUs provided including current best-of-class K80 HPC GPUs as well as older K40 and K20 GPUs. All benchmarks run using Shannon utilize the NVIDIA CUDA 7.5 SDK and the GNU 4.9.2 compiler.

3.2.4 White: IBM POWER8 and NVIDIA Kepler Testbed

White is the newest test bed machine provided in Sandia's Advanced Architecture test bed program and provides access to dual-socket, 10-core per socket IBM POWER8 processors with a single K40 GPU attached to each socket via PCIe. Nodes are interconnected with FDR InfiniBand which is also the first deploy of this technology from Mellanox at Sandia. For benchmark runs on this system we utilize the IBM XL 13.1.2 compiler and the GNU 4.9.2 compiler.

3.2.5 Hammer: ARM64 Testbed

Hammer is a 46-node installation of ARM64 processors designed by Applied Micro (APM). These single-socket processors provide 8-cores and fully comply with the ARMv8 ISA specification. Each core runs at 2.4GHz. Four channels of DDR3 memory are connected giving memory bandwidth which is similar to an Intel Sandy Bridge socket. For benchmark runs on these machines we use the GNU 4.9.2 compiler which was the first to provide optimization support for the ARM64 ISA.

3.3 Baseline and Non-Kokkos Implementations of LULESH

3.3.1 Baseline MPI-Only Implementation

The LULESH mini-application version 2.0 [5, 25] is provided by LLNL for codesign and benchmarking activities. The code comes with versions written using MPI, OpenMP, CUDA and OpenACC, the latter are written by NVIDIA. When downloaded, LULESH uses preprocessor directives to compile in OpenMP pragmas and headers as well as calls to the MPI runtime. For our purposes we have developed an MPI-only version of the code, which we may refer to as 'Serial' since it contains no on-node parallelism, by removing all OpenMP statements. The MPI-only version is regarded as our baseline application for developing peer implementations since it more closely represents the code with which many ASC application developers will be starting to parallelize on the Trinity and Sierra platforms.

3.3.2 Baseline OpenMP Implementation

As stated, the baseline versions of the code provided by LLNL include an OpenMP implementation which we call the "OpenMP Original" variant. This code was included in the CORAL procurement and has been run on a wide variety of platforms and compilers.

3.3.3 RAJA Implementations

The development RAJA code provided by LLNL during the L2 includes two variants of the LULESH miniapp. The first termed 'basic' utilizes RAJA for parallel-execution dispatch effectively replacing OpenMP parallel-for and reduction parameters with RAJA equivalents. The second version, which is designed to improve performance on some platforms, uses data structures known as Index Sets. These allow for contiguous ranges

of data structures within the application to be represented specifically improving the level of vectorization which may be possible. We term this implementation 'RAJA-IndexSet'.

3.3.4 Minimal OpenMP Implementation

In the minimal OpenMP implementation, developed by Sandia, we utilize the Intel Analyzer and Inspector XE toolkits to add parallelism to the baseline MPI-only version of the code. These tools allow us to profile the hotspots of the miniapp and then, under guidance from the tool, gradually add parallel dispatch to expensive compute loops. Thread data race conditions can then be detected (which are a feature of the serial LULESH algorithm) and corrected. In this implementation we employ the use of atomic operations on the domain data-structure to ensure threads are not able to overwrite each others updates. The purpose of this implementation is to provide insight into a minimal version of the code which may be developed using semi-automated or tool-assisted processes.

3.3.5 Optimized OpenMP Implementation

The optimized OpenMP implementation of LULESH uses experiences from Sandia's Application Performance Team which have been gathered over four years of porting codes to Xeon, Xeon-Phi and POWER processors/co-processors. The changes implemented in the code include:

- **Extensive Use of the `restrict` and `const` Keywords**: which enable the compiler to more accurately reason about variable/memory use. The effect is to provide more efficient instruction generation which often includes higher level of vectorization.
- **Efficient Reductions**: reductions are re-implemented to utilize OpenMP-reduction constructs. Our experience has shown that the compiler generates more efficient code when these are used.
- **Thread Memory Buffer Hoisting**: in the original OpenMP implementation provided by LLNL buffers are repeatedly allocated and then freed. In the optimized variant these are hoisted out of parallel loops and replaced by schemes which keep the buffer live allowing it to grow when needed. The effect is to significantly reduce calls to memory allocation/deallocation routines which are typically synchronization points in threaded sections.
- **Use of OpenMP 4.0 SIMD Pragmas**: we make extensive use of SIMD pragmas where the executing code can be shown to be safe. In our analysis of the baseline LULESH OpenMP implementation from LLNL many compilers are unable to verify that loop dependency is safe, where these can be safely reasoned about and we expect performance improvements we can force vector instruction to be generated.

3.4 Kokkos Implementations of LULESH

3.4.1 Minimal-CPU Kokkos Implementation

The minimal-CPU Kokkos variant is roughly equivalent in its approach to the Minimal OpenMP variant developed by Sandia. It modifies the serial version of LULESH to use Kokkos:: parallel_for and Kokkos::parallel_reduce to parallelize loops. Examples of loop transformation are shown below.

```
for (Index_t i = 0; i < numElem; ++i) {
  ...
}
```

Listing 3.1: Example Original For Loop

```
Kokkos::parallel_for ( numElem, [&] (const Index_t& i) {
  ...
});
```

Listing 3.2: Example For Loop Rewritten into Kokkos

Additionally, atomic operations are employed to resolve write conflicts. In contrast to the recommended way of using Kokkos for new applications we utilize *capture by reference* semantics to create C++11 Lambdas. This allows the data structures to remain unchanged including the use of the std::vector container.

```
    for (Index_t lnode = 0; lnode < 8; ++lnode) {
      Index_t gnode = elemToNode[lnode];
      domain.fx(gnode) += fx_local[lnode];
      domain.fy(gnode) += fy_local[lnode];
      domain.fz(gnode) += fz_local[lnode];
    }
```

Listing 3.3: Example Domain Update in Original Code

```
    for (Index_t lnode = 0; lnode < 8; ++lnode) {
      Index_t gnode = elemToNode[lnode];
      Kokkos::atomic_add(&domain.fx(gnode), fx_local[lnode]);
      Kokkos::atomic_add(&domain.fy(gnode), fy_local[lnode]);
      Kokkos::atomic_add(&domain.fz(gnode), fz_local[lnode]);
    }
```

Listing 3.4: Domain Update using Kokkos Atomics Operations

Finally, we modified the error handling code from the original serial algorithm in LULESH to use parallel reductions when error cases are identified. This eliminates the need to call MPI process abort functions directly from loops which prevents vectorization and execution on GPUs.

The changes described allow the serial code to now run in thread-parallel execution on CPU architectures like traditional Xeon processors, POWER8 processors and Intel's Xeon Phi many-core accelerator. It represents a low optimization state for the type of code we expect to run on the Trinity Phase II platform.

3.4.2 Minimal-GPU Kokkos Implementation

In order to execute the code on GPUs more changes are required. In particular it is necessary to capture variables by copy, so that a self contained functor can be given to the GPU to execute there. A consequence is that class instances captured by the lambda function must have their member functions marked appropriately with **const**. In LULESH this entails the Domain class and its data accessor functions. Furthermore those accessor functions must be marked with the KOKKOS_INLINE_FUNCTION macro which allows for the correct mark-up needed to compile code for NVIDIA's CUDA or the Kalmar compiler for AMD Fusion APU devices. An example of this change is:

```
class Domain {
  ...
  Real_t &x(const Index_t idx) { return m_x[idx]; }
};
```

Listing 3.5: Original Domain Definition Code from LULESH

```
class Domain {
  ...
  KOKKOS_INLINE_FUNCTION Real_t &x(const Index_t idx) const { return m_x[idx]; }
};
```

Listing 3.6: Modified Domain Definition allowing execution on the GPU

Another consequence of capturing by copy is that std::vector containers can no longer be used. The effect of continuing std::vector usage while capturing by copy would be that at each kernel launch all data allocations referenced inside the loop body would be copied into new allocations. Further, std::vector is not supported by the NVIDIA CUDA or the AMD Kalmar Fusion APU backends. In order to address this difficulty, Kokkos provides a more or less "drop-in" replacement for std::vector containers by using the Kokkos::vector class. It has a similar API to std::vector but utilizes Kokkos View semantics to ensure assignments are shallow copies.

```
class Domain {
  ...
  std::vector<Real_t> m_x;
};
```

Listing 3.7: Original Domain Definition in LULESH

```
class Domain {
  ...
  Kokkos::vector<Real_t> m_x;
};
```

Listing 3.8: Kokkos Domain Definition using Kokkos Vector Replacements

We also replace raw memory allocations with Kokkos equivalents to ensure memory allocations can be correctly mapped to GPU backends.

3.4.3 Kokkos Optimization Level 1

The first set of optimization addresses excessive data reallocation. LULESH uses large temporary allocations which are allocated and then deallocated for subsets of kernels. Instead of allocating and reallocating the actual memory, we added a large buffer allocation and a function which creates views into those buffer allocations.

```
//replace
Real_t *fx_elem = Allocate<Real_t>(numElem8);
Real_t *fy_elem = Allocate<Real_t>(numElem8);
Real_t *fz_elem = Allocate<Real_t>(numElem8);
...
Release(&fz_elem);
Release(&fy_elem);
Release(&fx_elem);
```

Listing 3.9: Original LULESH Element Allocation

```
ResizeBuffer((numElem8*sizeof(Real_t)+4096)*3);
Real_t *fx_elem = AllocateFromBuffer<Real_t>(numElem8);
Real_t *fy_elem = AllocateFromBuffer<Real_t>(numElem8);
Real_t *fz_elem = AllocateFromBuffer<Real_t>(numElem8);
```

Listing 3.10: Modified LULESH Element Allocation

Next we utilized NVIDIA's visual profiler to determine the biggest bottlenecks in GPU performance, since the relative performance of the CPU and GPU execution was poor compared to the relative hardware capabilities. The biggest issue highlighted by the profiler is the low occupancy achieved by a number of kernels. Much of the register pressure (i.e. the size of the state of a given thread) comes from temporary static sized arrays. The most expensive kernel calls is a sub function which calculates values for three static sized arrays corresponding to three spacial dimensions. Since the calculations are independent one can break that down into three successive calculations for a single dimension, and thus reuse the temporary static sized arrays.

```
...
xd1[..] = ..;
yd1[..] = ..;
...
CalcElemFBHourglassForce(xd1, yd1, zd1, hourgam, coefficient,
                         hgfx, hgfy, hgfz);
```

Listing 3.11: Original Hourglass Calculation Sequence in LULESH

```
...
xd1[..] = ..;
...
CalcElemFBHourglassForce(xd1, hourgam, coefficient, hgfx);
...
yd1[..] = ..;
...
CalcElemFBHourglassForce(yd1, hourgam, coefficient, hgfy);
...
```

Listing 3.12: Modified Hourglass Calculation Sequence in LULESH

This represents a commonly useful technique, that in cases where register pressure is high, it is better to isolate the calculations than interleaving them, even though the latter can provide more instruction parallelism.

3.4.4 Kokkos Optimization Level 2

The second level of optimization addresses data layout and access properties, as well as introduces hierarchical parallelism where appropriate. A defining feature of Kokkos are memory layouts for multi-dimensional views. In LULESH a number of linearized 2D allocations are used, which in the previous variants were simple pointers with hard coded indexing. In this version those pointers are replaced by actual 2D views for which in most cases the default layout for each architecture is the correct one. Furthermore there are a number of data streams for which gather operations are performed. Replacing those with Kokkos views using the Kokkos::RandomAccess memory trait can improve performance on GPUs.

```
Kokkos::View<const Real_t*, Kokkos::MemoryTraits<Kokkos::RandomAccess> > m_c_x ;
...
KOKKOS_INLINE_FUNCTION Real_t c_x(const Index_t idx) const { return m_c_x[idx]; }
```

Listing 3.13: Kokkos Random Access Memory Traits

Some of the kernels expose a natural hierarchical parallelism, *i.e.* they have nested loops suited for parallelization. Using Kokkos nested parallel calls can help improve performance on GPUs by exposing more parallelism and thus increasing the occupancy. In particular a technique was employed where the inner most loop is mapped to a ThreadVectorRange policy and the outer loop is split between a team-level parallel loop and a thread level parallel loop. Effectively we create artificial work sets for teams. This approach can help with cache locality by making threads in the same team collaborate on cache usage vs. competing for scarce resources. In particular kernels which show a gather behavior with reuse of data, tend to benefit from using hiearchical parallelism.

```
Kokkos::parallel_for("CalcFBHourglass B", numNode, KOKKOS_LAMBDA(const int gnode) {
  Real_t fx_tmp = Real_t(0.0);
  for (Index_t i = 0; i < count; ++i)
    fx_tmp += fx_elem[cornerList[i]];
  domain.fx(gnode) += fx_tmp;
```

Listing 3.14: Original Hourglass Kernel-B Calculation in Kokkos

```
Kokkos::parallel_for ("CalcFBHourglass B",
                      Kokkos::TeamPolicy<>((numNode+127)/128,team_size,2),
  KOKKOS_LAMBDA (const typename Kokkos::TeamPolicy<>::member_type& team) {
  const Index_t gnode_begin = team.league_rank()*128;
  const Index_t gnode_end =
     (gnode_begin + 128<numNode)?gnode_begin + 128:numNode;
  Kokkos::parallel_for(Kokkos::TeamThreadRange(team,gnode_begin,gnode_end),
    [&] (const Index_t& gnode) {
    reduce_double3 f_tmp;
    Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(team,count),
      [&](const Index_t& i,double3& tmp) {
      tmp.x += fx_elem[cornerList[i]] ;
    },f_tmp);
```

```
    Kokkos::single(Kokkos::PerThread(team), [&](){ domain.fx(gnode) += f_tmp.x ;});
  });
```

Listing 3.15: Modified Hourglass Kernel-B Calculation using Hierarchical Parallelism

3.4.5 Kokkos Optimization Level 3

The LULESH mini-application consists of many successive loops which go over the same range and are trivially data parallel (i.e. there is no reduction, scatter or gather operation. Those loops can be merged which results in better cache reuse and reduced memory traffic. For parallel execution it also better amortizes the start up costs of parallel regions.

We merged kernels in two functions: EvalEOSForElems and CalcEnergyForElems resulting in another significant performance improvement on GPUs.

3.5 LULESH on AMD Fusion APUs

For the last months Sandia National Laboratories has been collaborating with AMD on several topics. These include: (1) an AMD Fusion APU backend for Kokkos and, (2), deployment of the HSA runtime on Sandia's Cooper Fusion Architecture test bed. The Kokkos backend utilizes AMD's Kalmar compiler which is currently under active development. The Kalmar compiler provides a parallel programming environment based on Microsoft's C++AMP. While the latter is not yet completely finished (a number of features are not implemented yet, with an expected completion time around the SC15 conference), all capabilities necessary to run the Minimal-GPU variant and the optimization level 1 variant of LULESH are available. That experiment was successfully performed on an AMD test platform with Kaveri APUs. We do not report performance results here, since the software stack is still far away from a release and the hardware is a consumer grade product with little double precision compute capabilities. Hence the performance is currently not representative of what we believe HPC variants of AMD's APUs will deliver in the future.

3.6 Code Development and ATS Platforms

The porting of full ASC applications to future ATS platforms will no doubt be a time consuming process, not least because the algorithms used in many existing production settings are well optimized for serial environments. We note that our research activities using the Mantevo mini-app suite have shown that data races between threads are difficult to debug and optimize as code porting activities take place.

In Figure 3.1 we show a possible mapping of our various implementations to a timeline for future ATS platforms. Our conjecture is that initial ports of applications to machines will likely not be well optimized but will be performed to get code onto the platform and utilize the available hardware. Performance will likely not be a first priority in these ports. Over time, as programmer expertise grows and the associated debugging and profiling infrastructure is improved, levels of performance will increase. We separate out the continued use of directives from C++ abstractions as the path for Trilinos and several important code bases at Sandia is slowly converging to the use of Kokkos throughout the software stack. Our inclusion of RAJA is to show how we believe these ports of LULESH will compare noting that the optimized GPU variant has not yet been developed by LLNL. The eventual aim of the Kokkos project is to see widespread adoption of many of the core concepts introduced in the C++20 Language Specification and beyond but we recognized that continued development to support specializations for HPC (for instance the inclusion of checkpoints or other reliability infrastructure) may be required in the future (see the dashed outlines for possible future version of the Kokkos programming model).

Our approach takes three main phases: (1) the initial porting of parallel execution dispatch to future hardware, we term these ports "minimal" in that they achieve only the basics neccessary to execute on Trinity Phase-II and Sierra; (2) the second, longer, phase from 2016 to 2018 requires modification of the application data structures. This is likely to be a significant undertaking requiring considerable development but will yield greater performance on GPU-based systems such as Sierra. Finally, (3), from 2018 onwards,

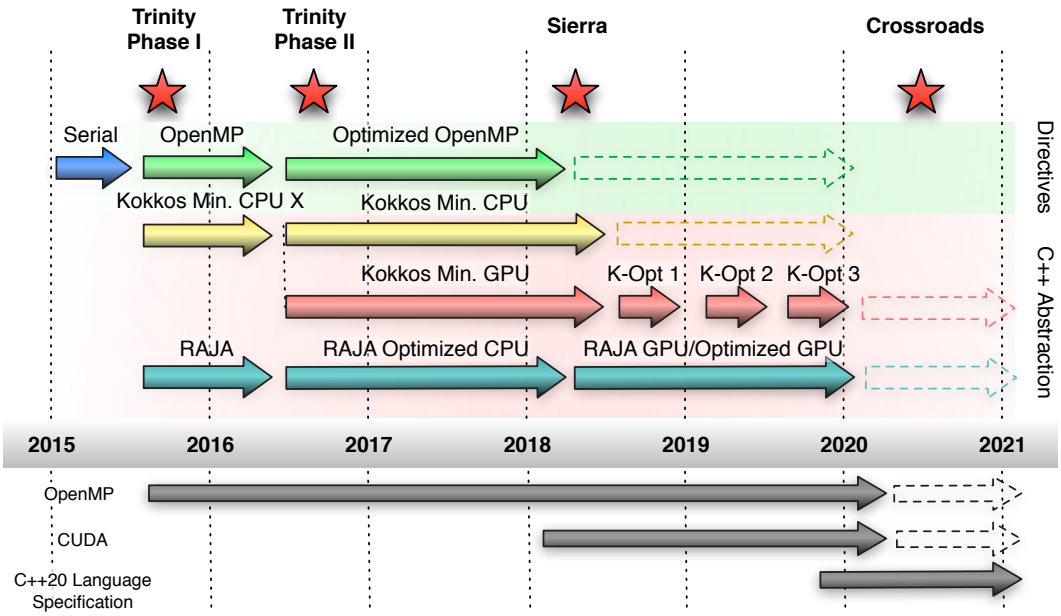


Figure 3.1: Code Development and Benchmark Versions mapped to ATS Platforms

application developers will slowly begin to optimize the applications that have been ported to next generation architectures delivering much greater levels of performance.

In the latter sections of this document, we compare the performance and associated changes in application code (which we term "programmer productivity") for the implementation variants show in Figure 3.1. This provides a possible overview of the tradeoffs which our code groups may make in deciding how quickly to port and optimize codes in the future. Put differently, what levels of performance we might be able to expect from our codes at what point in time and for what level of programmer effort.

3.7 Performance Portability for LULESH Implementations

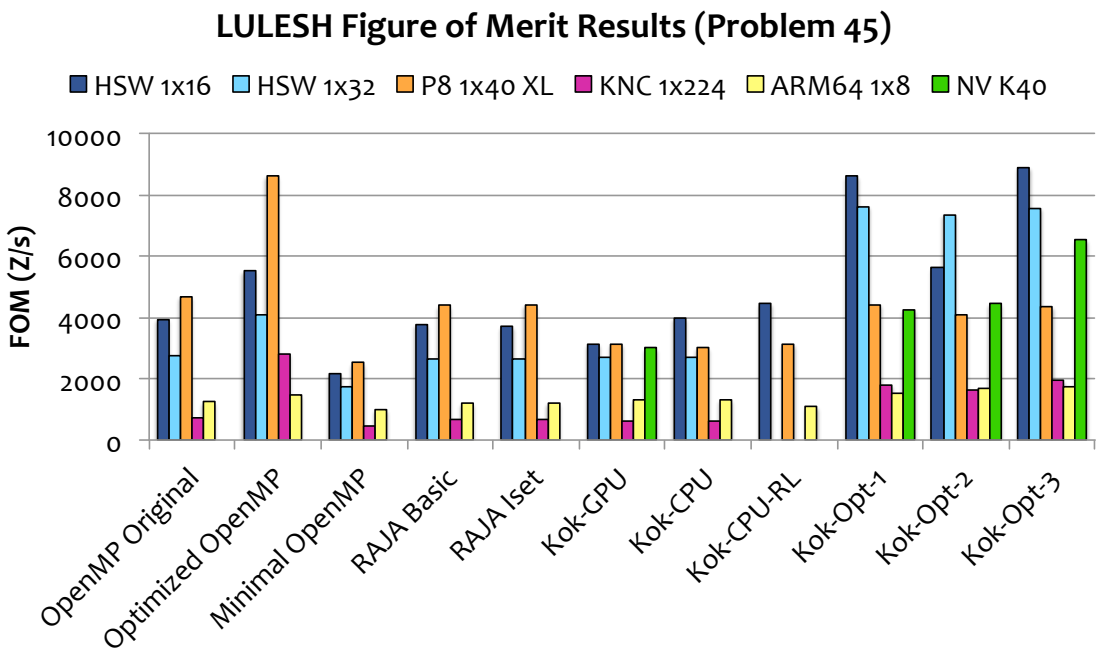
3.7.1 Environment Configuration

For each of the runtimes gathered we utilized our experiences from Sandia's ASC architecture testbed benchmarking to configure each run for optimized MPI process placement and thread affinity. Unless otherwise stated we force nodes to be partitioned evenly by MPI ranks and request thread affinities be applied that match each partition. On all platforms except the POWER8 we use default OpenMP runtime settings which typically, in our experience, provide the highest performance. For the POWER8 systems we explicitly tell the OpenMP runtime that binding must be applied as this is not the default when using the IBM-XL runtime.

The MPI and compiler environments deployed on the test bed are custom build by Sandia's research team using the latest versions of NUMA libraries, the Hardware-Locality library [12] and OpenMPI [21]. Over time we have shown that these configurations provide high performance for the systems being used in this study.

3.7.2 Performance Comparison

For the performance comparison we report the harmonic mean of LLNL-coded Figure of Merit (FOM) from a minimum of 10 runs. Variations are recorded and show typically differences of 1-3% between runs on



the same machine. The simulations are run to completion, that means no iteration count limit was given. We ran three different sizes (45, 60, 90) based on guidance from LLNL Lulesh developers. The associated memory footprint is 100 MB, 220 MB and 700 MB respectively.

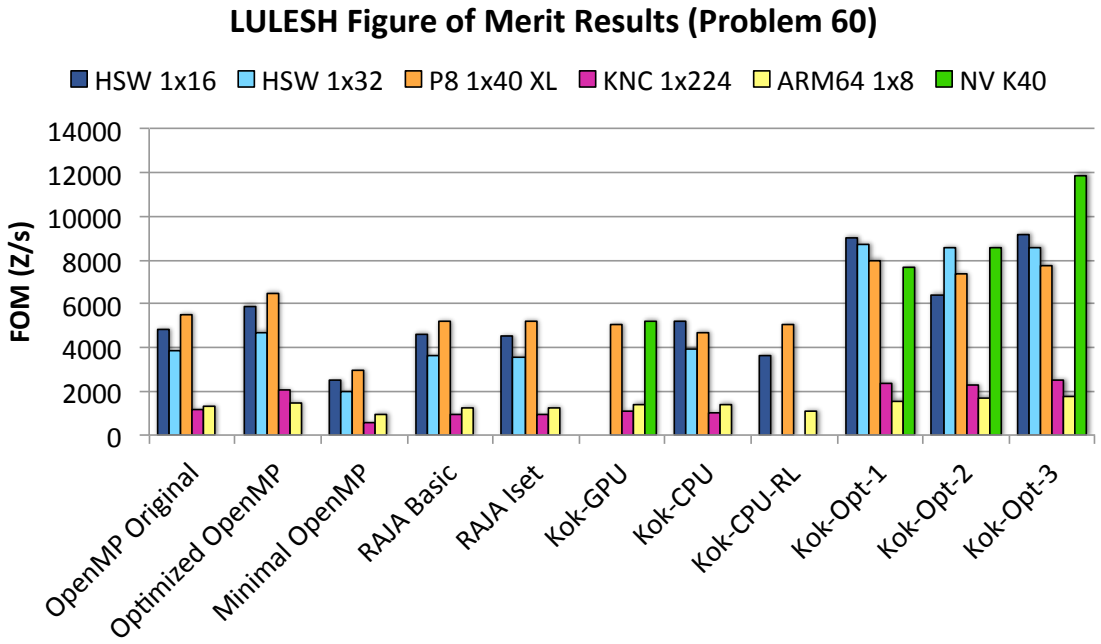
The performance data shows relatively consistent performance across all "basic" implementations. The only outlier is the "Minimal OpenMP" variant, which performs at about 50% of the other variants. The optimized Kokkos variants consistently outperform the basic variants by 60-90%. It is noteworthy that already at the first optimization level most of the performance benefit for CPU like architectures is achieved, while GPU performance significantly increases with the using more aggressive optimization.

3.8 LULESH Binary Sizes

In Figure 3.2 we show the binary sizes associated with the Kokkos, RAJA and pure OpenMP programming models. We breakdown the binary size associated with each model by the number of instructions generated during compile and the binary space allocated for initialized and uninitialized static data structures. We note that pure OpenMP binaries are the smallest in terms of both instructions and associated data allocations. Kokkos binaries have additional instructions and data allocations required for the processing of Kokkos View structures and runtime handling. Both initialized and uninitialized data structures, which make up a significant proportion of the final binary size in each implementation, are roughly static across variants representing what we believe to be a static overhead. We attribute some of the increase in instruction bytes to the routines associated with the Kokkos runtime however, the additional use of templates and different optimizations to the binary are very likely to factor into this observation also.

3.9 LULESH Compile Time

The heavy of C++ abstraction layers creates the potential for increased compile times due to additional parsing, processing and optimization steps required to generate a binary. Figure 3.3 shows the compile times using the Intel 15.2.164 compiler on the Sandia Shepard platform. These times utilize the platform specific options to the compiler supplied by the RAJA and Kokkos build system. In assessing the compiler output we are able to relate increased compile times with additional processing of the compiler inlining features. In



particular we see RAJA provides higher than default maximum inline thresholds which when removed drop compile and link times to be close to the OpenMP implementations. Our conclusion is that the addition of C++ layers is more likely to require heavy inlining support (which is seen in the higher Kokkos compile times where default thresholds are used) and therefore their use in larger production settings is likely to result in longer compile and link times.

3.10 Programmer Productivity Metrics for LULESH Implementations

3.10.1 Calculating Programmer Productivity

The Software Engineering community as a whole continues to research into the measurement of programmer productivity and code complexity. Although a number of approaches have been proposed [24, 14] there is no consistent approach to measurement. In order to provide a consistent measurement we have developed a simple approach which is outlined as follows:

1. **Remove Programmer Comments**: comments in code vary considerably across implementations as each programmer describes implementation specifics in a variety of ways. These are stripped out to ensure analysis of differences does attribute any inherent code difference to these comments.

2. **Format using the `clang-format` utility**: we provide a consistent formatting of source code using the LLVM formatting utilities set to the 'Google' coding style as this allows for longer code lines. The automation of code formatting means we apply a consistent scheme to all files. In some cases minor hand tweaks to the formatting need to be applied for small differences; we have attempted to perform these using the same approach across all implementations to provide fair analysis.

3. **Code Change Sites using Apple FileMerge**: each LULESH implementation is compared for code site changes using Apple's FileMerge developer utility. The baseline source code is our MPI-only variant of LULESH. This tool aggregates multiple sequential different lines to provide an accurate heuristic for the number of places in the code where developer modifications have been made.

4. **Source code lines changed using `diff`**: finally, we utilize the UNIX diff utility to count lines in the implementations which would have to be added or removed when compared to the baseline MPI-only

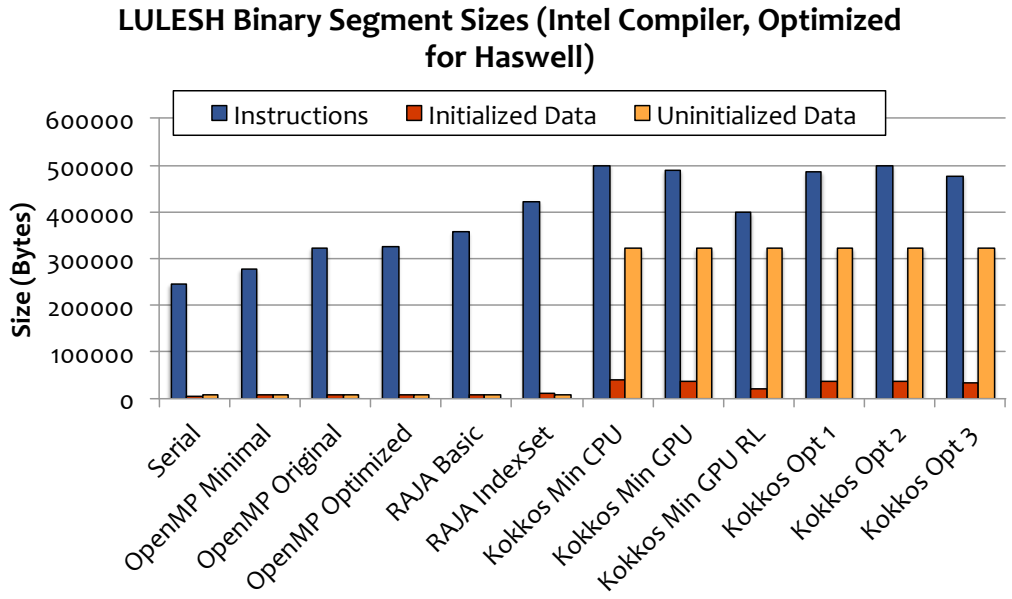


Figure 3.2: LULESH Binary Sizes by Programming Model (compiled using the Intel 15.2 compiler with optimization for the Haswell platform specified)

variant.

We note that this approach does not always provide an exact metric but our experience, which has included hand tweaking **all** implementations to reduce errors, the results are representative of the amount of effort our research team has had to apply in developing each code.

3.10.2 Sites of Changes in Code for LULESH Implementations

LULESH has 37 loops which are parallelized in the Minimal-CPU variant. The initial parallelization effort is very similar across OpenMP, RAJA and Kokkos with OpenMP being the least invasive. Modifying code to execute on the GPU with Kokkos doubles the number of change sites from about 125 to 250. But most of those changes are technical in nature requiring the addition of correct mark-up of member functions, and the replacement of std :: vector with Kokkos::vector. While tedious we do not expect that these changes will be as complicated as identifying parallelism and solving thread-safety issues. The non-expert variant follows a lot of the coding recommendations of Kokkos and thus is much closer to the minimal-GPU than the minimal-CPU variant. The reason that the Minimal-CPU variant has less changes than the RAJA variant is that it achieves thread-safety through use of atomic operations instead of changing the algorithm.

In Figure 3.4 we present the sites at which changes to the source code have been made. OpenMP provides the lowest number of sites because it does not require closing brackets to be added to loops as both RAJA and Kokkos need for lambda functions to parse correctly. Ensuring execution on the GPU requires an additional 20 - 30 code sites to be modified. The lowest number of modification sites in the C++ abstraction models (RAJA and Kokkos) occurs with the Kokkos capture-by-reference implementation (Kokkos-CPU-RL) because it requires no modification to the application data structures. We note that the effect of this is considerably poorer application performance.

3.10.3 Source Code Changes for LULESH Implementations

The number of source code lines added, removed and final total is presented in Figure 3.5. The Minimal OpenMP implementation requires the fewest changes but provides poor application performance. Some

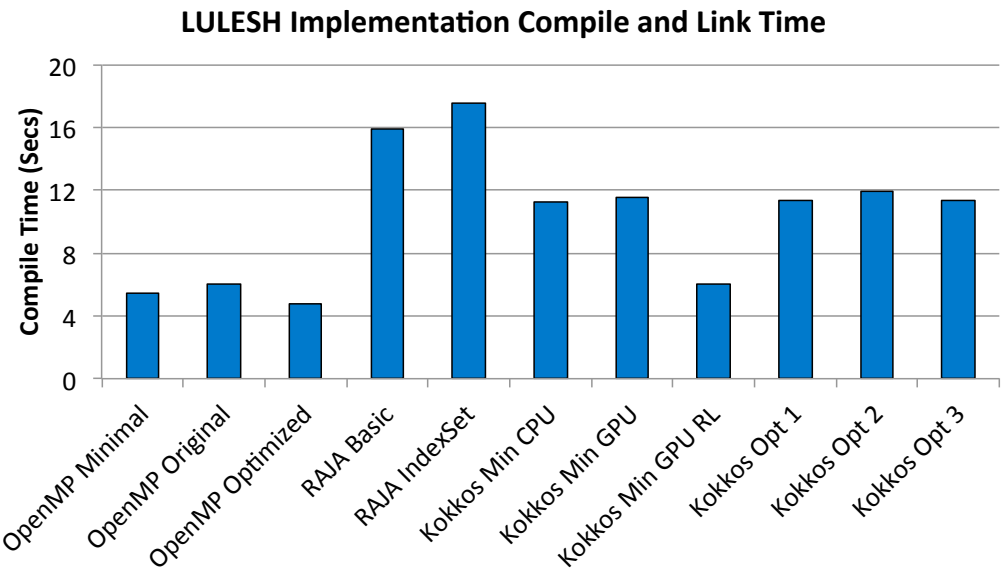


Figure 3.3: LULESH Compile Times by Programming Model (compiled using the Intel 15.2 compiler with optimizations for the Shepard Haswell platform)

variation exists between the OpenMP, RAJA and Kokkos implementations because the default time stepping calculation in RAJA is poorly framed for parallel execution and so is modified in these implementations to execute more efficiently.

The conclusion comparing directives to the C++ abstraction layers is that approximately similar lines of code must be modified between the implementations which acceptable levels of performance are desired. There is a clear correlation between improving performance and higher levels of SLOC modification.

3.11 Analysis of MiniAero

MiniAero is a mini-application for the simulation of compressible flow computational fluid dynamics (CFD) problems [19, 20]. It is available through the Mantevo project suite of mini-applications [6]. MiniAero solves the compressible Navier-Stokes equations and employs a cell-centered unstructured mesh finite volume method (first- or second-order accuracy) with explicit Runge-Kutta time integration. An internal mesh generator produces a structured mesh of hexahedral elements, however, unstructured mesh data structures are employed. MiniAero uses MPI for inter-node communication and Kokkos for intra-node parallelism. A layer of ghost cells is communicated across MPI process boundaries. The cell-centered finite volume method involves a loop over all the faces. The flux is calculated across the faces and summed into the two adjacent cells. This will lead to thread conflicts, and MiniAero has two approaches for resolving the thread conflicts: a "gather-sum" approach and an atomic fetch and add approach. The gather-sum approach is a two stage process. For the first stage a loop is performed over the faces. The fluxes are computed then stored at the faces. The second stage is a loop over all the elements, summing the fluxes on the six faces. Since the FY2014 ASC co-design milestone review, there have been various extensions to MiniAero. From the aspects of the physical model and discretization, they include the addition of:

- viscous terms to the Euler equations to solve the Navier-Stokes equations
- second-order discretization for the convective fluxes.

The second-order discretization for the convective fluxes was an important addition as full application codes are second order or higher. This also required the addition of flux limiters. All the results presented employ

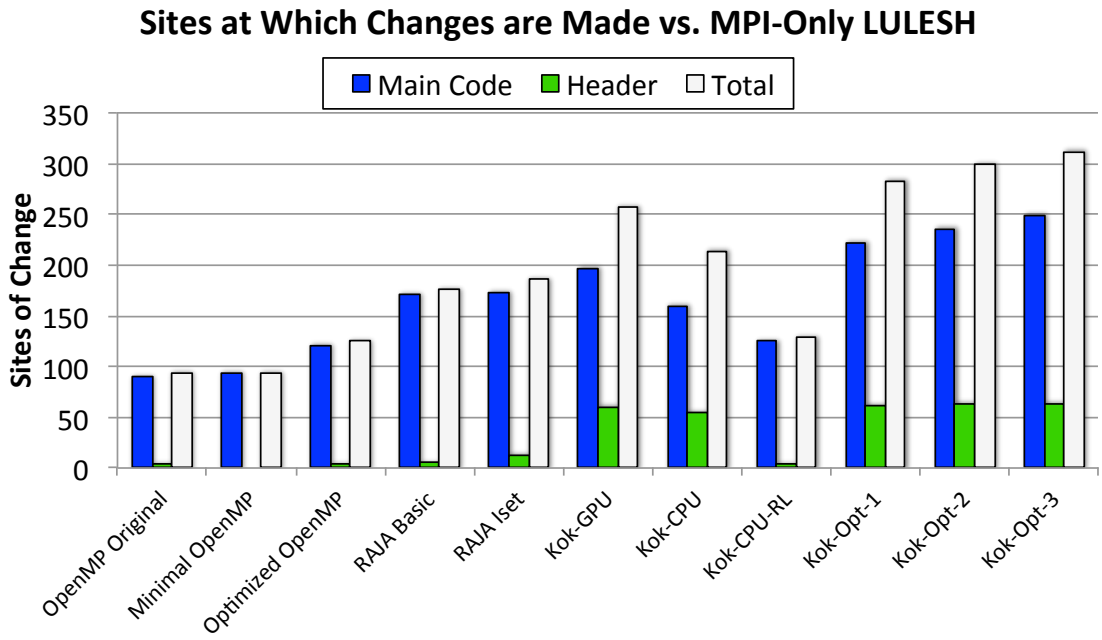


Figure 3.4: Sites in the LULESH source code at which lines of code are changed (multiple lines of code may be changed at each site)

the second-order discretization. A first order discretization is too simple to be representative of a full application code. There were also several important bug fixes and improvements:

- A key bug was fixed that involved incorrect handling of the ghost element values.
- A memory scaling bottleneck during the mesh generation was removed. As the focus of MiniAero is on the solver performance, and the mesh generation is a preprocessing step, the improved algorithm exploited the fact that the internal mesh generator employed a structured hexahedral mesh. The six neighboring processes are known in advance so the inter-process communication is limited to those six neighboring processes. This replaced an all-to-all communication for discovery of neighboring processes.
- The original restriction of powers-of-two process decomposition has been removed. The improved algorithm that allows non-powers-of-two decomposition also improved load balance.

3.11.1 Performance Analysis of MiniAero

In the ASC L2 Trilab codesign milestone for FY14, MiniAero was analyzed for on-node performance using several ASC platforms including Sandy Bridge, BlueGene/Q, the Intel Xeon Phi Knights Corner accelerator, AMD's Interlagos processor and NVIDIA's K20 Kepler GPU for the first order method. In the following sections we show updated scaling analysis on newer platforms for the second order method and show multi-node analysis. The single node comparisons include a comparison of the effect of the choice of number of MPI processes and OpenMP threads for both a Haswell and POWER8 compute node. The test case employed for the following performance analysis is a 30° ramp in a Mach 2.5 inviscid flow with second-order method. Note that the problem size varied with the different studies.

On-Node Performance Comparison

We present a comparison of the performance for the architectures described above in Figure 3.6 (some descriptions repeated from Section 3.2 for convenience). The performance is given in terms of the number

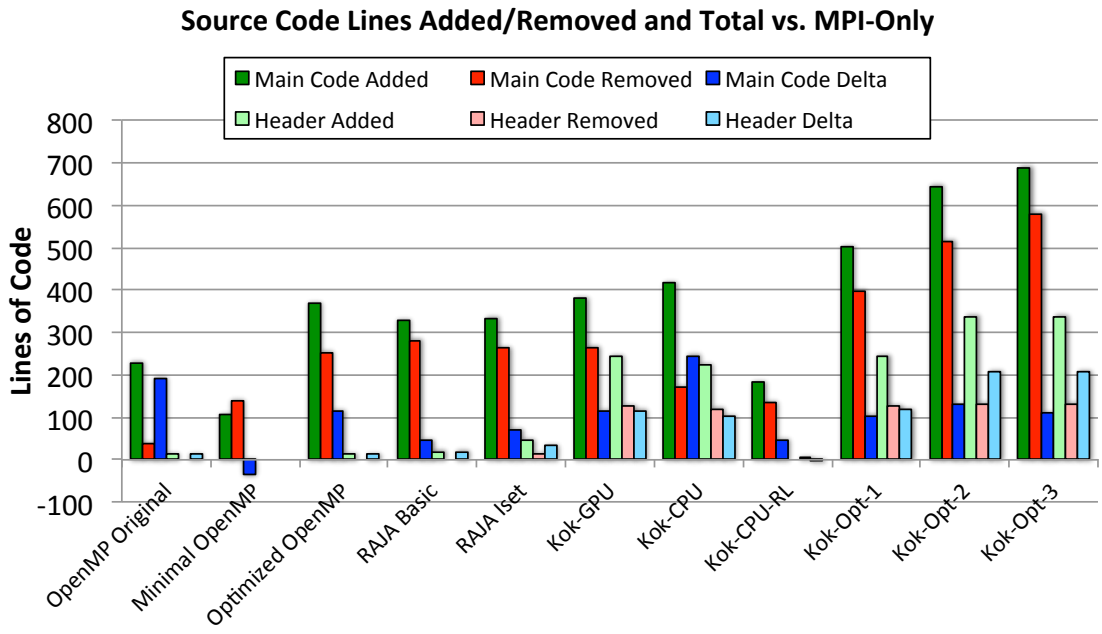


Figure 3.5: Source Code lines added, removed and total for LULESH implementations

of cell updates per compute node per second per time step. The problem size is 128x64x64 (516,000) cells and is dictated by the requirement to fit in the 6GB memory of the KNC.

- Shannon K80 (1 out of the 2 GPUs); atomics gives higher performance than gather-sum
- White dual-socket POWER8; 20 MPI processes each with 8 OpenMP threads was optimal, i.e. 1 MPI process with 8 OpenMP threads per core; gather-sum gives higher performance than atomics
- Shepard dual-socket Haswell; 2 MPI processes each with 32 OpenMP threads was optimal (used both SMT threads on a core); gather-sum gives slightly higher performance than atomics
- TLCC2 dual-socket Sandy Bridge (2.6GHz Intel Xeon E5-2670); MPI-only with 16 MPI processes was optimal; built with Intel 14.0.4.211 compiler; gather-sum gives higher performance than atomics
- Compton single KNC (1.1GHz 57-core); 1 MPI process with 224 OpenMP threads; atomics gives higher performance than gather-sum
- BG/Q compute node (single socket with 16 1.6GHz A2 cores for compute, each supporting 4 hardware threads); 1 MPI process with 64 OpenMP threads; built with GNU 4.7.2 compiler; gather-sum gives higher performance than atomics

As expected, the K80, POWER8 and Haswells give considerably higher performance than the three other architectures (other architectures are older architectures, e.g. Sandy Bridge has half the number of cores as Haswell and KNC has shortcomings which are hoped to be remedied by KNL, while a BG/Q socket uses substantially less power than the others).

A comparison of the performance on a single dual-socket Haswell compute node on Shepard for the combinations of MPI processes and OpenMP threads using hyperthreading, from 64 MPI processes (MPI-only) to a single MPI process with 64 OpenMP threads, is presented in Figure 3.7a. The mesh used for this study is 1024x128x128 (16.8 million) cells and is a factor of 32 times larger than the mesh used for the study in the previous figure (Figure 3.6) and is intended to have a memory usage closer to that used for a full application. For the 64 MPI process case, the memory usage is about 90GB, while for the 64 OpenMP thread case the memory usage is about 40GB. For this test case, the use of hyperthreading increases the

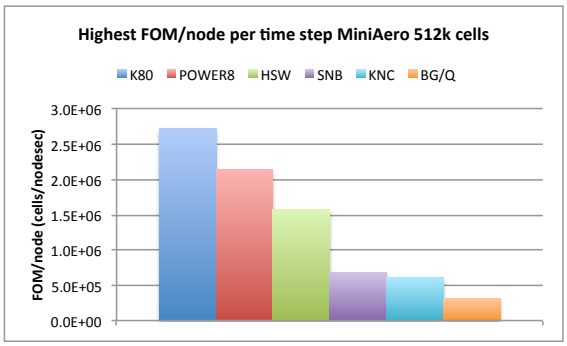
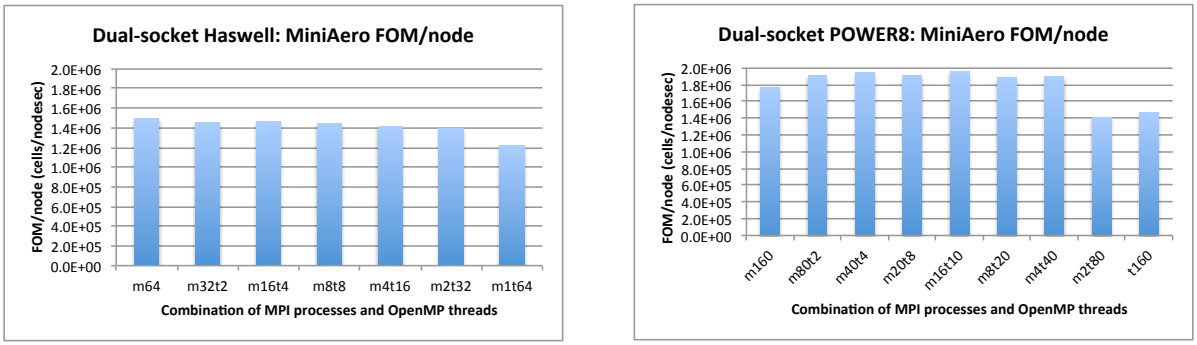


Figure 3.6: MiniAero highest performance (cell updates per node per second per time step) for 128x64x64 (516,000) cells for: Shannon K80 (1 out of the 2 GPUs), White dual-socket POWER8, Shepard dual-socket Haswell, TLCC2 dual-socket Sandy Bridge, Compton single KNC card (1 MPI process with 224 OpenMP threads, BG/Q compute node (1 MPI process with 64 OpenMP threads)

performance. The atomics approach (presented in the figure) gives slightly higher performance than the gather-sum approach. The performance for the different combinations of MPI processes and OpenMP threads are effectively the same with the exception of the case with 64 OpenMP threads. This is due to having a single MPI process with threads on two sockets, i.e. two NUMA regions. Although the use of threads may not have a performance advantage over MPI-only for this test case, it definitely has a memory advantage as the MPI-only case requires a factor of 2.25 of the memory required by the case with 64 threads.

A comparison of the performance on a single dual-socket POWER8 compute node on White for the combinations of MPI processes and OpenMP threads using hyperthreading, from 160 MPI processes (MPI-only) to a single MPI process with 160 OpenMP threads in presented in Figure 3.7b. The mesh used is 1024x128x128 (16.8 million) cells and is the same as that used for the Haswell study. The gather-sum approach (presented in the figure) provides higher performance than the atomics approach. For this test case, the performance for the combinations of MPI processes and OpenMP threads are effectively the same with the exception of the MPI-only case (160 MPI processes) and the two cases where an MPI process has threads that span more than one NUMA region (the case with 2 MPI processes each with 80 OpenMP threads and 1 MPI process with 160 OpenMP threads).



(a) (b)

Figure 3.7: MiniAero performance for 1024x128x128 (16.8 million) cells comparing the peformance for the different combinations of MPI processes and threads for: (a) dual-socket Haswell (b) dual-socket POWER8.

Strong Scaling

Strong scaling studies of the optimized MiniAero code were performed on two platforms, Mutrino and Volta. Results for a 1024x64x64 (4.2 million) cell mesh are shown in Figures 3.8–3.9. The figures plot performance per node against number of nodes. Performance is given in cell updates per second. The same metric will

be used for all following plots. A combination of MPI processes and OpenMP threads is used in a fashion so that 32 (i.e. all) cores per node are used on Mutrino and 8 cores per socket are utilized on Volta. As seen in the figures, as the number of nodes increases, the application scales the same regardless of the how the ranks and threads are implemented. The only exception is the use case of 1 MPI rank with 8 threads per socket on Volta (i.e. 1 MPI rank with 16 threads, 8 on each socket), which is about 20% slower due to a single MPI rank spanning more than one NUMA region. This is consistent with the study in Figure 3.7 where the performance is similar as long as each socket has at least one MPI process (the performance decreases for the case for one MPI process for two sockets, with OpenMP threads on both sockets). In Figure 3.8 the most threads used is 16 OpenMP threads for one MPI process, but all 16 OpenMP threads are on the same socket.

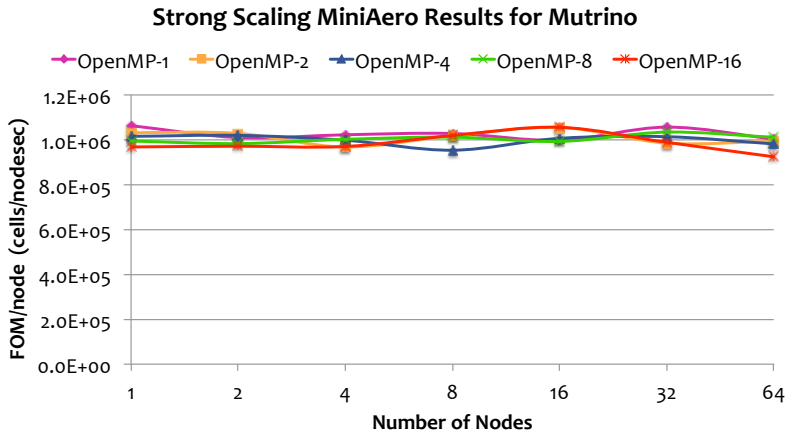


Figure 3.8: MiniAero strong scaling on Mutrino with 1024x64x64 (4.2 million) cell mesh.

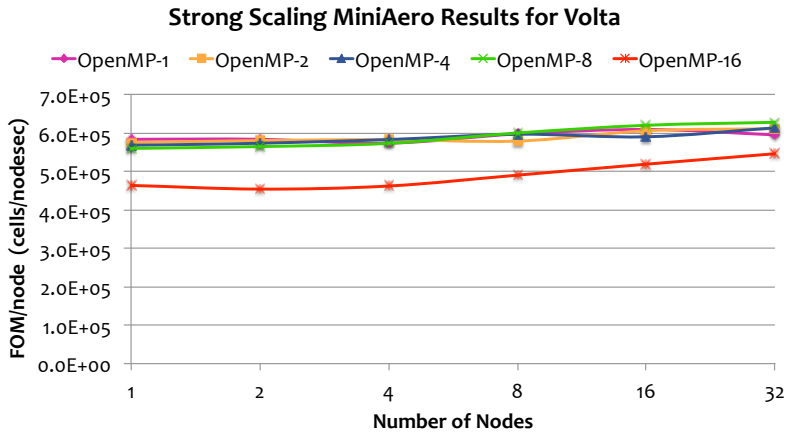


Figure 3.9: MiniAero strong scaling on Volta with 1024x64x64 (4.2 million) cell mesh.

Weak scaling

Weak scaling studies are performed on several platforms, although in some of these studies, we use a smaller problem size due to memory constraints. Figure 3.10 presents a weak scaling study with 1024x64x64 (4.2 million) cell mesh per compute node for Mutrino.

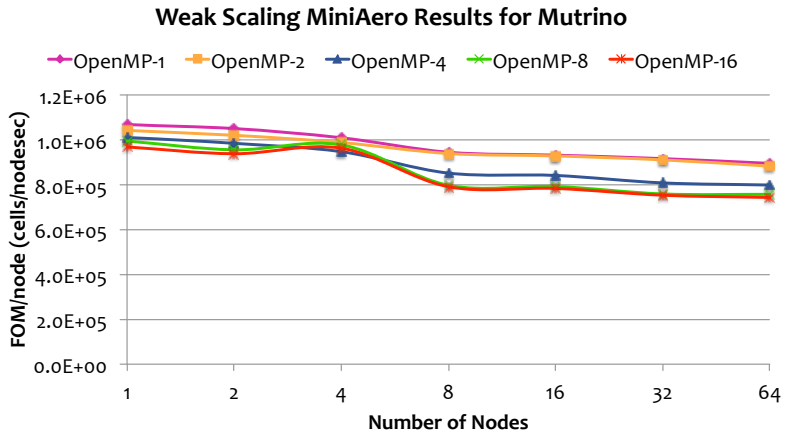


Figure 3.10: MiniAero weak scaling on Mutrino with 1024x64x64 (4.2 million) cell mesh per compute node.

Figures 3.11–3.13 present weak scaling studies for Shannon K80, Compton KNC and Vulcan BG/Q platforms, respectively. These studies employ 128x64x64 (516,000) cells per GPU, KNC or BG/Q compute node. The problem size was dictated by the requirement to fit in the 6GB memory of the KNC. For each KNC have a single MPI process with 224 OpenMP threads. For a BG/Q compute node, all 64 hardware threads are employed, either with 64 MPI processes or 64 OpenMP threads. The largest BG/Q runs are on 4096 nodes with 2 billion degrees of freedom. For the MPI-only case, using 64 MPI processes per compute node gives a total of 262,144 MPI processes. The performance of the atomics version of MiniAero is considerably worse than the gather-sum version on BG/Q, but atomics are slightly better than gather-sum on the KNC, and atomics are significantly better than gather-sum on the GPU. From Figure 3.13 one can see that MPI+OpenMP threads gives higher performance than MPI-only for the larger numbers of nodes.

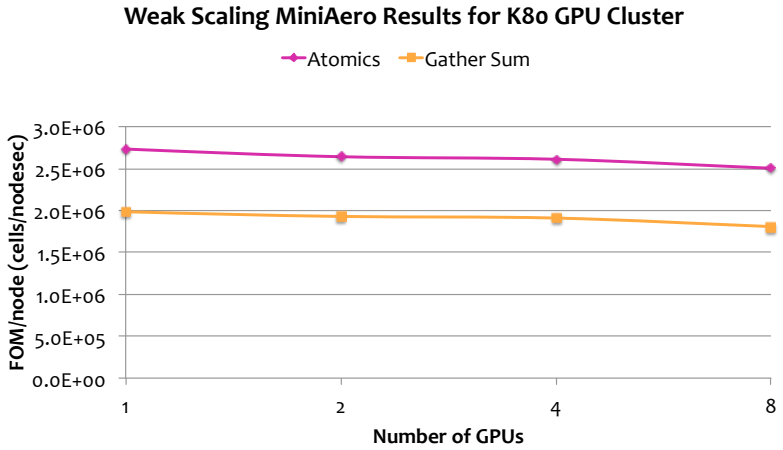


Figure 3.11: MiniAero weak scaling on Shannon K80 platform. 128x64x64 (516,000) cells per GPU (half of the K80)).

A weak scaling study for the average memory usage per compute node corresponding to Figure 3.13 is presented in Figure 3.14. As expected, gather-sum has a larger memory requirement than atomics because of the need to store fluxes at the faces. Also as expected, MPI+OpenMP has a lower memory requirement

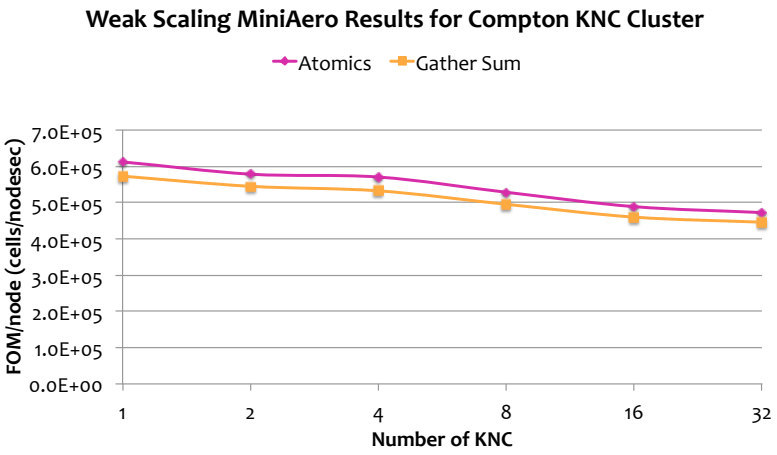


Figure 3.12: MiniAero weak scaling on Compton KNC platform (using both KNC cards per node). 128x64x64 cells (516,000) per KNC card (has 6GB RAM). One MPI process per KNC with 224 OpenMP threads.

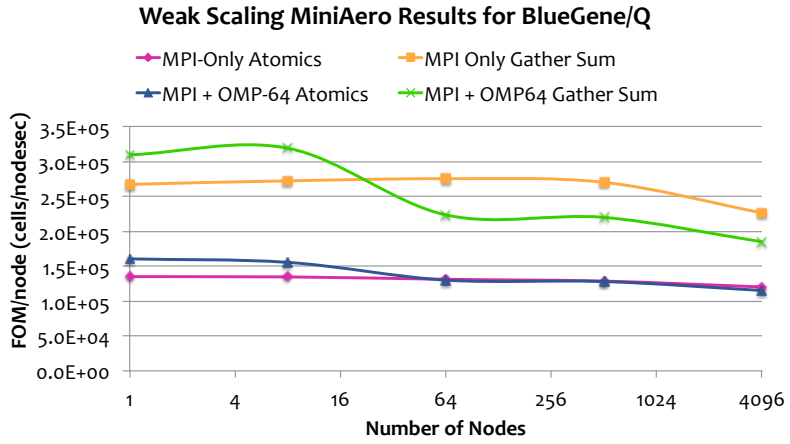


Figure 3.13: MiniAero weak scaling on LLNL Vulcan BG/Q platform. 128x64x64 (516,000) cells per node. For MPI-only case, used 64 MPI processes per node. For MPI+OpenMP case, use 1 MPI process with 64 OpenMP threads per node. 262,144 MPI processes for the MPI-only case on 4096 nodes.

than MPI-only because the latter has more replication of data for ghosting. Note that as the problem size is scaled up, the average memory usage per compute node is increasing faster for the MPI-only case while the memory usage for the MPI+OpenMP threads case stays mostly flat. For the MPI-only case at 4096 compute nodes, 262,144 MPI processes are used. We have previously observed large memory increases for other miniapps as well as application codes at very large numbers of MPI processes. Clearly the use of MPI and threads provides a memory advantage over MPI-only. Combining the use of atomic operations with threading achieves a memory footprint reduction of more than 1/3 for large node counts.

3.11.2 MiniAero binary size

A chart for the miniAero binary size for Haswell is presented in Figure 3.15. As with the LULESH binary size presentation, we breakdown the binary size by the number of instructions generated during compile and the binary space allocated for initialized and uninitialized static data structures.

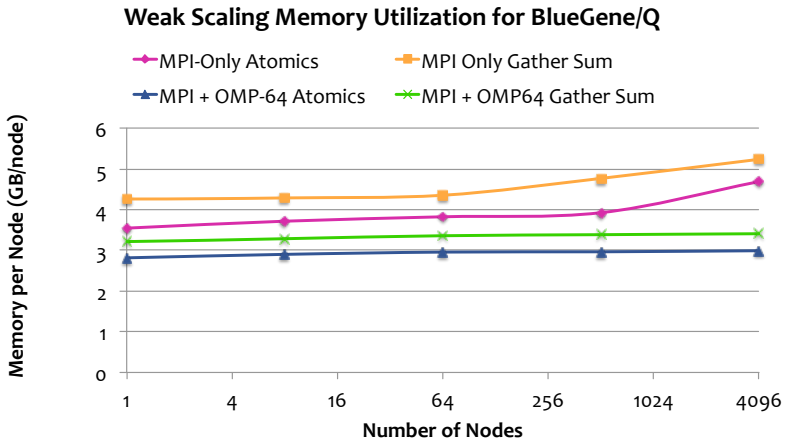


Figure 3.14: MiniAero weak scaling on LLNL Vulcan BG/Q platform memory usage. 128x64x64 (516,000) cells per compute node. For MPI-only case, used 64 MPI processes per compute node. For MPI+OpenMP case, use 1 MPI process with 64 OpenMP threads per compute node.

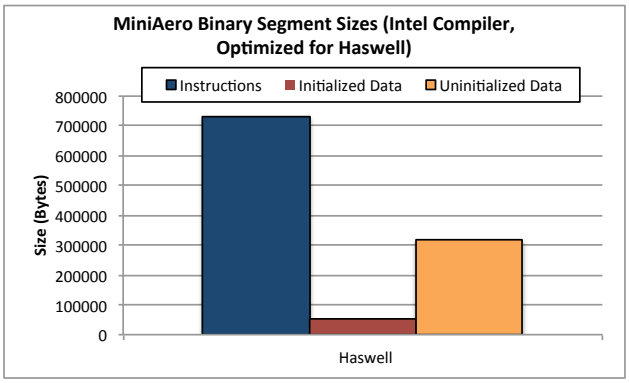


Figure 3.15: MiniAero binary size for Haswell (compiled using Intel 16.0.042 compiler).

3.11.3 Vectorization and Instruction Analysis of MiniAero

The ATS-1 Trinity Phase-I and Phase-II deployments will both bring new processors and instruction set architectures to DOE computing facilities. In Phase-I Intel Xeon Haswell processors will be used which will provide AVX2 instructions that maintain the 256-bit width of AVX found in TLCC-2 machines but augment the capabilities with vectorized integer operations, gather/scatter instructions and fused-multiply-add for floating point. The availability of gather/scatter memory access instructions has the potential to improve vectorization opportunities as operands can now be more easily gather sparsely in memory. In the Phase-II deployment of Trinity application developers will need to make their code work on the Intel Knights Landing Xeon Phi processor (KNL). The KNL will offer wider (512-bit) vector operations and additional capabilities in the form of lane-masking, gather/scatter conflict detection and others.

In Figure 3.16 we present a breakdown of the dynamic instructions executed by MiniAero on Sandy Bridge, Haswell and Knights Landing emulation environments. The instructions recorded relate only to the use of the vector-units (so this is a subset of the total instructions executed but includes all floating point operations). The Figure shows that approximately 80% of all vector instructions executed on Sandy Bridge and Haswell architectures and 75% of instructions on Knights Landing are executed over a scalar operand.

The respective width of the instructions executed, 128-, 256- and 512-bit, is shown in green, yellow and red respectively. For KNL roughly 10% of instructions utilize the new SIMD vector units.

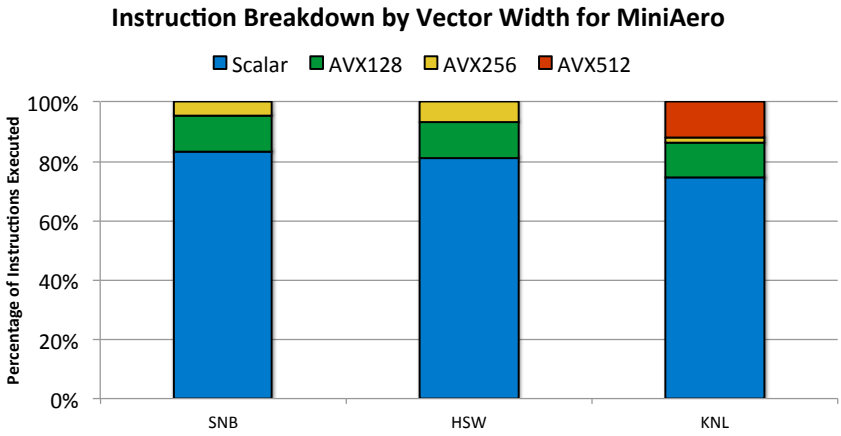


Figure 3.16: Instruction Analysis of MiniAero for Sandy Bridge (SNB), Haswell (HSW) and Knights Landing (KNL)

Given that the majority of floating point compute capability is being directed toward SIMD vector units in future designs the inability of application code to vectorize indicates a potential concern for future runtime performance. This highlights a recurring theme from analysis of our mini-applications and larger production codes – vectorization levels are often very low. Our runtime analysis, from which these results are derived, allows us to inspect application functions and even basic blocks by SIMD instruction execution allowing us to locate the most frequently executed functions which do not have high levels of vectorization. The approach used in this study is being actively developed and enhanced to support application code teams in preparation for the Trinity Phase-II deployment in 2016.

3.12 Lessons Learned

Our analysis of performance shows that we continue to experience poorer levels of optimization than we would like, particularly when C++ Lambda functions are used to introduce both Kokkos and RAJA into applications. We note that in many cases (and reflected in our MiniAero results) the level of vectorization which is achieved is very low. Given the growing importance of efficient vectorization for our codes, particularly for the Trinity Phase-II Knights Landing deployment, this presents a particular worry. We continue to explore the possibilities of having greater support for vectorization within the Kokkos framework but realize that this needs to be heavily balanced with the ability to produce guaranteed safe code and that this process does not in fact slow the code down by forcing vectorization when efficiency heuristics can show this not to be beneficial.

Our expectation is that in the initial ports of applications to utilize Kokkos (which we have termed minimal in these studies), safety issues will still largely be present and will be significant frustration for users who will need to be constantly weary of thread-safe issues. However, as developers become more acustomized to these issues and coding for these problems becomes natural we expect to be able to introduce greater direction for the compiler to force vector instruction sequences to be generated.

This continues to emphasize the important of working with vendors to ensure compiler optimization is a high priority item. Further, working across the trilabs to understand these issues has, and will continue to be, an important part of our collaborative codesign process.

As outlined in our L2 results presentation, performance analysis of C++ abstractions continues to be challenging, in part because the HPC community has asked tool developers to map analysis of routines directly to pragmas. This leads to a number of issues most notably that in OpenMP programs performance is often attributed to Kokkos and RAJA runtime headers rather than the calling application. Our development of the KokkosP interfaces to address this issue required communication with vendors. In an ideal world connection to the programming model via these interfaces opens the door for accurate profiling with parallel

context awareness. Agreeing on similar interfaces across the Trilab community is a possibility which may need to be explored. This continues to be the subject of further work at Sandia.

Another issue explored in this L2 was the importance of "productivity" as a metric for introducing programming models to our code base. Given the considerable breadth and size of the application portfolio at Sandia and the other trilabs, developing lighter weight methods of introducing parallelism so that step refinement is possible is clearly profitable. Our experiences in this L2 marry with feedback from our production users who want to be able to introduce on-node parallelism into their codes with as fewer modifications to the application code base as possible. Once these initial parallel kernels are ported and running then a post-introduction optimization phase will allow for greater performance to be achieved with further code refactoring. This process was reflected in our "swim-lane" approach outlined in this report.

Finally, we continue to believe that a collaborative approach to codesign which seeks to build on best practices between the trilabs is the most productive outcome of this L2. Our discussions with the RAJA team at LLNL and the users of Kokkos at LANL continue to educate us on issues which affect the broad community. Collaboration through the ATS Centers of Excellence and our plans for the FY16 Trilab L2 will continue to ensure that communication is enhanced and deepened for all of our benefit.

Chapter 4

Los Alamos National Laboratories

4.1 Introduction

The primary distinction of our research strategy is the addition of a system-level runtime investigation using the Legion programming model and runtime library. We focus on only one proxy application, but with two distinct runtime implementations that target different levels of parallelism within a system. In particular, we target both internode and intranode parallelism using what are effectively functional programming models: A task-graph, data-centric model (Legion), and a fine-grained, single-address-space model (Kokkos). These are targeted at system and node level resources, respectively. Our goals are similar to those stated in § 2.1. Additionally, LANL hopes to help our collaborators improve and vet their implementations (Raja and Kokkos) by adding coverage of a larger set of applications and user experiences.

Although not a part of this milestone, we hope that our work will lay the foundation for a system-wide model that combines both task-graph and fine-grained models into a sustainable approach to exascale.

4.2 SNAP Proxy Application

SNAP serves as a proxy application to model the performance of a modern discrete ordinates neutral particle transport application. SNAP may be considered an update to Sweep3D, intended for hybrid computing architectures. It is modeled off the Los Alamos National Laboratory code PARTISN. PARTISN solves the linear Boltzmann transport equation (TE), a governing equation for determining the number of neutral particles (e.g., neutrons and gamma rays) in a multi-dimensional phase space. SNAP itself is not a particle transport application; SNAP incorporates no actual physics in its available data, nor does it use numerical operators specifically designed for particle transport. Rather, SNAP mimics the computational workload, memory requirements, and communication patterns of PARTISN. The equation it solves has been composed to use the same number of operations, use the same data layout, and load elements of the arrays in approximately the same order. Although the equation SNAP solves looks similar to the TE, it has no real world relevance.

The solution to the time-dependent TE is a "flux" function of seven independent variables: three spatial (3-D spatial mesh), two angular (set of discrete ordinates, directions in which particles travel), one energy (particle speeds binned into "groups"), and one temporal. PARTISN, and therefore SNAP, uses domain decomposition over these dimensions to coherently distribute the data and the tasks associated with solving the equation. The parallelization strategy is expected to be the most efficient compromise between computing resources and the iterative strategy necessary to converge the flux.

The iterative strategy is comprised of a set of two nested loops. These nested loops are performed for each step of a time-dependent calculation, wherein any particular time step requires information from the preceding one. No parallelization is performed over the temporal domain. However, for time-dependent calculations two copies of the unknown flux must be stored, each copy an array of the six remaining dimensions. The outer iterative loop involves solving for the flux over the energy domain with updated information about coupling among the energy groups. Typical calculations require tens to hundreds of groups, making

the energy domain suitable for threading with the node's (or nodes') provided accelerator. The inner loop involves sweeping across the entire spatial mesh along each discrete direction of the angular domain. The spatial mesh may be immensely large. Therefore, SNAP spatially decomposes the problem across nodes and communicates needed information according to the KBA method. KBA is a transport-specific application of general parallel wavefront methods. Nested threads, spawned by the energy group threads, are available to use in one of two ways. Per one approach, nested threads may be used to further parallelize the work to sweep different energy groups assigned to a main-level thread. This option is still experimental and has only been implemented to work in the case of using a single MPI process. Alternatively, nested threads are used to perform "mini KBA" sweeps by concurrently operating on cells lying on the same diagonal of spatial sub-domains already decomposed across the distributed memory architecture (i.e., different MPI ranks). Lastly, although KBA efficiency is improved by pipelining operations according to the angle, current chipsets operate best with vectorized operations. During a mesh sweep, SNAP operations are vectorized over angles to take advantage of the modern hardware.

SNAP should be tested with problem sizes that accurately reflect the types of calculations PARTISN frequently handles. The spatial domain shall be decomposed to 2,000–4,000 cells per node (MPI rank). Each node will own all the energy groups and angles for that group of cells; typical calculations feature 10–100 energy groups and as few as 100 to as many as 2,000 angles. Moreover, sufficient memory must be provided to store two full copies of the solution vector for time-dependent calculations. The preceding parameters assume current trends in available per core memory. Significant advances or detriments affecting this assumption shall require reconsideration of appropriate parameters per compute node.

SNAP was developed by Joe Zerr (CCS-2) and Randy Baker (CCS-2) at Los Alamos National Laboratory. Randy Baker is also the lead developer of the PARTISAN code.

4.3 Legion Investigation

4.3.1 Programming Model & Runtime

Legion is a data-centric programming model for writing high-performance applications for distributed heterogeneous architectures. Making the programming system aware of the structure of program data gives Legion programs three advantages:

User-Specification of Data Properties:
Legion provides abstractions for programmers to explicitly declare properties of program data including organization, partitioning, privileges, and coherence. Unlike current programming systems in which these properties are implicitly managed by programmers, Legion makes them explicit and provides the implementation for programmers.

Automated Mechanisms:
Current programming models require developers to explicitly specify parallelism and issue data movement operations. Both responsibilities can easily lead to the introduction of bugs in complex applications. By understanding the structure of program data and how it is used, Legion can implicitly extract parallelism and issue the necessary data movement operations in accordance with the application-specified data properties, thereby removing a significant burden from the programmer.

User-Controlled Mapping:
By providing abstractions for representing both tasks and data, Legion makes it easy to describe how to map applications onto different architectures. Legion provides a mapping interface which gives programmers direct control over all the details of how an application is mapped and executed. Furthermore, Legions understanding of program data makes the mapping process orthogonal to correctness. This simplifies program performance tuning and enables easy porting of applications to new architectures.

There are three important abstractions in the Legion programming model:

Logical Regions:

Logical regions are the fundamental abstraction used for describing program data in Legion applications. Logical regions support a relational model for data. Each logical region is described by an index space of rows (either unstructured pointers or structured 1D, 2D, or 3D arrays) and a field space of columns. Unlike other relational models, Legion supports a different class of operations on logical regions: logical regions can be arbitrarily partitioned into sub-regions based on index space or sliced on their field space. Data structures can be encoded in logical regions to express locality with partitioning and slicing describing data independence.

Tree of Tasks using Regions:

Every Legion program executes as a tree of tasks with a top-level task spawning sub-tasks which can recursively spawn further sub-tasks. All tasks in Legion must specify the logical regions they will access as well as the privileges and coherence for each logical region. Legion enforces a functional requirement on privileges which enables a hierarchical and distributed scheduling algorithm that is essential for scalability.

Mapping Interface:

Legion makes no implicit decisions concerning how applications are mapped onto target hardware. Instead mapping decisions regarding how tasks are assigned to processors and how physical instances of logical regions are assigned to memories are made entirely by mappers. Mappers are part of application code and implement a mapping interface. Mappers are queried by the Legion runtime whenever any mapping decision needs to be made. Legion guarantees that mapping decisions only impact performance and are orthogonal to correctness which simplifies tuning of Legion applications and enables easy porting to different architectures.

Legion Contributors:

Stanford	**Los Alamos**	**NVIDIA**
Sean Treichler	Pat McCormick	Michael Bauer (Stanford Site)
Elliot Slaughter	Charles Ferenbaugh	
Wonchan Lee	Samuel Gutierrez	
Zhihao Jia	Kei Davis	
Alex Aiken		

4.3.2 SNAP - Legion

The Legion implementation of SNAP for this milestone is designed to test Legion developer productivity, provide a target example for a Legion abstraction layer, and test ideas for system-level portability and performance improvements over MPI. We implemented about 90% of the original Fortran version of SNAP in C++ using the Dragon abstraction layer, developed by Joshua Payne, to remove much of the Legion boilerplate code. Since Legion is a task-graph model we needed to choose implementation strategies for data layout, task-execution layout, and which of the original Fortran subroutines to wrap in tasks. One major goal here was to implement the KBA-sweep using only data-dependencies between decomposed domains. By correctly identifying the data requirements of each task performing the KBA-sweep, the runtime can build an execution graph which not only correctly performs a complicated sweep, but also finds locations where sweeps in multiple directions can be performed simultaneously and places where non-related tasks can be executed while parts of the sweep are still executing.

For this implementation, we choose to create a different logical region for each "chunk" of the decomposed domain. In MPI terms, this corresponds to a different logical region for each rank. It would be possible to do a single logical region for the entire domain, and then partition that region into subregions for each rank, but at the current time it was not feasible due to index space sizes being limited by 32-bit pointers. In the following example, we create a logical region that contains the regions for each sub-domain. We then loop over all of the sub-domains and create a logical region for that subdomain.

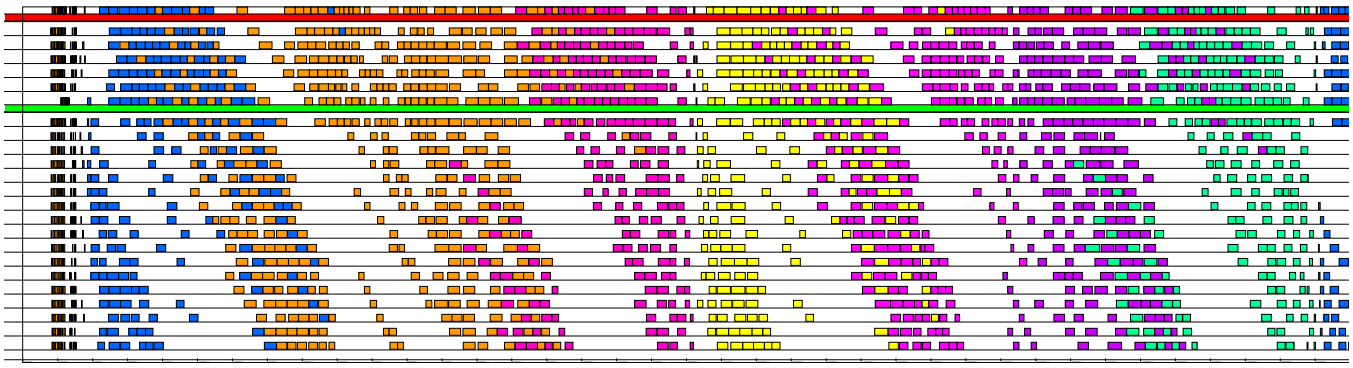


Figure 4.1: A single KBA-sweep of the Legion version of SNAP 6x6x6.

```
LRWrapper chunkArrays; // Contains regions for each sub-domain
chunkArrays.create(ctx,runtime,"chunkArrays",{nCx,nCy,nCz},
                   FIDs::Array::angFluxes,LRWrapper());

RegionRequirement rr(chunkArrays.lr,WRITE_ONLY,EXCLUSIVE,chunkArrays.lr);
    rr.add_field(FIDs::Array::angFluxes);
// Mapping our global container to a physical region
PhysicalRegion pr = runtime->map_region(ctx,rr);

// The accessor to the data stored in the physical region
LegionMatrix<LRWrapper> angFluxes(chunkArrays,pr,FIDs::Array::angFluxes);

for(int k=0;k<nCz;k++)
  for(int j=0;j<nCy;j++)
    for(int i=0;i<nCx;i++)
      angFluxes(i,j,k) = createLRwrapper(ctx,rt,"angFluxes",{nang,nx_l,ny_l,nz_l,ng,noct},
                  FIDs::AngFlux::ptr_in,double(),
                  FIDs::AngFlux::ptr_out,double());
```

The second design choice to be made for this implementation is how to break the problem down into asynchronous tasks. For several cases, this was straightforward: inner and outer source calculations, the convergence tests, and the tasks to save the fluxes from the last iteration. These functions were broken up into a single task per sub-domain, although they could potentially be further divided up into one task per sub-domain per group. However, at that point the work load becomes too fine and the overhead of the task-launch outweighs the cost of the task. For the KBA-sweep, there is one task per sub-domain per octant, or $nCx * nCy * nCz * noct$ tasks. For example, in a run with a domain decomposed into 8x8x8 chunks the total number of tasks is 7168.

4.3.3 Performance

At first glance the performance for the Legion version of SNAP leaves something to be desired. In its current state, the Legion implementation is at best 8x slower than the Fortran version and 50x slower at worst. However, there are several very important things that should be understood before dismissing the Legion runtime.

First, this comparison is between a first-pass Legion implementation and a highly optimized MPI-OpenMP Fortran version. Second, we were experiencing significant issues with the default Legion mapper, and thus were not able to run multi-node tests. Third, we did not have time to implement an OpenMPversion, and thus we were unable to test fine-grained parallelism (Note that the potential to combine the Legion and Kokkos implementations of SNAP would resolve this issue.) When comparing the results for a 4x4x4 grid versus an 8x8x8 grid, presented in Table 4.1, at larger global dimensions the 8x8x8 is faster than the 4x4x4. This leads us to believe that we are not engaging as many cores as we could be if we had a fine-grain

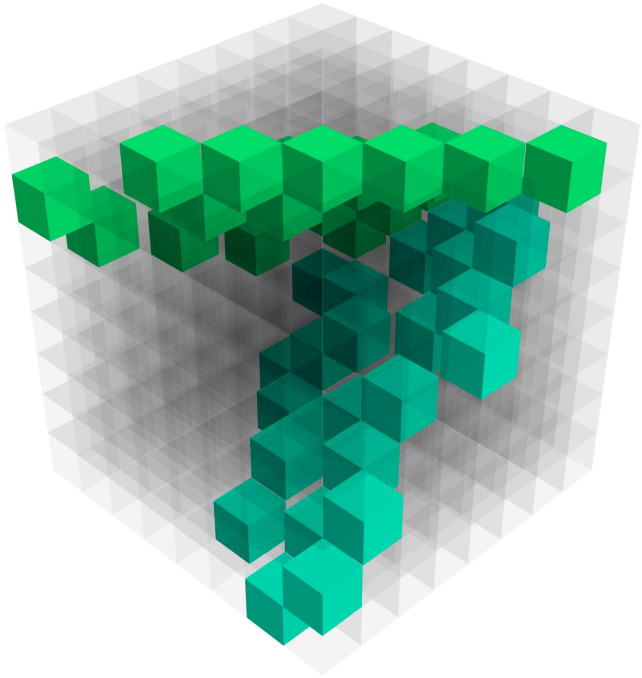


Figure 4.2: Visualization of a single KBA-sweep of the Legion version of SNAP 6x6x6.

parallel model embedded within the task. Fourth, Legion's default mapper currently performs a very poor job at memory management and it is likely that we are seeing many more memory copies than are strictly necessary. Below, note the scaling relative to global domain size between the two versions. We expect to see similar scaling relative to problem size, however the current Legion runtime scales much less favorably.

Global Dims	Proc Dims	Fortran	Legion
32x32x32	4x4x4	0.7659	8.51917
64x64x64	4x4x4	1.6531	60.9977
96x64x64	4x4x4	2.2266	90.7992
32x32x32	8x8x8	4.2654	8.09240
64x64x64	8x8x8	9.7494	35.2876
96x64x64	8x8x8	10.654	49.1353

Table 4.1: Timing comparison of a single KBA-Sweep Fortran SNAP vs Legion SNAP. Single node, 32 logical cores.

We do not believe that any of the above performance comments are the result of design decisions with Legion's model, i.e. they can be solved with additional development time for Legion to mature. We have a close collaboration with the primary Legion developers, and we are working to provide feedback to help improve both the Legion model and the evolving implementation. Two priorities that have come out of this work are the desire that Legion have a better default mapper, and turn-key support for MPI-Legion interoperability.

4.3.4 Portability

One of the major selling points of Legion is portability of a "correct" algorithm. In essence, the application developer does not need to be aware of which node, numa-domain, GPU, or, generically, address space that

the data might live on because this information is abstracted away by the runtime. Again, the default mapper requires some improvement, but with minor changes we were able to simultaneously generate both GPU and CPU versions of simple kernels using the Legion with our Dragon abstraction layer. The design of Dragon is such that it will enable us to generate execution kernels that can take advantage of multi-core, multi-gpu, and multi-node resources, a feature that is outside the scope of Legion's programming model (Legion handles the data motion required to satisfy a task's region requirements, but does not seek to provide a low-level model for data-parallel arithmetic.)

```
class DAXPY
{
private: float a;
public: // Required members
  static const int SINGLE = false;
  static const int INDEX = true;
  static const int MAPPER_ID = 0;
public: // Required methods

  IndexKernelArgs genArgs(Context ctx, HighLevelRuntime* runtime,
        float _a,LRWrapper _x,LRWrapper _y,LRWrapper _z,int nSub) :
  {
    a = _a;
    IndexKernelArgs args;
    LPWrapper xp,yp,zp;
    xp.simpleSubPart(ctx,runtime,_x,nSub);
    yp.simpleSubPart(ctx,runtime,_y,nSub);
    zp.simpleSubPart(ctx,runtime,_z,nSub);

    args.add_arg(xp,0,READ_ONLY,EXCLUSIVE);
    args.add_arg(yp,0,READ_ONLY,EXCLUSIVE);
    args.set_result(zp,0,EXCLUSIVE); // Must be WRITE_ONLY

    return args;
  }
  // Dummy function for type analysis
  static float evaluate_s(int i,LegionMatrix<float> _x,LegionMatrix<float> _y){}

  __host__ __device__
  float evaluate(int i,LegionMatrix<float> _x,LegionMatrix<float> _y)
  {
    // This shows how you can use the __CUDA_ARCH__ macro to differentiate host code from
        device code
  #ifdef __CUDA_ARCH__
    printf("GPU: x(%i) + y(%i) = %f + %f\n",i,i,_x(i).cast(),_y(i).cast());
  #else
    printf("CPU: x(%i) + y(%i) = %f + %f\n",i,i,_x(i).cast(),_y(i).cast());
  #endif
    return a*_x(i) +_y(i);
  }
};
```

4.3.5 Productivity

Productivity is difficult to quantify when comparing two implementations that differ by both language and runtime. Implementing SNAP in base Legion would be onerous, so we chose to first develop an abstraction layer, Dragon, to hide much of the boilerplate code required by the Legion runtime. Historically, this is a very common approach, e.g., users generally do not make direct calls to MPI throughout their codes. Using this abstraction layer, we were able to produce a rewrite of SNAP that required approximately 2/3 of the lines of the original Fortran code. There are, however, some issues, mainly associated with the difficulty in debugging Legion applications and the default mapper. It can sometimes be a complex task to determine the cause of a runtime error. We have experienced many errors caused by bugs in the default mapper, and a few related to garbage collection. Tracking down the precise cause of errors can be both extremely difficult and

frustrating. These challenges are being communicated to the Leigon devleopers, and will likely be addressed as Legion matures.

These things being said, there is substantial promise to the Legion programming model when one considers that Legion allows the developer to program in a way that can ignore data dependencies that do not effect the current task, i.e., once a task has defined its region requirements, it can safely operate on its local data without having to reason about side effects. This is a natural consequence of pure functional models. However, Legion exposes an elegant interface for describing computational regions, and the Legion runtime obviates the time-consuming logic required to explicitly update dependencies between tasks.

4.4 Kokkos Investigation

4.4.1 Programming Model

Kokkos implements 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 (e.g `parallel_for`) and data management (Kokkos View). Kokkos is designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. For parallel execution, it can use OpenMP, Pthreads, CUDA, and other threading APIs as backend programming models, while for memory layout, it supports x86 Haswell, Xeon Phi, and Power8, among others.

The core developers of Kokkos are Carter Edwards and Christian Trott at the Computer Science Research Institute (CSRI) of the Sandia National Laboratories.

4.4.2 SNAP - Kokkos

The Kokkos implementation of SNAP for this milestone is designed to test ideas for node-level portability and performance improvements over more direct, hard-coded implementation strategies. Starting from the original Fortran version of SNAP with hand-tuned OpenMP, we began by isolating the core KBA sweep kernel, `dim3_sweep`, and removing the OpenMP directives from it. Using this kernelized version as a basis, we then implemented two different versions of the sweep algorithm in C++: (1) a *direct* version that simply wraps each for-loop with Kokkos parallel invocation calls, and (2) a more sophisticated, hierarchically parallel version that seeks to expose the maximum parallelism available from this numerical method. With expert help from the Kokkos developers, a third version was developed; a modification of the hierarchical parallelism version with performance enhancements. These implementations are discussed in more detail in the following sections. We reference a similar representative section of code from each, save the optimized version, beginning with the Fortran version in Figure 4.3.

```
!------------------------------------------------------------------------
!
!       Compute the numerator for the update formula
!------------------------------------------------------------------------

        pc = psi + psii(:,j,k)*mu*hi + psij(:,ic,k)*hj + psik(:,ic,j)*hk
        IF ( vdelt /= zero ) pc = pc + vdelt*ptr_in(:,i,j,k)
```

Figure 4.3: A representative sample of Fortran implementation of SNAP.

Direct Implementation

In the direct implementation, a sample of which is shown in Figure 4.4, we simulated the effects of giving an application kernel to an application programmer who would not have foreknowledge of fine-grained parallel execution or data layout abstraction APIs. We assumed that they would proceed, more or less, line by line and transform the implicit parallel loops of the kernelized Fortran version by writing a functor for each class

of implicit array algebra. That is to say, we assumed enough general computing knowledge to facilitate learning the basics of Kokkos (how to allocate data in a Kokkos View, how to invoke a parallel region using a functor (including the Kokkos specific aspects thereof), how to generalize and classify array updates), but not enough sophistication to use all of the advanced features of Kokkos. We then designed the generic functors to fit the roles of the array algebra in the kernelized Fortran version and replicated it using these functors. It should be noted that this type of implementation does what is colloquially known as "parallelizing the inner loop" and is known to perform poorly due to thread contention. However, given the assumptions we started with, this is the type of implementation that would result. Moreover, we are given the ability to measure the difference in performance, relatively and absolutely, as compared to later versions.

```
////////////////////////////////////////////////////////////////////////////////
// H - compute the numerator for the update formula
////////////////////////////////////////////////////////////////////////////////
{
//          pc = psi + psii(:,j,k)*mu*hi + psij(:,ic,k)*hj + psik(:,ic,j)*hk // pc(nang)
//          IF ( vdelt /= zero ) pc = pc + vdelt*ptr_in(:,i,j,k)
}
      psii_slice = subview( psii, ALL(), j, k );
      psij_slice = subview( psij, ALL(), ic, k );
      psik_slice = subview( psik, ALL(), ic, j );
      parallel_for( nang,
          update_Aa_Bb_C1C2c_D1D2d_E1E2e
            < device_type, view_t_1d,        // pc
              view_t_1d,                     // psi
              view_t_1d_s, view_t_1d,        // psii_slice
              view_t_1d_s, view_t_1d,        // psij_slice
              view_t_1d_s, view_t_1d >       // psik_slice
            ( pc, c0,
              psi, c1,
              psii_slice, mu, hi,
              psij_slice, hj, c1,
              psik_slice, hk, c1 ) );
      if ( vdelt != c0 ) {
        ptr_in_slice = subview( ptr_in, ALL(), i, j, k );
        parallel_for( nang, update<device_type,view_t_1d,view_t_1d_s>( pc, c1, ptr_in_slice,
             vdelt ) );
      }
```

Figure 4.4: A representative sample of the direct Kokkos implementation of SNAP.

Hierarchical Parallelism Implementation

The hierarchical parallelism version (heretofore, HP), was designed to use the set of features that would give the basis of performance using Kokkos. HP in Kokkos allows for three layers of parallelism, to wit: the outermost layer is threads (typically, real cores), shown in Figure 4.5, then threads in a thread group (e.g. Hyperthreads), shown in Figure 4.6, and the innermost is vectorized instructions, where we revisit the representative code sample in Figure 4.7. We identified the appropriate places in SNAP where both parallel performance and correctness could be achieved by mapping SNAP to three level parallelism. Energy groups were mapped to threads, diagonals to thread groups, and (what was implicit in Fortran) array arithmetic to vector regions. The intention was that version, HP, should correctly use Kokkos, and allow mapping of parallelism to multiple architectures with a single codebase, but to leave on the table some room for Kokkos specific, but still portable, optimizations.

Optimized HP Implementation

The Optimized Hierarchical Parallelism version was developed by consulting Kokkos developers (specifically, Christian Trott) and sharing initial performance numbers of the HP implementation. Christian was able to

```
const team_policy_t policy( N, hyper_threads, vector_lanes );

  parallel_for( policy, dim3_sweep2<  team_policy_t,
                                      view_1d_t, view_2d_t, view_3d_t, view_4d_t,
                                      view_5d_t, view_6d_t, view_7d_t >
                                    ( M, L,
                                      ng, cmom, noct,
                                      nx, ny, nz, ichunk,
                                      diag,
                                      id, ich, oct,
                                      jlo, jhi, jst, jd,
                                      klo, khi, kst, kd,
                                      psii, psij, psik,
                                      jb_in, kb_in,
                                      qim, qtot, ec,
                                      mu, w,
                                      wmu, weta, wxi,
                                      hi, hj, hk,
                                      vdelt, ptr_in, ptr_out,
                                      flux, fluxm, psi, pc,
                                      jb_out, kb_out,
                                      flkx, flky, flkz,
                                      hv, fxhv, dinv,
                                      den, t_xs ) );
```

Figure 4.5: An example of the outermost layer of parallelism in Kokkos as used in the HP Kokkos implementation of SNAP.

```
parallel_for( Kokkos::TeamThreadLoop( team_member, M ), [&]( const int& mm ) { // diag loop
```

Figure 4.6: An example of the middle layer of parallelism in Kokkos as used in the HP Kokkos implementation of SNAP.

suggest optimizations to be made, including thread specific scratch memory, and even a fix for a parallel correctness issue which resulted from a misunderstanding of which parallel regions could be active simultaneously in the Kokkos HP model. It is expected that this implementation have the superior performance of the three, and that it perform approximately as well as the original OpenMP Fortran kernelized version.

4.4.3 Performance

We tested three different configurations of problem size, versus the three Kokkos implementations and the Fortran amongst various hardware architectures available. The problem size configurations tested were denoted A, B, and C. Configuration A featured a spatial domain of 8x8x8, 40 energy groups, and 240x8 angles. B tested a domain of 16x16x16, 80 energy groups, and 360x8 angles. Finally, C tested a 32x32x32 domain, 120 energy groups, and 480x8 angles. We made use of computing resources hosted in the Darwin cluster at LANL, and the White cluster at SNL. We tested the kernelized version of Fortran SNAP both with (F O) and without OpenMP (F NO). We tested all three implementations of Kokkos SNAP, including the Direct (DK), Hierarchical Parallelism (KHP), and Optimized HP (OPT), using thread backends of Serial (S), OpenMP (O), and PThreads (T).

The direct implementation (DK) fared as expected, poorly. In fact, when run on Haswell-equipped nodes, in OpenMP and Pthreads mode, it would actually take more time than when run in Serial mode, due to thread contention, which was verified using Intel's diagnostic tool, VTune. The HP Kokkos version was an improvement over the direct implementation, and the Optimized HP (OPT) was the best of the three, as

```
////////////////////////////////////////////////////////////////////////////////
// H - compute the numerator for the update formula
////////////////////////////////////////////////////////////////////////////////
{
//          pc = psi + psii(:,j,k)*mu*hi + psij(:,ic,k)*hj + psik(:,ic,j)*hk // pc(nang)
//          IF ( vdelt /= zero ) pc = pc + vdelt*ptr_in(:,i,j,k)
}
        parallel_for( Kokkos::ThreadVectorLoop( team_member, L ), [&]( const int& ll ) {
          pc(ll,mm) = psi(ll,mm) + psii(ll,j,k,g)*mu(ll)*hi + psij(ll,ic,k,g)*hj(ll)
                                 + psik(ll,ic,j,g)*hk(ll);
          if ( vdelt(g) != c0 ) {
            pc(ll,mm) = pc(ll,mm) + vdelt(g) * ptr_in(ll,i,j,k,oct,g);
          }
        }); // ThreadVectorLoop H
```

Figure 4.7: An example of the innermost layer of parallelism in Kokkos as used in the HP Kokkos implementation of SNAP.

seen in Figures 4.8 and 4.9, when tested on Haswell processors. In the end, there were combinations of problem size and threading model for which OPT performed competitively and better than Fortran (with OpenMP turned back on), as seen in Figures 4.10, which compares HP versions of SNAP with Fortran on an E5-2698, and 4.11, which compares just the OPT Kokkos SNAP and Fortran SNAP with OpenMP run on the two processor platforms in ATS1 and ATS2, Haswell, and Power8 as a stand-in for future Power processors. ***This is an excellent result in favor of Kokkos, in that, code written in this model can perform as well as Fortran on current and future ASC platforms.***

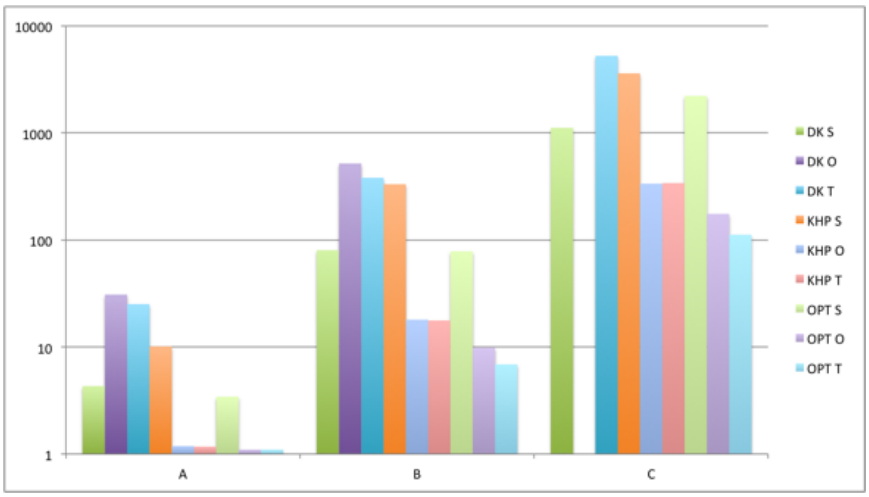


Figure 4.8: Results from running the Kokkos implementations of SNAP, Direct (DK), Hierarchical Parallelism (KHP), and Optimized HP (OPT), with thread models Serial (S), OpenMP (O), and PThreads (T), using test configurations A, B, and C, on an E5-2660 Haswell equipped compute node in the Darwin cluster. The y-axis is wall-clock time and thus, within a test configuration, lower is better.

4.4.4 Portability

We had access to compute nodes that included Haswell (similar to those in ATS1, Trinity), Xeon Phi (KNC), Nvidia K40, and Power8. Of those four architectures, we were able to run the same code base of the Kokkos implementations on all but the K40. There were additional modifications needed, above and beyond

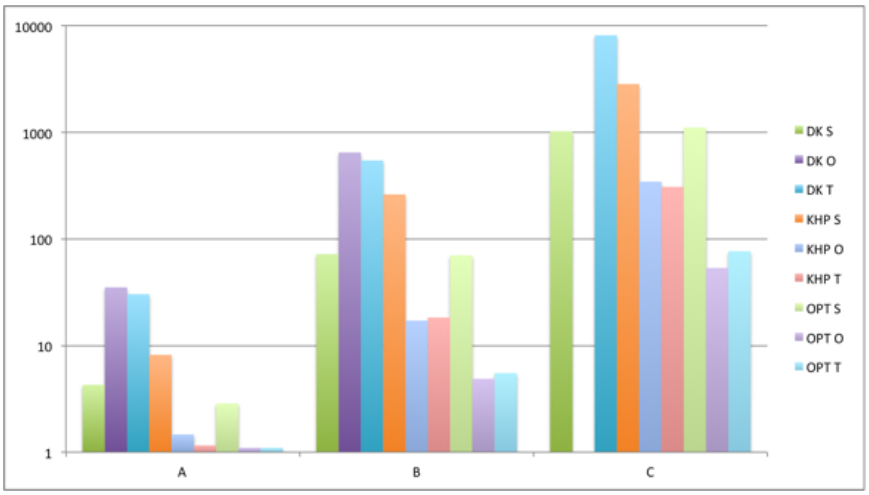


Figure 4.9: Results from running the Kokkos implementations of SNAP, Direct (DK), Hierarchical Parallelism (KHP), and Optimized HP (OPT), with thread models Serial (S), OpenMP (O), and PThreads (T), using test configurations A, B, and C, on an E5-2698 (as in Trinity) Haswell equipped compute node in the Darwin cluster. The y-axis is wall-clock time and thus, within a test configuration, lower is better.

using Kokkos datatypes, in order to support sweep scheduling data structures that would be necessary to successfully run using the CUDA Kokkos device.

4.4.5 Productivity

If Kokkos-equipped code were to be written in the same fashion as the direct implementation, it is possible that productivity could be either lower (more lines of code necessary), or higher (re-use of generic functors). For the HP implementations, both base and optimized, sections of computational code expressed as lambdas in Kokkos format are generally the same number of lines of code as Fortran, with similar appearance, including only an extra line at the start of the vectorized parallel region (see Figure 4.7).

4.5 Conclusions

This work has established that we can use functional programming models to implement discrete ordinates type sweep methods, and that, in so doing, it is possible to make gains in performance, portability and productivity. An important consequence of this work is the improved relationship with our sister laboratories, and the close collaboration with the developers of the two models that we investigated. Both Legion and Kokkos show tremendous promise, and we will continue our close collaboration with Stanford and SNL to improve and integrate these projects into our set of tools. In particular, both of these models are currently being vetted for inclusion in our ATDM effort. We are also collaborating with SNL and LLNL to increase NNSA representation on the C++ standard committee with the goal of incorporating parts of the Kokkos model into the 2020 standard.

4.5.1 Lessons Learned

The most significant lesson that we learned was the value and need for close collaboration with the developers of Legion and Kokkos. Expert training and direct collaboration with these developers was essential to avoiding bottlenecks and misunderstandings in our investigation. This aspect may become less important as these models mature, but for the time being, this interaction is critical.

Another important lesson is that it is still unclear where to draw a line between the system and node-level runtimes that we need in order to address a whole system. Legion offers a data model that can extend into lower-level address spaces, e.g., GPU or accelerator main memory. This ability is desirable because it

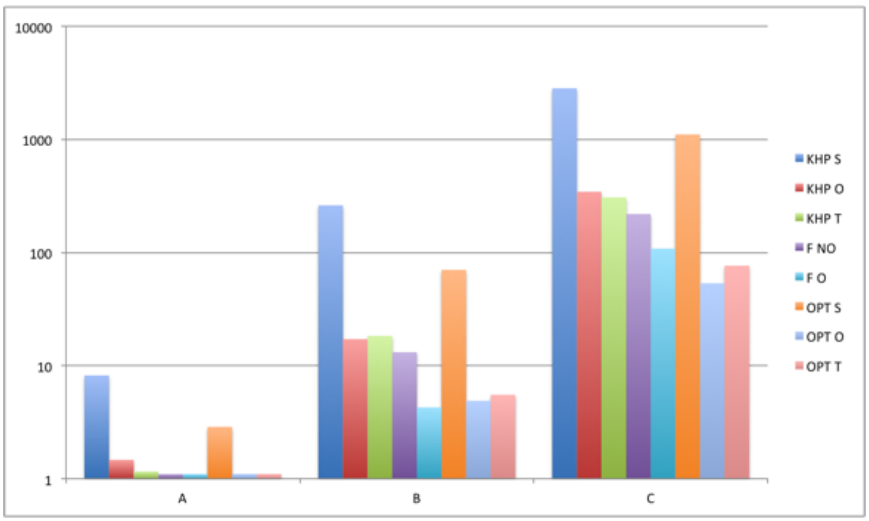


Figure 4.10: Results from running the Kokkos implementations of SNAP including Hierarchical Parallelism (KHP), and Optimized HP (OPT), with thread models Serial (S), OpenMP (O), and PThreads (T), as well as the Fortran version (F NO) and Fortran with OpenMP enabled (F O), using test configurations A, B, and C, on an E5-2698 (as in Trinity) Haswell equipped compute node in the Darwin cluster. The y-axis is wall-clock time and thus, within a test configuration, lower is better.

allows the Legion runtime to place data where they are needed before executing a task. However, similar functionality is provided by Kokkos, at least as an interface for moving data. While Legion might use the Kokkos interface to control data movement, there is possible contention in the control of data layout. Both the Legion and Kokkos project developers understand the importance of correct data layout for performance. Our inclination is that this is the domain of the node-level runtime, and is not necessarily a concern for Legion. However, there are obvious benefits in having the Legion runtime be aware of data layouts in order to provide data transposition. It remains unclear as to how these two runtimes could coordinate to provide a more seamless interface to developers. We believe that having logically hierarchical runtimes for system-level and node-level resources is good, as this recognizes a separation that makes it possible for humans to reason about the different levels of parallelism required to fully subscribe the whole system. However, without shared semantics for describing data layout, this will remain a challenge.

4.5.2 Future Work

As stated in our introduction, we would like to continue this work and bring our Legion and Kokkos implementations together into a single program. This work would help to advance several objectives: It would provide an answer to the simple question of compatibility, i.e., "Can the runtimes co-exists?"; It would help us to determine whether or not a single semantic programming model is possible or desirable; and It will continue to foster collaborations between LANL and the developers of these two projects. All of these are important goals that we will try to achieve.

Figure 4.11: Results from running the Fortran OpenMP (F O) and Optimized HP Kokkos (OPT O) with OpenMP, using test configurations A, B, and C, on E5-2698 (as in Trinity) and Power8 compute nodes, in the Darwin (LANL) and White (SNL) clusters, respectively.The y-axis is wall-clock time and thus, within a test configuration, lower is better.

Chapter 5

Conclusions

While this L2 milestone was designed so that each individual laboratory could further explore the potential of the programming models they focused on, and the conclusions specific to each of those models were elaborated upon in each chapter, the collaborative nature of the effort uncovered several common themes about lessons learned that we capture here as overall conclusions to the effort.

While it was clear that these models can and do provide performance portability, these tools are simply *enablers* of reaching that goal, not magic bullets that developers can naively adopt and expect immediate positive results. With time, these tools will continue to improve and harden to the point that developers who have a good understanding of how the languages features and underlying runtime systems are being used will find them much easier to use, but as with any complex and general-purpose solution, we are just beginning to approach that target. The lessons of co-design are clear - that these middleware layers must collaborate early and often with both the compiler and hardware vendors on one side, and a diverse set of applications on the other to optimize this process.

Training and documentation on the use of these models is also a critical next step in their development, and we are just beginning this process. In particular, the Kokkos team has a formal release process as part of the Trllinos suite of tools, is offering hands-on training sessions, and continuously improving the documentation. The RAJA team is developing a formal release process as part of their ATDM CS toolkit, developing additional documentation, and working closely with developers inside the LLNL production code teams on developing a core set of training materials. Likewise, the Legion team at Stanford has had training sessions, hack-a-thons with LANL and others, a user guide, and a number of examples from which developers can start to learn how to think about developing code using these task-based models. In all cases, improved access to online forums and wikis through which developers can share their experiences are another important goal.

However, training in the specifics of these models (particularly RAJA and Kokkos) are not sufficient. For shared memory programming with threads, developers must still understand how to write correct thread-safe code. For example, neither RAJA or Kokkos will automatically parallelize a loop that is inherently thread unsafe. Training in at least the basics of OpenMP, and preferably CUDA, are important prerequisites to using these tools, which can then take correct thread-safe code and enable portability and increased performance on multiple architectures.

Finally, the collaboration amongst the ASC labs has demonstrated that co-design with vendors does work, and is most effective when the labs can speak with a common voice about issues. This has been particularly valuable in the conversations with compiler writers, who are just now recognizing the power of C++ features, how we intend to use them, and what they must do to ensure that expected optimizations are not being flummoxed by these newer and non-traditional uses of the language.

In conclusion, the ASC tri-labs will continue to develop and advance these programming models through the co-design process: working together to identify common issues, with vendors to improve compiler technology and hardware features planned in their roadmaps, and with application teams to make these models usable and effective in a production environment. A followon ASC L2 milestone between the three labs is planned for FY16 which will take the lessons learned from this year, and demonstrate their use in either our production integrated codes or new ATDM next-gen code projects.

Appendix A

Technical Lessons Learned

This appendix is meant to be a central place to capture detailed lessons learned on advanced architectures through the execution of this L2. The data provided here is too detailed for the main report, but may be of value to others pursuing these programming models.

A.1 Issues Encountered

It is clear that both RAJA and Kokkos performance and flexibility depends strongly on good compiler and programming model support. During the course of this L2, and in other work, we have identified various issues that need to be addressed and/or which require deeper investigation to understand better. The following is a partial list of some key concerns:

- C++ abstractions typically add a $5-10\%$ performance overhead, which often varies by compiler and code context.
- Optimizations, such as SIMD vectorization, are often disabled when OpenMP is enabled.
- The `restrict` pseudo-keyword is only honored in very specific scopes, which also vary by compiler since restrict is not part of the C++ standard.
- The CUDA `device` attribute must be attached to a lambda function where it is defined to be used in a GPU kernel, which clutters application code, forces execution decisions to compile-time, and limits RAJA flexibility by forcing compile-time decisions about whether to run on the CPU or GPU.
- OpenMP 4.0 lacks unstructured data mapping support [29], which impedes its viability as a useful accelerator backend for abstraction- based models like RAJA that provide generic loop traversal templates which know nothing about the details of the loop body they are executing. OpenMP 4.1 is expected to address this [4].

Next, we discuss several CUDA topics and present issues to be aware of and best practices.

A.1.1 CUDA Unified Memory

Initialization. Managed Memory is setup via two constructs, either utilizing the CUDA API call cudaMallocManaged() or by defining a global variable using the CUDA __managed__ keyword. Similar to the C malloc() routine, cudaMallocManaged() does not initialize the allocated memory. We recommend using cudaMemset() to explicitly set the memory to 0 on the GPU device.

We have also observed slowdowns when initializing Managed Memory on the host, although the memory was allocated on the device (i.e., with `CUDA_VISIBLE_DEVICES` set). As usual, it is best to defer data movement between host and device until absolutely necessary. We discovered this while attempting to

generate a "single-source" code base which initially had the host assigned to consistently initialize allocated memory.

Synchronization. When both CPU and GPU are executing concurrently, the GPU has exclusive access to managed memory while a kernel is running. To avoid a segmentation fault, wait until the GPU completes it work and call cudaDeviceSynchronize(), or cudaStreamSynchronize() and cudaStreamQuery() assuming a return value of cudaSuccess.

Performance. When using CUDA Unified Memory (i.e., "managed memory"), it is important to set the environment variable `CUDA_VISIBLE_DEVICES` to target a specific GPU device (e.g., 0 or 1). Otherwise, a large performance penalty can result. Indeed, we have observed slowdowns by more than a factor of 20 in a CUDA variant of LULESH without this environment variable set properly. When the environment variable is not set, managed memory is allocated on the host allocation and setup in zero-copy (pinned) memory mode which means that GPU memory accesses are limited to PCI-express performance.

Much useful advice for CUDA GPU programming can be found on the *NVIDIA Parallel FORALL* Blog. For example, one useful pro-tip states: *"Unified Memory requires that all GPUs support peer-to-peer memory access, but this may not be the case where GPUs are connected to different I/O controller hubs on the PCI-Express bus. If the GPUs are not all peer-to-peer compatible, then allocations with cudaMallocManaged() falls back to device-mapped host memory (also known as "zero-copy" memory). Access to this memory is via PCI-express and has much lower bandwidth and higher latency. To avoid this fallback, you can use `CUDA_VISIBLE_DEVICES` to limit your application to run on a single device or on a set of devices that are peer-to-peer compatible" [22].*

CUDA_VISIBLE_DEVICES Settings.

The environment variable `CUDA_VISIBLE_DEVICES` discussed above can be used to target a single or multiple GPU devices [26]. Options include:

```
CUDA_VISIBLE_DEVICES=1        // Only device 1 will be visiable
CUDA_VISIBLE_DEVICES=0,1      // Devices 0 and 1 will be visible
CUDA_VISIBLE_DEVICES=''0,1''  // Same as above, quotation marks are optional
CUDA_VISIBLE_DEVICES=0,2,3    // Devices 0, 2, 3 visible; device 1 is masked
```

CUDA will enumerate the visible devices starting at zero. In the last case above, actual devices $0, 2, 3$ will appear numbered as $0, 1, 2$. If you set the order to $2, 3, 0$, those devices will be enumerated as $0, 1, 2$. If some value given to `CUDA_VISIBLE_DEVICES` is that of a device that does not exist, all valid devices with IDs before the invalid value will be enumerated, while all devices after the invalid value will be masked. To determine the device IDs available on your system, you can run the NVIDIA deviceQuery executable included in the CUDA SDK.

A.1.2 GPU floating point atomics

Constraints. While atomic functions that update a 32 or 64 byte word via read-modify-write are extremely convenient and simple to use, they present a significant performance challenge. On current Kepler-based Tesla GPU cards, atomics for double-precision floating point must be implemented (i.e., emulated) based on atomicCAS (atomic compare-and-swap) routines; this can slow an application significantly if it uses such an operation frequently. Single-precision atomics on Kepler are supported by hardware. But, even they should be used sparingly, if at all. Nonetheless, atomicCAS can be used to implement other atomic operations, such as a double-precision atomic add.

Compare-and-Swap Loop Design Pattern. The following two code-blocks illustrate an atomicCAS compare-and-swap loop design pattern. Jeff Preshing has a good explanation on his blog regarding this pattern and lock-free programming [27, 28]. The CAS loop repeatedly attempts to update a value at an address until it succeeds. Failure generally means that another thread succeessfully wrote to the memory address. When this occurs, the old value is updated from shared memory, and the loop repeats.

```
__device__ double atomicAdd(double* address, double val)
{
```

```
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
              __double_as_longlong(val + __longlong_as_double(assumed)));
        // Use integer comparison to avoid hang in case of NaN (NaN != NaN)
    } while (assumed != old);
    return __longlong_as_double(old);
}
```

```
// emulate atomic add for doubles
__device__ inline void atomicAdd(double *address, double value)
{
    unsigned long long oldval, newval, readback;
    oldval = __double_as_longlong(*address);
    newval = __double_as_longlong(__longlong_as_double(oldval) + value);
    while ((readback = atomicCAS((unsigned long long*)address, oldval, newval)) != oldval)
    {
        oldval = readback;
        newval = __double_as_longlong(__longlong_as_double(oldval) + value);
    }
}
```

Performance Comparison with Emulated Atomics. The following list summarizes throughput measurements for a reduction operation running on a GPU node (specifically, a K20 XM card on the LLNL IPA system). In particular, it illustrates typical performance degradation when using emulated atomics. The operation performs a sum reduction over a double array of length 1024 * 1024 * 32. Reduction is communication intensive (bandwidth limited) and so is measured in GB/s.

```
THREADS_PER_BLOCK 1024
Float : 4.58 GB/s
Double: 0.522 GB/s (main slow down due to software emulated atomicAdd)

THREADS_PER_BLOCK 512
Float : 13.42 GB/s
Double: 0.06GB/s

THREADS_PER_BLOCK 256
Float : 1.7 GB/s
Double: 0.023 GB/s

Benchmark: No atomic double
THREADS_PER_BLOCK 1024
Double: 6.79 GB/s
```

Avoiding Atomics. The previous results show that, when we remove atomics altogether, we achieve performance for the double precision sum reduction on par with float, where atomics for floats are supported in hardware and thus are very fast. The technique for removing atomics altogether in a reduction mainly involves a bit more book-keeping and finishing off the computation on the host. Here, each thread block uses a unique memory location for its update, which obviates the need for atomic operations. The intermediate updates from all thread blocks are now stored in device global memory can then be scanned by the host to generate the final reduced value.

Another technique found on the *NVIDIA Parallel FORALL* Blog entitled, *Optimized Filtering with Warp-Aggregated Atomics* [11] shows warp aggregation, which can reduce the number of atomic calls by up to $32X$ and achieves impressive speedups. The warp aggregation technique allows threads in a warp to perform local updates (to shared memory for example), and then assigns one thread to aggregate the result and update global memory. The following two figures from [11] show the potential impact on performance if atomics are not used sparingly and carefully.

Figure A.1 shows the precipitous drop in bandwidth proportional to number of atomics executed, or fraction of positive elements in an array, where the filter performs an atomic add operation when an element is positive. Figure A.2 illustrates a profound performance improvement when using warp-aggregation, which dramatically reduces the number of atomic calls.

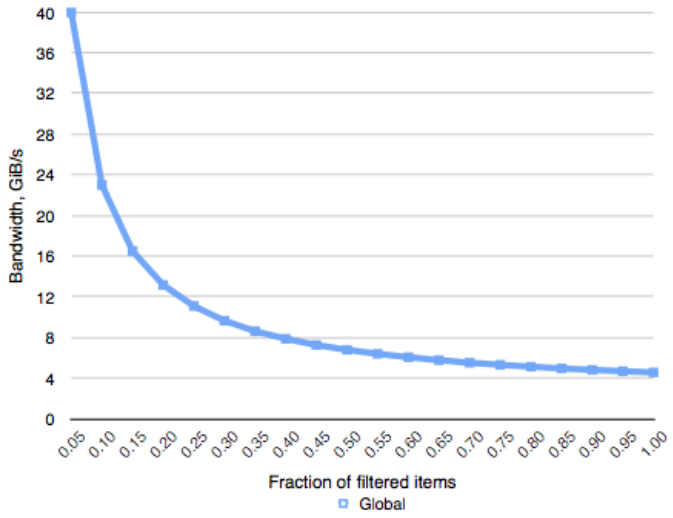


Figure A.1: Drop in bandwidth is proportional to atomics executed [11].

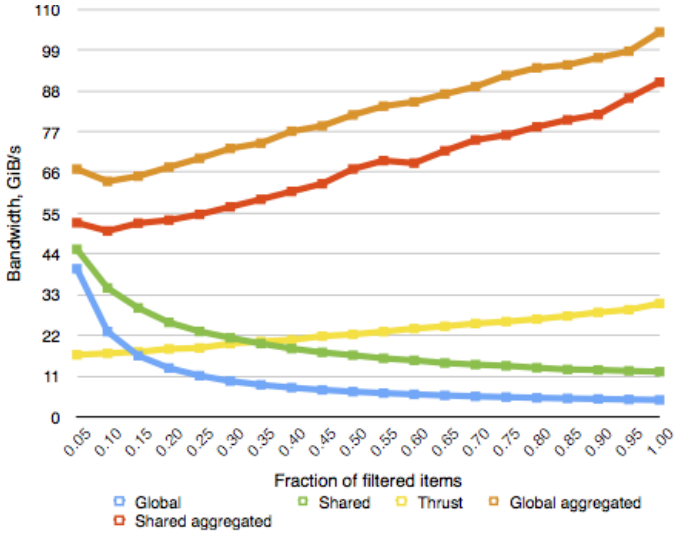


Figure A.2: Reducing the number of atomic operations significantly improves performance [11].

Bibliography

[1] Bolt c++ template library. http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library/.

[2] C++11 lambda functions. http://en.cppreference.com/w/cpp/language/lambda.

[3] Dee exascale co-design center for materials in extreme environments. http://exmatex.org.

[4] Final comment draft for the openmp 4.1 specification. http://openmp.org/mp-documents/OpenMP4.1_Comment_Draft.pdf.

[5] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[6] Improving Performance via Mini-applications. Technical Report SAND2009-5574.

[7] Intel cilk plus. https://www.cilkplus.org/.

[8] Nvidia cuda zone. https://developer.nvidia.com/cuda-zone.

[9] Occa web site. http://libocca.org.

[10] Thrust. http://docs.nvidia.com/cuda/thrust/index.html#axzz3lvtWvB8B.

[11] Andrew Adinetz. Optimized filtering with warp aggregated atomics. http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics.

[12] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.

[13] Barbara Chapman, Gabrielle Jost, and Ruud Van Der Pas. *Using OpenMP: Portable Parallel Shared Memory Programming*. The MIT Press, Cambridge, MA, 2008.

[14] Edward T. Chen. Program Complexity and Programmer Productivity. *IEEE Transactions on Software Engineering*, (3):187–194, 1978.

[15] member of stackoverflow community Dirk. Preventing destructor call after kernel call in cuda. http://stackoverflow.com/questions/19005360/preventing-destructor-call-after-kernel-call-in-cuda.

[16] H Carter Edwards and Daniel Sunderland. Kokkos Array Performance-Portable Manycore Programming Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10. ACM, 2012.

[17] H Carter Edwards, Daniel Sunderland, Vicki Porter, Chris Amsler, and Sam Mish. Manycore Performance-Portability: Kokkos Multidimensional Array Library. *Scientific Programming*, 20(2):89–114, 2012.

[18] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[19] Kenneth Franko. MiniAero: A performance portable mini-application for compressible fluid dynamics. In preparation.

[20] Kenneth J. Franko, Travis C. Fisher, Paul Lin, and Steven W. Bova. CFD for next generation hardware: Experiences with proxy applications. In *Proceedings of the 22nd AIAA Computational Fluid Dynamics Conference, June 22–26, 2015, Dallas, TX*. AIAA 2015–3053.

[21] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[22] Mark Harris. Parallel Forall Control GPU Visibility. `http://devblogs.nvidia.com/parallelforall/cuda-pro-tip-control-gpu-visibility-cuda_visible_devices`.

[23] Mark Harris. Unified memory in cuda 6. `http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6`.

[24] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Shadnaz Asgari, Victor Basili, Jeffrey K Hollingsworth, and Marvin V Zelkowitz. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 conference*, pages 35–35. IEEE, 2005.

[25] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[26] Chris Mason. Cuda visible device masking. `http://www.acceleware.com/blog/cudavisibledevices-masking-gpus`.

[27] Jeff Preshing. An-introduction-to-lock-free-programming. `http://preshing.com/20120612/an-introduction-to-lock-free-programming/`.

[28] Jeff Preshing. you-can-do-any-kind-of-atomic-read-modify-write-operation. `http://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/`.

[29] T. R. W. Scogland, J. Keasler, J. Gyllenhaal, R. Hornung, B. de Supinski, and H. Finkel. Supporting indirect data mapping in openmp. In *Proceedings of the 11th International Workshop on OpenMP (IWOMP 2015), October 1–2, 2015, Aachen, Germany.* to appear.

[30] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, Volume 1, The MPI Core.* The MIT Press, Cambridge, MA, 1998.