LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-788757

# RAJA: Portable Performance for Large-Scale Scientific Applications

D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, T. R. W. Scogland

September 4, 2019

RAJA: Portable Performance for Large-Scale Scientific Applications
Denver, CO, United States
November 1, 2019 through November 8, 2019

## Disclaimer

# 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 designs ranging from homogeneous CPU machines to GPU or FPGA accelerated systems and many-cores. Achieving good application performance often requires choosing a programming model that is best suited to a particular platform. For large codes that are used daily in production, and are under continual development, architecture-specific ports are untenable. Maintainability requires single-source 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 exploit various programming models, and thus architectures, with a single-source code base. We describe preliminary results with RAJA in three large production applications at Lawrence Livermore National Laboratory, observing $17\times$, $13\times$ and $12\times$ speedup on GPU-only over CPU-only nodes with a single-source codebase in all cases.**

## I. Introduction

In recent decades, performance of high-performance computing (HPC) applications has increased dramatically without requiring major code changes. Developers have been able to focus on algorithm advances and new capabilities as architectures remained largely homogeneous and CPU clock rates increased. Coarse-grained distributed parallelism, MPI typically, was sufficient to achieve high performance on a wide range of platforms. There was little need to adopt additional or alternate on-node parallelism, such as OpenMP multithreading or tasking on-node.

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 procurements of large commodity systems (CTS) and advanced technology systems (ATS) in 3-5 year cycles. The current rapid pace of disruptive changes in ATS node architectures presents substantial performance portability challenges.

RAJA's design grew out of a need to continue supporting large scale production multi-physics applications in ASC, which place significant constraints on portable programming methodologies:

- **Large code bases.** Applications contain $O(100K) - O(1M)$ source lines and many numerical kernels (sometimes $O(15K)$). No kernels may dominate runtime so any portability approach must apply across most of the code base without per-kernel code modification or tweaking.
- **Platform diversity.** Codes are routinely run on laptops (Windows, Linux and Mac OS), commodity clusters, and first-of-a-kind advanced technology systems; they must run well on a diverse set of architectures at any given time.
- **Long service lives.** Codes are used in production daily for *decades* producing critical calculations over their entire operating life. Codes must remain viable over several platform generations.
- **Continual development.** New modeling capabilities are added throughout the lifetimes of most codes. Adopting new technologies cannot disrupt users and feature additions.

Given these constraints, platform-specific variants of such applications are not tenable due to limited developer resources, time and mission priorities. RAJA [1] is a C++ abstraction layer, developed at Lawrence Livermore National Laboratory (LLNL), that enables performance portability within the constraints of production applications. The main goal of RAJA is to enable *manageable performance portability*. Applications are not committed to fixed hardware or software technology choices. Currently, there is no clear "best choice" for all architectures. They also have the option of adopting RAJA incrementally at any scale, from a single kernel at a time to a library. Also, platform-specific data structure and parallel execution concerns must be insulated from source code that most developers work with regularly. Finally, portability must be built into a code and maintained over its lifetime without major disruption to developer and user productivity.

RAJA targets loop kernel 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. Its developers include: computer science researchers and application developers working closely with each other and with compiler, system software, and tool vendors. Thus, its requirements and features are determined directly from needs of production applications.

This paper makes the following specific contributions:

- Introduces RAJA, an open source C++ standards-based

portability layer;
- Presents an evaluation of RAJA in both benchmarks and early results from 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 an evaluation of RAJA in benchmarks and preliminary evaluation in three production applications. Lastly, we present RAJA limitations, future work and conclusions in Section VI and Section V.

## II. Background and Motivation

Portability can be achieved at the 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 multi-threading 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 such as 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 the 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 provides 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 refactoring 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 single-source veneer on top of OpenCL, which would provide portability and single source, but it is currently tied to a single closed-source implementation.

### 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 transfer useful information gained from these exercises back to ALE3D.

*2) Macros:* A simple technique to write portable application code is to use preprocessor macros for architecture-dependent parts of the code. At compile time the macros are replaced with appropriate architecture-specific code. Such a model can achieve high portability and eliminates runtime overhead compared to using a given programming model directly. 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 focuses on portability to older C++ compilers and algorithm memory accesses. 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 several years ago to explore the potential of C++ abstractions to enable performance portability in LLNL ASC applications, which are written mostly in C++. These applications needed to prepare for Sierra [12], a 125 petaflop

advanced technology system – the first production GPU-enabled computing platform at LLNL – while maintaining high performance on other production systems with designs from commodity CPU to IBM BlueGene.

Without an abstraction layer, applications would need to use CUDA or OpenMP directly to target GPUs. CUDA, while a mature technology, was untenable due to substantial development, 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 refactoring, too many fundamental algorithmic or data structure changes, or lacking reasonable support for incremental adoption or both.

RAJA has matured into a powerful set of general performance portable capabilities (see Section III) that have enabled 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 a variety of looping patterns, such as fixed stride or arbitrary indexing (i.e., indirection), with a single kernel; portable reduction types that do not require reduction-specific loop execution mechanisms; heterogeneous memory space support that is decoupled from RAJA and optional and nested loop support 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 relatively minor source code changes. RAJA does this by encapsulating loop and region execution and keeping the application description of the computation the same. For example, consider the following C/C++-style *saxpy* 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];
});
```

The RAJA abstraction accepts a C++ functor, which is usually produced by using the C++11 lambda facility by the user, in this paper we refer to the loop body as the `lambda kernel body`. In real applications, kernel bodies are much larger than our trivial example, thus modifying the loop header only and leaving the kernel body unmodified means that most lines of code in an application do not change as the application is ported to RAJA.

[1]The `RAJA` namespace is included with `using namespace RAJA` for all examples for simplicity

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

### A. Concepts

*1) Execution Policies:* An execution policy is a C++ type that specifies how a loop kernel will execute. RAJA supports several programming model back-ends (described in Section III-B). An execution policy can specify which programming model back-end to use, 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. An execution policy *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 traversal templates, described below. RAJA provides a variety of execution policies; users can also define their own policies to specialize RAJA.

*2) Iteration Spaces:* 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. Objects that model the `RandomAccessContainer` concept are automatically iteration spaces. RAJA iteration space containers and generators are similar to C++ standard library containers and themselves 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 executed together as a unit. Segments directly map to simple for-loop patterns, such as a range or an indirection array. 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. RAJA segments of different types are interchangeable when describing the iteration space.

Index sets provide more 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 of the sparse

space are contiguous. Expressing such contiguous sub-ranges allows RAJA to vectorize the contiguous sub-ranges while still doing sparse operations on the rest.

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

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

Another set of traversal 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 types of scans: 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 traversals. They provide users the ability to perform reduction and atomic operations and are specialized on policies similar to traversal templates. RAJA supplies five different reduction operations, each accepting an execution 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

The *execution policy* used with a reduction must be compatible with the execution policy given to the `forall` or `kernel` construct in which the reduction is used. A simple example is shown below. Note that since they are independent of the traversal, arbitrary other computation can be done in the same loop, along with any number of reductions of arbitrary types. [2]

```
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, execution policies for atomics depend on the execution context in which they are used. RAJA also provides an "automatic" policy for atomics that deduces the correct policy type in GPU/CUDA, OpenMP, and sequential CPU execution contexts. An example of RAJA atomic usage is:

```
double* sum = 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, comp_atomic> sum(&v);
++sum;
sum += 5;
```

### B. Supported Backends

As of release v0.6.0 RAJA supports execution policies for the following back-ends:

- `sequential`: forced sequential execution;
- `simd`: forced SIMD optimizations;
- `loop`: allows compiler to apply SIMD optimizations, not forced;
- `openmp`: OpenMP without offload support;
- `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 (i.e., loops, reductions, scans, and atomics). 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 atomics and `kernel` execution policy specializations. An OpenACC back-end and support for reductions and `kernel` policies in OpenMP with target offload is under development as is a backend for AMD HIP.

### C. Policy Implementation

A RAJA execution policy ties a loop traversal 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 = iter.begin();
  auto end = iter.end();
  auto dist = iter.size();
  for (decltype(dist) i = 0; i < size; ++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 a loop index.

RAJA `seq_exec` and `simd_exec` are similar but have their for-loops decorated with OpenMP or compiler-specific pragmas that either request strictly sequential or SIMD execution, respectively. The CUDA back-end is more complex since

[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.

it must launch a GPU kernel. This kernel takes the 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 uses the `omp_parallel_exec` specialization to establish a parallel region: the 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 can perform loop transformations such as tiling, collapse, and fusion. Kernel policies require that users specify which lambdas should be inserted and where. Multiple lambdas are supported, for example, when data initialization is required and a simple loop nest would not be functionally equivalent.

```
// a policy definition for a 2-nested loop
// with loop permutation
KernelPolicy<
  For<1, loop_exec,
    For<0, omp_parallel_for_exec, Lambda<0>>
  >
>

// a policy definition for a 2-nested loop
// with collapsing
KernelPolicy<
  Collapse<omp_parallel_collapse_exec,
    ArgList<1, 0>,
    Lambda<0>
  >
>
```

The primary difference for users between traversal concepts in RAJA is that `forall` uses a single iteration space object, while `kernel` supports multiple *Iterable* spaces passed as a tuple type. Depending on where the `Lambda` tag is placed within the kernel policy, more than one lambda argument may also be required, allowing not just nested loops, but irregularly nested loops.

In the example below, we show a RAJA kernel for a matrix multiplication:

```
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 via operators that perform integral offset computations. 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.

Various RAJA *layout* types are available to specify at `View` creation the multi-dimensional access pattern for a block of memory. Examples include a standard 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 permute the order of the indices, allowing the memory ordering to be shifted by compile-time calculations.

### E. Memory Model

From the beginning RAJA has made the decision to be agnostic to the mechanism used to make memory available on offload devices. The main reason for this is the overarching goal of remaining non-invasive, while some codes wish to explicitly manage their memory others want to rely on unified memory access or mechanisms they already have for this purpose. Notably, RAJA views and layouts work with arbitrary pointers, but do not themselves manage placement of memory.

That said, there is an associated library called CHAI [13], which performs automatic data copies between CPU and GPU memory spaces based on hooks into RAJA. CHAI complements RAJA by providing a *managed array* abstraction moves data to an execution memory space, as needed, based on RAJA policies.

```
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 `forall` and `kernel` kernels will execute based on the chosen execution policy, ensuring data is moved to the correct memory location. CHAI allowed incremental porting of codes to RAJA while providing effectively manual data management. Also, there was no need to choose between manual data management or vendor-specific solutions like NVIDIA Unified Memory (UM).

### 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, prudent application writers usually define execution policies within header files. Different header files can then be used to compile an application for different platforms. Also, similar loop structures may share execution policies across a large code base. RAJA promotes the notion of parametrization 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; however, 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.

| 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 | 2,388 | Intel Haswell (32 cores) | 9,688 KNL (68 cores) |

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.

## 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 in each case 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. We discuss this for RAJA standalone as well as GPU performance expectations for the applications in the following sections.

### 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 [14] contains a diverse set of kernels to assess performance of RAJA features with different programming models and compilers. Kernels come from stream benchmarks, LCALS [15], [16] and Polybench [17] 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 pre-defined size (number of loop iterations) and number of times it is run to generate execution timings. The Suite is configurable via command line arguments to perform various performance experiments: 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 to report: execution timings, speedup of each RAJA variant with respect to the reference variant, Figure of Merit (run time deviation between RAJA and reference variant), and result checksums (to verify each variant of a kernel produces the same results).

*b) Results:* Here, we present results for 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.

Due to space limitations, Figure 1 shows performance differences between RAJA and reference variants for Sequential, OpenMP, and CUDA as histograms aggregated over all Suite kernels. 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. We believe this to be due to difficulties that C++ compilers have optimizing OpenMP pragmas well when combined with C++ template abstractions because many RAJA kernels that are significantly slower or faster than reference are exercising identical RAJA mechanisms. This is a topic of investigation and we use the Suite to provide simple reproducers of optimization issues to compiler vendors. 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 note that the Suite isolates performance differences between RAJA and reference. In our experience, real applications do not exhibit this stark behavior.

### B. ARES

ARES is a massively parallel, multi-dimensional, multiphysics code at LLNL. Users run ARES for small serial calculations to large-scale simulations on millions of processors to simulate high-explosive experiments, Inertial Confinement Fusion modeling, and hydrodynamic instability experiments [18], [19]. ARES has over 700k lines of C/C++

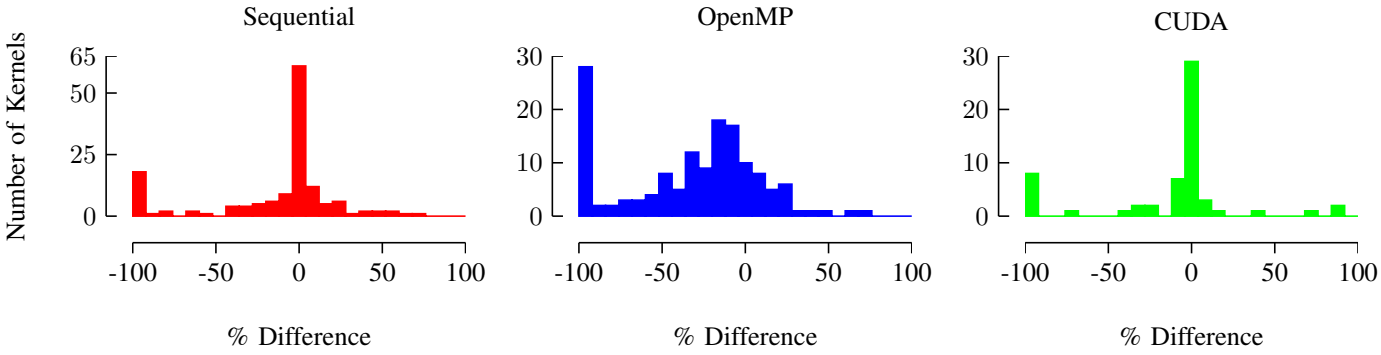


Fig. 1. Performance difference (%) between RAJA and reference variants of Performance Suite kernels on five HPC platforms. Positive values mean that the RAJA variant is faster than the reference.

| Kernel | Sequential | | OpenMP | | CUDA | |
|---|---|---|---|---|---|---|
| | Ref. | RAJA | Ref. | RAJA | Ref. | RAJA |
| PRESSURE | 1.1710 | 2.3769 | 0.2662 | 0.4972 | 0.0651 | 0.0660 |
| ENERGY | 1.7165 | 1.8971 | 0.2051 | 0.3010 | 0.0571 | 0.0566 |
| VOL3D | 1.1050 | 1.0947 | 0.0711 | 0.0848 | 0.0108 | 0.0100 |
| DEL_DOT_VEC_2D | 1.5785 | 1.5260 | 0.0921 | 0.1096 | 0.0131 | 0.0136 |
| FIR | 1.8179 | 1.9050 | 0.2439 | 0.1286 | 0.0122 | 0.0125 |
| LTIMES | 1.8655 | 1.8725 | 0.1026 | 0.1344 | 0.0200 | 0.0211 |

TABLE III
PERFORMANCE RESULTS FOR APPLICATION-BASED KERNELS.

| | 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 IV
ARES RAYLEIGH–TAYLOR PROBLEM: SPEEDUP ACROSS SYSTEMS

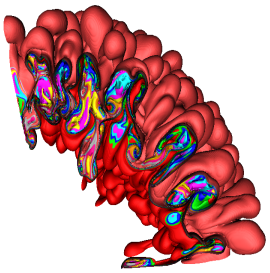

Fig. 2. ARES simulation: Rayleigh-Taylor mixing layer in a convergent geometry, 191.1 million zones.

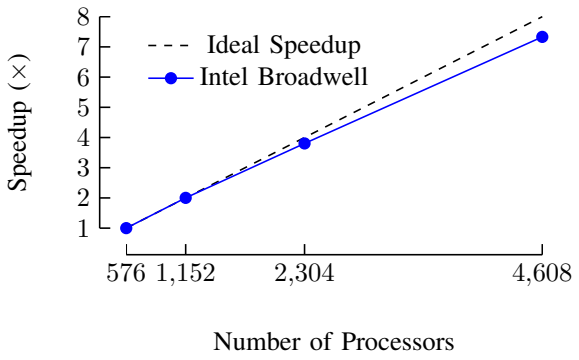


Fig. 3. ARES strong scaling up to 4,068 MPI ranks of Intel Broadwell.

code, uses MPI for distributed memory parallelism, and RAJA for fine-grained parallelism on CPUs and GPUs. Physics capabilities ported to RAJA currently include Lagrange and Arbitrary Lagrangian Eulerian (ALE) hydrodynamics, analytic Equations of State, grey radiation diffusion, material strength models, thermonuclear burn, and sliding surfaces.

We first consider a Rayleigh-Taylor mixing layer in a convergent geometry in Figure 2. The full 3D simulation ($4\pi$) 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 runtime 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 IV, 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 V; the problem can not run on 8 nodes of Manta due to memory constraints. While we have little experience to date running some of the physics packages used in this problem on GPUs, we already see speedup of over 9× on a GPU system vs. a commodity CPU cluster.

We are also beginning to realize Sierra capabilities to run high fidelity calculations. For example, we recently ran a 1.52 billion zone turbulent mixing problem, that would take a very large CPU-only resource allocation, ran in 213 minutes using 64 nodes of Sierra (256 V100 GPUs). This makes several

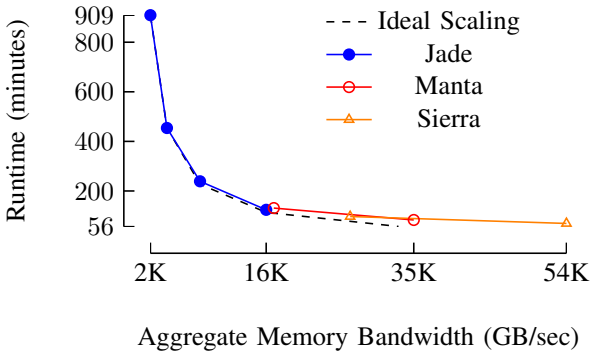


Fig. 4. ARES runtime compared to aggregate bandwidth of run configuration.

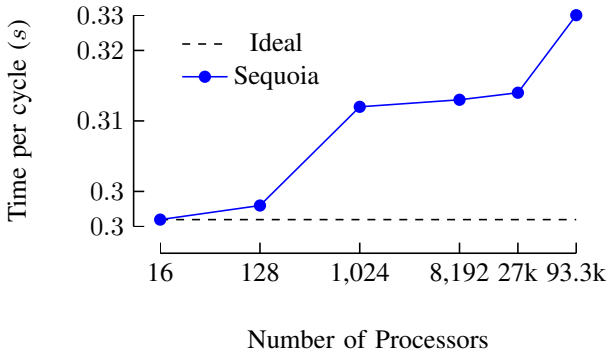


Fig. 5. ALE3D weak scaling up to 93,318 processors of Sequoia.

| | 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 V
ARES MULTIPHYSICS PROBLEM: SPEEDUP ACROSS SYSTEMS

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

high fidelity runs practical to complete in a work day, which is not possible otherwise.

### C. ALE3D

ALE3D is a 2D and 3D Arbitrary Lagrangian-Eulerian (ALE) multiphysics 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, and is used for small calculations on a commodity workstation, 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. Figure 5 shows ALE3D weak scaling up to 93,318 processes on the Sequoia system (See Table I for details).

Table VI presents runtime of 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 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. The results of these studies are presented in the same code, but changing the execution policy, allows ALE3D to achieve speedups of up to 5.8× when comparing a single GPU to one node of a CPU-based architecture like Intel "Haswell". Adding an additional 3 GPUs speeds the code up another 3.4×, meaning a whole node of GPUs can be up to 17× faster than a whole node of CPU resources.

### D. ARDRA

ARDRA [20] is a massively parallel neutral particle transport code at LLNL that models the nuclear interactions of unbound neutrons and gamma rays as they move through background materials in 1D, 2D and 3D geometries. ARDRA solves the Discrete Ordinates form the Linear Boltzmann Transport Equation[21], a PDE with unknowns spanning seven dimensions in time, angle, energy and space. ARDRA is used to model nuclear reactors, criticality experiments, shielding problems, model 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 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).

Due to RAJA, ARDRA now has a single source code that can execute on both CPU and GPU architectures. 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.

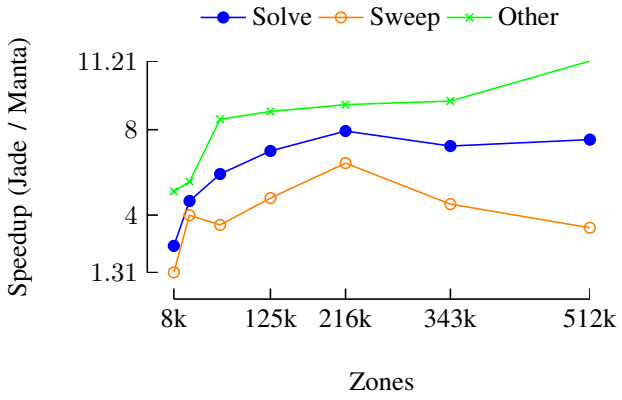


Fig. 6. ARDRA speedup on one node of Manta with 4 P100 GPUs vs. one node of Jade with 36 CPU cores using various zone sizes, 48 groups, 80 directions, P4 scattering.

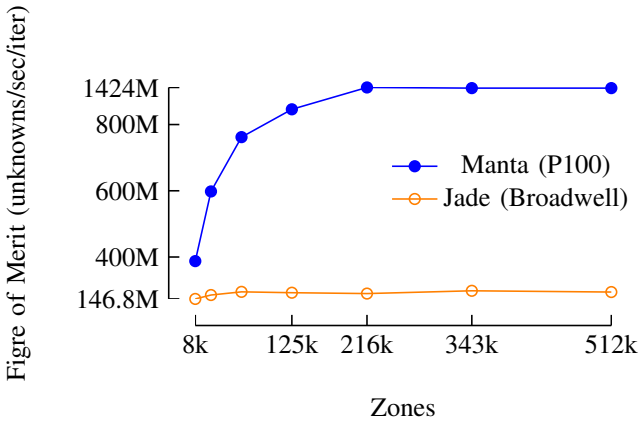


Fig. 7. ARDRA unknowns per second on one node of Manta with 4 P100 GPUs vs. one node of Jade with 36 CPU cores using various zone sizes, 48 groups, 80 directions, P4 scattering.

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. The first is to enable portability in production codes to realize performance on next generation supercomputers. The second is to provide a flexible programming model that can 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 production applications. This design has proven itself through its adoption in several production codes. Developers with little CS expertise find it straightforward to use; 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 code; RAJA integrates with existing algorithms and data structures and it does not require changes to loop bodies in most cases. RAJA promotes implementation flexibility via clean encapsulation. Changes to execution patterns can be propagated across a large code base by localizing type changes in header files; users achieve good performance by parallelizing loop patterns rather than individual kernels. Detailed performance tuning can be performed by experts without disrupting application source code that most developers work with.

RAJA has enabled rapid progress in enabling applications to prepare for Sierra while maintaining single-source applications in a production environment. Also, by easily accessing different programming models, application developers can bring the best tools (debuggers, thread checkers, etc.) to bear when porting code. In addition, developers can be 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 ASC production codes were presented. Prior to RAJA, each of these codes was parallelized solely for distributed memory parallelism via MPI. Each code developed a different approach for RAJA integration, 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.

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.

## VI. Future Work

Future work will focus on stabilizing interfaces, the expansion of RAJA back-ends 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. Better access to shared or constant memory on GPUs is a high priority for performance. We also plan to develop a flexible asynchronous queue mechanism that will support CUDA streams. 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 [22]. 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.

## 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-748585.

## 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] CHAI. [Online]. Available: https://github.com/LLNL/CHAI
[14] RAJAPerf. [Online]. Available: https://github.com/LLNL/RAJAPerf
[15] 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.
[16] F. H. McMahon, "The livermore fortran kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, 1986.
[17] L.-N. Pouchet. (2012) Polybench: The polyhedral benchmark suite. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench
[18] 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.
[19] 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.
[20] 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.
[21] E. E. Lewis and W. F. Miller, *Computational methods of Neutron Transport*. La Grange Park, IL, USA: American Nuclear Society, 1993.
[22] 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 Ph accelerator.

*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 primarily by varying the command line argument `sizefact`. This value is a multiple used to increase the number of iterations each kernel is run for. For example:

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

will run twice as many iterations for ever 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.