RAJA: Portable Performance for Large-Scale Scientific Applications

David Alexander Beckingsale, Jason Burmark, Rich Hornung, Holger Jones , William Killian,
Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, Thomas R. W. Scogland
Lawrence Livermore National Laboratory
Livermore, CA 94550
Emails: {beckingsale1, burmark1, hornung1, holgerjones, killian4, pearce8, kunen1, robinson96, ryujin1, scogland1}@llnl.gov

***Abstract*—Modern high-performance computing systems are diverse, with hardware designs ranging from homogeneous multi-core CPUs to GPU or FPGA accelerated systems. Achieving desirable application performance often requires choosing a programming model best suited to a particular platform. For large codes used daily in production that are under continual development, architecture-specific ports are untenable. Maintainability requires single-source application code that is performance portable across a range of architectures and programming models.**

In this paper we describe RAJA, a portability layer that enables C++ applications to leverage various programming models, and thus architectures, with a single-source codebase. We describe preliminary results using RAJA in three large production codes at Lawrence Livermore National Laboratory, observing $17\times$, $13\times$ and $12\times$ speedup on GPU-only over CPU-only nodes with single-source application code in each case.

I. Introduction

In recent decades, high-performance computing (HPC) application performance has increased dramatically without requiring major code changes. Developers have been able to advance algorithms and code capabilities while architectures have remained largely homogeneous and CPU clock rates have increased. Typically, coarse-grained distributed parallelism via MPI was sufficient to achieve high performance on relevant HPC platforms. There was little benefit to exposing fine-grained on-node parallelism, such as OpenMP multithreading or tasking.

Earlier architecture paradigm shifts, such as the transition from vector machines to symmetric multiprocessors (SMPs), were separated by decades. These gaps allowed developers time to rewrite applications, if needed. Currently, the Advanced Simulation and Computing (ASC) Program in the Department of Energy (DOE) interleaves procurement of large commodity technology systems (CTS) and advanced technology systems (ATS) in 3-5 year cycles. The rapid pace of disruptive changes in ATS node architectures presently forces developers to tackle substantial performance portability challenges.

RAJA grew out of a need to support large scale production multi-physics applications in the Lawrence Livermore National Laboratory (LLNL) ASC program on new ATS platforms, such as Sierra, which place significant constraints on programming methodologies:

- **Large code bases.** Applications contain $O(100K) - O(1M)$ source lines and many numerical kernels (sometimes $O(10K)$). Often, no kernels dominate run time; thus, any portability approach must apply across most of a codebase without per-kernel code modification or tweaking.
- **Platform diversity.** Codes are routinely run on laptops (Windows, Linux, Mac OS), commodity clusters, and first-of-a-kind ATS machines, so they must run well on a diverse set of architectures at any given time.
- **Long service lives.** Codes are used daily in production for *decades* to perform critical calculations, so they must remain viable over several platform generations.
- **Continual development.** New modeling capabilities must be added throughout the lifetimes of the codes to meet programmatic needs. Thus, adopting new technologies cannot disrupt developers or users.

Given these constraints, platform-specific variants of the applications are not tenable due to limited developer resources, time, and mission priorities. RAJA [1] is a C++ abstraction layer, developed at LLNL, that enables performance portability within the application constraints described above. The main goal of RAJA is to enable *manageable performance portability*, and avoid committing to fixed software technology choices based on current hardware designs, since there is no clear "best choice" for all architectures presently. Applications may choose to adopt RAJA incrementally at any scale, from a single kernel to an entire codebase. Also, platform-specific data accesses and parallel execution concerns can be insulated from source code that most developers work with daily. Finally, portability must be built into a code and maintained over its lifetime without major disruption to developer and user productivity.

RAJA targets loop-level parallelism for C++ applications by relying solely on standard C++11 language features for its external interface and common programming model extensions, such as OpenMP and CUDA, for its implementation. RAJA developers include: computer science researchers and application developers working closely together and with compiler, system software, and tool vendors. Thus, its requirements and features are determined directly from needs of production applications.

It is important to note that, from its start, RAJA has been designed to be agnostic to the strategy employed by users to manage memory, in particular to make simulation data available on offload devices. That said, there exist other

libraries we have developed to complement RAJA and simplify the management of hierarchical memory systems for users. These tools are discussed in Section III-E.

This paper makes the following specific contributions:

- It introduces RAJA, an open source C++ standard-based portability layer;
- It presents an evaluation of RAJA in both benchmarks and initial use in production applications.

In Section II, we discuss different approaches to portable application development and motivate RAJA. Then, we describe RAJA features in Section III. In Section IV, we present RAJA results from benchmarks and preliminary evaluation in three production applications. Lastly, we discuss RAJA limitations, future work and conclusions in Section VI and Section V.

II. Background and Motivation

Portability can be achieved at a programming model or application level. A portable programming model eliminates the need to implement a custom solution, but leaves applications at the mercy of vendor implementations of chosen models. Moreover, different models have unique programming characteristics and are not easily interchangeable. Yet, interchangeability is necessary to assess performance and manage portability. Application level techniques require significant effort, but reduce reliance on a specific implementation. In this section, we describe both programming model and application techniques.

A. Programming Models

There has been a clear trend in HPC toward node-level parallel programming models that extend programming languages, like C and C++, via compiler directives and library routines. OpenMP, OpenACC, CUDA, and other models can support multithreading and/or processor heterogeneity where CPUs and accelerators are combined. Compiler vendors support these models well, making them viable for production codes. However, no existing model is a clear best choice for all architectures and applications.

Directive offload models like OpenMP [2], OpenACC [3], and HMPP [4] use directives to extend base languages such as Fortran, C, and C++ with heterogeneous offload capabilities. OpenMP is best known for providing portable multithreading on shared-memory multicore systems. In simplest usage, a loop can be parallelized by adding an `omp parallel for` directive. More recent versions of OpenMP, as well as OpenACC and HMPP, have support for offloading regions of code and loops to (potentially) non-CPU devices. Support for accelerators in OpenMP continues to improve, with many features added over the past few years. While compiler support is improving, none of these models has a well-established base of implementations such that they can be viewed as portable for production codes today.

Block and grid models such as CUDA [5], OpenCL [6] and HCC [7] are usually built with their own compilers somewhat independent of a base language. CUDA and HCC take a single source approach, using the nvcc and HCC-clang compilers respectively, while OpenCL expects all host code to be compiled with a normal non-OpenCL toolchain and provides architecture-specific compilers for its kernel languages. They each provide a low-level interface to a relatively specific batch-processing device that models GPUs well. GPU threads are grouped as a grid of thread blocks, which are mapped to GPU multiprocessors. These models often require programmers to transform code significantly. With the exception of OpenCL, which requires even more code modification due to its lack of single source support, they are non-portable. The SYCL [8] extension to OpenCL takes a hybrid approach that provides a C++-like veneer on top of OpenCL, which should provide single source portability, but it is still maturing.

B. Application Level

Application level techniques leverage programming languages and libraries to provide portability between programming models and architectures. They are restricted to operate within the rules of the base language, but depend less on a specific implementation since they only require a compiler for the base language to produce working code.

1) Multiple code versions: One approach to portable code is to write a version of a code for each target platform and switch between implementations. This gives developers complete control to tune for each platform and some applications are small enough that this route is manageable. However, for the production codes we discuss in this paper, this is not a practical solution. For example, many variants of LULESH, a proxy for ALE3D (Section IV-C), have been developed [9]. But, there is no practical path to transition all of ALE3D to the approach taken in any one of these exercises.

2) Macros: A simple technique to write portable application code is to use preprocessor macros for architecture-dependent parts of a code. At compile time, macros are replaced with appropriate architecture-specific code. Such a model can achieve high portability and can eliminate runtime overhead compared associated with other abstraction approaches. However, macros can obfuscate code for debuggers and compiler diagnostics.

3) Existing Libraries: Libraries are another potential portability alternative to programming models. Notably the C++17 standard library offers parallel algorithms designed to be portable across architectures. While they are portable, memory handling in distributed memory nodes is ill-defined, and support for custom algorithms is limited. The Kokkos library [10] provides portability across memory systems and compute platforms. It has been shown to provide good performance and supports a wide range of programming model back-ends. While similar in spirit to RAJA, Kokkos constrains users to specific algorithm and memory access patterns and may be considered more disruptive for some users. In contrast, RAJA focuses on ease of expression and reducing impact on application code. Agency [11] is another C++ library with similar portability goals, focusing on describing execution by assigning work to groups of execution agents. However, it is an experimental effort and its long-term support is unclear.

C. Why RAJA?

RAJA began about 7 years ago to explore the potential of C++ abstractions to enable performance portability in LLNL ASC applications, which are written mostly in C++. We realized the potential advantages of an abstraction layer that could insulate large application codes from architectural differences and performance tuning exercises. The fact that these applications also needed to prepare for Sierra [12], a 125 petaflop ATS machine – the first production GPU-enabled computing platform at LLNL – helped focus RAJA development and assessment efforts. We hoped that we could develop the proper software tools that would allow us to port to GPU systems while maintaining high performance on other production systems with designs such as commodity CPU and IBM BlueGene architectures.

Without an abstraction layer, applications would have needed to use CUDA or OpenMP directly to target GPUs. CUDA, while a mature technology, was untenable due to substantial code rewriting, maintenance burden, and lack of portability. OpenMP GPU support, both features and compilers, were portable but nascent and not production ready at the time. The hope was that a C++ abstraction layer would enable access to different programming models and would not be overly disruptive to code team productivity. Existing C++ abstraction layers were viewed as insufficiently flexible, either requiring too much code rewriting, too many fundamental algorithmic or data structure changes, or lacking adequate support for incremental adoption.

RAJA has matured into a powerful set of general performance portable capabilities (see Section III) that have proven to be a boon to porting LLNL codes to Sierra. Also, as it has been adopted by different codes, important features have been developed that are unique to RAJA. These include: support for critical looping patterns, such as fixed stride or arbitrary indexing (i.e., indirection) within a single kernel, portable reduction types that do not require reduction-specific loop execution mechanisms, integration with a variety of heterogeneous memory space management approaches that are decoupled from RAJA, and support for complex nested loop patterns with facilities for generating multiple nesting orders and variations for performance tuning and specialization.

III. The RAJA Portability Layer

RAJA provides C++ abstractions that enable users to make their code portable with manageable source code changes. RAJA does this by encapsulating loop and region execution and keeping the application description of the computation within loops largely unchanged. For example, consider the following C/C++-style *daxpy* loop:

```
for (int i = 0; i < N; ++i)
{
  a[i] += c * b[i];
}
```

The equivalent RAJA version keeps the loop body the same[1]:

```
RangeSegment seg (0, N);
forall<loop_exec> (seg, [=] (int i) {
  a[i] += c * b[i];
});
```

[1]The RAJA namespace is omitted in all examples for readability

The RAJA abstraction accepts a C++ functor, which is usually produced by using the C++11 lambda facility by the user, which in this paper we refer to as a `lambda kernel body`. In real applications, kernel bodies are much larger than the one in the simple example above. Modifying the loop header only and leaving the kernel body unmodified means that most lines of code in a thread-safe application do not change as it is converted to use RAJA.

The daxpy example above shows three main RAJA concepts: 1) execution policies (`loop_exec`), 2) iteration spaces (`seg`), and 3) execution templates (`forall`). The following subsections describe these concepts.

A. Concepts

1) Execution Policy: An execution policy is a C++ type that specifies how a loop kernel will run. A policy can specify which programming model back-end to use (see Section III-B), or it may be a complex *composition* of simpler policies. Nesting or aggregation of execution policies gives users flexibility to easily access powerful features of parallel programming models, such OpenMP and CUDA. RAJA execution policies also encode traits that help drive code generation via template specialization: each policy defines a *policy type*, an *execution template*, a *launch category*, and an *execution platform*.

A *policy type* refers to a known *execution back-end*; e.g., `sequential`, `OpenMP`, `CUDA`, etc. An *execution template* type validates, at compile time, whether an execution policy is used within a correct context; e.g., a user should not use a loop execution policy within a `scan` or `reduction` type. A *launch category* specifies how code will be launched (synchronously, asynchronously, or undefined). Finally, an *execution platform* type describes where the loop will execute, and can be used to determine which memory space will be accessed. Policy types specialize RAJA execution templates, and RAJA provides a variety of execution policies; users can also define their own policies to customize RAJA.

2) Iteration Space: A RAJA iteration space defines a set of loop indices for a kernel. Index access operations are guaranteed to be constant in time and portable across single-core, multicore, and GPU offload execution. An object that models the *random access container* concept can be used as an iteration space. Indeed, RAJA iteration space containers and generators are similar to C++ standard library containers and conform to that concept.

RAJA defines two categories of iteration spaces: *segments* and *index sets*. A segment defines a set of loop indices to be executed as a unit, and an index set is a container of arbitrary segments to be run with a single kernel. Segments directly map to simple for-loop patterns and RAJA provides three segment types: `RangeSegment` which defines a stride-one index range, `RangeStrideSegment` which defines a constant-stride index range, and `ListSegment` which is an arbitrary set of indices, akin to an indirection array. Any RAJA segment type can be used with any loop body.

Index sets provide power and flexibility by enabling execution of a collection of segments, each of which can be operated on independently. Index sets require a two-level execution policy, one for iterating over segments and one for executing the segments. This is a specialization designed to help optimize performance and expressibility in sparse iteration spaces, especially in cases where sub-ranges contain contiguous indices. Expressing such sub-ranges as contiguous segments allows RAJA to vectorize portions of kernel execution while allowing irregular indexing patterns on the rest.

3) Execution Template: RAJA execution templates define operations performed on a lambda kernel body based on execution policy specialization and an iteration space object. RAJA provides several execution templates. Most common is `forall`, which follows the *parallel-for idiom* and maps directly to a traditional C-style for-loop, or a C++11 range based for-loop.

Some kernels have more complex structure, such as a loop nest, that do not map well to the `forall` construct. RAJA provides more complex execution templates via its `kernel` interface. A RAJA kernel template enables composition of multiple policies and iteration spaces to define a *kernel* structure within the C++ type system. RAJA `forall` and `kernel` policies are discussed in Subsection III-C.

Another set of execution templates that RAJA provides supports portable `scan` operations. Scans are used commonly in parallel work assignment, sorting, comparison, and stream compaction as a method to parallelize otherwise serial work. RAJA provides four scan types: inclusive, exclusive, inclusive in-place, and exclusive in-place. A RAJA `scan` template requires an *execution policy*, an input *iterable* or begin/end *iterators*, and an optional *operator*. RAJA has predefined operators for all C++ standard library function objects such as `plus` and `multiply` but can also take an arbitrarily complex user-defined operator to apply in a prefix scan pattern.

RAJA provides two other classes of templates that do not implement loop traversals. They provide users the ability to perform reduction and atomic operations specialized on policies similar to execution templates. RAJA supplies five different reduction operations, each accepting a reduction policy and an underlying storage type:

- `ReduceSum`: sum of all values
- `ReduceMax`: maximum value
- `ReduceMin`: minimum value
- `ReduceMaxLoc`: max value, plus index of max value
- `ReduceMinLoc`: min value, plus index of min value

A *reduction policy* must be compatible with the execution policy given to the `forall` or `kernel` construct in which the reduction is used (see example below). For instance, one cannot use an OpenMP reduction policy in a CUDA execution context. Since reductions are independent of the execution template, arbitrary other computation can be done in the same loop, along with any number of reductions of arbitrary types. [2]

[2]A RAJA reduction operation can only apply its reduction operation within a lambda expression. Setting a "local" value to some arbitrary expression result is not supported.

```
ReduceMaxLoc<RedPol, double> max(0, -1);
forall<ExecPol> (iSpace, [=] (int i) {
  max.maxloc(a[i], i);
});
double val = max.get();
int loc = max.getLoc();
```

RAJA portable atomic operations appear similar to the interface provided by CUDA. Like reductions, atomic policies depend on the execution context in which they are used. RAJA also provides an "automatic" atomic policy that deduces the correct atomic policy in GPU/CUDA, OpenMP, and sequential CPU execution contexts. An example of RAJA atomic usage is:

```
double* sum = new double[1];  *pi = 0.0;
forall<ExecPol> (iSpace, [=] (int i) {
  atomicAdd<auto_atomic>(sum, a[i]);
});
double res = *sum;
```

Lastly, RAJA provides atomic references and atomic views over containers that are compatible with arbitrary memory locations and work with all atomic policies. An example of usage is:

```
int v = 1;
AtomicRef<int, omp_atomic> sum(&v);
++sum;
sum += 5;
```

The RAJA `AtomicRef` interface is consistent with the C++20 `std::atomic_ref` definition.

B. Supported Back-ends

As of release v0.9.0, RAJA supports execution policies for the following back-ends:

- `sequential`: forced sequential execution;
- `simd`: forced SIMD optimizations;
- `loop`: allows compiler to optimize according to its heuristics;
- `openmp`: OpenMP CPU multithreading;
- `openmp_target`: OpenMP with target offload;
- `cuda`: NVIDIA CUDA execution; and,
- `tbb`: Intel Threading Building Blocks.

Currently, only `sequential`, `loop`, `openmp`, and `cuda` support all RAJA features described in the previous section. Other back-ends are works-in-progress and support a subset of features. For example, reductions are not guaranteed to be correct when using `simd`. The Intel `TBB` back-end lacks `kernel` execution policy specializations. Support for reductions and `kernel` policies in OpenMP with target offload is under development as is a full-featured back-end for AMD HIP.

C. Policy Implementation

A RAJA execution policy ties a loop execution to a particular programming model. For example, `loop_exec` uses a standard for-loop. The specialization of the `forall` traversal template looks like this:

```
auto begin = std::begin(iter);
auto end = std::end(iter);
auto dist = std::distance(begin, end);
for (decltype(dist) i = 0; i < dist; ++i) {
```

```
  loop_body(begin[i]);
}
```

A for-loop traverses the iterates in a segment. For each iterate, the lambda function representing the loop body is called with the appropriate loop index.

RAJA `seq_exec` and `simd_exec` are similar but use for-loops decorated with compiler-specific or OpenMP pragmas that enforce strictly sequential or SIMD execution, respectively. The CUDA back-end is more complex since it must launch a GPU kernel. This kernel takes a lambda as an argument and calls it with the appropriate indices on the GPU.

The OpenMP specializations parallelize for-loops through the use of OpenMP pragmas. The `omp_parallel_for_exec` policy is implemented in two steps. Internally, RAJA will first use the `omp_parallel_exec` specialization to establish a parallel region: then it will call the `omp_for_exec` specialization to distribute iterates to threads in the parallel region.

Encapsulating more complex loop structures is done through RAJA kernel policies. Kernel execution policies allow arbitrary nesting of wrapper and loop policies. A kernel execution policy can support arbitrary loop nests and specialized policies that transform loops via tiling, collapse, fusion, etc. Multiple lambdas are supported, for example, when data initialization is required and a simple loop nest would not be functionally equivalent. Kernel policies require users to indicate where a lambda should be inserted in the compiler-generated code. The following shows two examples:

```
// a policy for a double loop nest with OpenMP on inner loop.
KernelPolicy<
  For<1, loop_exec,
    For<0, omp_parallel_for_exec,
      Lambda<0>
    >
  >
>

// a policy for a collapsed double loop nest with OpenMP
KernelPolicy<
  Collapse<omp_parallel_collapse_exec, ArgList<1, 0>,
    Lambda<0>
  >
>
```

The primary difference for RAJA users between execution concepts is that `forall` uses a single iteration space object, while `kernel` supports multiple *iterable* spaces passed in a tuple type. The `Lambda` tag indicates where a lambda is to be inserted in the code; since multiple lambdas are supported, this allows irregularly nested loops.

In the following example, we show a RAJA kernel for a matrix multiplication using RAJA multi-dimensional data *views* (see Section III-D):

```
kernel<EXEC_POL>(make_tuple(col_range, row_range),
[=] RAJA_DEVICE(int col, int row) {
  double dot = 0.0;
  for(int k=0; k<N; ++k)
  {
    dot += Aview(row,k) * Bview(k,col);
  }
  Cview(row,col) = dot;
});
```

This kernel generates the equivalent of a two-level loop nest, one for rows and one for columns.

D. Views and Data Layouts

RAJA provides a *View* abstraction that wraps a pointer to a block of memory to simplify multi-dimensional indexing by hiding integral offset computations from user code. RAJA also has optional strongly-typed indices with `TypedView` so that users receive information about incorrect index usage at compile-time. The RAJA `AtomicViewWrapper` defines a view where all access and updates are performed atomically.

RAJA provides various *layout* types to specify at `View` creation the multi-dimensional access pattern for a block of memory. Examples include a common zero-based layout where the last index has stride-one data access, a non-zero-based (offset) index layout where the last index has stride-one data access, and a permuted layout that can change the order of the index stridings, allowing the memory ordering to be modified at compile-time.

E. Memory Model

From its start, RAJA has been designed to be agnostic to the strategy employed by users to manage memory, in particular to make simulation data available on offload devices. The main reason for this is the overarching goal to minimize invasiveness. While some codes may wish to explicitly manage memory, others may want to rely on unified memory or other mechanisms. Notably, RAJA views and layouts work with arbitrary pointers, but do not themselves manage memory placement.

That said, there are associated libraries, Umpire [13] and CHAI [14], that applications use to manage data in heterogeneous memory spaces. Umpire, which is used within CHAI and can also be used standalone, provides a portable memory management API that allows users to access a variety of vendor and open source memory tools. Umpire provides utilities for allocation/deallocation, transfers and other operations, and introspection. CHAI invokes data copies between CPU and GPU memory spaces based on hooks into RAJA. CHAI complements RAJA by providing a *managed array* abstraction that moves data to an execution memory space, as needed when a kernel is launched, based on a RAJA execution policy. How this appears in application code is illustrated below:

```
chai::ManagedArray<double> my_data(100);
// data transferred implicitly to GPU
forall<cuda_exec>(0, 100, [=] RAJA_DEVICE (int i) {
  my_data[i] = i * 3.14;
});

// copy data back for host use
double* my_data_ptr = (double*) my_data;
```

RAJA informs CHAI where a kernel will execute based on an execution policy, ensuring data resides in the correct memory space for kernel execution. CHAI allows incremental porting of codes to RAJA with the appearance of unified memory usage while providing effectively manual data management. CHAI allows developers to explore manual data

Platform	Nodes	CPUs per node	Accelerators per node
Sequoia	98,304	IBM BlueGene/Q (16 cores)	N/A
Zin	2,916	Intel Sandy Bridge (16 cores)	N/A
Jade	1,302	Intel Broadwell (36 cores)	N/A
HasGPU	20	Intel Haswell (20 cores)	4 NVIDIA K80 GPU
Manta	36	2 IBM Power8+ (20 cores)	4 NVIDIA P100 GPU
Sierra	4,320	2 IBM Power9 (44 cores)	4 NVIDIA V100 GPU
Cori (CPU)	2,388	Intel Haswell (32 cores)	N/A
Cori (KNL)	9,668	Intel Knights Landing (68 cores)	N/A

TABLE I
SYSTEMS USED IN RAJA CASE STUDIES.

Variant	Description
Sequential	Reference sequential impl.
RAJA Sequential	RAJA sequential impl.
OpenMP	Reference OpenMP CPU multithreading
RAJA OpenMP	RAJA OpenMP CPU multithreading
OpenMP-target	Reference OpenMP 4.5 GPU offload
RAJA OpenMP-target	RAJA OpenMP 4.5 GPU offload
CUDA	Reference CUDA kernel impl.
RAJA CUDA	RAJA CUDA impl.

TABLE II
KERNEL VARIANTS CURRENTLY IN THE RAJA PERFORMANCE SUITE.

transfers or a vendor-specific solution like NVIDIA Unified Memory (UM), without changing application source code.

The applications discussed in Section IV apply Umpire, CHAI, and other techniques in various ways and combinations to manage memory. We will describe how memory is managed in each case

F. Application Considerations

Applications that use RAJA can easily change how and where compute kernels run by switching execution policies. For rapid prototyping and portability, we promote the pattern of defining execution policies in header files. Then, an application can be easily recompiled to run on different platforms. Also, similar loop structures may share execution policies across a large codebase. RAJA promotes the notion of parameterization of loop classes so that classes of loops can be tuned rather than individual kernels.

RAJA provides abstractions to access and operate on data in a platform-independent way although RAJA does not provide a memory management model. Applications can use native memory management techniques, or other abstraction layers to ensure data is available in a RAJA kernel. In Section IV, we discuss how three applications address data management on heterogeneous architectures.

IV. RAJA USE CASE STUDIES

In 2014, LLNL ASC applications started to explore the impact of RAJA on source code and performance. Since then, several production codes have adopted RAJA to prepare for the Sierra system (Section II-C) with the hope that RAJA will also be a long-term performance portability solution. In this section, we describe the integration of RAJA into three large application codes and report performance on various computing platforms. We focus on comparing performance between CPU-only and heterogeneous GPU-based systems. The node architectures are summarized in Table I.

Before we begin discussing applications, we note several important points. Each code manages heterogeneous memory systems differently. Second, MPI usage for inter-node parallelism is unchanged. Third, each team ensures that its code remains correct via extensive regression test suites. Fourth, each code uses RAJA as a single-source model so there are no GPU-enabled version without RAJA to compare to. Fifth, each code team verifies that RAJA had no negative performance impact by ensuring that base case CPU run times of test suites do not degrade. One application tracks performance for each change committed to its source repository; others do so regularly, but less frequently.

The last two points imply that comparing RAJA GPU performance against a native CUDA or OpenMP implementation for a full application is difficult. However, we monitor the performance of RAJA kernels compared to non-RAJA variants using a suite described in the next section.

A. RAJA Performance Suite

As mentioned earlier, applications use RAJA as a single-source model. So, it is necessary to compare RAJA performance against native implementation performance another way. The RAJA Performance Suite [15] contains a diverse set of kernels to assess performance of RAJA features with different programming models and compilers. Kernels come from stream benchmarks, LCALS [16], [17] and Polybench [18] Suites, and real applications. Each kernel appears in RAJA and non-RAJA variants for different programming models, summarized in Table II.

a) Implementation: The reference Sequential variant for each kernel in the Performance Suite uses C-style for-loops. All other RAJA and non-RAJA variants (Table II) are based on that. All variants of each kernel share the same CPU data allocation/deallocation and initialization routines. GPU variants used manual CUDA or OpenMP API calls to copy data between host and device. Data allocation/deallocation, initialization and necessary transfers are not included in execution timings.

Each kernel has a default size (number of loop iterations) and number of times it is run to generate execution timings. The Suite is configurable at run time via command line arguments to select: kernel sizes, number of samples, subsets of kernels or variants to run, etc.

After the Suite is run, CSV-formatted text files are generated that report execution timings, speedup of each RAJA variant with respect to its reference variant, Figure of Merit (run time deviation between RAJA and reference variant), and result checksums (to verify kernel variants run correctly).

b) Results: We present results for the RAJA Performance Suite on all HPC architectures described in Table I. All variants supported on each platform were run, except for OpenMP-target which is incomplete in RAJA.

Figure 1 shows performance differences between RAJA and non-RAJA reference variants for Sequential, OpenMP, and CUDA as histograms aggregated over all Suite kernels. The histogram bins are limited to 100% faster or slower, and kernels

falling outside this range are placed in the first or last bin respectively. The clustering around 0% shows that most kernels perform similarly for RAJA and reference Sequential and CUDA variants. OpenMP shows a larger variance and more RAJA kernels being slower than reference. This is a topic of investigations and we plan to develop variants of kernels in the Suite that use C++ lambda expressions in loops that use OpenMP pragmas directly without RAJA to see if we can gain further insight. Currently, we believe the main issues are due to optimization difficulties that C++ compilers have when OpenMP pragmas are inserted within C++ template abstractions because many RAJA kernels that are noticeably slower or faster than reference are exercising identical RAJA constructs. Nevertheless, for 55% of the cases overall, RAJA performance is within 10% of reference variants, and in 69 out of 140 kernel and platform pairs (49%), performance of at least one RAJA variant is better than the reference. It is important to emphasize that the Suite isolates performance differences between RAJA and reference implementations for individual kernels. Real applications that are able to run with and without RAJA for sequential execution show nearly identical aggregate run times for both versions.

B. Ares

Ares is a massively parallel, multi-dimensional, multi-physics code at LLNL used a wide range of calculations and problem sizes on serial resources up to millions of processors to model high-explosive, inertial confinement fusion, and hydrodynamics experiments [19], [20]. Ares has over 700k lines of C/C++ code, uses MPI for distributed memory parallelism, and RAJA for fine-grained parallelism on CPUs and GPUs. To manage host-device data transfers on Sierra, it uses a combination of manual operations and unified memory. Physics capabilities ported to RAJA currently include: Lagrange and Arbitrary Lagrangian Eulerian (ALE) hydrodynamics, equations of state, grey radiation diffusion, material strength, thermonuclear burn, and sliding contact surfaces.

We first consider a Rayleigh-Taylor mixing layer in a convergent geometry in Figure 2. The full 3D simulation (4π) has 191.1 million zones, and converges in 14,500 time cycles. Figure 3 shows that RAJA-enabled Ares strong-scales well on CPU-only architectures. Because the radiation-hydrodynamics component of Ares is largely bandwidth bound, we compare the run times on Jade, Manta, and Sierra (system details in Table I) to the aggregate bandwidth of the system run configurations in Figure 4. Ares performance on configurations with similar aggregate bandwidth (e.g., 4,608 Intel Broadwell CPU or 32 NVIDIA V100 GPUs) is indeed similar. Runtime and across-system speedup is listed in Table III, illustrating that using GPUs via RAJA CUDA back-end results in 11×-13× speedup.

Next, we demonstrate a 191.1 million zone problem that uses ALE hydrodynamics, dynamic species, grey radiation diffusion, and thermonuclear burn. Runtime and speedup are shown in Table IV; the problem can not run on 8 nodes of Manta due to memory constraints. While we have only recently been running such multi-physics problems on GPUs, we are seeing speedups of over 9×, node-for-node, on GPU systems vs. commodity CPU clusters.

	Cores	Nodes	Agg.B/W (GB/sec)	Runtime (min)	Speedup
Jade	576	16	2,080	909	1
	1,152	32	4,160	454	2
	2,304	64	8,320	239	3.8
	4,608	128	16,640	124	7.3
Manta	32	8	17,600	131	6.9
	64	16	35,200	83	10.9
Sierra	32	8	27,200	97	9.6
	64	16	54,400	69	13

TABLE III
ARES RAYLEIGH–TAYLOR PROBLEM: SPEEDUP ACROSS SYSTEMS

	Cores	Nodes	Agg.B/W (GB/sec)	Runtime (min)	Speedup
Jade	576	16	2,080	164.9	1
	1,152	32	4,160	84.8	1.94
	2,304	64	9,320	43.0	3.83
	4,608	128	18,640	24.6	6.70
Manta	32	8	17,600	–	–
	64	16	35,200	17.7	9.31

TABLE IV
ARES MULTI-PHYSICS PROBLEM: SPEEDUP ACROSS SYSTEMS

We are beginning to realize the potential of Sierra to run high fidelity calculations. For example, for Sierra acceptance, we ran a 97.8 billion zone turbulent mixing problem, in 27 hours using nearly all of the machine (4,096 nodes – 16,384 GPUs). Such a simulation would not be possible on any CPU-only resource allocation. Thus, Sierra makes multiple, smaller-scale, but still very high fidelity runs, practical to complete in a work day, which will be a substantial boon to user productivity.

C. ALE3D

ALE3D is a 2D and 3D Arbitrary Lagrangian-Eulerian (ALE) multi-physics framework whose capabilities include: heat conduction, chemical kinetics and species diffusion, incompressible flow, diverse material models, chemistry models, multi-phase flow, and magneto-hydrodynamics. ALE3D contains over one million lines of C++ code, uses MPI for distributed memory parallelism. It is used for small calculations on commodity workstations, to massively parallel simulations running on hundreds of thousands of processors. RAJA is being integrated into ALE3D for fine-grained on-node parallelism.

Using RAJA, ALE3D currently targets both CPU and GPU architectures and can scale from a single workstation to thousands of processors. To manage host-device memory transfers on Sierra, it uses the CHAI managed array library (see Section III-E). Figure 5 shows ALE3D weak scaling up to 93,318 processes on the Sequoia system (See Table I for details).

Table V presents run times for two ALE3D problems, Sedov and Shaped Charge, on four architectures. The same source code is compiled for different architectures using appropriate RAJA back-ends. ALE3D achieves speedups of up to 5.8× when comparing a single GPU to one node of an Intel Haswell CPU architecture. Using all four GPUs on a node provides additional speedup of 3.4× over one GPU, resulting in overall

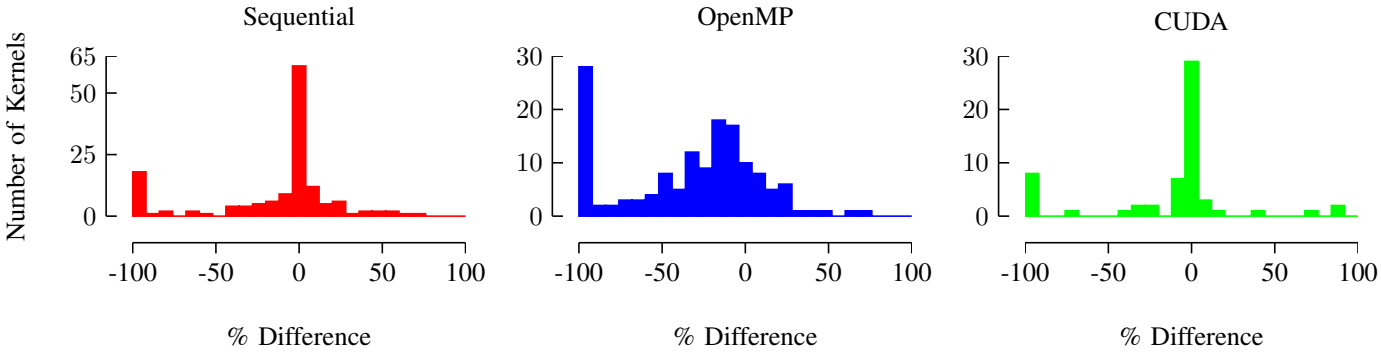

Fig. 1. Performance difference (%) between RAJA and reference variants of Performance Suite kernels on five HPC platforms. Positive values mean that the RAJA variant is faster than the reference.

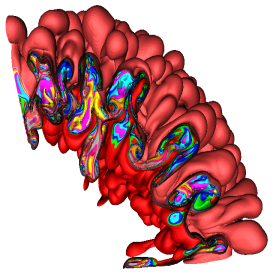

Fig. 2. Ares simulation: Rayleigh-Taylor mixing layer in a convergent geometry, 191.1 million zones.

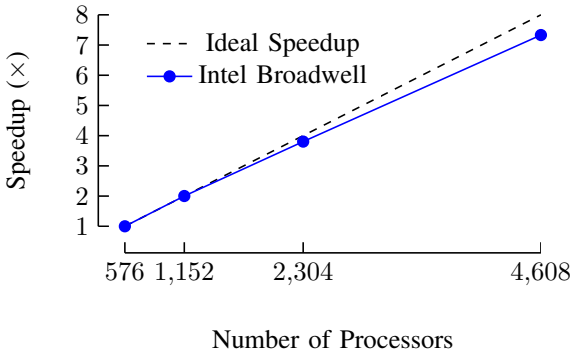

Fig. 3. Ares strong scaling up to 4,068 MPI ranks of Intel Broadwell.

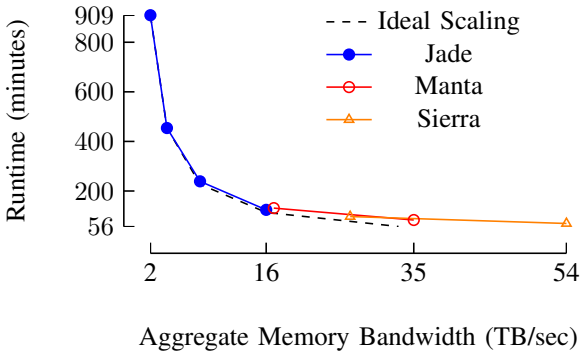

Fig. 4. Ares runtime compared to aggregate bandwidth of run configuration.

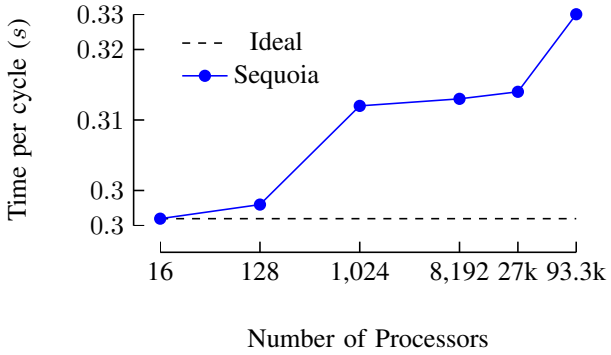

Fig. 5. ALE3D weak scaling up to 93,318 processors of Sequoia.

Problem	Jade	Zin	HasGPU	Manta (1 GPU)	Manta (4 GPU)
Sedov	7.319	10.23	8.288	1.794	0.616
Shaped Charge	113.229	-	173.362	67.187	19.8

TABLE V
ALE3D RUNTIME (SECONDS) USING A SINGLE NODE OF CPU OR GPU ARCHITECTURES WITH THE APPROPRIATE RAJA BACK-END.

speedup of 17× for one GPU-enabled node vs. one CPU-only node. For analyzing performance on both CPU and GPU architectures, the ALE3D team ran studies using three different input problems across four architectures. These used the same code, but changed execution policies to run on different architectures.

D. Ardra

Ardra [21] is a massively parallel neutral particle transport code at LLNL that solves the discrete ordinates form of the linear Boltzmann transport equation [22], a partial differential equation with unknowns spanning seven dimensions in time, angle, energy and space. Ardra is used to simulate nuclear interactions of unbound neutrons and gamma rays as they move through background materials in 1D, 2D and 3D geometries to model nuclear reactors, criticality experiments, shielding problems, detectors, and radiation dosages. Ardra has over 250k lines of C++, C and Fortran, and has historically used an MPI and serial programming model. It uses MPI to decompose space, angle, and energy (6-dimensions) across processes and

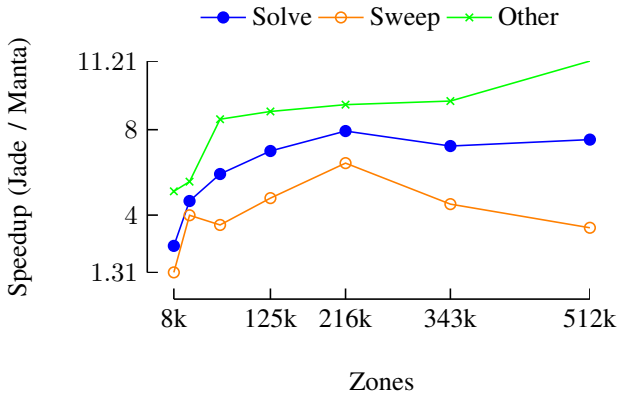

Fig. 6. Ardra speedup on one node of Manta with 4 P100 GPUs vs. one node of Jade with 36 CPU cores using various zone sizes, 48 groups, 80 directions, P4 scattering.

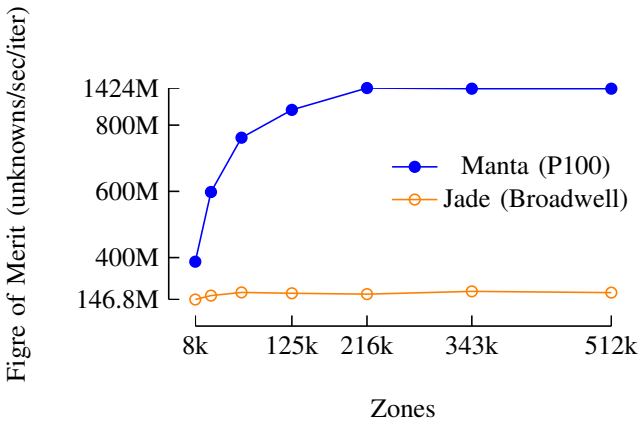

Fig. 7. Ardra unknowns per second on one node of Manta with 4 P100 GPUs vs. one node of Jade with 36 CPU cores using various zone sizes, 48 groups, 80 directions, P4 scattering.

serial execution within each process. Computation kernels in Ardra are mostly matrix-free matrix-vector operations. Workloads vary greatly, from single-process 1D calculations on thousands of unknowns on a personal computer to large 3D problems using 1.5 million MPI ranks and 47 trillion unknowns on Sequoia (system details in Table I).

Using RAJA, Ardra now has a single source code that can execute on both CPU and GPU architectures, including the ability to modify array layouts to optimize multi-dimensional data layouts for each architecture at compile time. To manage host-device data transfers on Sierra, it uses the CHAI managed array library (see Section III-E). Figure 6 shows speedups of various Ardra components when running on four NVIDIA P100 GPUs (one Sierra EA system node), compared to one node (36 cores) of Intel Broadwell CPU. *Sweep* speedup contains the major MPI communications algorithm, while *NonSweep* contains non-MPI algorithms. *Solve* is the combined speedup. This node-for-node comparison shows speedups of nearly 12×, depending on the problem size. A key benefit of the GPU architecture is its high throughput, which helps with high resolution calculations.

Figure 7 shows the overall figure of merit (unknowns per *second*) when running on CPU and GPU-based architectures. These results further highlight the increased efficiency of GPU architectures when dealing with larger problem sizes.

V. Conclusion

This paper presents RAJA, a C++ performance portability layer that is central to the current state of practice for running ASC applications on modern HPC systems at LLNL. The development of RAJA was motivated by the need to address a two-fold problem. First, large production codes require a model for single-source portability to run on advanced technology computing systems as well as various other platforms. Second, a programming model must be sufficiently flexible to adapt to changes in computer architecture trends. Since many applications at LLNL are under continuous development, maintaining hardware-specific versions is unrealistic, and would negatively impact scientific productivity.

RAJA design and features were motivated by key algorithmic patterns and maintenance requirements in real-world applications. This design has proven itself as RAJA has been adopted by several production codes. These application teams have found RAJA sufficiently flexible and robust to integrate it in unique code-specific ways that align with each team's tolerance for code disruption and constraints on code maintainability. Developers with little CS expertise find RAJA reasonably straightforward to use.This is critical since application teams are multi-disciplinary with most members lacking deep knowledge of hardware and parallel programming models. Incremental, selective adoption is achievable in a large codebase and RAJA integrates with existing algorithms and data structures without requiring changes to loop bodies in most cases. RAJA promotes implementation flexibility via clean encapsulation so that changes to execution patterns can be propagated across a large codebase by localizing type changes in header files. The result is that users can achieve good performance by introducing fine-grained parallelism for loops with similar patterns rather than tuning individual kernels. When needed, detailed performance tuning can be done by experts without disrupting application source code that most developers work with.

RAJA has enabled rapid progress across multiple application teams as they prepared for Sierra while maintaining single-source code in a production environment. Also, by easily accessing different programming models, the application developers were able to bring the best tools (debuggers, thread checkers, etc.) to bear when porting code. Furthermore, developers were insulated from negative productivity impacts due to immature vendor compilers and system software on new platforms.

To demonstrate RAJA's suitability as a performance portability model, results from benchmarks and three large-scale ASC production codes were presented in this paper. Prior to RAJA, each of these codes supported distributed memory parallelism via MPI only. Each code developed a different approach for RAJA integration and heterogeneous memory system management, yet arrived at similar results and conclusions. These applications show that RAJA has been used

to successfully port over 2 million lines of code so far. Most importantly, RAJA has enabled each code to develop a single-source, multi-architecture portability solution. GPU-only node runtimes for *Ares*, *ALE3D*, and *Ardra* have shown 13×, 17×, and 12× speedups over CPU-only node runs, respectively.

VI. Future Work

RAJA is in active development and will continue to evolve with the needs of applications as its adoption expands. We plan to share new user experiences and techniques for performance portable application codes in the future.

Future RAJA work will focus on stabilizing interfaces, adding new features, expanding back-end support for other programming models and platforms, and working with vendors and standards organizations to better support the performance and maintenance of portability layers like RAJA. We also plan to develop a flexible asynchronous queue mechanism that will streams for GPUs from multiple vendors. Research has shown great value in RAJA as an auto-tuning tool, but the cost of pre-compiling every variant into a binary can be prohibitive [23]. We are exploring JIT compilation for both CPU and GPU kernels for performance and runtime policy selection. Finally, the RAJA reduction interface, while easy to use compared to options that require explicit parameter passing or separate reduction loop execution methods, is far harder to optimize and more complicated than desired due to difficulty implementing the highly abstract interface. We plan to pursue an alternative reduction interface that, while being more verbose, should be much more performant, especially for OpenMP back-ends.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-788757.

References

[1] RAJA. [Online]. Available: https://github.com/LLNL/RAJA
[2] OpenMP ARB, "OpenMP application programming interface version 4.5," http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, Nov. 2015. [Online]. Available: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[3] "OpenACC 2.0 Application Programming Interface Specification," https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf, Jun. 2013.
[4] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A Hybrid Multi-Core Parallel Programming Environment," in *GPGPU 2007: Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming in CUDA," in *ACM Queue*, vol. 6, no. 2, 2008, pp. 40–53.
[6] "The OpenCL Specification," https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, Nov. 2012.
[7] "HCC: Heterogeneous Compute Compiler," https://gpuopen.com/compute-product/hcc-heterogeneous-compute-compiler/, 2015.
[8] L. Howes and M. Rovatsou, "Sycl integrates opencl devices with modern c++," *Khronos Group*, 2015.
[9] LULESH. [Online]. Available: https://codesign.llnl.gov/lulesh.php
[10] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257
[11] agency-library/agency. [Online]. Available: https://github.com/agency-library/agency
[12] Sierra. [Online]. Available: https://computation.llnl.gov/computers/sierra
[13] Umpire. [Online]. Available: https://github.com/LLNL/Umpire
[14] CHAI. [Online]. Available: https://github.com/LLNL/CHAI
[15] RAJAPerf. [Online]. Available: https://github.com/LLNL/RAJAPerf
[16] R. D. Hornung and J. A. Keasler, "A case for improved c++ compiler support to enable performance portability in large physics simulation codes," Tech. Rep. LLNL-TR-635681, 2013.
[17] F. H. McMahon, "The livermore fortran kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, 1986.
[18] L.-N. Pouchet. (2012) Polybench: The polyhedral benchmark suite. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench
[19] R. Darlington, T. McAbee, and G. Rodrigue, "A Study of ALE Simulations of Rayleigh-Taylor Instability," in *Computer Physics Communications*, vol. 135, 2001, pp. 58–73.
[20] B. E. Morgan and J. A. Greenough, "Large-Eddy and Unsteady RANS Simulations of a Shock-Accelerated Heavy Gas Cylinder," in *Shock Waves*, April 2015.
[21] U. Hanebutte and P. N. Brown, "Ardra, scalable parallel code system to perform neutron and radiation transport calculations," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-TB-132078, 1999.
[22] E. E. Lewis and W. F. Miller, *Computational methods of Neutron Transport*. La Grange Park, IL, USA: American Nuclear Society, 1993.
[23] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2017, pp. 307–316.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [RAJA: PORTABLE PERFORMANCE FOR LARGE-SCALE SCIENTIFIC APPLICATIONS]

A. Abstract

This artifact contains instructions on how to reproduce the experiments described in part IV. A of this paper, using the open-source RAJA Performance Suite. We do not provide instructions on how to reproduce experiments conducted using restricted applications, as these codes cannot be made publicly available. The output of the experiments is in text files in a CSV format, and in this paper was plotted using PGFPlots.

B. Description

1) Check-list (artifact meta information): Fill in whatever is applicable with some informal keywords and remove the rest

- **Algorithm:**
- **Program:** C++ code.
- **Compilation:**
- **Binary:** C++ executables generated by the RAJA Performance Suite.
- **Data set:** N/A.
- **Run-time environment:** The experiments were produced on five different high-performance computing platforms, with a range of CPUs and GPUs.
- **Hardware:**
- **Output:** timings, figures of merit, and speedup values.
- **Experiment workflow:** clone software, configure and build using provided scripts, run executable to generate results.
- **Publicly available?:** Yes.

2) How software can be obtained (if available): All open source software used in this paper is available on GitHub. RAJA can be obtained from the https://github.com/LLNL/RAJA, and the RAJA Performance Suite can be obtained from https://github.com/LLNL/RAJAPerf.

The experiments in this paper used the 0.2.3 version of the RAJA Performance Suite, which is available under the git tag "0.2.3".

3) Hardware dependencies: RAJA will run on most CPUs, and on NVIDIA GPUs. To generate the results in this paper, we used supercomputers with the following node configurations:

- Power8+ CPU and NVIDIA P100 GPU.
- IBM BlueGene/Q CPU.
- Intel Haswell and NVIDIA K80 GPU.
- Intel Haswell CPU and Intel Xeon Phi.

4) Software dependencies: RAJA requires a compiler with support for the C++11 standard. The oldest GCC version supported is GCC 4.9.3. RAJA also requires CMake 3.9.2.

5) Datasets: The RAJA Performance Suite can take as input command line arguments to change the default values for the included benchmark kernels. We used the default values. A complete list of arguments can be obtained by running `./rajaperf.exe --help`.

C. Installation

Clone the RAJA Performance Suite:

```
git clone --branch 0.2.3 --recursive https://github.com/LLNL/RAJAPerf.git
```

Run CMake to configure software, substituting the appropriate C++ compiler:

```
mkdir build && cd build
cmake -DCMAKE_CXX_COMPILER=<path to c++ compiler> ..
```

If running on a machine at Lawrence Livermore National Laboratory or Argonne National Laboratory, you can use the provided scripts to build with specific compilers:

```
./scripts/blueos_nvcc8.0_clang-coral.sh
cd build_blueos_nvcc8.0_clang-coral
make -j
```

D. Experiment workflow

To run the experiments, use the binary created by building the software (described in the previous section):

```
./rajaperf.exe
```

This will generate four files: `RAJAPerf-checksum.txt`, `RAJAPerf-timing.csv`, `RAJAPerf-fom.csv`, and `RAJAPerf-speedup.csv`

E. Evaluation and expected result

The file `RAJAPerf-checksum.txt` should contain two values for each kernel: the checksum value, and the checksum diff. The diff should be 0, indicating that there is no difference in solution between the baseline variant and any other.

F. Experiment customization

The experiment can be customized by varying the command line argument `sizefact`, which is a multiplier on the number of loop iterations for each kernel. For example:

```
./rajaperf.exe --sizefact 2.0
```

will run loops twice as long as the default for each kernel.

G. Notes

The performance of the RAJA Performance Suite is greatly influenced by the compiler version and compiler flags used during compilation. Using newer compilers will typically improve performance. For more information about using RAJA and the RAJA Performance Suite, please visit raja.readthedocs.io, or email the RAJA development team at raja-dev@llnl.gov.