RAJA Overview (extracted from ASC Tri-lab Co-design Level 2 Milestone Report 2015)

Rich Hornung, Holger Jones, Jeff Keasler, Rob Neely, Adam Kunen, Olga Pearce, LLNL

Lawrence Livermore National Laboratory

LLNL-TR-677453

U.S. DEPARTMENT OF ENERGY NNSA

RAJA Overview (extracted from ASC Tri-lab Co-design Level 2 Milestone Report 2015)

Rich Hornung, Holger Jones, Jeff Keasler, Adam Kunen, Rob Neely, Olga Pearce
Lawrence Livermore 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*. 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 application developers. The goal was 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. This report contains a description of the work that was completed at LLNL 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.
LLNL-TR-677453.

Contents

Chapter 1

RAJA Chapter from Milestone Report

1.1 Introduction

Many science and engineering HPC applications, including ASC codes, have used the portable and standard MPI (Message Passing Interface) library [19] 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 [12], Thrust [8], 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.

1.1.1 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 implementation 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.

1.1.2 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 1.1: Basic RAJA loop transformation

There are several key differences to note in the RAJA loop in Listing 1.1:

- 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 be 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.

1.1.3 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 1.1 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 1.1: 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 [10] or CilkPlus [5], or may launch the loop iterations as a CUDA kernel to run on a GPU [6]. We describe how this works in more detail in Section 1.1.4.
- **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 1.1 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 1.1.5, 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 **??**
- **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 1.3.

1.1.4 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.

1.1.5 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 1.2 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 1.2: 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 in 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 1.3 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 1.3: 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 1.4. IndexSets can be be used to reorder loop iterations to to achieve "data parallel" execution without modifying the loop body code. Figure 1.4 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.

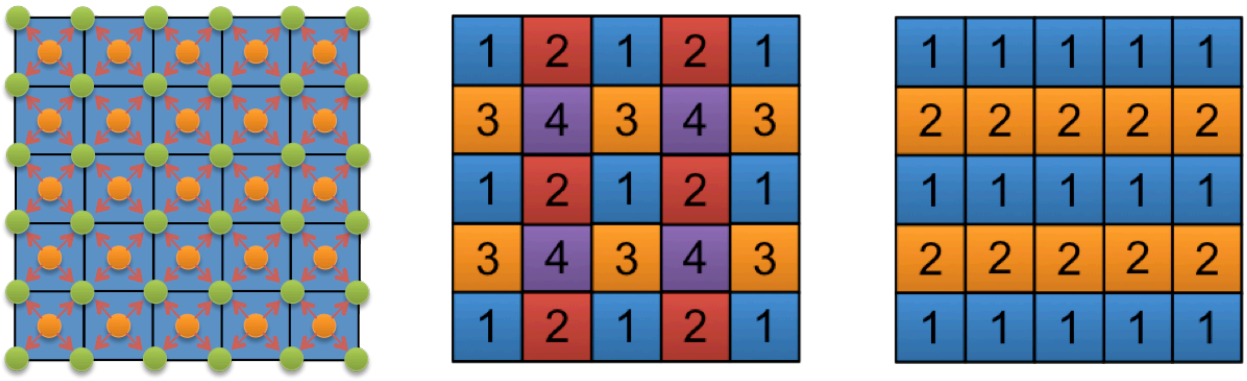

Figure 1.4: Zone-to-node sum operations (left), ordering option A (center), and ordering option B (right).

RAJA Segments can also represent arbitrary tilings of loop iterations that can be tuned and sized for specific architecture and memory configurations. Figure 1.5 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 1.5 can be transformed into the "compact" tiling in the lower part of the figure.

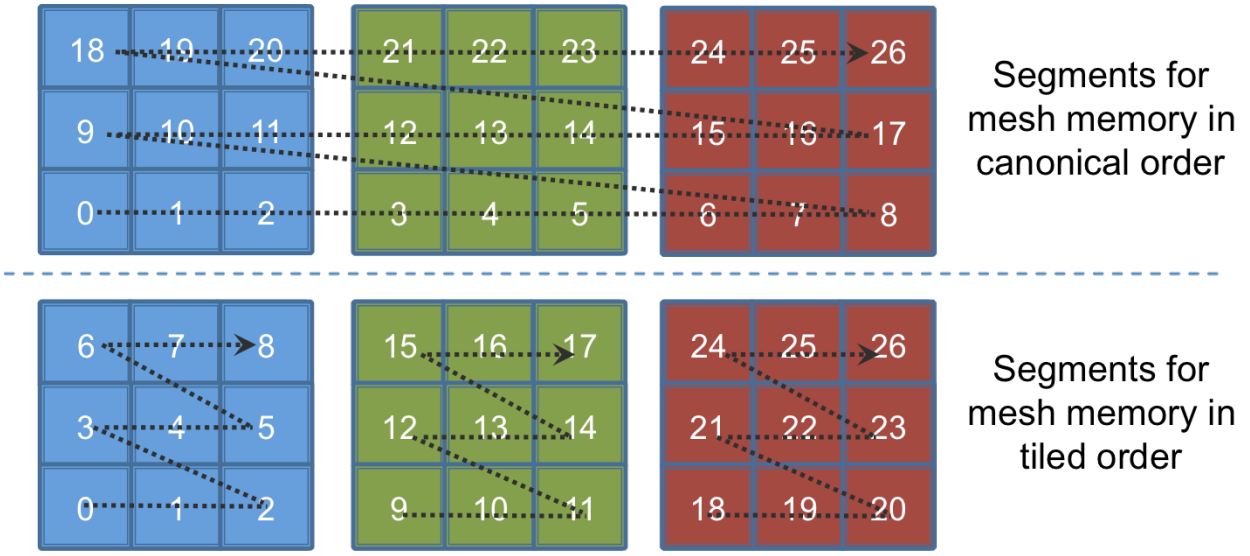

Figure 1.5: RAJA Segments represent arbitrary "tilings" of loop iterations that in turn touch data on a mesh. Colors represent Segments within an IndexSet. Numbers represent 1d array indices.

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 alloca-

tion 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. 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 1.6. 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.

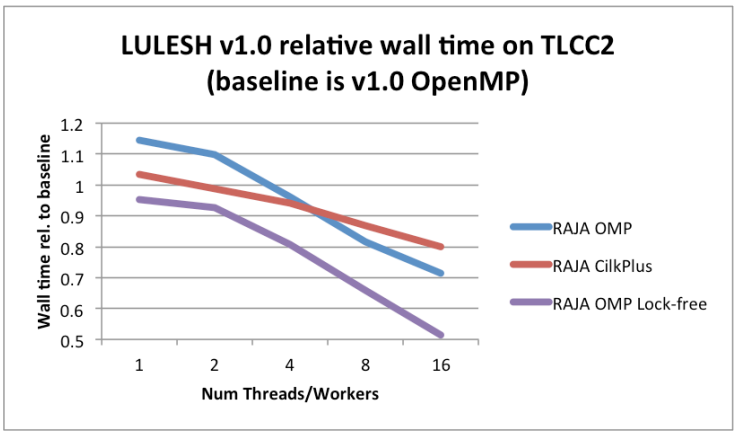

Figure 1.6: A strong-scaling runtime comparison of three different LULESH variants to the baseline (non-RAJA) OpenMP version.

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.

```
#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 1.2, 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.

1.2 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.

1.2.1 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 1.7 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 1.8 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.

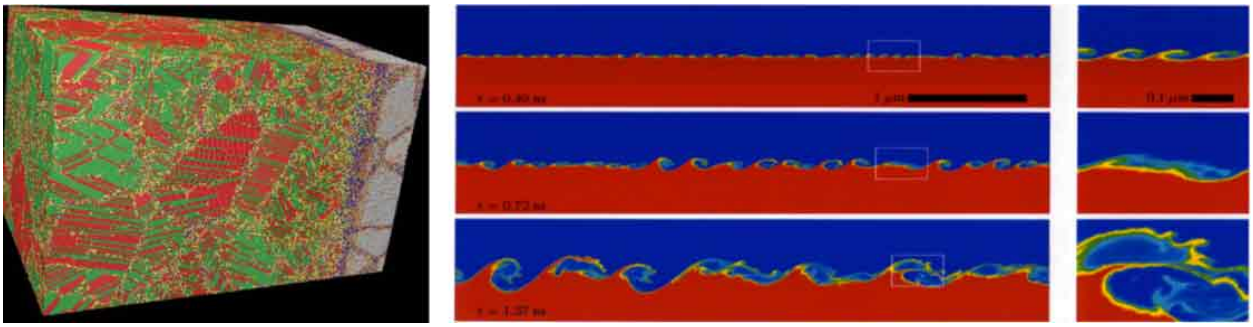

Figure 1.7: Two examples of such large-scale MD simulations, SPaSM and ddcMD, simulating solid and liquid problems.

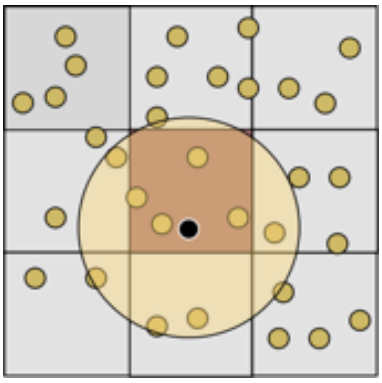

Figure 1.8: Cut-off distance in CoMD.

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 1.8 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.

1.2.2 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
        ...
```

1.2.3 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 1.9 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.

1.2.4 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.

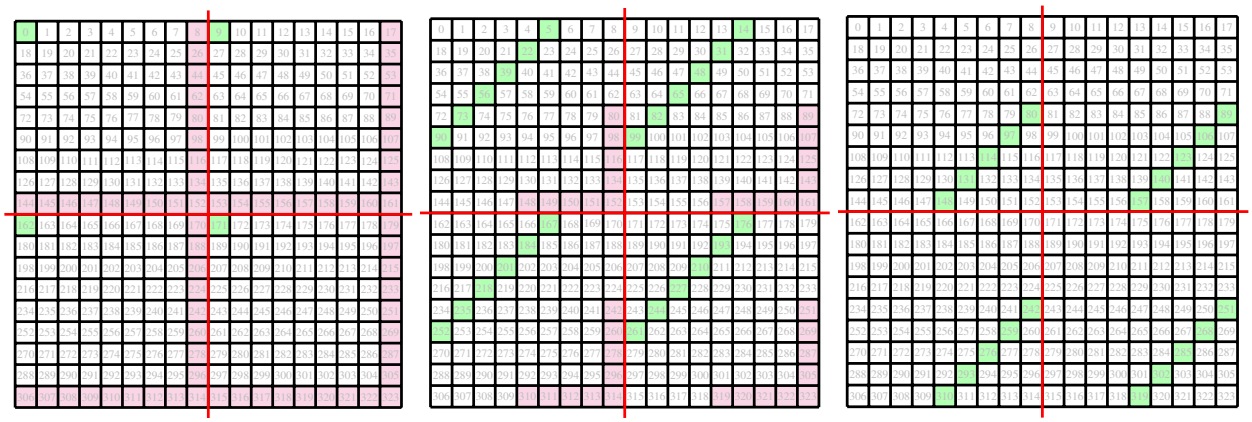

Figure 1.9: Wavefront Schedule

- **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.

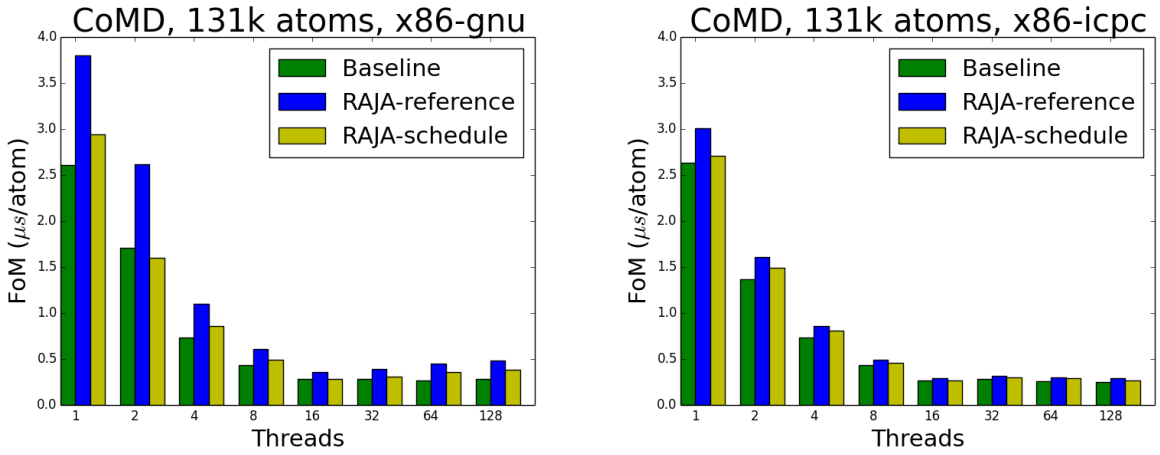

Figure 1.10: CoMD on x86, LJ force computation. Problem size 131K atoms.

Figure 1.10 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 1.11 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.

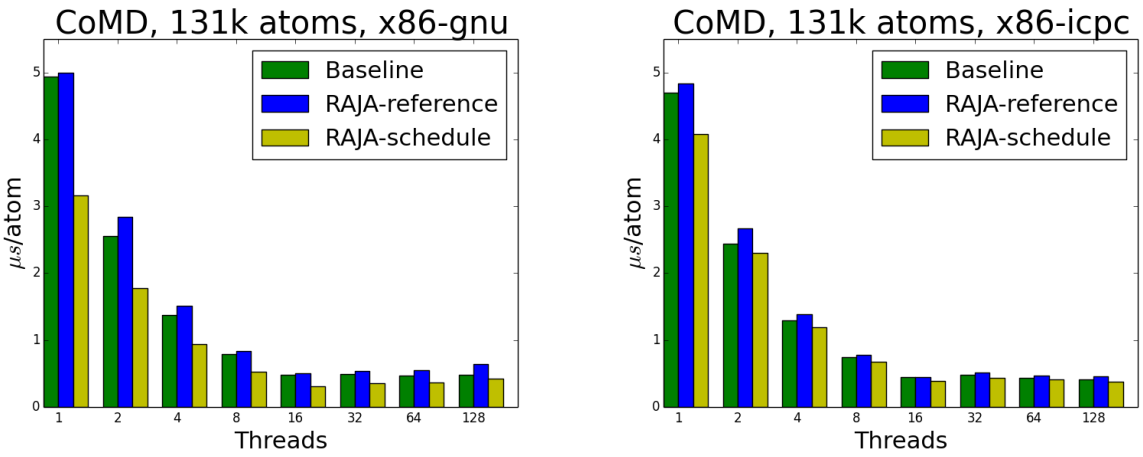

Figure 1.11: CoMD on x86, EAM force computation

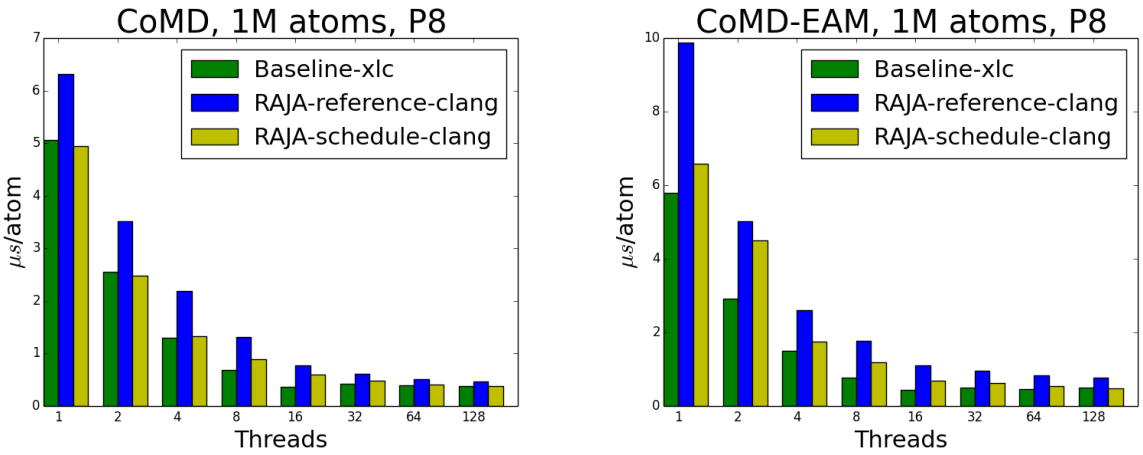

Figure 1.12: CoMD on Power8 with xlC and Clang compilers, LJ and EAM forces

Figure 1.12 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 1.13 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 1.14 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.

1.2.5 Productivity

We enabled compilation of CoMD with a C++ compiler, primarily by adding type casts. This change impacted 47 lines of code. Figure 1.15 shows the number of changes we made to the source code to convert it

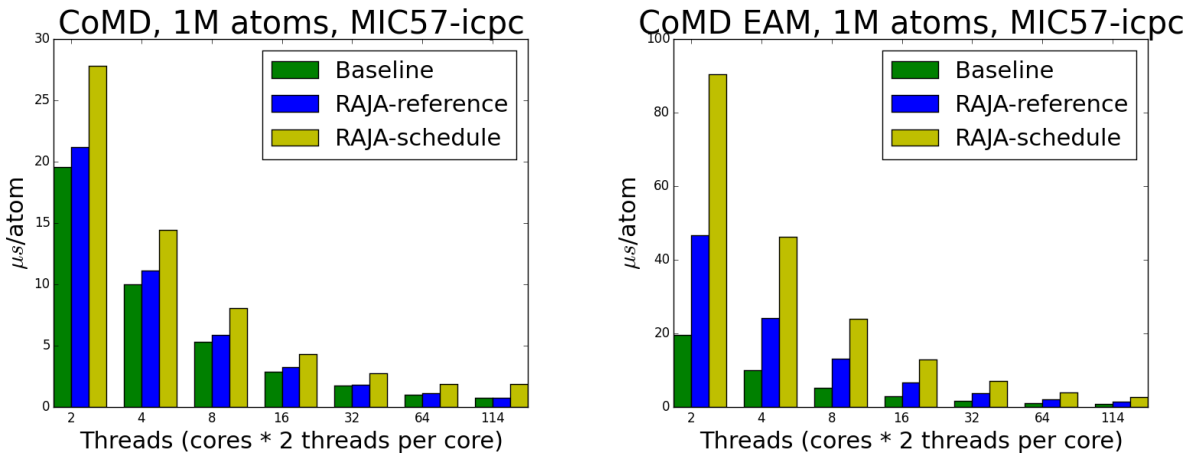

Figure 1.13: CoMD on MIC with Intel compiler, LJ and EAM forces. Problem size 1M atoms.

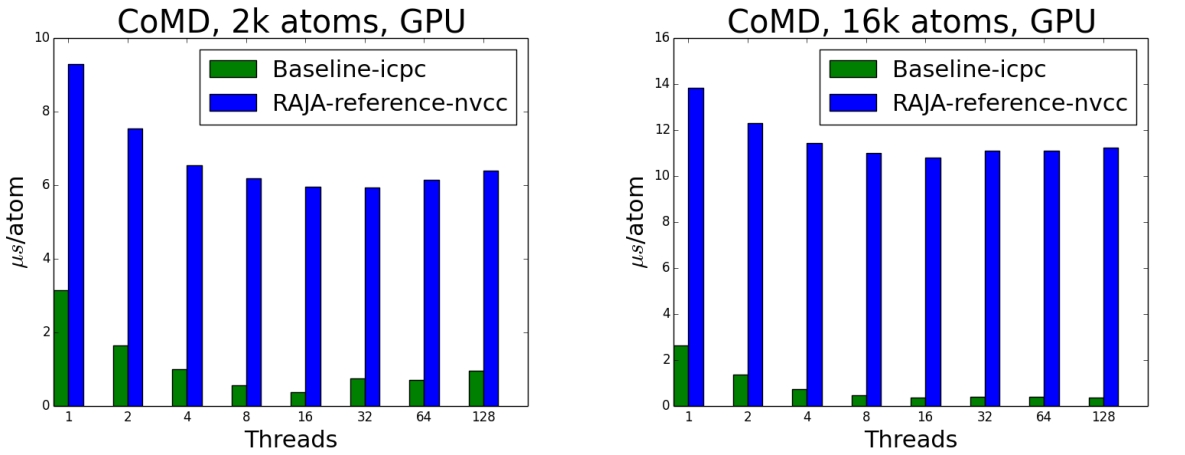

Figure 1.14: CoMD on GPU with NVIDIA compiler

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%.

1.2.6 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.

1.3 Kripke

1.3.1 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

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 1.15: 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`

1.3.2 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 [7] 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.

1.3.3 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.

1.3.4 CoE OpenMP and CUDA Variants

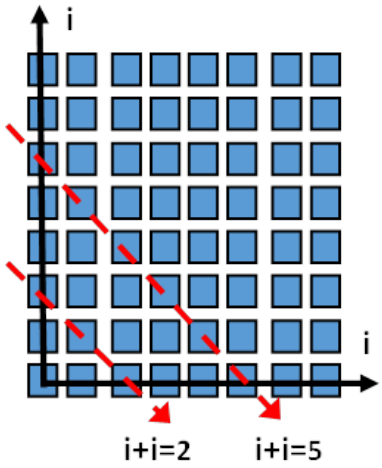

Figure 1.16: Hyperplane sweep method.

This variant is a cumulation of effort by Leopold Grinberg (IBM) and Steven Rennich (NVidia) under contract with the *Sierra Center of Excellence* at LLNL. They implemented a hyperplane sweep method, as seen in Figure 1.16, 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 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.

1.3.5 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.

1.3.6 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.

1.3.7 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 1.17 show the original Kripke results, the OCCA results (without manual optimizations),

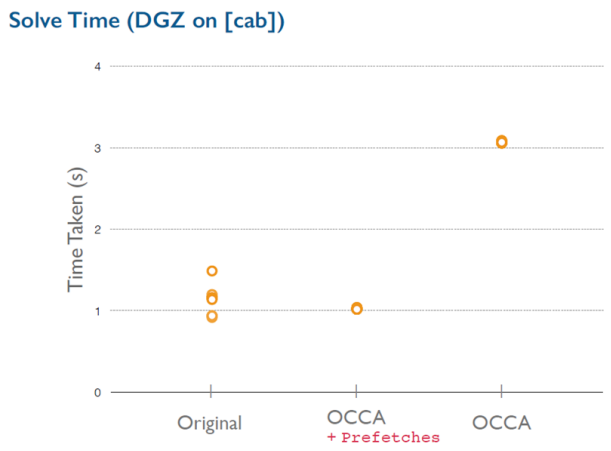

Figure 1.17: Performance of OCCA Kripke implementation, with and without manual "prefecth".

and the OCCA results with these manual "prefetch" optimizations. From these results we can directly see 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 an 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.

1.3.8 Performance Results

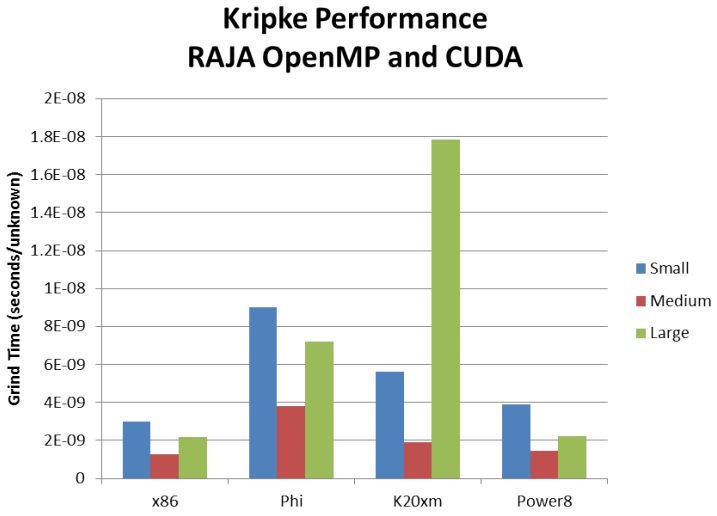

Figure 1.18: Performance of RAJA Kripke implementation across multiple architectures.

In Figure 1.18 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 1.19. 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.

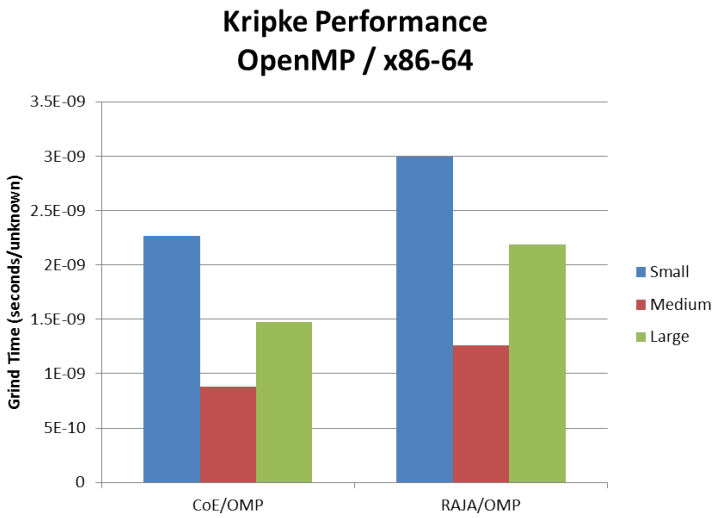

Figure 1.19: Comparison of CoE and RAJA OpenMP implementation on x86-64.

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 1.20. It's currently unclear 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 1.21 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 1.22.

1.3.9 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.

1.4 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

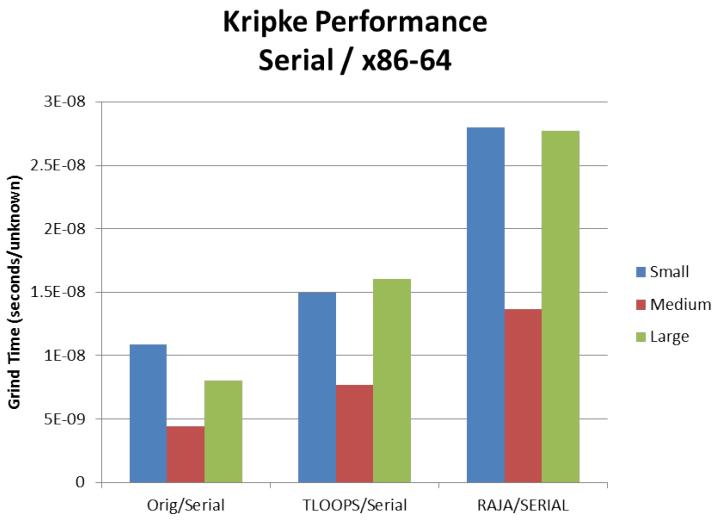

Figure 1.20: Comparison of serial performance of various Kripke variants.

the source code. Here, we have shown using CoMD (Section 1.2) and Kripke (Section 1.3) that basic RAJA transformations allow OpenMP CPU and CUDA GPU execution without further code changes. In particular, 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 1.2). 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 1.1.5, 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

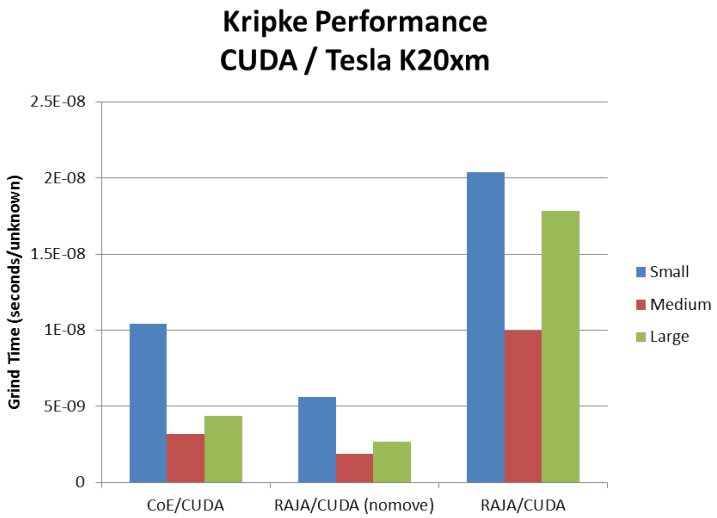

Figure 1.21: Comparison of CUDA (K20xm) performance of Kripke CoE and RAJA variants.

application source code.

1.4.1 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 [14]. 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 **??**, 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

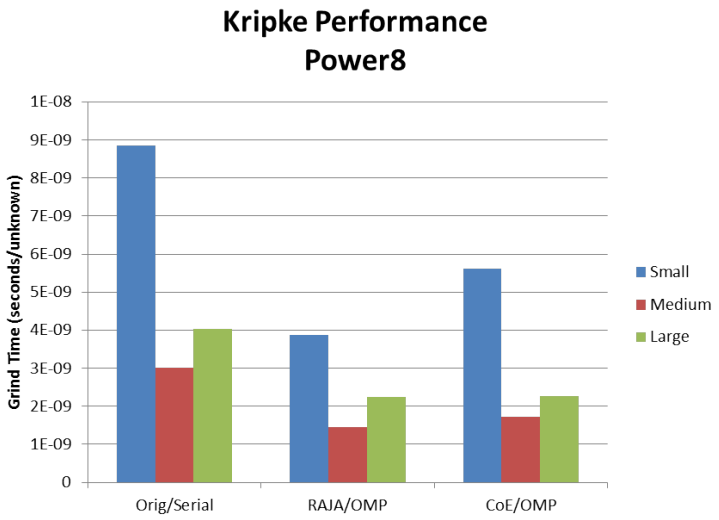

Figure 1.22: Comparison of Power8 performance of various Kripke variants.

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, 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;

   //
   // 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 cudaDeviceSynchronize() 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 [11]. 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 operates 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:
   // ...
};
```

1.4.2 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 **??** 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 2

Tri-lab L2 Milestone 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.

Chapter 3

Technical Lessons Learned and Issues Encountered

In this section, we describe some 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 interested in these programming models.

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 [18], 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.

3.1 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" [13].*

CUDA_VISIBLE_DEVICES Settings.

The environment variable `CUDA_VISIBLE_DEVICES` discussed above can be used to target a single or multiple GPU devices [15]. 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.

3.2 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 [16, 17]. 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* [9] 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 [9] show the potential impact on performance if atomics are not used sparingly and carefully.

Figure 3.1 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 3.2 illustrates a profound performance improvement when using warp-aggregation, which dramatically reduces the number of atomic calls.

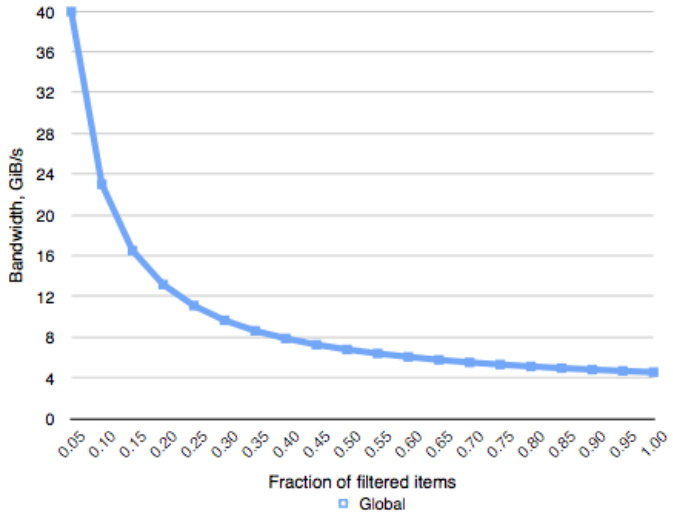

Figure 3.1: Drop in bandwidth is proportional to atomics executed [9].

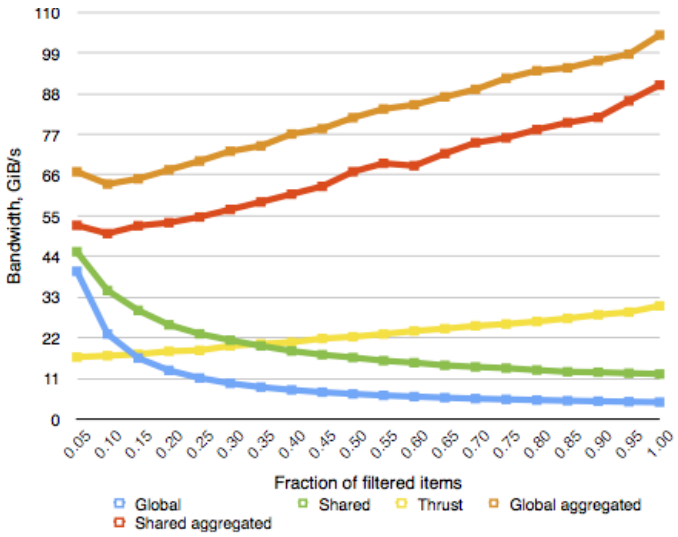

Figure 3.2: Reducing the number of atomic operations significantly improves performance [9].

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

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

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

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

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

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

[11] 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.

[12] 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.

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

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

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

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

[17] 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/.

[18] 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.

[19] 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.